COMPSCI 3MI3 - Principles of Programming Languages

Topic 10 - References

NCC Moore

McMaster University

Fall 2021

Adapted from "Types and Programming Languages" by Benjamin C. Pierce

Pure vs Impure Features

So far, all of the language features we have considered have been **pure**.

- That is, features whose operation and results stay within the confines of the existing program.

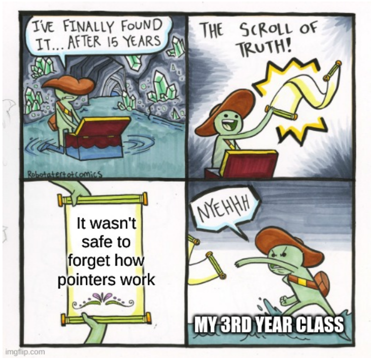
All practical languages, including the "purely function language" Haskell, must contain **impure** features in order to be useful.

- These are features which have effects exterior to the program itself. Examples include:
  - Assignment to mutable variables; file I/O; displaying data; network connections; interprocess communication; etc.
- These are more generally referred to as **computational effects**.

(Re)Enter Mutable References!

So far, our calculus has been incapable of describing and reasoning about computational effects.

- In this topic, we're going to show how **mutable references** can be formalized and added to our calculus.
- A mutable reference is a reference to a location in a data storage device.


I'VE FINALLY FOUND IT... AFTER 15 YEARS
THE SCROLL OF TRUTH!
It wasn't safe to forget how pointers work
NYEHHH
MY 3RD YEAR CLASS
imgflip.com

For Your Next Assignment...

Almost all programming languages provide an assignment operation, including Haskell!

- When you use the $<-$ operator to jailbreak[1] data from the IO monad you use assignment to do so!

In our case, assignment is an operation which changes the contents of a previously allocated piece of storage.

- For our discussion, we're going to abstract away messy details like what exactly the data is being stored in and how.

[1]that's not official terminology, that's just what it feels like.

How Does This Assignment Work?

In general, variable storage is comprised of three operations:

- Memory allocation, aka **referencing**.
- A “store” operation, aka **assignment**.
- A “retrieve” operation, aka **dereferencing**.

Depending on the programming language, some or all of these operations may be made implicit by the language’s grammar.

- Python hides allocation and retrieval, but storage is explicit.
- C/C++ hides retrieval, with allocation and storage being explicit.
- In ML, all three operations are explicit.

Retrieval is implicit in Python and C because, when we retrieve a variable’s value inside an expression, there is no explicit retrieval operator. In ML, there is an explicit retrieval operator.

Operation!

In order to formalize memory interactions, it makes sense to keep these three operations separate and explicit. We will therefore follow the ML model.

- We will use the expression `ref <value>` to denote the allocation of a memory cell.
  - Note that an initialization value will be required at the syntactic level.
  - This operator creates a reference to a newly allocated memory cell, which must be captured in order to be preserved.
  - We'll talk later about what does the capturing. For now, imagine something like a C++ program's namespace.
- We will use `r := <value>` to denote storage of a value in an *existing* memory cell.
- We will use `!r` to denote retrieval from a memory cell.

Typing Rules for Reference Operations

Of course, if we have any new additions to our language, we need to talk typing!

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{ref}\ \ t_1 : \mathit{Ref}\ \ T_1} \qquad \text{(T-Ref)}$$

$$\frac{\Gamma \vdash t_1 : \mathit{Ref}\ \ T_1}{\Gamma \vdash !t_1 : T_1} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \vdash t_1 : \mathit{Ref}\ \ T_1 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \mathit{Unit}} \qquad \text{(T-Assign)}$$

Making References

- For reference creation, it would make sense for the type of `ref <value>` to be the type of the value.
- However, this isn't accurate! `ref` doesn't create a value, it creates a **reference to** a value.
- Therefore, let's add a new type to our calculus.

$$
\begin{array}{rl}
\langle T \rangle & ::= \ \ldots \\
| & \text{Ref } \langle T \rangle
\end{array}
$$

So, just as C pointers are typed as *references to* some other data type, we set the type of our created reference as a **reference to** some other type already in our calculus.

