This is a "Literate Haskell" source file. It is meant to be read as documentation but it's also compilable source code. The text is formatted using Markdown so GitHub can render it nicely.

This module translates source code written in C into source code written in Rust. Well, that's a small lie. For both C and Rust source we use an abstract representation rather than the raw source text. The C representation comes from the language-c package, and the Rust representation is in Corrode's `Language.Rust.AST` module.

I've tried to stick to the standard features of the Haskell2010 language, but GHC's `ViewPatterns` extension is too useful for this kind of code.

```
{-# LANGUAGE ViewPatterns #-}
```

This module exports a single function which transforms a C "translation unit" (terminology from the C standard and reused in language-c) into a collection of Rust "items" (terminology from the Rust language specification). It imports the C and Rust ASTs, plus a handful of other useful data structures and control flow abstractions.

```
module Language.Rust.Corrode.C (interpretTranslationUnit) where

import Control.Monad
import Control.Monad.ST
import Control.Monad.Trans.Class
import Control.Monad.Trans.Except
import Control.Monad.Trans.RWS.Strict
import Data.Foldable
import qualified Data.Map.Lazy as Map
import qualified Data.IntMap.Strict as IntMap
import Data.Maybe
import Data.List
import Data.STRef
import qualified Data.Set as Set
import Language.C
import Language.C.Data.Ident
import qualified Language.Rust.AST as Rust
import Language.Rust.Corrode.CFG
import Language.Rust.Corrode.CrateMap
import Text.PrettyPrint.HughesPJClass hiding (Pretty)
```

This translation proceeds in a syntax-directed way. That is, we just walk down the C abstract syntax tree, and for each piece of syntax we find, we decide what the equivalent Rust should be.

A tool like this could use a more sophisticated strategy than the purely syntax-directed approach. We could construct intermediate data structures like most modern compilers do, such as a control flow graph, and do fancy analysis passes over those data structures.

However, purely syntax-directed translation has two advantages:

1. The implementation is simpler,
2. and it's easier to preserve as much structure as possible from the source program, which is one of Corrode's design goals.

So far, we haven't needed more sophisticated approaches, and we'll stick with syntax-directed style for as long as possible.

That said, we do need to maintain some data structures with intermediate results as we go through the translation.

Intermediate data structures

The C language is defined such that a compiler can generate code in a single top-to-bottom pass through a translation unit. That is, any uses of a name must come after the point in the translation unit where that name is first declared.

That's convenient for us, because it means we can easily keep the environment in a "state monad", and emit the translated Rust items using a "writer monad". These are fancy terms for objects that have some simple operations.

- The state monad has `get`, `put`, and `modify` operations that we use to query or update the current enviroment.
- The writer monad has a `tell` operation we use to add new Rust items to the output.

Actually, we use a combined type, `RWS`, which supports the operations of reader, writer, and state monads all in one.

You probably don't need to read any of the awful monad tutorials out there just to use these! The important point is that we have this type alias, `EnvMonad`, which you'll see throughout this module. It marks pieces of code which have access to the environment and the output.

```
type EnvMonad s = ExceptT String (RWST FunctionContext Output (EnvState s) (ST s))
```

In fact, we're going to wrap up the monad operations in some helper functions, and then only use those helpers everywhere else. But first, let's define our state and writer types.

Context

```
data FunctionContext = FunctionContext
    { functionReturnType :: Maybe CType
    , functionName :: Maybe String
    , itemRewrites :: ItemRewrites
    }
```

Output

As we proceed through the C source, we'll accumulate Rust definitions that we want to include in the final output. This type simply holds any data that we want to have bubble up from further down the parse tree. For example, a `struct` might be defined inside a declaration that's inside a loop that's inside a function. We need to get that struct out to its own item in Rust.

```
data Output = Output
    { outputItems :: [Rust.Item]
    , outputExterns :: Map.Map String Rust.ExternItem
    , outputIncomplete :: Set.Set String
    }
```

As we generate output from translating each piece of C syntax, we need a way to combine the output from different pieces. There's a standard Haskell typeclass (which is like a Rust trait) called `Monoid` which encapsulates this idea of combining pieces of output.

```
instance Monoid Output where
```

For our output to be a monoid, it needs to specify

- what empty output looks like (which is just output where every field is empty),

```
    mempty = Output
        { outputItems = mempty
        , outputExterns = mempty
```

```
        , outputIncomplete = mempty
        }
```

- and how to combine two different pieces of output (which we do by pairwise combining each field).

```
    mappend a b = Output
        { outputItems = outputItems a `mappend` outputItems b
        , outputExterns = outputExterns a `mappend` outputExterns b
        , outputIncomplete = outputIncomplete a `mappend` outputIncomplete b
        }
```

`emitItems` adds a list of Rust items to the output.

```
emitItems :: [Rust.Item] -> EnvMonad s ()
emitItems items = lift $ tell mempty { outputItems = items }
```

`emitIncomplete` records that we saw an incomplete type. For our purposes, an incomplete type is anything with a definition we can't translate, but we can still allow pointers to values of this type as long as nobody tries dereferencing those pointers.

```
emitIncomplete :: ItemKind -> Ident -> EnvMonad s CType
emitIncomplete kind ident = do
    rewrites <- lift (asks itemRewrites)
    unless (Map.member (kind, identToString ident) rewrites) $
        lift $ tell mempty { outputIncomplete = Set.singleton (identToString ident) }
    return (IsIncomplete ident)
```

Frequently, a type is incomplete only because it had a forward declaration; in that case, once the real declaration is seen, we shouldn't consider it incomplete any more. However, we choose not to rewrite the environment to reflect the newly-completed type. Instead, when we look something up, if we discover that it had an incomplete type, we re-check for whether that type has since been completed. The caller still needs to be prepared to handle an incomplete type, though.

```
completeType :: CType -> EnvMonad s CType
completeType orig@(IsIncomplete ident) = do
    mty <- getTagIdent ident
    fromMaybe (return orig) mty
completeType ty = return ty
```

Global state

The above `Output` type, used in the writer monad, accumulates bottom-up, from the leaves of the AST upward. So as we translate a particular expression, say, we can look at the output from its subexpressions, but we aren't allowed to examine any output from adjacent expressions, statements, or functions.

By contrast, sometimes we need to look at information we computed from arbitrary places earlier in the source code. For that we'll use a state monad to save whatever we might need later.

Let's divide state into two categories: what we'll call "global" and "scope-limited" state. Changes to scope-limited state, covered below, get undone whenever we leave the C scope that caused those changes. Global state, on the other hand, is permanent.

Sometimes we need to construct a unique name, because the Rust pattern we want to generate requires a name someplace where C did not require one. We'll follow a standard idiom for this and just generate a unique number each time we need a new name.

```
data GlobalState = GlobalState
    { unique :: Int
    , usedForwardRefs :: Set.Set Ident
    }
```

`uniqueName` generates a new name with the given base and a new unique number:

```
uniqueName :: String -> EnvMonad s String
uniqueName base = modifyGlobal $ \ st ->
    (st { unique = unique st + 1 }, base ++ show (unique st))
```

When a forward-declaration is referenced but we haven't seen the later definition yet, we need to keep track of the fact that eventually we'll need that symbol, so we can force it to be translated once we finally do see it.

```
useForwardRef :: Ident -> EnvMonad s ()
useForwardRef ident = modifyGlobal $ \ st ->
    (st { usedForwardRefs = Set.insert ident (usedForwardRefs st) }, ())
```

Scope-limited state

One of the biggest challenges in correctly translating C to Rust is that we need to know what type C would give every expression, so we can ensure that the Rust equivalent uses the same types. To do that, we need to keep track of all the names that are currently in scope, along with their type information.

C has three namespaces for variable, function, and type names:

- field names, introduced inside `struct` or `union` definitions;
- identifiers, introduced by declarations, which may be variables, functions, or typedefs;
- and tags, introduced with `struct <tag>`, `union <tag>`, or `enum <tag>` (C99 section 6.7.2.3).

For example, here is a legal snippet of C source, using the name `foo` in all three ways:

```
typedef struct foo { struct foo *foo; } foo;
```

Within a scope, names which are in the same namespace must refer to the same thing. For example, you can't declare both a `struct x` and an `enum x` in the same scope, because they both use the same tag. Similarly, you can't declare a `typedef` and a variable with the same name in the same scope, because both declare identifiers.

