CLE Type System

Benjamin Flin

August 2021

1 Introduction

2 CLE Types Grammar

The following is a small grammar for CLE types τ. l and r represent enclaves and can be thought of as arbitrary identifiers.

$$\tau, \pi, \gamma ::= \mathrm{l}\ \sigma \mid \mathrm{l}\ \sigma\ (\alpha_1, \ldots, \alpha_n) \rightarrow_\phi \theta \quad \text{cle type}$$

$\alpha, \beta, \theta, \sigma$ are sets of remote enclaves.

3 LLVM core grammar

The following source grammar represents an idealized subset of LLVM and teletyped versions of the types described above. Many operations are missing, but every missing operation can either be constructed from multiple instructions or is irrelevant to the CLE type inference and checking. The rules for cle types in the core grammar should represent the small grammar cle types above. My references for the grammar construction is from this[1] mail resource.

⟨*prog*⟩ ::= empty
  | ⟨*prog*⟩ ⟨*top-level-entity*⟩

⟨*top-level-entity*⟩ ::= ⟨*fun-def*⟩
  | ⟨*fun-decl*⟩
  | ⟨*global-def*⟩
  | ⟨*global-decl*⟩

⟨*global-def*⟩ ::= ⟨*global-ident*⟩ : ⟨*type*⟩ '`=`' ⟨*const*⟩'`;`'

⟨*global-decl*⟩ ::= ⟨*global-ident*⟩ : ⟨*type*⟩ '`;`'

⟨*fun-def*⟩ ::= '`define`' ⟨*global-ident*⟩'`(`' ⟨*params*⟩ '`)`' : ⟨*type*⟩ ⟨*func-body*⟩

[1] `https://lists.llvm.org/pipermail/llvm-dev/2018-June/123851.html`

⟨*fun-decl*⟩ ::= '`declare`' ⟨*global-ident*⟩'`(`' ⟨*params*⟩ '`)`' : ⟨*type*⟩ '`;`'

⟨*params*⟩ ::= empty
| ⟨*param-list*⟩

⟨*param-list*⟩ ::= ⟨*local-ident*⟩
| ⟨*param-list*⟩'`,`' ⟨*local-ident*⟩

⟨*global-ident*⟩ ::= '`@`'⟨*ident*⟩

⟨*local-ident*⟩ ::= '`%`'⟨*ident*⟩

⟨*func-body*⟩ ::= '`{`' ⟨*block-list*⟩ '`}`'

⟨*block-list*⟩ ::= ⟨*block*⟩
| ⟨*block-list*⟩ ⟨*block*⟩

⟨*block*⟩ ::= ⟨*ident*⟩'`:`' ⟨*instructions*⟩ ⟨*terminator*⟩

⟨*instructions*⟩ ::= empty
| ⟨*instructions*⟩ ⟨*instruction*⟩'`;`'

⟨*instruction*⟩ ::= ⟨*store-instr*⟩
| ⟨*load-instr*⟩
| ⟨*alloca-instr*⟩
| ⟨*gep-instr*⟩
| ⟨*call-instr*⟩
| ⟨*binary-instr*⟩
| ⟨*cast-instr*⟩
| ⟨*const-instr*⟩

⟨*terminator*⟩ ::= ⟨*br-term*⟩
| ⟨*ret-term*⟩

⟨*decl*⟩ ::= ⟨*local-ident*⟩ : ⟨*type*⟩

⟨*store-instr*⟩ ::= '`store`' ⟨*value*⟩, ⟨*local-ident*⟩

⟨*load-instr*⟩ ::= ⟨*decl*⟩ '`=`' '`load`' ⟨*value*⟩, ⟨*local-ident*⟩

⟨*alloca-instr*⟩ ::= ⟨*decl*⟩ '`=`' '`alloca`' ⟨*llvm-type*⟩

⟨*call-instr*⟩ ::= ⟨*decl*⟩ '`=`' ⟨*global-ident*⟩'`(`' ⟨*params*⟩ '`)`'