Dereference Went Over His Head

The dereferencing operation has very straightforward typing.

- The type contained within *Ref* is the type of the dereferenced reference.

Assignment, however, has some interesting subtleties:

- First, the assignment itself has type *Unit*, which will allow us to perform sequencing.
- Second, note that the term being assigned is constrained via antecedent to be the same type as the type contained by *Ref*.
- The term being written, and the cell being written to **must have the same datatype**, or the term is untypable.

A Sample Program

Using the above semantics, we are able to construct programs such as the following:

```
x = ref 0;
y = ref 0;
z = ref 0;
x := 2;
y := 2;
z := !x + !y;
!z
>> 4
```

Notice how we are initializing these three variables.

- The $=$ sign, rather than $:=$, tells us that we are not assigning the results of $ref\,0$ to a location represented by x.
- The use of $=$ is the same as expression renaming in pure lambda calculus (`add`, `pred`, etc.).

Alias James Bond

In the previous example, we used variables x, y and z to store *references to* some memory storage unit. Let's explore some of the subtleties of our memory system by considering the following program:

```
x = ref 5;
y = x;
x := 10;
!y
>> 10
```

- Note that, although we never assign to y, the changes to x are seen when we dereference *y*.
- This is because both x and y reference the same memory cell!
- In order for x and y to reference independent values, we would have had to allocate a new memory cell to y.

Replication in Snake Language

Aliasing is also something you can do in Python with mutable data structures.

```
>>> L = [1,2,3]
>>> M = L
>>> L[1] = "potato"
>>> M
[1,"potato",3]
```

- In Python, this is a well known noob trap. An inexperienced programmer will think they are copying the data structure, when they are in fact only copying the reference.
- Programming experts such as ourselves, however, can put aliasing to good use!

Sharing is Caring

We can pass the results of calculations around our program using aliased memory cells as *implicit communication channels*!

```
c = ref 0
inc_c = λx : Unit. (c := succ (!c); c!) ;
dec_c = λx : Unit. (c := pred (!c); c!) ;
inc_c unit;
>> 1
inc_c unit;
>> 2
dec_c unit;
>> 1
```

- We declared two functions, `inc_c` and `dec_c`, which'
  - assign to `c` either the successor or the predecessor of it's current value, and
  - give the current value of `c` as a result.

Decrement Inc.

If we then package `inc_c` and `dec_c` as a record[2] as follows...

```
o = {i = inc_c, d = dec_c};
```

- This allows us to pass around both functions (and their shared state) as a unit.
- In effect, we have constructed a very rudimentary **object**.
- In Object Oriented Programming, an object combines:
  - Some set of persistent data.
  - Operations over said data.

  And allows the programmer to work with the object as a singular entity.

[2]we'll be talking about records and other data structures next week

Garbage Collection

We've talked quite a bit about memory allocation, but what about memory **deallocation?**

- Our language intentionally does not contain an explicit deallocation operation. Such operations are not, in general, **type safe**.
- Consider the following scenario:
  - Process A allocates a *Nat* memory cell.
  - Process B creates an alias to said memory cell.
  - Process A deallocates the memory cell.
  - Process B now has a **Dangling Reference**.
- The fact that process B has a reference to a memory cell which is no longer allocated is *precisely* the sort of situation we want to avoid!

Smelly, Smelly Garbage

- What if our scenario continued:
  - Process C allocates a *Bool* memory cell, and is given *the same cell that process A received!*
- There is no guarantee that a deallocated memory cell won't be reused by the same program.
- If reused, there is no guarantee that a memory cell will receive the same type of value as before.
- Thus, Process B thinks it has a reference to a *Nat* memory cell, when in fact the memory cell contains a *Bool*.
- This is a clear violation of our type system!

Collection Day is Friday!

The way most modern languages solve the dangling reference problem is through a run-time process known as **garbage collection**.