Rust divides namespaces a different way. In Rust, types are in one namespace–not just `struct` and `enum` types, as in C, but also type aliases. Variables and functions are in another namespace. (Unit-like and tuple-like structs are actually in *both* namespaces, though fortunately Corrode never generates those kinds of structs.)

Fortunately, language-c does the hard work for us of disambiguating these different uses of identical-looking names. As we'll see later, whenever an identifier occurs, we can tell which namespace to use for it from the AST context in which it appeared.

We'll come back to our internal representation of C types at the end of this module.

The identifiers, typedefs, and tags that are visible at any given point in the program are limited by C's scoping rules. We package them up into a type. This type also carries a reference to the `GlobalState` value defined above.

```
data EnvState s = EnvState
    { symbolEnvironment :: [(Ident, EnvMonad s Result)]
    , typedefEnvironment :: [(Ident, EnvMonad s IntermediateType)]
    , tagEnvironment :: [(Ident, EnvMonad s CType)]
    , globalState :: GlobalState
    }
```

`modifyGlobal` updates the global state according to a specified transformation, and returns a value computed by that transformation.

```
modifyGlobal :: (GlobalState -> (GlobalState, a)) -> EnvMonad s a
modifyGlobal f = lift $ do
    st <- get
```

```
    let (global', a) = f (globalState st)
    put st { globalState = global' }
    return a
```

Some names are special in C or special in Rust, or both. We rename those as we encounter them. At the moment, only `main` gets this special treatment.

```
applyRenames :: Ident -> String
applyRenames ident = case identToString ident of
    "final" -> "final_"
    "fn" -> "fn_"
    "in" -> "in_"
    "let" -> "let_"
    "main" -> "_c_main"
    "match" -> "match_"
    "mod" -> "mod_"
    "proc" -> "proc_"
    "type" -> "type_"
    "where" -> "where_"
    name -> name
```

`get*Ident` looks up a name from the appropriate namespace in the environment, and returns the type information we have for it, or `Nothing` if we haven't seen a declaration for that name yet.

```
getSymbolIdent :: Ident -> EnvMonad s (Maybe Result)
getSymbolIdent ident = do
    env <- lift get
    case lookup ident (symbolEnvironment env) of
        Just symbol -> fmap Just symbol
        Nothing -> case identToString ident of
            "__func__" -> getFunctionName ""
            "__FUNCTION__" -> getFunctionName ""
            "__PRETTY_FUNCTION__" -> getFunctionName "top level"
            name -> return $ lookup name builtinSymbols
    where
    getFunctionName def = do
        name <- lift (asks functionName)
        let name' = fromMaybe def name
        return $ Just Result
            { resultType = IsArray Rust.Immutable (length name' + 1) charType
            , resultMutable = Rust.Immutable
            , result = Rust.Deref (Rust.Lit (Rust.LitByteStr (name' ++ "\NUL")))
            }
    builtinSymbols =
        [ ("__builtin_bswap" ++ show w, Result
            { resultType = IsFunc (IsInt Unsigned (BitWidth w))
                [(Nothing, IsInt Unsigned (BitWidth w))] False
            , resultMutable = Rust.Immutable
            , result = Rust.Path (Rust.PathSegments ["u" ++ show w, "swap_bytes"])
            })
        | w <- [16, 32, 64]
        ]
        ++
        [ ("__FILE__", Result
            { resultType = IsPtr Rust.Immutable charType
            , resultMutable = Rust.Immutable
```

```
            , result = Rust.MethodCall (
                    Rust.Call (Rust.Var (Rust.VarName "file!")) []
                ) (Rust.VarName "as_ptr") []
            })
        , ("__LINE__", Result
            { resultType = IsInt Unsigned (BitWidth 32)
            , resultMutable = Rust.Immutable
            , result = Rust.Call (Rust.Var (Rust.VarName "line!")) []
            })
        ]

getTypedefIdent :: Ident -> EnvMonad s (String, Maybe (EnvMonad s IntermediateType))
getTypedefIdent ident = lift $ do
    env <- gets typedefEnvironment
    return (identToString ident, lookup ident env)

getTagIdent :: Ident -> EnvMonad s (Maybe (EnvMonad s CType))
getTagIdent ident = lift $ do
    env <- gets tagEnvironment
    return $ lookup ident env
```

`add*Ident` saves type information into the environment.

```
addSymbolIdent :: Ident -> (Rust.Mutable, CType) -> EnvMonad s String
addSymbolIdent ident (mut, ty) = do
    let name = applyRenames ident
    addSymbolIdentAction ident $ return Result
        { resultType = ty
        , resultMutable = mut
        , result = Rust.Path (Rust.PathSegments [name])
        }
    return name

addSymbolIdentAction :: Ident -> EnvMonad s Result -> EnvMonad s ()
addSymbolIdentAction ident action = lift $ do
    modify $ \ st -> st { symbolEnvironment = (ident, action) : symbolEnvironment st }

addTypedefIdent :: Ident -> EnvMonad s IntermediateType -> EnvMonad s ()
addTypedefIdent ident ty = lift $ do
    modify $ \ st -> st { typedefEnvironment = (ident, ty) : typedefEnvironment st }

addTagIdent :: Ident -> EnvMonad s CType -> EnvMonad s ()
addTagIdent ident ty = lift $ do
    modify $ \ st -> st { tagEnvironment = (ident, ty) : tagEnvironment st }
```

`addExternIdent` saves type information into the environment like `addSymbolIdent`. But external declarations are not immediately added to the output. Most declarations in header files go unused in any given translation unit so including them in the output is just clutter; and even if a declaration is used, this translation unit may have a non-`extern` definition of the same symbol, in which case we should only emit the full definition. So this function records which extern declaration *would* be emitted if it turns out that we need it.

```
addExternIdent
    :: Ident
    -> EnvMonad s IntermediateType
    -> (String -> (Rust.Mutable, CType) -> Rust.ExternItem)
    -> EnvMonad s ()
```

```
addExternIdent ident deferred mkItem = do
    action <- runOnce $ do
        itype <- deferred
        rewrites <- lift $ asks itemRewrites
        path <- case Map.lookup (Symbol, identToString ident) rewrites of
            Just renamed -> return ("" : renamed)
            Nothing -> do
                let name = applyRenames ident
                let ty = (typeMutable itype, typeRep itype)
                lift $ tell mempty { outputExterns = Map.singleton name (mkItem name ty) }
                return [name]
        return (typeToResult itype (Rust.Path (Rust.PathSegments path)))
    addSymbolIdentAction ident action
```

Reporting errors

There will always be C source code that language-c parses happily but Corrode can't translate to Rust, and we should report as much helpful explanation as possible when that happens.

```
noTranslation :: (Pretty node, Pos node) => node -> String -> EnvMonad s a
noTranslation node msg = throwE $ concat
    [ show (posOf node)
    , ": "
    , msg
    , ":\n"
    , render (nest 4 (pretty node))
    ]
```

In some cases, we should be able to translate the given input, but the translation just hasn't been implemented yet.

```
unimplemented :: (Pretty node, Pos node) => node -> EnvMonad s a
unimplemented node = noTranslation node "Corrode doesn't handle this yet"
```

Some C source is illegal according to the C standard but is nonetheless syntactically valid. Corrode does not promise to detect all of these cases, but we can call `badSource` when we do detect such an error.

```
badSource :: (Pretty node, Pos node) => node -> String -> EnvMonad s a
badSource node msg = noTranslation node
    ("illegal " ++ msg ++ "; check whether a real C compiler accepts this")
```

Top-level translation

Since we're taking the syntax-directed approach, let's start by looking at how we handle the root of the C AST: the "translation unit". This is C's term for a single source file (`*.c`), but since this code operates on already pre-processed source, the translation unit includes any code that was `#included` from header files as well.

(It would be nice to be able to translate macros and conditionally-compiled code to similar constructs in Rust, but it's much more difficult to write a reasonable C parser for source code that hasn't been run through the C pre-processor yet. And that's saying something, as parsing even pre-processed C is hard. So language-c runs the pre-processor from either `gcc` or `clang` before parsing the source file you hand it. See Corrode's `Main.hs` for the driver code.)

`interpretTranslationUnit` takes a language-c AST representation of a translation unit, and returns a list of Rust AST top-level declaration items.

