TAL inference

Lily Lin

Code: https://github.com/rctcwyvrn/tal-inference

Run dockerfile: `run.sh`

1 Motivations

In the TAL paper we saw that typed assembly was able to provide a number of useful safety properties for assembly, like not getting stuck and not accessing unallocated memory. However we also saw that TAL requires a set of label types, either generated by a compiler or handwritten by the programmer. Since there are many existing assembly programs that lack these annotations, as well as compilers that aren't designed to output types in the back-end, an interesting idea would be to implement a type inference system for TAL.

This project attempts to do exactly that, inferring all types for TAL with jumps and pointers. The goals are to prevent the same classes of issues that the original TAL and the ATAPL TAL attempted to avoid, namely getting stuck and out of bounds memory errors.

2 Syntax

The instructions are

- `add r1 r2 v`
- `sub r1 r2 v`
- `mov r1 r2`
- `bnz r2 v`
- `load r1 r2 i`
- `store r1 i r2`
- `store-strong r1 i r2`
- `malloc r1 n`
- `commit r1`

which all follow the same runtime semantics as what we saw in ATAPL. The only difference is the `store-strong` instruction which supports changing the type of the value

3 Typing rules

3.1 Basic types

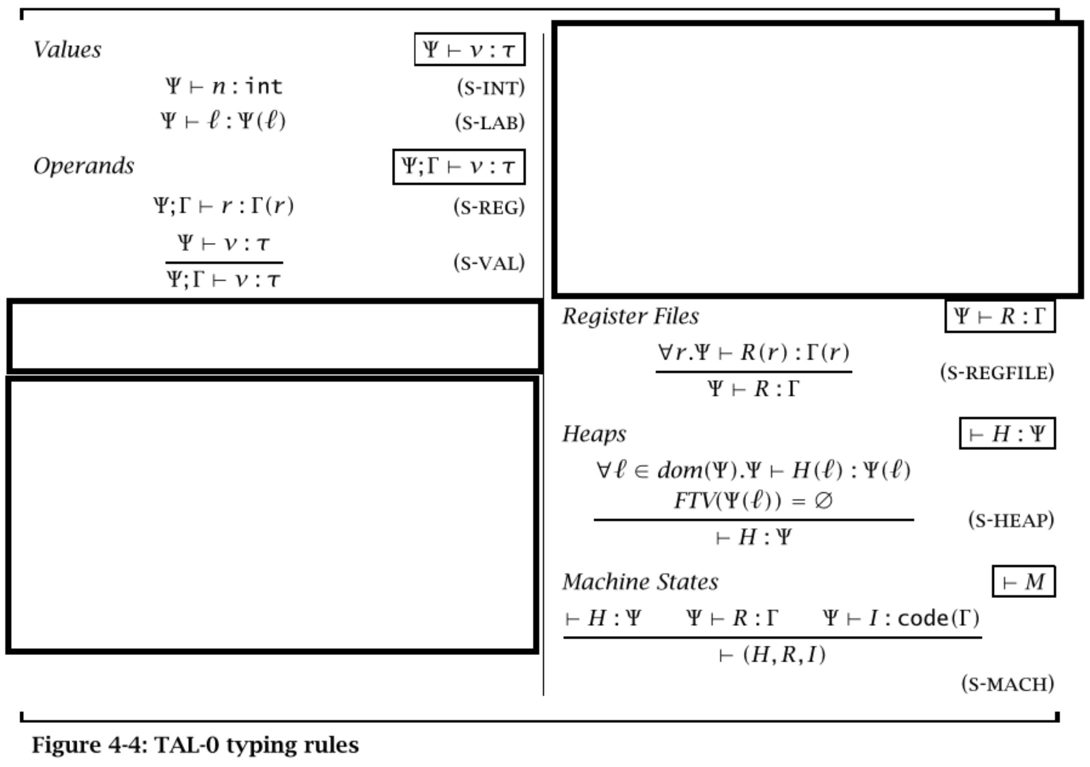
Our basic types are

- *int*
- *Top*

- $UniqPtr(\{l_i : \tau_i\}, \rho)$
- $Ptr(\{l_i : \tau_i\}, \rho)$
- $Code(\{r_i : \tau_i\})$

Let R be the set of registers and let $\Gamma[r_i]$ refer to τ_i for $\Gamma = \{r_i : \tau_i\}$. Let FRV and FTV be free rho and type variables respectively.

