1 The Howl Type Inference Algorithm

Howl's type inference system works on a *section* level. While in the current implementation each section is one statement, it's possible that inferring types at the block level will be supported in the future, so the option is being left open.

To infer types for a section, the following algorithm is used:

1. Initialize the section's *type environment* Γ.

   Each expression or sub-expression receives a *handle*, denoted H_n, which identifies it to the Howl compiler, and Γ is a mapping from handles H_n to types T_n.

   The initial types of each element of Γ are determined by the *setup rules*. Additionally, certain types of statement generate an additional handle H_S, also initialized to $*$.

2. Repeatedly apply the typing rules.

   Each *typing rule* specifies a set of source expressions or statements for which it is valid, a set of preconditions on the associated T_n types that must be met for it to be applied, and the associated transformation it performs on the elements of Γ.

   Once no further typing rules can be applied to Γ, type inference terminates.

3. Reduce the type objects generated by inference.

   Some type objects are not always in their simplest form. After inference completes, these can be *reduced* using a set of reduction rules on each object to produce uniquely determined types.

4. Associate the inferred types with their corresponding source expressions.

   Any expression that does not have an uniquely determined type associated with it is invalid.

2 Notation and Conventions

Each setup rule is notated as a mapping $E \to T$ where E represents a source statement or expression (possibly including handles for sub-expressions) and T represents the type assignments to be generated from that statement or expression.

Each typing rule is notated as a mapping $\{C_0 \dots C_n\} \to T$ where each C_n represents a precondition and T represents the rule's transformation. Both the expression class and precondition sets may also be $\forall$, to represent that the rule applies to all classes of expressions or may always be applied.

Type constants from the original Howl program are typeset in `monospace`.

3 Data Structures

3.1 Type Objects

The result of the type inference algorithm is a mapping from expression handles to type objects representing their type. Type objects admit the following operations:

- equality $T_a = T_b$
- partial ordering or *acceptance* $T_a \supseteq T_b$, representing that an object of type T_b can be assigned to a location of type T_a

Unless otherwise stated, type objects are assumed to be not equal and not accepting.

3.1.1 Error and Any

The *error type* $\emptyset$ represents the type of an invalid expression. In general, any transformation on type inputs including $\emptyset$ will also result in $\emptyset$. The semantically similar *any type* $*$ represents the type of an expression that has not yet had types inferred for it, or the type of an unconstrained parameter input.

Equality: $\emptyset = \emptyset, \quad * = *$

Ordering: $\forall T : \; T \neq \emptyset, \, \emptyset \not\supseteq T, \quad \forall T : * \supseteq T$

3.1.2 Type Constants

A type constant represents a single type from the original program, such as `i32` or `root.lib.String`.

Equality: Two type constants are equal if their string representations are equal.

Ordering: A type constant T_a accepts another type constant T_b if and only if:

- T_a and T_b both represent numeric types, or
- T_a represents a superclass of T_b, or
- T_a represents an interface implemented by T_b.

3.1.3 Type Aliases

A type alias represents the type of another handle, notated $[H_n]$.

3.1.4 Intersection Types

An intersection type represents the most general type accepted by two subtypes, notated $(T_a \cap T_b)$.

3.1.5 Union Types

An union type represents a type that may take on one of two different values depending on some to-be-determined condition, notated $(T_a \cup T_b)$.

4 Setup Rules

4.1 Assignment and Initialization

$$\begin{aligned} H_{lhs} \;\texttt{=}\; H_{rhs}\,\texttt{;} \; &\rightarrow H_{lhs} \mapsto (\Gamma_{H_{lhs}} \cap \Gamma_{H_{rhs}}), H_{rhs} \mapsto [H_{lhs}] \\ \texttt{let}\; T_{def} \;\texttt{=}\; H_{init}\,\texttt{;} \; &\rightarrow H_{init} \mapsto (\Gamma_{H_{init}} \cap T_{def}), H_S \mapsto [H_{init}] \end{aligned}$$

4.2 Constants

$$\begin{aligned} H_n : < \text{numeric literal} > \; &\rightarrow H_n \mapsto \texttt{numeric} \\ H_n : < \text{string literal} > \; &\rightarrow H_n \mapsto \star(\texttt{u8}) \end{aligned}$$

5 Typing Rules

5.1 Alias Dereferencing

$$[H_a] \;:\; \Gamma_{H_a} \text{ is uniquely determined} \rightarrow \Gamma_{H_a}$$

5.2 Intersection Failure

$$T_a \cap T_b \;:\; (T_a \not\supseteq T_b) \vee (T_b \not\supseteq T_a) \rightarrow \emptyset$$

5.3 Union-Intersection Resolution

$$T_a \cap (T_a \cup T_b) \rightarrow T_a$$

6 Examples