TODO: Use `thisModule :: ModuleMap` as the list of precisely which declarations should be public from this module. Allow `static` symbols to be made public this way, and non-`static` symbols to be hidden. For included symbols which are not defined in this translation unit, make their `extern` declaration public. Rename definitions as specified, too, but save the original name in a `#[link_name = "..."]` for externs and a `#[export_name = "..."]` for top-level definitions.

```
interpretTranslationUnit :: ModuleMap -> ItemRewrites -> CTranslUnit -> Either String [Rust.Item]
interpretTranslationUnit _thisModule rewrites (CTranslUnit decls _) = case err of
    Left msg -> Left msg
    Right _ -> Right items'
    where
```

Here at the top level of the translation unit, we need to extract the translation results from the writer and state monads described above. Specifically, we:

- set the environment to be initially empty;
- discard the final environment as we don't need any of the intermediate type information once translation is done;
- and get the final lists of items and externs that were emitted during translation.

```
    initFlow = FunctionContext
        { functionReturnType = Nothing
        , functionName = Nothing
        , itemRewrites = rewrites
        }
    initState = EnvState
        { symbolEnvironment = []
        , typedefEnvironment = []
        , tagEnvironment = []
        , globalState = GlobalState
            { unique = 1
            , usedForwardRefs = Set.empty
            }
        }
    (err, output) = runST (evalRWST (runExceptT (mapM_ perDecl decls)) initFlow initState)
```

With the initial environment set up, we can descend to the next level of the abstract syntax tree, where the language-c types tell us there are three possible kinds for each declaration:

```
    perDecl (CFDefExt f) = interpretFunction f
    perDecl (CDeclExt decl') = do
        binds <- interpretDeclarations makeStaticBinding decl'
        emitItems binds
    perDecl decl = unimplemented decl
```

Incomplete types (see `emitIncomplete` above) may be referenced early and then completed later. In that case we need to only emit the complete definition.

If they are never completed, however, then we need to emit some Rust type that can only be passed around by reference. We can't allow the Rust compiler to construct, copy, or consume values of the incomplete type, because we don't know how big it is. We also want each incomplete type to be distinct from all other incomplete types.

We meet those requirements by, for each incomplete type, creating a private `enum` type. We don't give the type any constructors, so new values can't be constructed and it can't be `match`ed on. Unlike the translation of other types, we don't declare this enum's `repr`, and we don't derive `Copy` or `Clone` for it.

```
    completeTypes = Set.fromList $ catMaybes
        [ case item of
```

```
            Rust.Item _ _ (Rust.Struct name _) -> Just name
            _ -> Nothing
        | item <- outputItems output
        ]
    incompleteTypes = outputIncomplete output `Set.difference` completeTypes
    incompleteItems =
        [ Rust.Item [] Rust.Private (Rust.Enum name [])
        | name <- Set.toList incompleteTypes
        ]
```

Next we remove any locally-defined names from the list of external declarations. We can't tell whether something is actually external when we encounter its declaration, but once we've collected all the symbols for this translation unit it becomes clear.

```
    itemNames = catMaybes
        [ case item of
            Rust.Item _ _ (Rust.Function _ name _ _ _) -> Just name
            Rust.Item _ _ (Rust.Static _ (Rust.VarName name) _ _) -> Just name
            _ -> Nothing
        | item <- outputItems output
        ]

    externs' =
        [ extern
        | (name, extern) <- Map.toList (outputExterns output)
        , name `notElem` itemNames
        ]
```

If a type declaration isn't actually used by any of the other declarations that we're keeping, then drop it. C headers declare lots of types, and most of them don't get used in any given source file.

This is not just a nice cleanup of the output. It also allows us to translate headers containing types we can't yet translate, as long as those types aren't actually used.

```
    items = incompleteItems ++ outputItems output
```

If there are any external declarations after filtering, then we need to wrap them in an `extern { }` block. We place that before the other items, by convention.

```
    items' = if null externs'
        then items
        else Rust.Item [] Rust.Private (Rust.Extern externs') : items
```

Declarations

C declarations appear the same, syntactically, whether they're at top-level or local to a function. `interpretDeclarations` handles both cases.

However, Rust has a different set of cases which we need to map onto: variables with `'static` lifetime are declared using `static` items, regardless of whether they're local or at top-level; while local variables are declared using `let`-binding statements.

So `interpretDeclarations` is parameterized by a function, `makeBinding`, which can construct a `static` item or a `let`-binding from non-static C variable declarations. In either case, the binding

- may be mutable or immutable (`const`);
- has a variable name;

- has a type (which we choose to be explicit about even for `let`-bindings where the type could generally be inferred);
- and may have an initializer expression.

The return type of the function depends on which kind of binding it's constructing.

```
type MakeBinding s a = (Rust.ItemKind -> a, Rust.Mutable -> Rust.Var -> CType -> NodeInfo -> Maybe CIni
```

It's up to the caller to provide an appropriate implementation of `makeBinding`, but there are only two sensible choices, which we'll define here for convenient use elsewhere.

FIXME: Construct a correct default value for non-scalar static variables.

```
makeStaticBinding :: MakeBinding s Rust.Item
makeStaticBinding = (Rust.Item [] Rust.Private, makeBinding)
    where
    makeBinding mut var ty node minit = do
        expr <- interpretInitializer ty (fromMaybe (CInitList [] node) minit)
        return $ Rust.Item attrs Rust.Public
            (Rust.Static mut var (toRustType ty) expr)
    attrs = [Rust.Attribute "no_mangle"]

makeLetBinding :: MakeBinding s Rust.Stmt
makeLetBinding = (Rust.StmtItem [], makeBinding)
    where
    makeBinding mut var ty _ minit = do
        mexpr <- mapM (interpretInitializer ty) minit
        return $ Rust.Let mut var (Just (toRustType ty)) mexpr
```

Now that we know how to translate variable declarations, everything else works the same regardless of where the declaration appears. Here are a sample of the simplest cases we have to handle at this point:

```
// variable definition, with initial value
int x = 42;

// variable declaration, which might refer to another module
extern int y;

// type alias
typedef int my_int;

// struct definition
struct my_struct {
    // using the type alias
    my_int field;
};

// variable definition using previously-defined struct
struct my_struct s;

// function prototypes, which might refer to another module
int f(struct my_struct *);
extern int g(my_int i);

// function prototype for a definitely-local function
static int h(void);
```

Some key things to note:

- C allows as few as **zero** "declarators" in a declaration. In the above examples, the definition of `struct my_struct` has zero declarators, while all the other declarations have one declarator each.
- `struct`/`union`/`enum` definitions are a kind of declaration specifier, like `int` or `const` or `static`, so we have to dig them out of the declarations where they appear—possibly nested inside other `struct` definitions, and possibly with variables immediately declared as having the new type. In Rust, each `struct` definition must be a separate item, and then after that any variables can be declared using the new types.
- Syntactically, `typedef` is a storage class specifier just like `static` or `extern`. But semantically, they couldn't be more different!

And here are some legal declarations which illustrate how weirdly complicated C's declaration syntax is.

- These have no effect because they only have declaration specifiers, no declarators:

  ```
  int;
  const typedef volatile void;
  ```

- Declare both a variable of type `int`, and a function prototype for a function returning `int`:

  ```
  int i, fun(void);
  ```

- Declare variables of type `int`, pointer to `int`, function returning pointer to `int`, and pointer to function returning `int`, respectively:

  ```
  int j, *p, *fip(void), (*fp)(void);
  ```

- Declare type aliases for both `int` and `int *`:

  ```
  typedef int new_int, *int_ptr;
  ```

OK, let's see how `interpretDeclarations` handles all these cases. It takes one of the `MakeBinding` implementations and a single C declaration. It returns a list of bindings constructed using `MakeBinding`, and also updates the environment and output as needed.

```
interpretDeclarations :: MakeBinding s b -> CDecl -> EnvMonad s [b]
interpretDeclarations (fromItem, makeBinding) declaration@(CDecl specs decls _) = do
```

First, we call `baseTypeOf` to get our internal representation of the type described by the declaration specifiers. That function will also add any `struct`s defined in this declaration to the environment so we can look them up if they're referred to again, and emit those structs as items in the output.