3.2 Heap, registers, values



Values $\boxed{\Psi \vdash v : \tau}$

$$\Psi \vdash n : \texttt{int} \quad \text{(S-INT)}$$

$$\Psi \vdash \ell : \Psi(\ell) \quad \text{(S-LAB)}$$

Operands $\boxed{\Psi ; \Gamma \vdash v : \tau}$

$$\Psi ; \Gamma \vdash r : \Gamma(r) \quad \text{(S-REG)}$$

$$\frac{\Psi \vdash v : \tau}{\Psi ; \Gamma \vdash v : \tau} \quad \text{(S-VAL)}$$

Register Files $\boxed{\Psi \vdash R : \Gamma}$

$$\frac{\forall r . \Psi \vdash R(r) : \Gamma(r)}{\Psi \vdash R : \Gamma} \quad \text{(S-REGFILE)}$$

Heaps $\boxed{\vdash H : \Psi}$

$$\frac{\begin{array}{c}\forall \ell \in dom(\Psi) . \Psi \vdash H(\ell) : \Psi(\ell) \\ FTV(\Psi(\ell)) = \varnothing\end{array}}{\vdash H : \Psi} \quad \text{(S-HEAP)}$$

Machine States $\boxed{\vdash M}$

$$\frac{\vdash H : \Psi \qquad \Psi \vdash R : \Gamma \qquad \Psi \vdash I : \texttt{code}(\Gamma)}{\vdash (H, R, I)} \quad \text{(S-MACH)}$$

Figure 4-4: TAL-0 typing rules

Figure 1: Relevant ATAPL TAL-0 typing rules (Source: Figure 4-4, page 148)

As you might expect, the typing rules are going to be very similar to the original TAL systems, but now with constraints being generated. First we follow identical rules for well typing of a value, operand, heap and register as seen in ATAPL.

3.3 Instruction sequences

We have slightly different rules for instruction sequences. Firstly, an instruction sequence now generates a set of constraints

$$\frac{\Psi ; \Gamma \vdash i : (\Gamma, C) \to (\Gamma_2, C') \quad \Psi ; \Gamma_2 \vdash I : (\Gamma_2, C') \to (\Gamma_3, C'')}{\Psi ; \Gamma \vdash i ; I : (\Gamma, C) \to (\Gamma_3, C'')} \text{(SeqC)}$$

The interpretation of this rule is that an instruction sequence $i; I$ maps a pair of initial register state and preconditions, to a pair of register state after execution and additional constraints that must hold for valid computation. What this rule is doing is threading the state and constraint accumulators accordingly.

An instruction sequence is then well typed if we can type and satisfy all constraints with variables as the initial state

$$\frac{\Gamma_1 = \{r : X_r | r \in R\} \quad X_r \in FTV(\Psi) \quad \Psi; \Gamma_1 \vdash I : (\Gamma_1, \emptyset) \to (\Gamma, C)}{\Psi \vdash I : Code(L(S(\Gamma_1)))}\text{(SeqTy)}$$

where S is the mapping generated from solving C and L is a substitution from $X_r \to Top$.

3.4 Jumps

The last piece of the puzzle is how to typecheck a `jump` terminal. As mentioned during the presentation, jumps include subtyping as well as a bunch of *very* questionable checks to avoid creating recursive types while still constraining labels properly. I don't know how to write it out formally so here's an approximation

When typechecking a `bnz v/jump v`

- $\Psi, \Gamma \vdash v \Rightarrow T$
- Let $T' = Code(\{r : T_r | r \in R\})$ where T_r is a fresh type variable
- Add the constraint $T = T'$ to C'
- If T is a variable corresponding to a parameter for this label, then for each register $r \in R$, add constraint $\Gamma[r] = T_r$ to C' if $\Gamma[r]$ is an *int*, *Ptr*, or *UniqPtr*.
- Add the constraint $\Gamma <: \{r : T_r | r \in R\}$ to C'
- The `bnz/jump` then has type $(\Gamma, C) \to (\Gamma, C')$