- Garbage collection requires access to the table of allocated memory cells.
- The program is periodically paused, and checked that it contains at least one valid reference to each entry in the table.
- Any orphaned memory cells are then deallocated.
  - Generally, it is impossible to reconstruct a reference to a memory cell once lost.
  - Deallocation of memory cells to which no references point is generally considered safe.
- Garbage collection is like the sound effects in a video game. It's doing it's job if you forget it's even there.

Modelling Memory Storage

Modelling Memory Storage

One question that should occur naturally to us, now that we have expanded our set of types to include references, is what does a reference value look like?

- So far, we have been renaming such values with variables, without actually exploring the concept.
- In our previous example programs, we have been using our intuitions about how computer memory works to fill in *how* memory is stored and retrieved.

Memories of the Real World

In most computer languages, memory is modelled as an array of bytes.

- The run-time system keeps track of what bytes are currently in use.
- When more memory is requested, the size of the requested memory is passed to the run-time system.
- A free region large enough for the data being allocated is found (or not).
- The run-time system then marks that region as being used, and returns the index of the newly allocated region.

This is essentially how dynamic memory allocation works in C. The address returned by `malloc()` is a **memory reference value**.

Abstraction Distraction!

For our purposes, such messy implementation details are unnecessary.

- The first thing we don't need to know is how the values are being stored.
  - i.e., that the data is stored in binary code in a RAM chip (for example).
- We can think of our **memory store** as an array of **values**, rather than of **bytes**.
- We also don't need to know that references themselves are numbers.

Memory Location Set!

- Let us take memory references to be elements of some uninterpreted set $\mathcal{L}$ of **store locations**.
  - l will be a metavariable ranging over $\mathcal{L}$.
  - We will add l to the terms of our language, to mean "a memory location value."
- The memory store itself can then be simply a partial function from locations $l \in \mathcal{L}$ to our set of values $\mathcal{V}$
  - We will use the metavariable μ to range over the *set of possible memory stores*.
  - In our equations, we will use μ to mean "some arrangement of memory."

From now on, we will refer to references as **locations**, and the memory storage device as simply the **store**.

Getting C Sick

By modelling memory this way, we are *intentionally distancing ourselves* from the way that memory is managed by C (and family).

- Because we are not interpreting our locations as numbers, we don't have to worry about what happens if we try to perform arithmetic over them.
- While extremely useful, pointer arithmetic is *notoriously problematic* when it comes to type systems.
    - If some location n is typed `float`, this tells us nothing about what might be stored at a location like $n + 4$.
    - How would we go about typing an operation like this without actually executing the program?

The Mysterious 151st Pokémon

We can conceive of the context provided by μ similarly to how we already talk about the typing context Γ.

- Rather than being attached to the typing relation, μ will be attached to the terms themselves:

$$t \mid \mu \tag{1}$$

When considering the evaluation of t, it is possible that evaluation will have a side effect on μ. We therefore modify the general form of evaluation, $t \to t'$ by adding two different state contexts:

$$t \mid \mu \to t' \mid \mu' \tag{2}$$

μtual Affection

- If no side effects happened, $\mu = \mu'$.
- Otherwise, the difference between μ and μ' represents the effect on the state of the evaluation of the term t.
- In effect, we have enriched our notion of *abstract machines*, so that a machine state is not just a program counter (represented as a term), but a program counter plus the current contents of the store.

Teenage μtant Ninja Turtles!

Let's refactor our evaluation rules for pure λ-Calculus in light of the new rules.

$$(\lambda x : T_{11}.t_{12})v_2 \mid \mu \to [x \mapsto v_2]t_{12} \mid \mu \qquad \text{(E-AppAbs)}$$

$$\frac{t_1 \mid \mu \to t_1' \mid \mu'}{t_1\ t_2 \mid \mu \to t_1'\ t_2 \mid \mu'} \qquad \text{(E-App1)}$$

$$\frac{t_2 \mid \mu \to t_2' \mid \mu'}{t_1\ t_2 \mid \mu \to t_1\ t_2' \mid \mu'} \qquad \text{(E-App2)}$$

μ Suede Shoes

Let's examine some of the subtleties.