If there are no declarators, then `baseTypeOf`'s side effects are the only output from this function.

```
    (storagespecs, baseTy) <- baseTypeOf specs
```

If there are any declarators, we need to figure out which kind of Rust declaration to emit for each one, if any.

```
    mbinds <- forM decls $ \ declarator -> do
```

This function is only used for what `language-c` calls "toplevel declarations", and as such, "the elements of the non-empty `init-declarator-list` are of the form `(Just declr, init?, Nothing)`. The declarator `declr` has to be present and non-abstract and the initialization expression is optional."

```
        (decl, minit) <- case declarator of
            (Just decl, minit, Nothing) -> return (decl, minit)
            (Nothing, _, _) -> badSource declaration "absent declarator"
            (_, _, Just _) -> badSource declaration "bitfield declarator"

        -- FIXME: if `specs` is a typedef reference, dig more derived out of that.
        (ident, derived) <- case decl of
            CDeclr (Just ident) derived _ _ _ -> return (ident, derived)
            _ -> badSource decl "abstract declarator"
```

```
deferred <- derivedDeferredTypeOf baseTy decl []
case (storagespecs, derived) of
```

Each `typedef` declarator is added to the environment. They must not have an initializer and do not return any declarations, so the only effect of a `typedef` is to update the environment.

> **TODO**: It would be nice to emit a type-alias item for each `typedef` and use the alias names anywhere we can, instead of replacing the aliases with the underlying type everywhere. That way we preserve more of the structure of the input program. But it isn't always possible, so this requires careful thought.

```
(Just (CTypedef _), _) -> do
    when (isJust minit) (badSource decl "initializer on typedef")
    addTypedefIdent ident deferred
    return Nothing
```

Non-`typedef` declarations may have an initializer. Each declarator is added to the environment as a new symbol.

Static function prototypes don't need to be translated because the function definition must be in the same translation unit. We still need to have the function's type signature in the environment though.

```
(Just (CStatic _), CFunDeclr{} : _) -> do
    addSymbolIdentAction ident $ do
        itype <- deferred
        useForwardRef ident
        return (typeToResult itype (Rust.Path (Rust.PathSegments [applyRenames ident])))
    return Nothing
```

Other function prototypes need to be translated unless the function definition appears in the same translation unit; do it and prune duplicates later.

```
(_, CFunDeclr{} : _) -> do
    addExternIdent ident deferred $ \ name (_mut, ty) -> case ty of
        IsFunc retTy args variadic ->
            let formals =
                    [ (Rust.VarName argName, toRustType argTy)
                    | (idx, (mname, argTy)) <- zip [1 :: Int ..] args
                    , let argName = maybe ("arg" ++ show idx) (applyRenames . snd) mname
                    ]
            in Rust.ExternFn name formals variadic (toRustRetType retTy)
        _ -> error (show ident ++ " is both a function and not a function?")
    return Nothing
```

Non-function externs need to be translated unless an identical non-extern declaration appears in the same translation unit; do it and prune duplicates later.

```
(Just (CExtern _), _) -> do
    addExternIdent ident deferred $ \ name (mut, ty) ->
        Rust.ExternStatic mut (Rust.VarName name) (toRustType ty)
    return Nothing
```

Declarations with storage class `static` always need to construct Rust static items. These items always have an initializer, even if it's just the zero-equivalent initializer. We use the caller's `fromItem` callback to turn the item (actually an `ItemKind`) into the same type that we're returning for other bindings.

```
(Just (CStatic _), _) -> do
    IntermediateType
        { typeMutable = mut
```

```
            , typeRep = ty } <- deferred
        name <- addSymbolIdent ident (mut, ty)
        expr <- interpretInitializer ty (fromMaybe (CInitList [] (nodeInfo decl)) minit)
        return (Just (fromItem
            (Rust.Static mut (Rust.VarName name) (toRustType ty) expr)))
```

Anything else is a variable declaration to translate. This is the only case where we use `makeBinding`. If there's an initializer, we also need to translate that; see below.

```
    _ -> do
        IntermediateType
            { typeMutable = mut
            , typeRep = ty } <- deferred
        name <- addSymbolIdent ident (mut, ty)
        binding <- makeBinding mut (Rust.VarName name) ty (nodeInfo decl) minit
        return (Just binding)
```

Return the bindings produced for any declarator that did not return `Nothing`.

```
    return (catMaybes mbinds)

interpretDeclarations _ node@(CStaticAssert {}) = unimplemented node
```

Initialization

The general form of initialization, described in C99 section 6.7.8, involves an initializer construct. The interface we want to expose for translating C initializers is fairly simple: given the type of the thing we are trying to initialize and the C initializer, produce a Rust expression that corresponds to the C expression that would have been initialized:

```
interpretInitializer :: CType -> CInit -> EnvMonad s Rust.Expr
```

Unfortunately, we have to delay the actual implementation of this function until the end of this section when we will have all the necessary pieces.

The problem is that in C, there are just too many ways of expressing the same initializations. Take, for example, the following struct definitions

```
struct Foo { struct Bar b; int z; }
struct Bar { int x; int y; }
```

Then, the following are all equivalent

```
struct Foo s = { 1, 2, 3 }
struct Foo s = { .b = { .x = 1, .y = 2 }, .z = 3 }
struct Foo s = { .b.y = 1, .b = { 1, 2 }, 3 }
struct Foo s = { (struct Bar) { 1, 2 }, 3 }
```

We need some canonical form for manipulating and composing initializer expressions. Then, we can deal with C initialization expressions in several steps: start by converting them to a canonical form, compose them together accordingly, and finally convert them to Rust expressions.

Our canonical form may have a base expression, which (if we're initializing an aggregate) may have some of its fields overridden. If a base expression is present, it overrides all previous initializers for this object. Otherwise, all fields not specified in the `IntMap` will get initialized to their zero-equivalent values.

```
data Initializer
    = Initializer (Maybe Rust.Expr) (IntMap.IntMap Initializer)

scalar :: Rust.Expr -> Initializer
scalar expr = Initializer (Just expr) IntMap.empty
```

Notice that combining initializers is an associative binary operation. This motivates us to use the `Monoid` typeclass again to represent the operation for combining two initializers.

```
instance Monoid Initializer where
```

- The identity element in this case will be the empty initializer. This is because whenever it is combined with another initializer (either from the left or the right), the result is just the other initializer.

```
    mempty = Initializer Nothing IntMap.empty
```

- When combining two initializers, the one on the right overrides/shadows definitions made by the one on the left.

```
    mappend _ b@(Initializer (Just _) _) = b
    mappend (Initializer m a) (Initializer Nothing b) =
        Initializer m (IntMap.unionWith mappend a b)
```

Now, we need to concern ourselves with constructing these initializers in the first place. We will need to keep track of the current object (see point 17 of section 6.7.8). A `Designator` describes a position inside a type.

```
type CurrentObject = Maybe Designator

data Designator
  = Base CType
```

- encodes the type of the base object pointed to

```
  | From CType Int [CType] Designator
  deriving(Show)
```

- encodes the type of the object pointed to, its index in the parent, remaining fields in the parent, and the parent designator

In several places, we need to know the type of a designated object.

```
designatorType :: Designator -> CType
designatorType (Base ty) = ty
designatorType (From ty _ _ _) = ty
```

Then, given a list of designators and the type we are currently in, we can compute the most general possible current object.

```
objectFromDesignators :: CType -> [CDesignator] -> EnvMonad s CurrentObject
objectFromDesignators _ [] = pure Nothing
objectFromDesignators ty desigs = Just <$> go ty desigs (Base ty)
    where

    go :: CType -> [CDesignator] -> Designator -> EnvMonad s Designator
    go _ [] obj = pure obj
    go (IsArray _ size el) (CArrDesig idxExpr _ : ds) obj = do
        idx <- interpretConstExpr idxExpr
        go el ds (From el (fromInteger idx) (replicate (size - fromInteger idx - 1) el) obj)
    go (IsStruct name fields) (d@(CMemberDesig ident _) : ds) obj = do
        case span (\ (field, _) -> applyRenames ident /= field) fields of
            (_, []) -> badSource d ("designator for field not in struct " ++ name)
            (earlier, (_, ty') : rest) ->
                go ty' ds (From ty' (length earlier) (map snd rest) obj)
    go ty' (d : _) _ = badSource d ("designator for " ++ show ty')
```

However, since it is possible for some entries in an initializer to have no designators (in which case the initializer implicitly applies to the next object), we need a way to calculate the most general next object from

the current one (provided we haven't reached the end of the thing we are initializing).

```
nextObject :: Designator -> CurrentObject
nextObject Base{} = Nothing
nextObject (From _ i (ty : remaining) base) = Just (From ty (i+1) remaining base)
nextObject (From _ _ [] base) = nextObject base
```

The type of an initializer expression is compatible with the type of the object it's initializing if either:

- Both have structure type and they're the same `struct`,
- Or neither have structure type.

