Garter

We will be implementing "pattern matching" with basic destructuring for Snake. First, we allow programmers to define *custom types* that have a fixed number (can be zero) of *data fields* . Then, programmers can use the `match` expression to match a Snake value against all types (primitive and custom), and run different code according to the match result.

Concrete syntax

<expr>: ...

| `type` <typedefs> `in` <expr>

| `match` <expr> `default` <expr> `:` <cases> `end`

<typedefs>:

| <typedef> `,` <typedefs>

| <typedef>

<typedef>:

| *IDENTIFIER* `(` <ids> `)`

| *IDENTIFIER*

<primtype>:

| `Num`

| `Bool`

| `Array`

| `Func`

<cases>:

| <case> <cases>

| <case>

<case>:

| `case` <typedef> `=>` <expr> `end`

| `case` <primtype> `(` <id> `)` `=>` <expr> `end`

Examples

```
type Some(value), None in
let a = Some(true) in
match a default false:
  case Some(o) => o end
  case None => [] end
end
```

should produce `true`.

```
type Burger(sauce, ham) in
type Ginger(size) in
match Ginger(2) default 5:
  case Burger(sauce, ham) => sauce end
  case Array(arr) => arr end
  case Func(f) => f end
  case Num(n) => n end
  case Bool(b) => b end
end
```

should produce `5`.

Abstract syntax

```rust
#[derive(Clone, Debug, PartialEq, Eq)]
pub enum Exp<Ann> {
    // ...
    TypeDefs(Vec<(String, Vec<String>)>, Ann),
    Match {
        expr: Box<Exp<Ann>>,
        default: Box<Exp<Ann>>,
        arms: Vec<(SnakeType, Vec<String>, Box<Exp<Ann>>)>,
        ann: Ann
    },
    // intermediate forms
    MakeInstance {
        typetag: u64,
        fields: <Box<Exp<Ann>>
        ann: Ann
    },
    MatchType {
        expr: Box<Exp<Ann>>,
        typetag: u64,
        ann: Ann
    },
    GetFields(Box<Exp<Ann>>, Ann),
}

#[derive(Copy, Clone, Debug, PartialEq, Eq)]
pub enum SnakeType {Num, Bool, Array, Func, Custom(String)}
```

Notes:

- The new intermediate forms will not appear until after the `resolve_types` pass.
- For `MakeInstance`, `fields` must represent a Snake array.
- For the `Vec<String>` in `TypeDefs`, we actually only care about its length. Notice that the field identifiers in type definition statements do not matter - these are merely placeholders (or comments).

`TypeDefs` and `Match` will not appear in `sequentialize`, but we do need to extend `SeqExp` to account for the new intermediate forms:

```
#[derive(Copy, Clone, Debug, PartialEq, Eq)]
pub enum SeqExp<Ann> {
    // ...
    MakeInstance {
        typetag: u64,
        fields: ImmExp,
        ann: Ann
    },
    MatchType {
        expr: ImmExp,
        typetag: u64,
        ann: Ann
    },
    GetFields(ImmExp, Ann),
}
```

Semantics

Compile-time errors

We will add the following compile-time errors:

1. Matching against undefined custom type:

   ```
   type Some(value) in
   match Some(true) default false:
     case Some(val) => val end
     case None => [] end
   end
   ```

   The compiler reports an error "use of undefined type `None`".

2. Destructuring with wrong arity:

   ```
   type Some(value), None in
   match Some(true) default false:
     case Some(v1, v2) => v1 end
     case None => [] end
   end
   ```

   The compiler reports an error "destructuring an instance of `Some` with wrong arity".

3. Defining the same custom type twice together:

```
type Some(value), Some in 0
```

The compiler reports an error "custom type `Some` defined repeatedly". Note: overloading is not supported.

4. Duplicate match arms:

```
type Some(value) in
match Some(true) default false:
  case Some(v) => v end
  case Some(v) => [] end
end
```

The comiler reports an error "custom type `Some` used repeatedly in one match expression".