- Note in the two congruence rules E-App1 and E-App2 we presume μ may be effected by the evaluation of t_1 and t_2.
  - Whether or not this is true, the change is reflected in the overall evaluation.
- However, with E-AppAbs, we have the same μ before and after.
  - This is one way of stating that function application *has no side effects!*
  - The only operations we have so far which effect state are allocation and assignment. Since there is no sub-evaluation implied, we know that μ will remain unchanged.
    - This property will be very useful later on!

Here are my Terms.

Formally, we will add the following terms to our calculus.

$$
\begin{aligned}
\langle t \rangle ::= \; & \ldots \\
\mid \; & \text{ref t} \\
\mid \; & \text{!t} \\
\mid \; & \text{t := t} \\
\mid \; & l
\end{aligned}
$$

Further, our complete list of values becomes:

$$
\begin{aligned}
\langle v \rangle ::= \; & \lambda\, \text{x:}\langle T \rangle.\langle t \rangle \\
\mid \; & \text{unit} \\
\mid \; & l
\end{aligned}
$$

For Internal Use Only!

While we have added l to our calculus, we do not necessarily intend for a programmer to ever use it.

- We intend for l to encode intermediate, implicit results of calculations, rather than explicit ones.
- In other words, we are working on the **internal language**.
  - l is available for constructing derived forms, but our goal is that locations can be safely ignored by the programmer.[3]

[3]unlike in C programming.

Dereferencing

Let's now formalize our evaluation rules for *dereferencing*.

$$\frac{t_1 \mid \mu \to t_1' \mid \mu'}{!t_1 \mid \mu \to !t_1' \mid \mu'} \qquad \text{(E-Deref)}$$

$$\frac{\mu(l) = v}{!l \mid \mu \to v \mid \mu} \qquad \text{(E-DerefLoc)}$$

By now these rules should have a familiar form.

- The congruence rule E-Deref lets us evaluate t_1 until a value is reached.
- Then, if a location is the result of evaluating t_1, and if $l \in dom(\mu)$, we can replace $!l$ with the retrieved value.
- Note that dereferencing anything other than l is disallowed by the set of evaluation rules available.

Assignment

$$\frac{t_1 \mid \mu \to t_1' \mid \mu'}{t_1 := t_2 \mid \mu \to t_1' := t_2 \mid \mu'} \qquad \text{(E-Assign1)}$$

$$\frac{t_2 \mid \mu \to t_2' \mid \mu'}{v_1 := t_2 \mid \mu \to v_1 := t_2' \mid \mu'} \qquad \text{(E-Assign2)}$$

$$l := v_2 \mid \mu \to unit \mid [l \mapsto v_2]\mu \qquad \text{(E-Assign)}$$

- Once again, we are required to fully evaluate t_1 and t_2 via congruence rules before performing the assignment.
- The assignment itself requires a location and a value.
- The notation $[l \mapsto v_2]\mu$ means "a store which maps l to v, with all other locations mapping to the same things as in μ."

Allocation

$$\frac{t_1 \mid \mu \to t_1' \mid \mu'}{\mathit{ref}\ t_1 \mid \mu \to \mathit{ref}\ t_1' \mid \mu'} \qquad \text{(E-Ref)}$$

$$\frac{l \notin \mathit{dom}(\mu)}{\mathit{ref}\ v_1 \mid \mu \to l \mid (\mu, l \mapsto v_1)} \qquad \text{(E-RefV)}$$

To evaluate allocation...

- We must first evaluate t_1 to a value using the congruence rule E-Ref.
- Then, we select a *fresh location* l not already used in μ.
- We extend μ with a mapping between this fresh location and the given value.
- The term itself directly evaluates to this fresh location.

Store Typing

Insert Meme Here

Let's Talk Typing!

Now that we have formalized the storage and retrieval of **values** from storage, one question remains, "What about the typing?"

- We will answer this question by elaborating on some of our natural intuitions.

We could start by saying that the type of a location in storage should be the type of the value stored there. This would yield a typing rule such as the following.

$$\frac{\Gamma \vdash \mu(l) : T_1}{\Gamma \vdash l : \mathit{Ref} T_1} \tag{3}$$

That is, to type a location, we look up the value at that location and type it.