In the latter case we don't check what type they are, because we can cast any scalar type to any other as needed.

```
compatibleInitializer :: CType -> CType -> Bool
compatibleInitializer (IsStruct name1 _) (IsStruct name2 _) = name1 == name2
compatibleInitializer IsStruct{} _ = False
compatibleInitializer _ IsStruct{} = False
compatibleInitializer _ _ = True
```

We've used the expression "the most general (object)" several times. This is because designators alone aren't actually enough to determine exactly what gets initialized—we also need to compare types.

An initializer expression initializes the current object, if the two have compatible types. Otherwise, we extend the designator to refer to the first sub-object of the current object and check whether *that* object is compatible with the initializer. We keep descending through first sub-objects until we run out of sub-objects without finding any compatible objects.

```
nestedObject :: CType -> Designator -> Maybe Designator
nestedObject ty desig = case designatorType desig of
    IsArray _ size el -> Just (From el 0 (replicate (size - 1) el) desig)
    ty' | ty `compatibleInitializer` ty' -> Just desig
    IsStruct _ ((_ , ty') : fields) ->
        nestedObject ty (From ty' 0 (map snd fields) desig)
    _ -> Nothing
```

Given these helpers, we are now in a position to translate C initializers to our initializers.

When we have a list of expressions, we start by parsing all of the designators into our internal representation.

```
translateInitList :: CType -> CInitList -> EnvMonad s Initializer
translateInitList ty list = do

    objectsAndInitializers <- forM list $ \ (desigs, initial) -> do
        currObj <- objectFromDesignators ty desigs
        pure (currObj, initial)
```

Next, we have to choose the starting current object (`base`). For aggregate types, the first current object points to their first field but for scalar types it points to the primitive itself. For example

```
struct point { int x, y };

int i = { 1, 3 };
struct point p = { 1, 3 };
```

In the first example, the whole of `i` gets initialized to `1` (and `3` is ignored) since `i` is not a struct. On the other, in the second example, it is the fields of `p` that get initialized to `1` and `3` since `p` is a struct.

```
    let base = case ty of
                  IsArray _ size el -> From el 0 (replicate (size - 1) el) (Base ty)
```

```
                IsStruct _ ((_,ty'):fields) -> From ty' 0 (map snd fields) (Base ty)
                _ -> Base ty
```

Finally, we are ready to calculate the initializer for the whole list. We walk through the list of designators and their initializers from left to right passing along the current object as we go (in case the initializer that follows has no designator).

When we get to the end, we throw away the final current object as initializer lists never affect the current object of their enclosing initializer.

```
    (_, initializer) <- foldM resolveCurrentObject (Just base, mempty) objectsAndInitializers
    return initializer
```

Resolution takes a current object to use if no designator is specified. It returns the new current object for the next element to use, and the above `Initializer` type representing the part of the object that this element initialized.

```
resolveCurrentObject
    :: (CurrentObject, Initializer)
    -> (CurrentObject, CInit)
    -> EnvMonad s (CurrentObject, Initializer)
resolveCurrentObject (obj0, prior) (obj1, cinitial) = case obj1 `mplus` obj0 of
    Nothing -> return (Nothing, prior)
    Just obj -> do
```

If the initializer provided is another initializer list, then the initializer has to be for the current object. If it is just an initializer expression, however, we translate the expression to find out what type it has, and find the corresponding sub-object to initialize using `nestedObject`.

```
        (obj', initial) <- case cinitial of
            CInitList list' _ -> do
                initial <- translateInitList (designatorType obj) list'
                return (obj, initial)
            CInitExpr expr _ -> do
                expr' <- interpretExpr True expr
                case nestedObject (resultType expr') obj of
                    Nothing -> badSource cinitial "type in initializer"
                    Just obj' -> do
                        let s = castTo (designatorType obj') expr'
                        return (obj', scalar s)
```

Now that we've settled on the right current object and constructed an intermediate `Initializer` for it, we need to wrap the latter in a minimal aggregate initializer for each designator in the former.

```
        let indices = unfoldr (\o -> case o of
                                Base{} -> Nothing
                                From _ j _ p -> Just (j,p)) obj'
        let initializer = foldl (\a j -> Initializer Nothing (IntMap.singleton j a)) initial indices

        return (nextObject obj', prior `mappend` initializer)
```

Finally, we can implement the full `interpretInitializer` function we declared near the beginning of this section.

Inside an initializer list, we used `nestedObject` above to search for the right sub-object to apply an initializer expression to. But here, outside an initializer list, C99 section 6.7.8 paragraph 13 seems to disallow anything but immediately compatible types; and GCC, Clang, and ICC all reject sub-object typed scalar initializers for structs.

```
interpretInitializer ty initial = do
    initial' <- case initial of
        CInitExpr expr _ -> do
            expr' <- interpretExpr True expr
            if resultType expr' `compatibleInitializer` ty
                then pure $ scalar (castTo ty expr')
                else badSource initial "initializer for incompatible type"
        CInitList list _ -> translateInitList ty list

    zeroed <- zeroInitialize initial' ty
    helper ty zeroed

    where
```

The simplest type of initialization in C is zero-initialization. It initializes in a way that the underlying memory of the target is just zeroed out.

```
    zeroInitialize i@(Initializer Nothing initials) origTy = completeType origTy >>= \ t -> case t of
        IsBool{} -> return $ scalar (Rust.Lit (Rust.LitBool False))
        IsVoid{} -> badSource initial "initializer for void"
        IsInt{} -> return $ scalar (Rust.Lit (Rust.LitInt 0 Rust.DecRepr (toRustType t)))
        IsFloat{} -> return $ scalar (Rust.Lit (Rust.LitFloat "0" (toRustType t)))
        IsPtr{} -> return $ scalar (Rust.Cast 0 (toRustType t))
        IsArray _ size _ | IntMap.size initials == size -> return i
        IsArray _ size elTy -> do
            elInit <- zeroInitialize (Initializer Nothing IntMap.empty) elTy
            el <- helper elTy elInit
            return (Initializer (Just (Rust.RepeatArray el (fromIntegral size))) initials)
        IsFunc{} -> return $ scalar (Rust.Cast 0 (toRustType t))
        IsStruct _ fields -> do
            let fields' = IntMap.fromDistinctAscList $ zip [0..] $ map snd fields
            let missing = fields' `IntMap.difference` initials
            zeros <- mapM (zeroInitialize (Initializer Nothing IntMap.empty)) missing
            return (Initializer Nothing (IntMap.union initials zeros))
        IsEnum{} -> unimplemented initial
        IsIncomplete _ -> badSource initial "initialization of incomplete type"
    zeroInitialize i _ = return i

    helper _ (Initializer (Just expr) initials) | IntMap.null initials = return expr
    helper (IsArray _ _ el) (Initializer expr initials) = case expr of
        Nothing -> Rust.ArrayExpr <$> mapM (helper el) (IntMap.elems initials)
        Just _ -> unimplemented initial
    helper strTy@(IsStruct str fields) (Initializer expr initials) =
        Rust.StructExpr str <$> fields' <*> pure expr
        where
        fields' = forM (IntMap.toList initials) $ \ (idx, value) ->
            case drop idx fields of
            (field, ty') : _ -> do
                value' <- helper ty' value
                return (field, value')
            [] -> noTranslation initial ("internal error: " ++ show strTy ++ " doesn't have enough fiel
    helper _ _ = badSource initial "initializer"
```

Function definitions

A C function definition translates to a single Rust item.

```
interpretFunction :: CFunDef -> EnvMonad s ()
interpretFunction (CFunDef specs declr@(CDeclr mident _ _ _ _) argtypes body _) = do
```

Determine whether the function should be visible outside this module based on whether it is declared `static`.

```
    (storage, baseTy) <- baseTypeOf specs
    (attrs, vis) <- case storage of
        Nothing -> return ([Rust.Attribute "no_mangle"], Rust.Public)
        Just (CStatic _) -> return ([], Rust.Private)
        Just s -> badSource s "storage class specifier for function"
```

Note that `const` is legal but meaningless on the return type of a function in C. We just ignore whether it was present; there's no place we can put a `mut` keyword in the generated Rust.

```
    let go name funTy = do
```

Definitions of variadic functions are not allowed because Rust does not support them.

```
            (retTy, args) <- case funTy of
                IsFunc _ _ True -> unimplemented declr
                IsFunc retTy args False -> return (retTy, args)
                _ -> badSource declr "function definition"
```

Translating `main` requires special care; see `wrapMain` below.

```
            when (name == "_c_main") (wrapMain declr name (map snd args))
```

Open a new scope for the body of this function, while making the return type available so we can correctly translate `return` statements.

```
            let setRetTy flow = flow
                    { functionReturnType = Just retTy
                    , functionName = Just name
                    }
            f' <- mapExceptT (local setRetTy) $ scope $ do
```

Add each formal parameter into the new environment, as a symbol.