3.5 Subtyping

The plan is to use subtyping instead of polymorphism to allow "forgetting" that certain registers were set, similar to what was original TAL. So we add a subtyping rule for contexts and pointers, however note that there is no generic subsumption rule. Subtyping only appears in constraints generated by jumps as seen in the above rule. We only require weakening since pointer indicies are inherently ordered and this is done by quantifying over the labels on the right hand side.

$$\frac{\forall r \in R \vdash \Gamma_1[r] <: \Gamma_2[r]}{\vdash \Gamma_1 <: \Gamma_2}\text{(Gamma subtype)}$$

$$\frac{}{\vdash \tau <: Top}\text{(Top)}$$

$$\frac{\forall k_i \vdash \tau_i <: \mu_i}{\vdash UniqPtr(\{l_i : \tau_i\}) <: UniqPtr(\{k_i : \mu_i\})}\text{(Unique pointer subtype)}$$

$$\frac{\forall k_i \vdash \tau_i <: \mu_i}{\vdash Ptr(\{l_i : \tau_i\}) <: Ptr(\{k_i : \mu_i\})}\text{(Pointer subtype)}$$

3.6 Inference

We have a simple bidirectional inference rule, which just looks the register up in the context. This always works because the context will be filled with type variables at the start of each instruction sequence (`SeqTy` rule).

$$\frac{\Gamma[r] = T}{\Psi, \Gamma \vdash r \Rightarrow T}\text{(Infer register)}$$

$$\frac{\Psi[l] = T}{\Psi, \Gamma \vdash l \Rightarrow T}\text{(Infer label)}$$

$$\frac{}{\Psi, \Gamma \vdash n \Rightarrow \mathit{int}}\text{(Infer int)}$$

3.7 Constraint generation

Next we have the rules for checking instructions, starting with `mov` and arithmetic. Recall from the rule for instruction sequences that an instruction I under heap Ψ has type $(\Gamma_1, C) \to (\Gamma_2, C')$

$$\frac{\Psi, \Gamma \vdash r2 \Rightarrow T}{\Psi \vdash \text{mov r1 r2} : (\Gamma, C) \to (\Gamma[r1/T], C \cup \{T \neq \mathit{UniqPtr}\})}\text{(Mov)}$$

Let A be one of the arithmetic operations, $A \in \{\text{add}, \text{sub}\}$.

$$\frac{\Psi, \Gamma \vdash r2 \Rightarrow T_2 \quad \Psi, \Gamma \vdash v \Rightarrow T_3}{\Psi \vdash \text{A r1 r2 v} : (\Gamma, C) \to (\Gamma[r1/\mathit{int}], C \cup \{T_2 = \mathit{int}, T_3 = \mathit{int}\})}\text{(Arithmetic)}$$

3.7.1 Memory

The memory based instructions are a bit more complex since they involve rho variables.

$$\frac{}{\Psi \vdash \text{malloc r1 n} : (\Gamma, C) \to (\Gamma[r1/T], C)}\text{(Malloc)}$$

where $T = \mathit{UniqPtr}(\{i : \mathit{int} | i \in [n]\}, \emptyset)$

Most rules involve generating a fresh ρ and including it in the constraints, for example in load:

$$\frac{\Psi, \Gamma \vdash r2 \Rightarrow T \quad \rho \in FRV \quad T_2 \in FTV}{\Psi \vdash \text{load r1 r2 n} : (\Gamma, C) \to (\Gamma[r1/T_2], C \cup \{T = \mathit{Ptr}(\{n : T_2\}, \rho)\})}\text{(Load)}$$

You might notice that this rule appears to reject loading from a unique pointer. So how does the typechecker implement this rule? The answer is that there's a very sketchy rule in the unification algorithm that `UniqPtr <: Ptr`... which is really informal and should be fixed. Either with a more formal subtyping or with two different load instructions like store.

