Enum Translation with Type Constraints

Minwook Lee
ryan-lee@kaist.ac.kr

I. Notation and Assumptions

- We consider a type as **a set of values**. Therefore, int is equivalent to $\mathbb{Z}$.
  - Every set operation can be applied to types as well. For example, $\mathsf{int} \cup \epsilon = \mathbb{Z}$ and $\epsilon \subseteq \mathbb{Z}$.
  - Every unary or binary operation can be applied to types as well. For example:

$$\tau_1 \ binop \ \tau_2 = \{v_1 \ binop \ v_2 \mid v_1 \in \tau_1 \wedge v_2 \in \tau_2\}.$$

- Type constraint of the identifier x is of the form $[\![x]\!] = \langle \lfloor x \rfloor, \lceil x \rceil \rangle$.
  - $\lfloor x \rfloor$ is a set of possible values that can be assigned to the identifier.
  - $\lceil x \rceil$ is the required type for x based on the type annotation and its usage context.
  - Every set operation can be applied to type constraints as well. For example:
    * $[\![x]\!] \subseteq [\![y]\!]$ iff $\lfloor x \rfloor \subseteq \lfloor y \rfloor \wedge \lceil x \rceil \subseteq \lceil y \rceil$
    * $[\![x]\!] \cup [\![y]\!] = \langle \lfloor x \rfloor \cup \lfloor y \rfloor, \lceil x \rceil \cup \lceil y \rceil \rangle$
    * $[\![x]\!] \ binop \ [\![y]\!] = \langle \lfloor x \rfloor \ binop \ \lfloor y \rfloor, \lceil x \rceil \ binop \ \lceil y \rceil \rangle$

II. Basic MIR

A. Abstract Syntax

$$\begin{array}{rcl}
n & \in & \mathbb{Z} \\
x, y & \in & Id \\
f & := & \overline{B : b}; \overline{x : \tau} \\
b & := & \overline{s}; t \\
s & := & x = n \mid x = y \mid x = unop \ y \mid x = y_1 \ binop \ y_2 \\
t & := & \mathsf{goto} \ B \mid \mathsf{switch} \ x \ \overline{n : B} \mid \mathsf{return} \\
\epsilon & \in & Enum \\
\tau & := & \mathsf{int} \mid \epsilon
\end{array}$$

Fig. 1. Basic MIR Abstract Syntax

The abstract syntax of the basic MIR is shown in Fig 1. In MIR, a function f is represented with the type `Body`[1], which denotes a control flow graph (CFG) consisting of multiple basic blocks. Each basic block b is represented with the type `BasicBlockData`[2] and consists of multiple statements followed by a single terminator. In a basic block, statements do not have any control flow effect, and only terminators affect the control flow. A basic block is associated with a unique identifier B, represented with the type `BasicBlock`[3]. A statement s has the type `Statement`[4]. While various kinds of statements exist in MIR, we only need to consider assignments. The left-hand side of an assignment is a variable x, and the right-hand side is either an integer n, a variable, a unary operation, or a binary operation. A terminator t has the type `Terminator`[5] and is either a jump to another basic block, a switch, or a return. When returning, the return value is always the value stored in the variable named `_0`. If a function has n parameters, their values are stored in the variables `_1`, `_2`, ..., `_n`. Note that a function declares the type of each variable as well. A type τ is either `int` or a C `enum` type ϵ.

B. Generating Type Constraints

The rules for type constraints in basic MIR are shown in Fig 2.

[1] https://doc.rust-lang.org/beta/nightly-rustc/rustc_middle/mir/struct.Body.html
[2] https://doc.rust-lang.org/beta/nightly-rustc/rustc_middle/mir/struct.BasicBlockData.html
[3] https://doc.rust-lang.org/beta/nightly-rustc/rustc_middle/mir/struct.BasicBlock.html
[4] https://doc.rust-lang.org/beta/nightly-rustc/rustc_middle/mir/struct.Statement.html
[5] https://doc.rust-lang.org/beta/nightly-rustc/rustc_middle/mir/struct.Terminator.html

For the function f:

$$\overline{B : b}; \overline{x : \tau} \quad : \quad \lceil \overline{x} \rceil \subseteq \overline{\tau}$$