> **XXX**: We currently require that every parameter have a name, but I'm not sure that C actually requires that. If it doesn't, we should create dummy names for any anonymous parameters because Rust requires everything be named, but we should not add the dummy names to the environment because those parameters were not accessible in the original program.

```
                formals <- sequence
                    [ case arg of
                        Just (mut, argident) -> do
                            argname <- addSymbolIdent argident (mut, ty)
                            return (mut, Rust.VarName argname, toRustType ty)
                        Nothing -> badSource declr "anonymous parameter"
                    | (arg, ty) <- args
                    ]
```

If control falls of the end of the function we return `0` in the main function (C99 section 5.1.2.2.3, "Program termination".). For other functions we insert a `return;` statement to catch problems with `void` functions which miss a `return;` (for value returning functions this return statement should be deleted due to being unreachable).

```
            let returnValue = if name == "_c_main" then Just 0 else Nothing
                returnStatement = Rust.Stmt (Rust.Return returnValue)
```

Interpret the body of the function.

```
            body' <- cfgToRust declr (interpretStatement body (return ([returnStatement], Unreachabl
```

The body's Haskell type is `CStatement`, but language-c guarantees that it is specifically a compound statement (the `CCompound` constructor). Rather than relying on that, we allow it to be any kind of statement and use `statementsToBlock` to coerce the resulting Rust statement into a Rust block.

Since C doesn't have Rust's syntax allowing the last expression to be the result of the function, this generated block never has a final expression.

```
            return (Rust.Item attrs vis
                (Rust.Function [Rust.UnsafeFn, Rust.ExternABI Nothing] name formals (toRustRetType
                    (statementsToBlock body')))

        emitItems [f']
```

Add this function to the globals before evaluating its body so recursive calls work. (Note that function definitions can't be anonymous.)

```
    ident <- case mident of
        Nothing -> badSource declr "anonymous function definition"
        Just ident -> return ident

    let name = applyRenames ident
    let funTy itype = typeToResult itype (Rust.Path (Rust.PathSegments [name]))
    deferred <- fmap (fmap funTy) (derivedDeferredTypeOf baseTy declr argtypes)
    alreadyUsed <- lift $ gets (usedForwardRefs . globalState)
    case vis of
        Rust.Private | ident `Set.notMember` alreadyUsed -> do
            action <- runOnce $ do
                ty <- deferred
                go name (resultType ty)
                return ty
            addSymbolIdentAction ident action
        _ -> do
            ty <- deferred
            addSymbolIdentAction ident $ return ty
            go name (resultType ty)
```

Special-case translation for `main`

In C, `main` should be a function with one of these types:

```
int main(void);
int main(int argc, char *argv[]);
int main(int argc, char *argv[], char *envp[]);
```

In Rust, `main` must be declared like this (though the return type should usually be left off as it's implied):

```
fn main() -> () { }
```

So when we encounter a C function named `main`, we can't translate it as-is or Rust will reject it. Instead we rename it (see `applyRenames` above) and emit a wrapper function that gets the command line arguments and environment and passes them to the renamed `main`.

```
wrapMain :: CDeclr -> String -> [CType] -> EnvMonad s ()
wrapMain declr realName argTypes = do
```

We decide what the wrapper should do based on what argument types the C `main` expects. The code for different cases is divided into two parts: some setup statements which `let`-bind some variables; and a list of expressions to be passed as arguments to the real `main` function, which presumably use those `let`-bound variables.

```
    (setup, args) <- wrapArgv argTypes
```

The real `main` will have been translated as an `unsafe fn`, like every function we translate, so we need to wrap the call to it in an `unsafe` block. And we need to pass the exit status code that it returns to Rust's `std::process::exit` function, because Rust programs don't return exit status from `main`.

```
    let ret = Rust.VarName "ret"
    emitItems [Rust.Item [] Rust.Private (
        Rust.Function [] "main" [] (Rust.TypeName "()") (statementsToBlock (
            setup ++
            [ bind Rust.Immutable ret $
                Rust.UnsafeExpr $ Rust.Block [] $ Just $
                call realName args ] ++
            exprToStatements (call "::std::process::exit" [Rust.Var ret])
        )))]
    where
```

Writing AST constructors by hand is tedious. Here are some helper functions that allow us to write shorter code. Each one is specialized for this function's needs.

- `bind` creates a `let`-binding with an inferred type and an initial value.

  ```
  bind mut var val = Rust.Let mut var Nothing (Just val)
  ```

- `call` can only call statically-known functions, not function pointers.

  ```
  call fn args = Rust.Call (Rust.Var (Rust.VarName fn)) args
  ```

- `chain` produces method calls, but takes the object as its last argument, so it reads in reverse.

  ```
  chain method args obj = Rust.MethodCall obj (Rust.VarName method) args
  ```

  `a().b().c()` is written as:

  ```
  chain "c" [] $ chain "b" [] $ call "a" []
  ```

Now let's examine the argument types. If `main` is declared `(void)`, then we don't need to pass it any arguments or do any setup.

```
    wrapArgv [] = return ([], [])
```

But if it's declared `(int, char *argv[])`, then we need to call `std::env::args_os()` and convert the argument strings to C-style strings.

```
    wrapArgv (argcType@(IsInt Signed (BitWidth 32))
            : IsPtr Rust.Mutable (IsPtr Rust.Mutable ty)
            : rest) | ty == charType = do
        (envSetup, envArgs) <- wrapEnvp rest
        return (setup ++ envSetup, args ++ envArgs)
        where
        argv_storage = Rust.VarName "argv_storage"
        argv = Rust.VarName "argv"
        str = Rust.VarName "str"
```

```
        vec = Rust.VarName "vec"
        setup =
```

Convert each argument string to a vector of bytes, append a terminating NUL character, and save a reference to the vector so it is not deallocated until after the real `main` returns.

Converting an `OsString` to a `Vec<u8>` is only allowed if we bring the Unix-specific `OsStringExt` trait into scope.

```
            [ Rust.StmtItem [] (Rust.Use "::std::os::unix::ffi::OsStringExt")
            , bind Rust.Mutable argv_storage $
                chain "collect::<Vec<_>>" [] $
                chain "map" [
                    Rust.Lambda [str] (Rust.BlockExpr (Rust.Block
                        ( bind Rust.Mutable vec (chain "into_vec" [] (Rust.Var str))
                        : exprToStatements (chain "push" [
                                Rust.Lit (Rust.LitByteChar '\NUL')
                            ] (Rust.Var vec))
                        ) (Just (Rust.Var vec))))
                ] $
                call "::std::env::args_os" []
```

In C, `argv` is required to be a modifiable NULL-terminated array of modifiable strings. We have modifiable strings as array-backed vectors in `argv_storage`, so now we need to construct an array-backed vector of pointers to those strings.

Once we save these pointers, we *must not* do anything that might change the length of the vectors we got them from, as that could re-allocate the backing array and invalidate our pointer.

`Iterator.chain` produces a new iterator which yields all the elements of the original iterator, followed by anything in the second iterable. We just want to add one thing at the end of the array—namely, a NULL pointer—and conveniently, the `Option` type is iterable.

```
            , bind Rust.Mutable argv $
                chain "collect::<Vec<_>>" [] $
                chain "chain" [call "Some" [call "::std::ptr::null_mut" []]] $
                chain "map" [
                    Rust.Lambda [vec] (chain "as_mut_ptr" [] (Rust.Var vec))
                ] $
                chain "iter_mut" [] $
                Rust.Var argv_storage
            ]
```

In C, `argc` is the number of elements in the `argv` array, but not counting the terminating NULL pointer. So we pass the number of items that are in `argv_storage` instead.