5. Using custom type *constructors* improperly:

```
type Some(value) in
Some
```

The compiler reports an error "custom type constructor `Some` called without data fields".

```
type None in
None(0)
```

The compiler reports an error "custom type constructor `None` called with data fields".

Representation of custom type values

Each custom type will have its own unique *type tag*. Values of custom types will be stored on the heap using the following layout:

```
|          type tag          |
|      number of fields      | <-- this is the beginning of an array
|          field 1           |
|          field 2           |
|            ...             |
```

and their pointers will be tagged with `0b101`.

Shadowing

Functions, variables and custom type constructors, with or without data fields, can shadow each other as long as they have the same name. In other words, constructors, just like functions, can be semantically treated as values.

In this example:

```
def Foo(): 1 in
type Foo in
Foo()
```

The compiler will report an error because `Foo` on the 3rd line calls the *single-variant* (does not have data fields) custom type `Foo`'s constructor improperly. The function `Foo()` has been shadowed.

This example:

```
type Foo in
type Foo(a) in
Foo(1)
```

is valid and produces a value with the type tag corresponding to the second `Foo` type definition because the first one has been shadowed. Note that though overloading is not supported, shadowing of constructors is allowed.

Match expressions

The semantics of `match` is straightforward. If no arm matches the expression successfully, the value that follows `default` will be returned.

Equality

For equality checking, we will stick to reference semantics for custom type values, with the exception that all instances of one single-variant custom type are considered equal. However, an instance of one single-variant custom type is not considered equal to an instance of another shadowing / shadowed single-variant custom type with the same name. For example:

```
type Ginger in
let a = Ginger, b = Ginger in
a == b
```

produces `true`, but

```
type Ginger in
let a = Ginger in
type Ginger in
let b = Ginger in
a == b
```

produces `false`.

```
type Ginger(size) in
let a = Ginger(1), b = Ginger(1) in
a == b
```

produces `false` because `a` and `b` are stored at different locations on the heap, and `Ginger` here is not a single-variant type.

Printing custom type values

Calling `print` with a custom type value will print the type tag followed by the fields as an array. For example:

```
type Some(val), None in
print(Some(2))
```

will print

```
<1 : [2]>
```

Transformations

We will add a compiler pass called `resolve_types` right before `lambda_lift`. Consider this example:

```
type Ginger(size) in
if true:
  type Fish in
  match Ginger(3) default 5:
    case Ginger(size) => size end
    case Func(f) => f(9) end
    case Num(n) => n end
  end
else:
  0
```

This pass will tag `Ginger` with `0` and `Fish` with `1`. It then desugars all type definitions to functions or variables:

```
def Ginger(size):
  MakeInstance(0, [size])
in
if true:
  let Fish = MakeInstance(1, []) in
  match Ginger(3) default 5:
    case Ginger(size) => size end
    case Func(f) => f(9) end
    case Num(n) => n end
  end
else:
  0
```

and desugars the `match` syntax using `if` expressions:

```
def Ginger(size):
  MakeInstance(0, [size])
in
if true:
  let Fish = MakeInstance(1, []) in
  let matchee = Ginger(3),
      fields = GetFields(matchee) in
  if MatchType(matchee, 0):
    let size = fields[0] in
    size
  else: if isfunc(matchee):
    matchee(9)
  else: if isnum(matchee):
    matchee
  else:
    5
else:
  0
```

`MakeInstance`, `MatchType` and `GetFields` are intermediate forms (just like `MakeClosure`).

`MakeInstance(TypeTag, SnakeVal)` takes a type tag (Rust value!) and a Snake array, and returns a Snake value of the given type constructed with field values specified in the array. Again, the return value will be in the form of a pointer tagged `0b101`.

`MatchType(SnakeVal, TypeTag)` takes a Snake value and a type tag. It returns `true` if and only if the Snake value is of the custom type corresponding to the type tag.

`GetFields(SnakeVal)` takes a Snake value and returns a Snake array containing all the data fields in this Snake value if the given value is a custom type instance. Otherwise it returns a dummy `false` (or any other Snake value). By design, if the given value is primitive, we will not fall into a `match` arm that tries to access the return value of `GetFields`, so we are safe.