μBackpack

Monday 30th March, 2015

μBackpack is a simplified version of Backpack which is easier to implement and explain. Changes:

- A reexport may not be used to implement a declaration in a signature, *unless* the implementation comes textually before all signature, including included signatures (i.e., absent mutual recursion.)
- Type identity is computed by considering *all* of the holes in the package.
- No aliases, unless shaping is used.
- Mutual recursion is not permitted, unless shaping is used.

Generally speaking, you are subject to `hs-boot` style restrictions when mutual recursion is used, and no restrictions otherwise.

1 Things that don't work

No re-exports Signatures cannot be implemented using a module which reexports the entity in question, unless the implementation preceeds the signature. This example is rejected:

```
package p where
  A :: [x ::  Bool]
  B = [x = True]
  A = [import B; export (x)]
```

This restriction is the same as the restriction in `hs-boot` files.

No module-level granularity In the following example, `B1.B` and `B2.B` are not type-equal:

```
package p where
  A :: [data A]
  B = [data B = B]

package q where
  A1 = [data A = A1]
  A2 = [data A = A2]
  include p (A ↦ A1, B ↦ B1)
  include p (A ↦ A2, B ↦ B2)
```

No inter-package mutual recursion Inter-package mutual recursion requires a separate shaping pass to determine module identities (for same-package mutual recursion, the identity is known in advance.)

2 Worked example

```
package p where
  A :: [data A; x ::  A]
  B = [import A; data B = B A]
package r where
  P  = [data A = A; x = A]
  A2 = [import P; export (A)]
  include p
  include q
  include p (A ↦ A2, B ↦ B2)
  D  = [...stuff ...]

package q where
  A :: [data A; y ::  A]
  C = [import A; data C = C A]
```

This example exercises a few features that Backpack provides:

- `A` is never instantiated, so we can only typecheck these packages.
- `A` from `p` and `q` are linked together.
- A fresh instance of `A` from `p` is linked against `A2`.

At the end of type-checking `r`, we have the following mapping of module names to identities:

- A has identity α_A. When we import A, we actually see *two* interfaces: one which exports α_A.`A` and α_A.`x` (from `p`), and one which exports α_A.`A` and α_A.`y` (from `q`). An import of A effectively imports the merge of these two interfaces, but this merging process is left to be handled at the Haskell source level.[1] The mapping of interfaces is a local property: for example, when type-checking B, only one interface is available (the one from `p`).
- B has identity ν_B, where $\nu_B = \mathsf{p}(\mathsf{A} \mapsto \alpha_A):\mathsf{B}$ (read this as, "module B from package `p` with A instantiated with α_A"). Its interface exports ν_B.`B` with a single constructor ν_B.`B` :: α_A.`A` $\to$ ν_B.`B`
- C has identity ν_C, where $\nu_C = \mathsf{q}(\mathsf{A} \mapsto \alpha_A):\mathsf{C}$. Its interface exports ν_C.`C` with a single constructor ν_C.`C` :: α_A.`A` $\to$ ν_C.`C`
- P has identity ν_P, where $\nu_P = \mathsf{r}(\mathsf{A} \mapsto \alpha_A):\mathsf{P}$. Its interface exports ν_P.`A` and ν_P.`x`.
- A2 has identity ν_{A2}, where $\nu_{A2} = \mathsf{r}(\mathsf{A} \mapsto \alpha_A):\mathsf{A2}$. Its interface exports ν_P.`A` (nota bene, this is *not* ν_{A2}.`A`!)
- B2 has identity ν_{B2}, where $\nu_{B2} = \mathsf{p}(\mathsf{A} \mapsto \nu_{A2}):\mathsf{B}$. Its interface exports ν_{B2}.`B` with a single constructor ν_{B2}.`B` :: α_A.`A` $\to$ ν_{B2}.`B`
- D has identity ν_D, where $\nu_D = \mathsf{r}(\mathsf{A} \mapsto \alpha_A):\mathsf{D}$. Its interface exports stuff.

3 The rules

Nomenclature We use *interface* (τ) to refer to the type of a module.

