Hole instance renumbering for unique hole closures

Jonathan Lam

2022/01/13

1 Motivation

The original motivation is the performance problem stated in the GitHub issue #536 on the Hazel repository. The problem is that there is a blowup of numbered (tracked) hole instances in the final `HoleInstanceInfo.t` object. These hole instances are counted in `Program.renumber` right after evaluating the expression. The renumbering operation gives a unique instance number to each hole that is encountered (`Program.renumber_result_only`), and then recurses through the environments of the holes (`Program.renumber_sigmas_only`), renumbering the holes in the substituted expressions.

The problem is that it performs a full DFS through every hole instance, and considers every hole instance distinct from other hole instances. The problem is that it causes an absurdly large number of instances for each hole, even in programs where the evaluation takes many fewer steps. This may confuse the Hazel user: two hole instances that share the same environment, and thus will necessarily have the same value when the hole expression is filled, will have different hole instance numbers.n

The solution is to unify all instances of a hole which share the same (physically equal) environment, and thus keep track only of **unique hole closures** (instances of a hole with the same environment). This will require changes to hole `InstancePath.ts`. This will be greatly helpful for the fill-and-resume operation described in the POPL 2019 paper.

1.1 Other performance optimizations

When investigating the above issue, I realized a few other inefficiencies that worsen the performance issue. `Program.get_result` is called extraneously many times (four times per `ModelAction.t`). The program result can be stored in the `Model.t`, and the program does not have to be re-evaluated after every action: it should only be re-evaluated after appropriate edit actions and can be memoized with respect to the `UHExp.t` program expression.

The exact memoization and structural sharing characteristics of environments (e.g., the implementation of `Environment.extend`) may slightly affect

performance, but are not nearly as important as the exponential slowdown experienced with the hole renumbering.

2 Illustration

Consider the following Hazel program:

```
let a = (* hole 1 *) in
let b = \x.{(* hole 2 *)} in
b 4 + b 5 + (* hole 3 *)
(* yields:
 * (hole 2:1 with env e1) + (hole 2:2 with env e2) + (hole 3:1 with env e3)
 * with e1 = { a = (hole 1:1 with env nil), x = 4 },
 *      e2 = { a = (hole 1:2 with env nil), x = 5 },
 *      e3 = { a = (hole 1:3 with env nil), b = \x.(hole 2) }
 *)
```

The comment informally represents the formal syntax of empty holes (the scenario also applies to nonempty holes without modification). The statement `hole u:i with env s` indicates a hole with hole number (`MetaVar.t`) u, hole instance (`MetaVarInstance.t`) i, and hole environment (`Environment.t`, or σ) s. For each hole instance, a new entry is added to the `HoleInstanceInfo.t`.

The inefficiency is that the hole 1 creates three instances, but the instances will have the same value no matter what expression is assigned to a. This is because they share the same environment[1] (the empty environment). Note that hole 2 necessarily needs two instances, because it exists in a closure which is invoked with separate environments. (Thus, filling in hole 2 may produce different values at instances 2:1 and 2:2).

The inefficiency is amplified in the following simple example, which sparked the performance issue. See the below example.

```
let a = (* hole 1 *) in
let b = (* hole 2 *) in
let c = (* hole 3 *) in
(* hole 4 *)
```

[1] We need to be careful what we mean by "same environment." Here I mean physically equal environments, i.e., the same instance of an environment. However, we can also have holes with structurally but not physically equal environments. For example, the two instances of hole 2 in the above example would have structurally equal environments if the arguments to b in the function applications were the same. In this case, filling in the hole would result in the same value in all of those holes, but if one of those environments is modified, then the filled holes may evaluate to different expressions. Thus it may be more intuitive to only group holes with the physically equal environments.

3 Proposed solution

3.1 Evaluation with environments

The solution is very simple if evaluation is implemented with environments rather than substitution. This means that variables will be given meaning through lookup in the current eval-time environment, rather than by substitution during binding. This has the ordinary performance benefit over substitution, i.e., variable lookups will be lazily performed only on evaluated branches whereas substitution will act on all branches of the expression. The fixpoint form is also not necessary for recursion, due to the lazy nature of lookup.

The evaluation of holes is straightforward as well: the current evaluation environment becomes the hole's environment. Note that there are some places where evaluation does not reach a hole, such as a hole in a function body; in the case of unevaluated expressions, the current environment needs to be recursively assigned to all holes within the unevaluated expression. Nonempty holes are the same, except that the hole expression is also evaluated as normal.

Evaluation with environments was implemented as my starter project.

3.2 Renumbering operation

Now that an evaluation environment has been introduced, we simply can avoid creating new hole closures if a hole's environment has already been encountered in a previous instance of the same hole. To do this, we can check structural equality of the environments. Alternatively, we can associate a unique identifier to each `Environment.t` (e.g., a simple global counter that increments every time an environment is extended, similar to `MetaVarGen.next`) and perform a lookup by this identifier when attempting to assign a closure number to a hole in the result. If a hole closure with the same environment exists, then reuse the closure's id; otherwise generate a new closure id that is unique to this hole.

3.3 Hole instance paths to hole closure parents

Since there may be multiple instantiations of a hole closure, all with different paths, the idea of an instance path doesn't make sense as it did for hole instances. Instead, there may be multiple parents to any hole closure. This may be represented by a DAG, which is represented using an adjacency list representation.

Currently, `HoleInstanceInfo.t` is defined as:

```
MetaVarMap.t(list((Environment.t, InstancePath.t)))
```

With these changes, the structure that keeps track of hole closures can be defined as:

```
MetaVarMap.t(EnvironmentMap.t((int, Environment.t, list(HoleClosure.t))))
```

where `EnvironmentMap = IntMap` and `HoleClosure.t` is defined analogously to `HoleInstance.t`. This is interpreted as: each hole u maps to a map from environment ID's to the information about each hole closure. Each hole closure has an integer ID, an associated environment shared by all instances of this hole closure, and a list of "parent" hole closures.

4 Implications

The performance implication is already well-discussed in the motivation.

This will notably increase the performance of the fill-and-resume operation (which is not implemented yet). This is because the fill-and-resume operation requires filling in each hole instance and then continuing evaluation. The fewer the unique hole instances to calculate, the more efficient the fill-and-resume operation will be. The use of hole closures rather than hole instances, and the adjacency list representation, do not complicate the fill-and-resume operation.