⟨*gep-instr*⟩ ::= ⟨*decl*⟩ '`=`' '`gep`' ⟨*local-ident*⟩'`,`' ⟨*nats*⟩

⟨*binary-instr*⟩ ::= ⟨*decl*⟩ '`=`' ⟨*value*⟩ ⟨*binop*⟩ ⟨*value*⟩

⟨*cast-instr*⟩ ::= ⟨*decl*⟩ '`=`' '`cast`' ⟨*local-ident*⟩ ⟨*llvm-type*⟩

⟨*nats*⟩ ::= ⟨*nat*⟩
| ⟨*nat*⟩, ⟨*nats*⟩

⟨*br-term*⟩ ::= '`br`' ⟨*local-ident*⟩, ⟨*local-ident*⟩, ⟨*local-ident*⟩ ⟨*ret-term*⟩ ::= '`ret`'
⟨*local-ident*⟩

⟨*value*⟩ ::= ⟨*const*⟩
| ⟨*local-ident*⟩
| ⟨*global-ident*⟩

⟨*const*⟩ ::= ⟨*integer-literal*⟩
| ⟨*bool-literal*⟩
| ⟨*float-literal*⟩
| ⟨*unit-literal*⟩
| ⟨*struct-const*⟩
| ⟨*array-const*⟩

⟨*type*⟩ ::= ⟨*llvm-type*⟩ '`+`' ⟨*cle-type*⟩
| ⟨*llvm-type*⟩

⟨*llvm-type*⟩ ::= ⟨*int-type*⟩
| ⟨*float-type*⟩
| ⟨*unit-type*⟩
| ⟨*array-type*⟩
| ⟨*pointer-type*⟩
| ⟨*struct-type*⟩
| ⟨*function-type*⟩

⟨*function-type*⟩ ::= '`(`' ⟨*llvm-type-list*⟩ '`)`' '`->`' ⟨*llvm-type*⟩

⟨*array-type*⟩ ::= '`[`' ⟨*nat*⟩ '`x`' ⟨*llvm-type*⟩ '`]`'

⟨*pointer-type*⟩ ::= ⟨*llvm-type*⟩'`*`'

⟨*struct-type*⟩ ::= '`{`' ⟨*llvm-type-list*⟩ '`}`'

⟨*llvm-type-list*⟩ ::= ⟨*llvm-type*⟩
| ⟨*llvm-type-list*⟩'`,`' ⟨*llvm-type*⟩

⟨*int-type*⟩ ::= '`i`'⟨*nat*⟩

⟨*float-type*⟩ ::= '`float`'
| '`double`'

⟨*unit-type*⟩ ::= '`unit`'

⟨*cle-type*⟩ ::= ⟨*enclave*⟩ ⟨*remote-enclaves*⟩
  | ⟨*enclave*⟩ ⟨*remote-enclaves*⟩ '`(`' ⟨*cle-args*⟩ '`)`' '`[`' ⟨*remote-enclaves*⟩ '`]`' '`->`' ⟨*remote-enclaves*⟩

⟨*cle-args*⟩ ::= ⟨*cle-args*⟩, ⟨*remote-enclaves*⟩
  | empty

⟨*remote-enclaves*⟩ ::= ⟨*remote-enclaves*⟩ '`+`' ⟨*enclave*⟩
  | empty

4 Small examples

This global viarable is in enclave purple, and is not shareable.

```
@foo : i64 + "purple" = 1;
```

This global variable is in enclave orange, and is shareable with enclave purple.

```
@foo : i64 + "orange" "purple" = 1;
```

The following function has a proposed type `"orange" "purple" (empty, "purple" | empty) [empty] -> empty`. It is in enclave "orange", and is callable from remote enclave "purple". The first argument is not shareable, and the second argument is shareable with purple. Any variable bound in the body must not be shareable, as well as the return type.

```
define @average(%0, %1) : (double, double) -> double
 + "orange" "purple" (empty, "purple" + empty) [empty] -> empty

{
   %2 : double + "orange" = %0 + %1;
   %3 : double + "orange" = %2 / 2.0;
   ret %3
}
```

5 Type rules

Here we assume all functions and global variables have cle types associated with them. We will focus on how to infer such types in the next section. There are several types of judgements, each of which is enumerated below:

5.1 Judgements

1. $\Gamma \vdash e : \tau$. Top-level entity or instruction e has type τ.
2. $\Gamma \vdash b :_\pi \gamma$. Basic block or set of basic blocks b has type π for all variables bound in instructions, type γ for the terminator.