For `main`'s second argument, we pass a pointer to the array that backs `argv`. After this we *must not* do anything that might change the length of `argv`, and we must ensure that `argv` is not deallocated until after the real `main` returns.

```
        args =
            [ Rust.Cast (chain "len" [] (Rust.Var argv_storage)) (toRustType argcType)
            , chain "as_mut_ptr" [] (Rust.Var argv)
            ]
```

We can't translate a program containing a `main` function unless we know how to construct all of the arguments it expects.

```
    wrapArgv _ = unimplemented declr
```

After handling `argc` and `argv`, we might see a third argument, conventionally called `envp`. On POSIX systems, we can just pass the pointer stored in a global variable named `environ` for this argument.

```
    wrapEnvp [] = return ([], [])
    wrapEnvp [arg@(IsPtr Rust.Mutable (IsPtr Rust.Mutable ty))] | ty == charType
        = return (setup, args)
        where
        environ = Rust.VarName "environ"
        setup =
            [ Rust.StmtItem [] $
                Rust.Extern [Rust.ExternStatic Rust.Immutable environ (toRustType arg)]
            ]
        args = [Rust.Var environ]
    wrapEnvp _ = unimplemented declr
```

Statements

In order to interpret `break` and `continue` statements, we need a bit of extra context, which we'll discuss as we go. Let's wrap that context up in a simple data type.

```
data OuterLabels = OuterLabels
    { onBreak :: Maybe Label
    , onContinue :: Maybe Label
    , switchExpression :: Maybe CExpr
    }

newtype SwitchCases = SwitchCases (IntMap.IntMap (Maybe Result))

instance Monoid SwitchCases where
    mempty = SwitchCases IntMap.empty
    SwitchCases a `mappend` SwitchCases b = SwitchCases $
        IntMap.unionWith (liftM2 eitherCase) a b
        where
        eitherCase lhs rhs = Result
            { resultType = IsBool
            , resultMutable = Rust.Immutable
            , result = Rust.LOr (toBool lhs) (toBool rhs)
            }
```

Inside a function, we find C statements. Unlike C syntax which is oriented around statements, Rust syntax is mostly just expressions. Nonetheless, by having `interpretStatement` return a list of Rust statements (instead of just a Rust expression), we end up being able to get rid of superfluous curly braces.

```
type CSourceBuildCFGT s = BuildCFGT (RWST OuterLabels SwitchCases (Map.Map Ident Label) (EnvMonad s)) [

interpretStatement :: CStat -> CSourceBuildCFGT s ([Rust.Stmt], Terminator Result) -> CSourceBuildCFGT s

interpretStatement (CLabel ident body _ _) next = do
    label <- gotoLabel ident
    (rest, end) <- interpretStatement body next
    addBlock label rest end
    return ([], Branch label)

interpretStatement stmt@(CCase expr body node) next = do
    selector <- getSwitchExpression stmt
    let condition = CBinary CEqOp selector expr node
```

```
    addSwitchCase (Just condition) body next
interpretStatement stmt@(CCases lower upper body node) next = do
    selector <- getSwitchExpression stmt
    let condition = CBinary CLndOp
            (CBinary CGeqOp selector lower node)
            (CBinary CLeqOp selector upper node)
            node
    addSwitchCase (Just condition) body next
interpretStatement (CDefault body _) next =
    addSwitchCase Nothing body next
```

A C statement might be as simple as a "statement expression", which amounts to a C expression followed by a semicolon. In that case we can translate the statement by just translating the expression.

If the statement is empty, as in just a semicolon, we don't need to produce any statements.

```
interpretStatement (CExpr Nothing _) next = next
```

Otherwise, the first argument to `interpretExpr` indicates whether the expression appears in a context where its result matters. In a statement expression, the result is discarded, so we pass `False`.

```
interpretStatement (CExpr (Just expr) _) next = do
    expr' <- lift $ lift $ interpretExpr False expr
    (rest, end) <- next
    return (resultToStatements expr' ++ rest, end)
```

We could have a "compound statement", also known as a "block". A compound statement contains a sequence of zero or more statements or declarations; see `interpretBlockItem` below for details.

In the case of an empty compound statement, we make an exception of our usual rule of not simplifying the generated Rust and simply ignore the "statement".

```
interpretStatement (CCompound [] items _) next = mapBuildCFGT (mapRWST scope) $ do
    foldr interpretBlockItem next items
```

This statement could be an `if` statement, with its conditional expression and "then" branch, and maybe an "else" branch.

The conditional expression is numeric in C, but must have type `bool` in Rust. We use the `toBool` helper defined below to convert the expression if necessary.

In C, the "then" and "else" branches are each statements (which may be compound statements, if they're surrounded by curly braces), but in Rust they must each be blocks (so they're always surrounded by curly braces).

Here we use `statementsToBlock` to coerce both branches into blocks, so we don't care whether the original program used a compound statement in that branch or not.

```
interpretStatement (CIf c t mf _) next = do
    c' <- lift $ lift $ interpretExpr True c
    after <- newLabel

    falseLabel <- case mf of
        Nothing -> return after
        Just f -> do
            (falseEntry, falseTerm) <- interpretStatement f (return ([], Branch after))
            falseLabel <- newLabel
            addBlock falseLabel falseEntry falseTerm
            return falseLabel
```

```
    (trueEntry, trueTerm) <- interpretStatement t (return ([], Branch after))
    trueLabel <- newLabel
    addBlock trueLabel trueEntry trueTerm

    (rest, end) <- next
    addBlock after rest end

    return ([], CondBranch c' trueLabel falseLabel)
interpretStatement stmt@(CSwitch expr body node) next = do
    (bindings, expr') <- case expr of
        CVar{} -> return ([], expr)
        _ -> lift $ lift $ do
            ident <- fmap internalIdent (uniqueName "switch")
            rhs <- interpretExpr True expr
            var <- addSymbolIdent ident (Rust.Immutable, resultType rhs)
            return
                ( [Rust.Let Rust.Immutable (Rust.VarName var) Nothing (Just (result rhs))]
                , CVar ident node
                )

    after <- newLabel
    (_, SwitchCases cases) <- getSwitchCases expr' $ setBreak after $
        interpretStatement body (return ([], Branch after))

    let isDefault (Just condition) = Left condition
        isDefault Nothing = Right ()
    let (conditions, defaults) = IntMap.mapEither isDefault cases
    defaultCase <- case IntMap.keys defaults of
        [] -> return after
        [defaultCase] -> return defaultCase
        _ -> lift $ lift $ badSource stmt "duplicate default cases"

    entry <- foldrM conditionBlock defaultCase (IntMap.toList conditions)

    (rest, end) <- next
    addBlock after rest end

    return (bindings, Branch entry)
    where
    conditionBlock (target, condition) defaultCase = do
        label <- newLabel
        addBlock label [] (CondBranch condition target defaultCase)
        return label
```

`while` loops are easy to translate from C to Rust. They have identical semantics in both languages, aside from differences analagous to the `if` statement, where the loop condition must have type `bool` and the loop body must be a block.

```
interpretStatement (CWhile c body doWhile _) next = do
    c' <- lift $ lift $ interpretExpr True c
    after <- newLabel

    headerLabel <- newLabel
    (bodyEntry, bodyTerm) <- setBreak after $ setContinue headerLabel $
```

```
        interpretStatement body (return ([], Branch headerLabel))

    bodyLabel <- newLabel
    addBlock bodyLabel bodyEntry bodyTerm

    addBlock headerLabel [] $ case toBool c' of
        Rust.Lit (Rust.LitBool cont) | cont /= doWhile ->
            Branch (if cont then bodyLabel else after)
        _ -> CondBranch c' bodyLabel after

    (rest, end) <- next
    addBlock after rest end

    return ([], Branch (if doWhile then bodyLabel else headerLabel))
```

C's `for` loops can be tricky to translate to Rust, which doesn't have anything quite like them.