The environment While type-checking a package, we have the following pieces of information:

- A package environment Δ, which holds the package definitions of all packages in scope.
- The current package name P and hole instantiation $\mathcal{H}$, such that for any module m defined in the package, $P(\mathcal{H}):m$ is the identity of that module (package level granularity).
- The package context Γ. This is a mapping from a module name to its identity and interfaces (plural!), accumulated from the modules/signatures we have type-checked and put into scope under this name.
- **Optional.** An implementation cache Φ. This is a mapping from module identity to interface for each identity that has a final implementation type-checked. This context is used to determine if an implementation comes textually before a signature (if so, when type checking the signature, the module will be present in the implementation cache). In the absence of this cache, reexports are never allowed in implementations (the `hs-boot` restriction always applies).

To begin type-checking of a particular indefinite package, generate fresh module variables α for each hole of the package (this is the only time module variables are generated). Set P to the name of the package, and $\mathcal{H}$ to be a mapping of each hole to the fresh module identity variables and begin type-checking each binding starting with an empty package context and implementation cache.

[1] From an implementation perspective, this is helpful because otherwise, it's not really clear who should actually do signature merging. Deferring merges to import time is convenient to implement.

Calculate holes As a pre-processing step, we elaborate the syntax with the holes each package requires. This proceeds recursively: the holes of a package are any signatures it defines, as well as the renamed holes of any packages it includes, minus any implemented modules. We assume for any package definition in the package environment Δ, we can tell what holes it requires. In the worked example, the set of holes for each of the packages is just A.

Typing modules A module definition $m = [M]$ is straightforwardly type-checked by using the Haskell level typing judgment $\Gamma; \nu_0 \vdash M : \tau$, where $\nu_0 = P(\mathcal{H}){:}m$. The resulting interface τ is brought into the package context as the logical binding $m \mapsto \nu_0@\tau$; we also record $\nu_0 : \tau$ in the implementation cache.

Typing signatures A signature definition $m :: [S]$ is type-checked with the Haskell level typing judgment $\Gamma; \nu_0 \vdash S : \tau$, where $\nu_0 = \mathcal{H}(m)$ if this hole was instantiated externally, or $\nu_0 = \Gamma(m)$ if it was instantiated internally. If the identity $\mathcal{H}(m)$ is already in the physical cache Φ, then the identities of the declarations in the signature are taken from the interface $\Phi(\mathcal{H}(m))$ (with the types checked for consistency); otherwise, they are given *fresh* identities originating from $\mathcal{H}(m)$ (in the same manner as an hs-boot file). The resulting type τ is brought into the package context as the logical binding $p \mapsto \mathcal{H}(m)@\tau$.

From the worked example, when A is type-checked from p of the first inclusion, $\mathcal{H}(\mathsf{A}) = \alpha_A$, there is no implementation in Φ, so the type `T` is assigned the original name α_A.`T`. In the second inclusion, $\mathcal{H}(\mathsf{A}) = \mathsf{r}(\mathsf{A} \mapsto \alpha_A){:}\mathsf{A2}$, which is implemented, so we look at the original name of `T` in $\mathsf{A2}$ and assign our type `T` the same: $\mathsf{r}(\mathsf{A} \mapsto \alpha_A){:}\mathsf{P}$.`T` (notably, $\mathsf{A2}$ is no where to be seen in this identifier, as it would be under the rule for holes.)

Typing includes An include of package P with renaming r is typed by recursively typechecking the source pointed to by P ($\Delta(P)$), but with an adjusted $\mathcal{H}'$ given thinning and renaming (and an empty context Γ'. In particular: the new $\mathcal{H}'$ is computed by taking the holes of P, renaming them according to r, and then finding the identities in $\mathcal{H}$ or Γ. In the case of cross-package mutual recursion, this lookup would fail (the shaping pass serves as an oracle which provides the correct identity).

In the worked example, we type-check with the following parameters:

- Top-level: $P = \mathsf{r}$, $\mathcal{H} = A \mapsto \alpha_A$
- `include p`: $P = \mathsf{p}$, $\mathcal{H} = A \mapsto \alpha_A$
- `include q`: $P = \mathsf{q}$, $\mathcal{H} = A \mapsto \alpha_A$
- `include p (A as A2, B as B2)`: $P = \mathsf{p}$, $\mathcal{H} = A \mapsto \nu_{A2}$

After recursively typechecking, we are left with a new context Γ'. The logical mapping of this context is renamed according to r, and then added to our current context (along with the implementation cache).