3. $\Gamma \vdash t :_\pi \gamma$. Terminator t has type γ and all referenced basic blocks, b are given type ${}_\pi\gamma$.

5.2 Rules for top-level entities

$$\frac{\tau = l\ \sigma}{\Gamma \vdash @x : \tau}\ \text{global-decl} \qquad \frac{\tau = l\ \sigma}{\Gamma \vdash @x = c : \tau}\ \text{global-def}$$

$$\frac{\begin{array}{c}\Gamma[@f \mapsto \tau, \%1 \mapsto l\ \alpha_1, \ldots, \%n \mapsto l\ \alpha_n] \vdash body :_{(l\ \phi)} l\ \theta \\ \tau = l\ \sigma\ (\alpha_1, \ldots, \alpha_n) \to_\phi \theta\end{array}}{\Gamma \vdash @f(\%1, \ldots, \%n)\ \{body\} : \tau}\ \text{fn-def}$$

$$\frac{\Gamma(@f) = \tau = l\ \sigma\ (\alpha_1, \ldots, \alpha_n) \to_\phi \theta}{\Gamma \vdash @f(\%1, \ldots, \%n)\ \{body\} : \tau}\ \text{fn-def-known}$$

5.3 Rules for basic blocks and instruction lists

$$\frac{\Gamma \vdash b :_\pi \gamma \qquad \Gamma', \%b :_\pi \gamma \vdash bbs :_\pi \gamma}{\Gamma \vdash b\ bbs :_\pi \gamma}\ \text{fn-body}$$

$$\frac{\Gamma \vdash instrs : \pi \qquad \Gamma' \vdash term :_\pi \gamma}{\Gamma \vdash \%b : instrs\ term :_\pi \gamma}\ \text{bb-unknown}$$

$$\frac{\Gamma(\%b) =_\pi \gamma}{\Gamma \vdash \%b : instrs\ term :_\pi \gamma}\ \text{bb-known}$$

$$\frac{\Gamma \vdash instr : \pi \qquad \Gamma[\%a \mapsto \pi] \vdash instrs : \pi}{\Gamma \vdash \%a = instr; instrs : \pi}\ \text{instrs}$$

5.4 Rules for special instructions and terminators

$$\frac{\Gamma(\%1) = l\ \alpha_1, \ldots, \Gamma(\%n) = l\ \alpha_n \qquad \Gamma \vdash @f : l\ \sigma\ (\alpha_1, \ldots, \alpha_n) \to_\phi \theta}{\Gamma \vdash \text{call}\ @f(\%1, \ldots, \%n) : l\ \theta}\ \text{call}$$

$$\frac{\Gamma(\%1) = l\ \alpha_1, \ldots, \Gamma(\%n) = l\ \alpha_n \qquad \Gamma \vdash @f : r\ \sigma\ (\beta_1, \ldots, \beta_n) \to_\phi \theta \qquad l \neq r \qquad l \in \sigma}{\Gamma \vdash \text{call}\ @f(\%1, \ldots, \%n) : \pi}\ \text{xd-call}$$

$$\frac{\Gamma(\%a) = \gamma \qquad \Gamma \vdash \%b_1 :_\pi \gamma \qquad \Gamma \vdash \%b_2 :_\pi \gamma}{\Gamma \vdash \text{br}\ \%a, \%b_1, \%b_2 :_\pi \gamma}\ \text{break}$$

$$\frac{\Gamma(\%a) = \gamma}{\Gamma \vdash \text{ret}\ \%a :_\pi \gamma}\ \text{ret}$$

5.5 General rule for instructions

The rules for all other instructions can be derived from general instruction form which takes in a number of arguments:

$$\text{instr } \%a_1, \ldots, \%a_n$$

Thus,

$$\frac{\Gamma(\%a_1) = \cdots = \Gamma(\%a_n) = \pi}{\Gamma \vdash \text{instr } \%a_1, \ldots \%a_n : \pi} \text{ ret}$$