Four Place Relation!

In effect, by making the type of a location dependent on the program state, we have changed the typing relation from:

- A three place relation from a typing context Γ and a term t to a type T. ($\Gamma \vdash t : T$)

Into:

- A four place relation from a typing context Γ, a memory state μ, and a term t to a type T. ($\Gamma \mid \mu \vdash t : T$)

Thus, our typing rule is refined to:

$$\frac{\Gamma \mid \mu \vdash \mu(l) : T_1}{\Gamma \mid \mu \vdash l : Ref T_1} \tag{4}$$

All the rest of our typing rules will need similar modification. Most won't do anything interesting with μ, but most will need to pass μ to sub-derivations.

Problematic Content

This approach has two big downsides, however.

- First, we have made typing a term dependent on typing another term. If we have a situation like the following, where terms stored in memory are dependent on other terms stored in memory, things can get quite inefficient.

$$
\begin{aligned}
(l_1 &\mapsto \lambda x : Nat.999, \\
l_2 &\mapsto \lambda x : Nat.(!l_1)x, \\
l_3 &\mapsto \lambda x : Nat.(!l_2)x, \\
l_4 &\mapsto \lambda x : Nat.(!l_3)x, \\
&\qquad\qquad\qquad\ldots)
\end{aligned}
$$

Not only could we potentially have to type a **lot** of locations to get the type of one location in particular, we have to *recalculate it each time!*

Even More Problematic Content

This is not even as bad as things can get! What if our memory locations were storing circular references?

$$(l_1 \mapsto \lambda x : \mathit{Nat}.(!l_2)x,$$
$$l_2 \mapsto \lambda x : \mathit{Nat}.(!l_1)x)$$

- Our current typechecker would get stuck in an infinite loop!
- It turns out that such structures do exist in practice, like doubly linked lists.

Sigma!

Surely we can make this more efficient. Let's recall the following:

- The type of a location is derivable upon allocation from the type of the instantiating value.
- According to our semantics, we are only allowed to assign values to locations of a matching type.
- The thing we are constantly recalculating *has been fixed from the start!*

So, rather than relying on our memory store μ, let's create a new **typing store** Σ!

- We are expanding the definition of allocation so that, when a new location is allocated, a corresponding entry in Σ is created.
- Just as μ is a partial function from locations to values, Σ is a partial function from locations to types.

Super Sigma Fighter!

$$\frac{\Sigma(l) = T_1}{\Gamma \mid \Sigma \vdash l : \mathit{Ref}\ \ T_1} \qquad \text{(T-Ref)}$$

- Γ starts off empty, and has typings added as the program is evaluated.
- Σ is used the same way.
- Just as an empty Γ is written $\emptyset$, an empty Σ will be written the same.
- As evaluation progresses, Σ will acquire more and more typings.

Typing Allocation

Now that we've got location typing under control, the rest of the typing rules fall into place much sac we would expect.

$$\frac{\Gamma \mid \Sigma \vdash t_1 : T_1}{\Gamma \mid \Sigma \vdash ref\ t_1 : Ref\ T_1} \qquad \text{(T-Ref)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : Ref\ T_{11}}{\Gamma \mid \Sigma \vdash !t_1 : T_{11}} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash t_1 : Ref\ T_{11} \qquad \Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 := t_2 : Unit} \qquad \text{(T-Assign)}$$

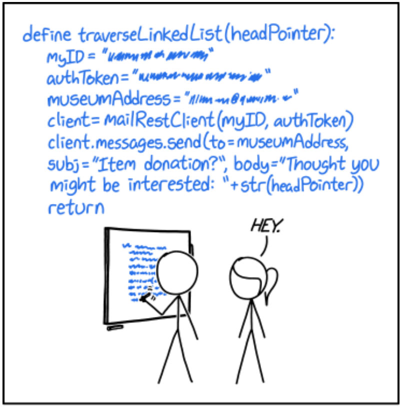
I Wish I'd Used a Serif Font For This Class

There is only one subtlety here:

- In T-Ref, note that, although this is an allocation operation, we make no changes to Σ via this typing rule.
  - This is because the typing store is updated by **evaluation**, not `typechecking`.
  - This is not implied by the inference rule for allocation in the textbook, as Σ doesn't appear anywhere in it.
  - The conjecture of your humble professor is that the textbook overlooked this, probably because it would mean carrying both μ and Σ through each evaluation rule as well.
  - If we added Σ to E-RefV, it would look something like this:

$$\frac{l \notin dom(\mu) \qquad \Gamma \mid \Sigma \vdash v_1 : T_1}{ref\ v_1 \mid \mu \mid \Sigma \rightarrow l \mid (\mu, l \mapsto v_1) \mid (\Sigma, l \mapsto T_1)} \quad \text{(E-RefV)}$$

Type Safety of References



```
define traverseLinkedList(headPointer):
    myID = "[illegible]"
    authToken = "[illegible]"
    museumAddress = "[illegible]"
    client = mailRestClient(myID, authToken)
    client.messages.send(to=museumAddress,
    subj="Item donation?", body="Thought you
     might be interested: "+str(headPointer))
    return
```

CODING INTERVIEW TIP: INTERVIEWERS GET REALLY MAD WHEN YOU TRY TO DONATE THEIR LINKED LISTS TO A TECHNOLOGY MUSEUM.

Progress and Preservation Strike Again!

Only one thing remains: to verify the soundness of this extension of our calculus.

- Remember, we demonstrate the soundness of our type system by demonstrating:
  - Progress
    - A well-typed term is either a value, or may be evaluated once under single-step operational semantics. That is, a well-typed term is not currently stuck.
  - Preservation
    - If a term is well-typed, and we evaluate it once under single-step operational semantics, the resulting term is also well typed.
- Progress is easy enough to be made an assignment question, but preservation is somewhat more interesting.

Poorly Preserved

Since we have extended both the evaluation and typing relations with value and type storage, we need to update our preservation theorem accordingly.

- Intuitively, we might construct the following.

$$(\Gamma \mid \Sigma \vdash t : T) \wedge (t \mid \mu \to t' \mid \mu') \implies (\Gamma \mid \Sigma \vdash t' : T) \quad \text{(Wrong!)}$$

- There is nothing in this approach that requires consistency between the way Σ and μ have been constructed.
- There is an implicit assumption that, if you look up a location in both μ and Σ, the type given by Σ will be valid for the value found in μ.
- Essentially, we need to define well-typedness for stores.

Well Typed Documents

Definition

A store μ is said to be **well typed** with respect to a typing context Γ and a store typing Σ if:

- $\mathit{dom}(\mu) = \mathit{dom}(\Sigma)$
- $\forall l \in \mathit{dom}(\mu) \mid \mu(l) : \Sigma(l)$

We write this $\Gamma \mid \Sigma \vdash \mu$

Less Poorly Preserved

Let's include this new idea of the well-typedness of a store to our preservation theorem.

$$\begin{aligned} &(\Gamma \mid \Sigma \vdash t : T) \\ \land\ &(t \mid \mu \to t' \mid \mu') \\ \land\ &(\Gamma \mid \Sigma \vdash \mu) \\ \implies &(\Gamma \mid \Sigma \vdash t' : T) \end{aligned}$$

We're getting closer, but this overlooks one crucial fact. The typing store may change during evaluation!

Preservation

We already recognized that μ can grow through evaluation, so we can hypothesize the existence of some Σ' to which a similar transformation has been applied.

THEOREM: [Preservation]

$$\begin{aligned}
& (\Gamma \mid \Sigma \vdash t : T) \\
& \wedge (t \mid \mu \to t' \mid \mu') \\
& \wedge (\Gamma \mid \Sigma \vdash \mu) \\
\implies & (\exists \Sigma' \supseteq \Sigma \mid \\
& \qquad (\Gamma \mid \Sigma' \vdash t' : T) \\
& \quad \wedge \ (\Gamma \mid \Sigma' \vdash \mu') \\
&)
\end{aligned}$$

Weakness is our Strength

One subtlety here is the fact that we've made no attempt to formalize how Σ' might be derived from Σ.