This is a non-compositional typing rule.

Merging package contexts If you have two bindings $m \mapsto \nu : \overline{\tau}$; check that the identities ν agree, and that all of the types are consistent with one another. If so, create a new binding with the lists of τ concatenated together. As an optimization, if one τ is from the final implementation, all other τs can be discarded (as they are guaranteed to be subsets of the final τ).

Merging implementation caches Cache merging is a trivial union; given two equal physical module identities their interfaces are guaranteed to be the same.

4 Complications

4.1 Signature consistency

Suppose that you want to use a signature, but you don't want to export it:

```
package p (P) where          package q where
  A :: [data A = A Bool]       include p
  P = [import A; ...]          Q = [import P; ...]
```

Evidently, we can't *completely* eliminate A, since eventually we'll need to fill it in with an implementation. Thus, the thinning must actually be shorthand for making a logical mapping in the context to an *empty* list of interfaces. In this example, in the context visible from Q, there is a binding for A which has module identity α_A but no interfaces.

The bug If we modify q to be:

```
package q where
  include p
  A :: [data A = A Int]
  Q = [import P; ...]
```

Both A from p and A from q have the same module identity, but their signatures are incompatible. However, while type-checking, we only check for signature consistency when (1) there is a backing implementation, and (2) when merging logical contexts. The first check is inapplicable in this case (A is a hole), and the second check does not see the inconsistency, because τ from p was dropped.

The fix The correct approach is to *keep* the types in the mapping (for consistency checking), but mark them as hidden (so they are not loaded on import).

4.2 Hole duplication

Hole duplication is when an indefinite module identity α is available from a package under multiple names. While this is technically disallowed under the current syntax in the absence of aliases, there are easy to imagine syntactic extensions which achieve this, e.g.

```
package p (P) where          package q where
  A :: [data A]                include p (A ↦ A1, A ↦ A2)
                               include p (A ↦ A3)
```

In this example, A1 and A2 have the same identity, while A3 has a different identity. Another example would be to have the multiple renaming occur in the thinning specification of a package.

The bug If the holes of q are specified to be A1, A2 and A3, then when type-checking q we will create three fresh identities, and A1.A and A2.A will not unify, even though they should.

The fix For every module identity, there must be a module name which is nominated as the *principal* hole: this is the only module which is included in the list of holes. This is the *only* hole which can be linked against to set the identity of the hole; any other occurrence of the module identity is ineligible. Thus, in the previous example, if A1 is the principal hole, then the holes of q are A1 and A3, and there is one valid linking:

```
package r-ok where           package r-failed where
  A1 = [data A = A]            A2 = [data A = A]
  include q                    include q
```

4.3 Module binding reordering

We stated that re-exports are not allowed if the implementation preceeds the signature, as in this example:

```
package p where
  A :: [x ::  Bool]
  C = [import A; y = True]
  B = [x = True]
  A = [import B; y = False; export (x, y)]
  D = [import A; import C; ...]
```

However, in this particular example, B does not depend on the signature of A, so the bindings in this package could be reordered to restore ordering:

```
package p where
  B = [x = True]
  A = [import B; y = False; export (x, y)]
  A :: [x ::  Bool]
  C = [import A; y = True]
  D = [import A; import C; ...]
```

This reordering is always possible as long as there is no mutual recursion.

The bug We've changed the semantics of the package: in the original example, only `x` was visible in C; in the reordered example, both `x` and `y` are visible.

The fix Float implementations to be type-checked as early as possible, but do not add the resulting interface to the package context until you arrive at their original definition site:

```
package p where
  B = [-- hidden; x = True]
  A = [-- hidden; import B; y = False; export (x, y)]
  A :: [x ::  Bool]
  C = [import A; y = True]
  B = [-- visible ]
  A = [-- visible ]
  D = [import A; import C; ...]
```

In this example, the A signature is type-checked with $\Gamma(\mathsf{A}) = \mathsf{p{:}A}$, which is present in the implementation cache; however, C is type-checked in a package context that only has τ from the signature, and not the implementation.