For the statement s:

$$\begin{array}{rcl} x = n & : & n \in \lfloor x \rfloor \\ x = y & : & [\![y]\!] \subseteq [\![x]\!] \\ x = unop\ y & : & unop\ \lfloor y \rfloor \subseteq \lfloor x \rfloor \\ x = y_1\ binop\ y_2 & : & \lfloor y_1 \rfloor\ binop\ \lfloor y_2 \rfloor \subseteq \lfloor x \rfloor \end{array}$$

Fig. 2. Basic MIR Type Constraints

C. *Resolving Type Constraints*

We resolve the type constraints with the following principles:

- We always choose the smallest set for $\lfloor x \rfloor$ that satisfies the type constraints.
  - This increases the coverage where the enum translation can be applied.
- We always choose the largest set for $\lceil x \rceil$ that satisfies the type constraints.
  - This prevents the redundant enum translation.

D. *Examples*

Assume $\epsilon = \{0, 1\}$.

1) Simple case: The following is an example of a basic MIR function:

$$x : \epsilon;\ y : \mathbb{Z};\ \{y = 1;\ x = y;\ \mathsf{return}\}$$

We can derive the following conditions:

$$\begin{array}{rcl} x : \epsilon & : & \lceil x \rceil \subseteq \epsilon \\ y : \mathbb{Z} & : & \lceil y \rceil \subseteq \mathbb{Z} \\ y = 1 & : & 1 \in \lfloor y \rfloor \\ x = y & : & [\![y]\!] \subseteq [\![x]\!] \end{array}$$

By resolving the type constraints we have:

$$[\![x]\!] = \langle \{1\}, \epsilon \rangle \quad [\![y]\!] = \langle \{1\}, \epsilon \rangle.$$

Now we deduce that $\lfloor x \rfloor \subseteq \lceil x \rceil$ and $\lfloor y \rfloor \subseteq \lceil y \rceil$. Therefore, we now safely modify the function to:

$$\cdots;\ y : \epsilon;\ \cdots.$$

Also, the translation doesn't introduce the unsafe code since all required types are valid Rust enum types.

2) Solving more cases with type constraints: The following is an example of a basic MIR function showing how deriving value constraints would work in case where subtype inference could not be applied:

$$x : \epsilon;\ y : \mathbb{Z};\ \{y = -1;\ x = -y;\ \mathsf{return}\}$$

We can derive the following conditions:

$$\begin{array}{rcl} x : \epsilon & : & \lceil x \rceil \subseteq \epsilon \\ y : \mathbb{Z} & : & \lceil y \rceil \subseteq \mathbb{Z} \\ y = -1 & : & -1 \in \lfloor y \rfloor \\ x = -y & : & -\lfloor y \rfloor \subseteq \lfloor x \rfloor \end{array}$$

By resolving the type constraints we have:

$$[\![x]\!] = \langle \{1\}, \epsilon \rangle \quad [\![y]\!] = \langle \{-1\}, \mathbb{Z} \rangle.$$

Now we deduce that all $\lfloor x \rfloor \subseteq \lceil x \rceil$ and $\lfloor y \rfloor \subseteq \lceil y \rceil$. Therefore, we can translate the integer type to the Rust enum type. However, this procedure would introduce unsafe code on $x = -y$ while translating the type of x to be ϵ.

$$
\begin{array}{rcl}
s & := & \cdots \mid x = \&y \mid x = *y \mid \; *x = y \\
\tau & := & \cdots \mid *\tau
\end{array}
$$

Fig. 3. MIR Abstract Syntax with Pointers

III. Adding pointers

A. Abstract Syntax

The abstract syntax of MIR with pointers is shown in Fig 3.

We denote the abstract cell $c \in \mathit{Cell}$ as the set of all locations while $[\![c]\!] = \langle \lfloor c \rfloor, \lceil c \rceil \rangle$ denotes the type constraint of the value which is located on c. The pointer type $*\tau$ denotes the set of cells that can be dereferenced to a value of type τ. If x can be dereferenced to a value of type τ_1 and $\tau_1 \subseteq \tau_2$, then x can also be dereferenced to a value of type τ_2. This means that $*\tau_1 \subseteq *\tau_2$.