- We did this a bit earlier, when talking about E-RefV.
- The claim that we're making here is substantially weaker than "Σ' has this precise relation to Σ."
- If we can prove this weaker claim, "μ' and t' are both well typed under some expanded version of Σ," we have achieved our goal without having to get into a lot of extraneous detail.
    - Essentially, adding this extra detail substantially complicates the theorem without making it any more useful.

Make Life Rue the Day It Gave Cave Johnson Lemmas!

In order to prove preservation, we will need the following three technical lemmas.

LEMMA: [Preservation Over Substitution]

$$(\Gamma, x : S \mid \Sigma \vdash t : T) \wedge (\Gamma \mid \Sigma \vdash s : S) \implies (\Gamma \mid \Sigma \vdash [x \mapsto s]t : T]) \tag{5}$$

LEMMA: [Preservation Over Storage]

$$(\Gamma \mid \Sigma \vdash \mu) \wedge (\Sigma(l) = T) \wedge (\Gamma \mid \Sigma \vdash v : T) \implies (\Gamma \mid \Sigma \vdash [l \mapsto v]\mu)) \tag{6}$$

LEMMA: [Weakening Over Typing Stores]

$$(\Gamma \mid \Sigma \vdash t : T) \wedge (\Sigma' \supseteq \Sigma) \implies (\Gamma \mid \Sigma' \vdash t : T) \tag{7}$$

I Don't Want Your Stupid Lemmas!

For the previous lemmas, proofs are not interesting enough for us to go through in detail.

- Preservation over substitution is proven in the same way as we proved it in topic 8.
- Preservation over storage is immediate from the definition of $\Gamma \mid \Sigma \vdash \mu$.
- Weakening over typing stores can be shown via a fairly easy induction.

Preservation VIII

Proof Sketch Of Preservation

- Straightforward induction on evaluation derivations, using the provided lemmas, plus the inversion property of our new typing rules (itself a straightforward extension of the inversion lemma of simply typed λ-Calculus).

Failure Can Still Be Progress!

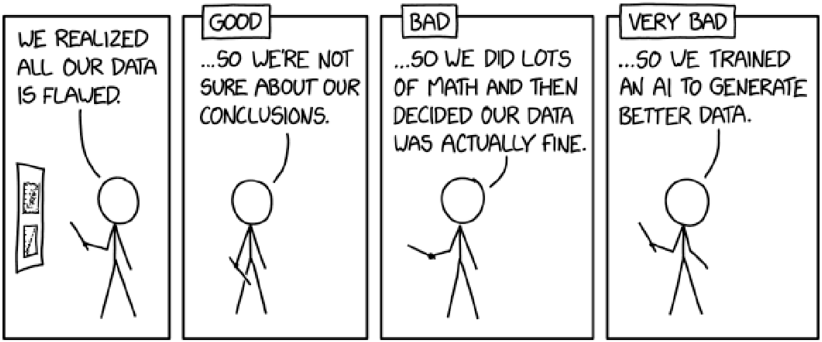
The theorem of progress, formally stated is:

THEOREM: [Progress]
Suppose t to be a closed, well typed term (that is, $\emptyset \mid \Sigma \vdash t : T$ for some T and Σ). Then either t is a value, or else, for any store μ such that $\emptyset \mid \Sigma \vdash \mu$, there is some term t' and store μ' such that $t \mid \mu \rightarrow t' \mid \mu'$.

Proof Sketch

- Straightforward induction on typing derivations, following the pattern established in topic 8.
- The canonical forms lemma needs two additional cases, stating that all values of type *Ref T* are locations, and similarly for *Unit*.

Last Slide Comic


WE REALIZED ALL OUR DATA IS FLAWED.
GOOD
...SO WE'RE NOT SURE ABOUT OUR CONCLUSIONS.
BAD
...SO WE DID LOTS OF MATH AND THEN DECIDED OUR DATA WAS ACTUALLY FINE.
VERY BAD
...SO WE TRAINED AN AI TO GENERATE BETTER DATA.