Speaking of store

$$\frac{\Psi, \Gamma \vdash r1 \Rightarrow T_1 \quad \Psi, \Gamma \vdash r2 \Rightarrow T_2 \quad \rho \in FTV}{\Psi \vdash \text{store r1 n r2} : (\Gamma, C) \to (\Gamma, C \cup \{T_1 = \mathit{Ptr}(\{n : T_2\}, \rho)\})}\text{(Store weak)}$$

Again, this interally uses the sketchy subyping system to allow weak stores to unique pointers... Luckily store strong doesn't need it. We do however have to make sure we aren't copying a unique pointer. Note that T_3 is completely free because we don't care what was stored in the pointer at that index before the `store-strong`

$$\frac{\Psi, \Gamma \vdash r1 \Rightarrow T_1 \quad \Psi, \Gamma \vdash r2 \Rightarrow T_2 \quad \rho \in FTV \quad T_3 \in FTV}{\Psi \vdash \text{store-strong r1 n r2} : (\Gamma, C) \to (\Gamma[r1/Ptr(\{n : T_2\}, \rho)], C')}\text{(Store strong)}$$

$$\text{where } C' = C \cup \{T_1 = Ptr(\{n : T_3\}, \rho), T_2 \neq UniqPtr\}$$

Finally, we have `commit`

$$\frac{\Psi, \Gamma \vdash r \Rightarrow T_1 \quad \rho \in FTV}{\Psi \vdash \text{commit r} : (\Gamma, C) \to (\Gamma[r/Ptr(\emptyset, \rho)], C \cup \{T_1 = UniqPtr(\emptyset, \rho)\})}\text{(Commit)}$$

4 Implementation details

4.1 Constraint generation

The initial setup of the context for each labelled block is handled by `check_block` (`checker.rs:303`), which initializes the context with type variables in each register.

Each rule is straightforwardly implemented in `check_instruction` (`checker.rs:214`) using the `constrain_register`, `constrain_value`, and `update_register` functions. Each function modifies the current context and constraint list as expected.

Jumps are handled by `constrain_jump` (`checker.rs:152`), which implements the sketchy rules from above.

4.2 Unification

Unification is handled by `Unifier` while `Satisfier` handles the subtyping rules (both in `unify.rs`)

The basic unification algorithm is fairly straightforward (`unify.rs:220`), however things get more complicated for unifying pointers.

A brief rundown of the implementation of pointer unification

1. The entrypoint is `unify_ptrs` (`unify.rs:64`). This function checks the two pointers to see if the pointers have rho variables and calls into either `unify_rho_id_with_mapping` if there is one or `unify_rho_mappings` if there are none.
2. `add` (`unify.rs:121`) and `unify_subtract` (`unify.rs:173`) implement addition and subtraction on records. Subtraction may generate additional constraints which is why it has the `unify_` prefix.
3. `unify_rho_id_with_mapping` (`unify.rs:197`) takes in a rho variable and a record type that it has to satisfy. If the variable has already been bound then we check if the current binding is satisfactory in `unify_rho_mappings`. Otherwise we just bind the rho variable
4. `unify_rho_mappings` (`unify.rs:143`) takes two record types and checks if they're equal by calling `unify_rho_entries` (`unify.rs:158`) on each element of the record. `unify_rho_entries` (`unify.rs:158`) mostly just calls into `constrain`, though it exists because `RhoEntry` contains the `Absent` variant. I initially added it because I thought it would be needed but it turns out I didn't, so this extra complexity is just a leftover from that.

The post unification mapping is closed by `chase_to_root` (`unify.rs:263`) and `chase_all_to_root` (`unify.rs:297`). This is where type variables are lifted to `Top` if they weren't bound (the mapping L in `SeqTy`). `chase_all_to_root` creates a direct mapping from variable id to a `TyU` (short for unified type, which does not contain the `UnifVar` variant and instead contains `Any`). This mapping is then used to initialize a `Satisfier`

4.3 Subtyping

The two subtyping rules are handled by `Satisfier` in `satisfy_jump` (`unify.rs:458`) and `satisfy_rho` (`unify.rs:372`). Both are called through `satisfy` (`unify.rs:504`) which simply loops through the subtyping constraints remembered from each jump and calls `satisfy_jump`.

The not equal constraints are also checked by `Satisfier` in `check_neq` (`unify.rs: 530`)

5 Running

- To change the test case that gets run, edit `test.rs` or `main.rs`
- To change the number of registers, edit `MAX_REGISTER` in `checker.rs`