B. Generating Type Constraints

We choose an Andersen-style pointer analysis, since we do not expect real-world C code to use extensive pointer operations that would require flow-sensitive analysis for enum types.

The rules for type constraints in MIR with pointers are shown in Fig 4.

For the statement s:

$$
\cdots
$$

$$
\begin{array}{lcllcl}
x = \&y & : & \forall\tau, & y \subseteq \lfloor x \rfloor & \wedge & \tau \subseteq \lceil y \rceil \Rightarrow *\tau \subseteq \lceil x \rceil \\
x = *y & : & \forall\tau\forall c, & c \in \lfloor y \rfloor \Rightarrow [\![c]\!] \subseteq [\![x]\!] & \wedge & *\tau \subseteq \lceil y \rceil \Rightarrow \tau \subseteq \lceil x \rceil \\
*x = y & : & \forall\tau\forall c, & c \in \lfloor x \rfloor \Rightarrow [\![y]\!] \subseteq [\![c]\!] & \wedge & \tau \subseteq \lceil y \rceil \Rightarrow *\tau \subseteq \lceil x \rceil
\end{array}
$$

Fig. 4. MIR Type Constraints with Pointers

C. Examples

Assume $\epsilon = \{0, 1\}$. The following is an example of a basic MIR function with pointers:

$$
x : *\mathbb{Z};\; y : \mathbb{Z};\; z : \epsilon;\; \{y = 1;\; x = \&y;\; z = *x;\; \mathsf{return}\}
$$

Each statement and variable declaration generates the following type constraints:

$$
\begin{array}{rcl}
x : *\mathbb{Z} & : & \lceil x \rceil \subseteq *\mathbb{Z} \\
y : \mathbb{Z} & : & \lceil y \rceil \subseteq \mathbb{Z} \\
z : \epsilon & : & \lceil z \rceil \subseteq \epsilon \\
y = 1 & : & 1 \in \lfloor y \rfloor \\
x = \&y & : & \forall\tau,\; y \subseteq \lfloor x \rfloor \;\wedge\; \tau \subseteq \lceil y \rceil \Rightarrow *\tau \subseteq \lceil x \rceil \\
z = *x & : & \forall\tau\forall c,\; c \in \lfloor x \rfloor \Rightarrow [\![c]\!] \subseteq [\![z]\!] \;\wedge\; *\tau \subseteq \lceil x \rceil \Rightarrow \tau \subseteq \lceil z \rceil
\end{array}
$$

Now we resolve the type constraints as follows:

$$
\begin{array}{l}
\lfloor y \rfloor = \{1\} \\
\lfloor x \rfloor = \{y\} \\
[\![y]\!] \subseteq [\![z]\!] \\
\tau \subseteq \lceil y \rceil \Rightarrow *\tau \subseteq \lceil x \rceil \Rightarrow \tau \subseteq \lceil z \rceil \subseteq \epsilon \\
\therefore \lceil x \rceil \subseteq *\epsilon \;\wedge\; \lceil y \rceil \subseteq \epsilon \;\wedge\; \lceil z \rceil \subseteq \epsilon
\end{array}
$$

By resolving the type constraints we have:

$$
[\![x]\!] = \langle \{y\},\, *\epsilon \rangle \quad [\![y]\!] = \langle \{1\},\, \epsilon \rangle \quad [\![z]\!] = \langle \{1\},\, \epsilon \rangle.
$$

IV. Adding structs and arrays

A. Abstract Syntax

The abstract syntax of MIR with structs and arrays is shown in Fig 5. The array type $\tau[]$ denotes the set of arrays whose elements are of type τ. i.e., if a variable x has the type $\tau[]$ and y has the type $\mathbb{Z}$, then $x[y] \subseteq \tau$.

$$
\begin{array}{rcl}
F & \in & \mathit{Field} \\
S & \in & \mathit{Struct} \\
s & := & \cdots \mid x.F = y \mid x = y.F \mid x[x'] = y \mid x = y[y'] \\
\tau & := & \cdots \mid S \mid \tau[] \\
p & := & \overline{S\{\overline{F:\tau}\}}; \overline{f}
\end{array}
$$

Fig. 5. MIR Abstract Syntax with Structs and Arrays

B. Generating Type Constraints

The rules for type constraints in MIR with structs and arrays are shown in Fig 6.

