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 levels and can be thought of as arbitrary identifiers.

$$
\begin{array}{lr}
\tau, \pi, \gamma ::= \mathrm{l}\ \sigma \mid \mathrm{l}\ \sigma\ (\alpha_1, \ldots, \alpha_n) \rightarrow_\phi \theta & \text{cle type} \\
\sigma, \phi, \theta, \alpha, \beta ::= \epsilon \mid \sigma + r & \text{remote levels}
\end{array}
$$

Note: remote levels are treated as sets.

3 LLVM core grammar

The following source grammar represents an idealized subset of LLVM. 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*⟩ ‘;’

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

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

⟨*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*⟩ ::= ⟨*level*⟩ ⟨*remote-levels*⟩
| ⟨*level*⟩ ⟨*remote-levels*⟩ ‘`(`’ ⟨*cle-args*⟩ ‘`)`’ ‘`[`’ ⟨*remote-levels*⟩ ‘`]`’ ‘`->`’ ⟨*remote-levels*⟩

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

⟨*remote-levels*⟩ ::= ⟨*remote-levels*⟩ ‘`+`’ ⟨*level*⟩
| empty

4 Small examples

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

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

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

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

The following function is in level "orange", and has a cdf with remote level "purple". The first argument is not shareable, the second argument is shareable with purple. Any variable bound in the body must not be shareable, as well as the return type.

Here I added ‘empty’ in-place of empty fields for clarity.

```
define @average(%0, %1) : (double, double) -> double
 + "orange" "purple" (empty, "purple") [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. CLE is a flow-sensitive model and the types of variables may change as the program is checked. To reason through this in the system, a type judgement will also include an output context, which may be used in the conclusions of judgements.

There are several types of judgements, each of which is enumerated below:

5.1 Judgements

1. $\Gamma \vdash e : \tau$. Top-level entity e has type τ. Top-level judgements are not flow-sensitive.

2. $\Gamma \vdash b :_\pi \gamma, \Gamma'$. Basic block or set of basic blocks b has type π for all variables bound in instructions, type γ for the terminator and produces a new context Γ'.

3. $\Gamma \vdash t :_\pi \gamma, \Gamma'$. Terminator t has type γ and all referenced basic blocks, b are given type ${}_\pi\gamma$.

5.2 Rules for top-level entities

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

$$\frac{\begin{array}{c}\Gamma, @f : \tau, \%1 : l\ \alpha_1, \ldots, \%n : l\ \alpha \vdash \mathit{body} :_{(l\ \phi_m)} l\ \theta_m, \Gamma' \\ \tau = l\ \sigma\ (\alpha_1, \ldots, \alpha_n) \rightarrow_\phi \theta\end{array}}{\Gamma \vdash @f(\%1, \ldots \%n)\ \{\mathit{body}\} : \tau}\ \text{fn-def}$$

$$\frac{\tau = l\ \sigma\ (\alpha_1, \ldots, \alpha_n) \rightarrow_\phi \theta}{\Gamma \vdash @f : \tau, \Gamma}\ \text{fn-decl}$$

5.3 Rules for basic blocks and instruction lists

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

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

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

$$\frac{\Gamma \vdash \mathit{instr} : \pi, \Gamma' \qquad \Gamma' \vdash \mathit{instrs} : \pi, \Gamma''}{\Gamma \vdash \mathit{instr}; \mathit{instrs} : \pi, \Gamma''}\ \text{instrs}$$

5.4 Rules for special instructions and terminators

$$\frac{\Gamma(\%1) = l\ \alpha_1, \ldots, \Gamma(\%n) = l\ \alpha_n \qquad \begin{array}{c}\Gamma(@f) = l\ \sigma\ (\beta_1, \ldots, \beta_n) \rightarrow_\phi \theta \\ \gamma = l\ \theta\end{array} \qquad \begin{array}{c}\alpha_1 \cap \beta_1 \neq \epsilon \\ \vdots \\ \alpha_n \cap \beta_n \neq \epsilon\end{array}}{\Gamma \vdash \text{call } @f(\%1, \ldots, \%n) : \pi, \Gamma, \%1 : l\ \alpha_1 \cap \beta_1, \ldots, \%n : l\ \alpha_n \cap \beta_n}\ \text{call}$$

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

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

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