For the struct definition $S\{\overline{F:\tau}\}$:

$$S\{\overline{F:\tau}\} \quad : \quad \forall F \in S, \ \Rightarrow \lceil S.F \rceil \subseteq \tau$$

For the statement s:

$$
\begin{array}{rcl}
 & \cdots & \\
x : S & : & \forall F \in S, \ [\![x.F]\!] \subseteq [\![S.F]\!] \\
x.F = y & : & [\![y]\!] \subseteq [\![x.F]\!] \\
x = y.F & : & [\![y.F]\!] \subseteq [\![x]\!] \\
x[x'] = y & : & \forall \tau, \tau \subseteq \lfloor y \rfloor \Rightarrow \tau[] \subseteq \lfloor x \rfloor \quad \wedge \quad \tau \subseteq \lceil y \rceil \Rightarrow \tau[] \subseteq \lceil x \rceil \\
x = y[y'] & : & \forall \tau, \tau[] \subseteq \lfloor y \rfloor \Rightarrow \tau \subseteq \lfloor x \rfloor \quad \wedge \quad \tau[] \subseteq \lceil y \rceil \Rightarrow \tau \subseteq \lceil x \rceil
\end{array}
$$

Fig. 6. MIR Type Constraints with Structs and Arrays

V. Adding function calls

A. Abstract Syntax

The abstract syntax of MIR with function calls is shown in Fig 7.

$$
\begin{array}{rcl}
g & \in & \mathit{Function} \\
p & := & \overline{S\{\overline{F:\tau}\}}; \overline{g:f} \\
t & := & \cdots \mid x = g(\overline{y}); B
\end{array}
$$

Fig. 7. MIR Abstract Syntax with Function Calls

We denote the local variable x in the function g as $g.x$. For the function call $x = g(\overline{y})$, we assume that y_r is the local variable in g function which is assigned as the return value. Also, we assume that $\overline{x_p}$ is the list of local variables in the current context, which is assigned to each parameter $\overline{y}$. The procedure of function application can be described as follows:

- We assign each value in $\overline{x_p}$ to the corresponding parameter in g. i.e., $\overline{g.y = x_p}$.
- Evaluate the function g with the current context.
- Then assign the return value $g.y_r$ to x.

B. Generating Type Constraints

The rules for type constraints in MIR with function calls are shown in Fig 8.

$$\cdots$$

$$x = g(\overline{y}); B \quad : \quad \overline{[\![x_p]\!] \subseteq [\![g.y]\!]} \quad \wedge \quad [\![g.y_r]\!] \subseteq [\![x]\!]$$

Fig. 8. MIR Type Constraints with Function Calls

VI. Adding function pointers

A. Abstract Syntax

The abstract syntax of MIR with function pointers is shown in Fig 9.

$$\begin{array}{rcl} s & := & \cdots \mid x = g \\ t & := & \cdots \mid x = y(\overline{y'}); B \\ \tau & := & \cdots \mid \overline{\tau} \to \tau' \end{array}$$

Fig. 9. MIR Abstract Syntax with Function Pointers

B. Generating Type Constraints

The rules for type constraints in MIR with function pointers are shown in Fig 10.

$$\cdots$$

For the function type $\overline{\tau} \to \tau'$:

$$\overline{\tau_1} \to \tau_1' \subseteq \overline{\tau_2} \to \tau_2' \quad : \quad \overline{\tau_2 \subseteq \tau_1} \quad \wedge \quad \tau_1' \subseteq \tau_2'$$

For the function pointer g:

$$\lceil g \rceil = \lfloor g \rfloor = \{\overline{g.y} \to g.y_r\}$$

For the statement s:

$$x = g \quad : \quad [\![g]\!] \subseteq [\![x]\!]$$

For the terminator t:

$$x = y(\overline{y'}); B \quad : \quad \forall g \in \lfloor y \rfloor, \; \overline{[\![y']\!] \subseteq [\![g.y]\!]} \; \wedge \; [\![g.y_r]\!] \subseteq [\![x]\!]$$

Fig. 10. MIR Type Constraints with Function Pointers