Oh, the initialization/declaration part of the loop is easy enough. Open a new scope, insert any assignments and `let`-bindings at the beginning of it, and we're good to go. Note that we need to wrap everything in a block if we have `let`-bindings, but not otherwise.

```
interpretStatement (CFor initial mcond mincr body _) next = do
    after <- newLabel

    ret <- mapBuildCFGT (mapRWST scope) $ do
        prefix <- case initial of
            Left Nothing -> return []
            Left (Just expr) -> do
                expr' <- lift $ lift $ interpretExpr False expr
                return (resultToStatements expr')
            Right decls -> lift $ lift $ interpretDeclarations makeLetBinding decls

        headerLabel <- newLabel
        incrLabel <- case mincr of
            Nothing -> return headerLabel
            Just incr -> do
                incr' <- lift $ lift $ interpretExpr False incr
                incrLabel <- newLabel
                addBlock incrLabel (resultToStatements incr') (Branch headerLabel)
                return incrLabel

        (bodyEntry, bodyTerm) <- setBreak after $ setContinue incrLabel $
            interpretStatement body (return ([], Branch incrLabel))

        bodyLabel <- newLabel
        addBlock bodyLabel bodyEntry bodyTerm

        cond <- case mcond of
            Just cond -> do
                cond' <- lift $ lift $ interpretExpr True cond
                return (CondBranch cond' bodyLabel after)
            Nothing -> return (Branch bodyLabel)
        addBlock headerLabel [] cond

        return (prefix, Branch headerLabel)
```

```
    (rest, end) <- next
    addBlock after rest end

    return ret

interpretStatement (CGoto ident _) next = do
    _ <- next
    label <- gotoLabel ident
    return ([], Branch label)
```

`continue` and `break` statements translate to whatever expression we decided on in the surrounding context. For example, `continue` might translate to `break 'continueTo` if our nearest enclosing loop is a `for` loop.

```
interpretStatement stmt@(CCont _) next = do
    _ <- next
    val <- lift (asks onContinue)
    case val of
        Just label -> return ([], Branch label)
        Nothing -> lift $ lift $ badSource stmt "continue outside loop"
interpretStatement stmt@(CBreak _) next = do
    _ <- next
    val <- lift (asks onBreak)
    case val of
        Just label -> return ([], Branch label)
        Nothing -> lift $ lift $ badSource stmt "break outside loop"
```

`return` statements are pretty straightforward—translate the expression we're supposed to return, if there is one, and return the Rust equivalent—except that Rust is more strict about types than C is. So if the return expression has a different type than the function's declared return type, then we need to insert a type-cast to the correct type.

```
interpretStatement stmt@(CReturn expr _) next = do
    _ <- next
    lift $ lift $ do
        val <- lift (asks functionReturnType)
        case val of
            Nothing -> badSource stmt "return statement outside function"
            Just retTy -> do
                expr' <- mapM (fmap (castTo retTy) . interpretExpr True) expr
                return (exprToStatements (Rust.Return expr'), Unreachable)
```

Otherwise, this is a type of statement we haven't implemented a translation for yet.

TODO: Translate more kinds of statements. `:-)`

```
interpretStatement stmt _ = lift $ lift $ unimplemented stmt
```

Inside loops above, we needed to update the translation to use if either `break` or `continue` statements show up. These functions run the provided translation action using updated `break`/`continue` expressions.

```
setBreak :: Label -> CSourceBuildCFGT s a -> CSourceBuildCFGT s a
setBreak label =
    mapBuildCFGT (local (\ flow -> flow { onBreak = Just label }))

setContinue :: Label -> CSourceBuildCFGT s a -> CSourceBuildCFGT s a
setContinue label =
    mapBuildCFGT (local (\ flow -> flow { onContinue = Just label }))
```

```haskell
getSwitchExpression :: CStat -> CSourceBuildCFGT s CExpr
getSwitchExpression stmt = do
    mexpr <- lift $ asks switchExpression
    case mexpr of
        Nothing -> lift $ lift $ badSource stmt "case outside switch"
        Just expr -> return expr

addSwitchCase :: Maybe CExpr -> CStat -> CSourceBuildCFGT s ([Rust.Stmt], Terminator Result) -> CSourceB
addSwitchCase condition body next = do
    condition' <- lift $ lift $ mapM (interpretExpr True) condition
    next' <- interpretStatement body next
    label <- case next' of
        ([], Branch to) -> return to
        (rest, end) -> do
            label <- newLabel
            addBlock label rest end
            return label
    lift $ tell $ SwitchCases $ IntMap.singleton label condition'
    return ([], Branch label)

getSwitchCases :: CExpr -> CSourceBuildCFGT s a -> CSourceBuildCFGT s (a, SwitchCases)
getSwitchCases expr = mapBuildCFGT wrap
    where
    wrap body = do
        ((a, st), cases) <- censor (const mempty)
            $ local (\ flow -> flow { switchExpression = Just expr })
            $ listen body
        return ((a, cases), st)
```

C allows `goto` statements and their corresponding labels to appear in any order. So the first time we encounter the name of a C label, we assign it a CFG label, regardless of whether that's in a labeled statement or a `goto` statement. Then we save the CFG label we chose so we can make sure to use the same one for all future references.

```haskell
gotoLabel :: Ident -> CSourceBuildCFGT s Label
gotoLabel ident = do
    labels <- lift get
    case Map.lookup ident labels of
        Nothing -> do
            label <- newLabel
            lift (put (Map.insert ident label labels))
            return label
        Just label -> return label
```

Once we've built a control-flow graph for a sequence of statements, we need to extract equivalent Rust control flow expressions.

```haskell
cfgToRust :: (Pretty node, Pos node) => node -> CSourceBuildCFGT s ([Rust.Stmt], Terminator Result) -> [
cfgToRust _node build = do
    let builder = buildCFG $ do
            (early, term) <- build
            entry <- newLabel
            addBlock entry early term
            return entry
    (rawCFG, _) <- evalRWST builder (OuterLabels Nothing Nothing Nothing) Map.empty
```

This is not always possible without either introducing new variables that weren't in the original program, or duplicating parts of the code. At the moment we choose to report a translation error in those cases, and for any other control-flow patterns we don't know how to recognize yet.

```
    let cfg = depthFirstOrder (removeEmptyBlocks rawCFG)
    let (hasGoto, structured) = structureCFG mkBreak mkContinue mkLoop mkIf mkGoto mkMatch cfg
    return $ if hasGoto then declCurrent : structured else structured
    where
```

The `CFG` module is agnostic to both source and target languages, so we need to tell it which common patterns apply to Rust and how to construct the appropriate Rust AST.

Order matters in this list. Every block that matches a particular pattern will be transformed before trying the next pattern. It's important that `ifThenElse` come after `while`, because otherwise perfectly good `while` loops may get transformed into an `if` nested inside a `loop`.

```
    loopLabel l = Rust.Lifetime ("loop" ++ show l)
    mkBreak l = exprToStatements (Rust.Break (fmap loopLabel l))
    mkContinue l = exprToStatements (Rust.Continue (fmap loopLabel l))
    mkLoop l b = exprToStatements (Rust.Loop (Just (loopLabel l)) (statementsToBlock b))
    mkIf c t f = exprToStatements (simplifyIf c (statementsToBlock t) (statementsToBlock f))

    currentBlock = Rust.VarName "_currentBlock"
    declCurrent = Rust.Let Rust.Mutable currentBlock Nothing Nothing
    mkGoto l = exprToStatements (Rust.Assign (Rust.Var currentBlock) (Rust.:=) (fromIntegral l))
    mkMatch = flip (foldr go)
        where
        go (l, t) f = exprToStatements (Rust.IfThenElse (Rust.CmpEQ (Rust.Var currentBlock) (fromIntegra
```

To generate code that's as clear as possible, we handle some interesting special cases for `if` statements.

- When both the true and false branches are empty, we don't need an `if`-statement at all. We just need to evaluate the condition for its side effects.

```
        simplifyIf c (Rust.Block [] Nothing) (Rust.Block [] Nothing) =
            result c
```

- When just the true branch is empty, we should compute the opposite of the condition, and swap the branches. Otherwise we'd get an expression like `if ... {} else { ... }`, which just looks silly. This pattern can show up in surprising places, such as translation of some `continue` statements.

```
        simplifyIf c (Rust.Block [] Nothing) f =
            Rust.IfThenElse (toNotBool c) f (Rust.Block [] Nothing)
```

- Otherwise, just construct a regular `if` statement.

```
        simplifyIf c t f = Rust.IfThenElse (toBool c) t f
```

Blocks and scopes

In compound statements (see above), we may encounter local declarations as well as nested statements. `interpretBlockItem` produces a sequence of zero or more Rust statements for each compound block item.

```
interpretBlockItem :: CBlockItem -> CSourceBuildCFGT s ([Rust.Stmt], Terminator Result) -> CSourceBuild
interpretBlockItem (CBlockStmt stmt) next = interpretStatement stmt next
interpretBlockItem (CBlockDecl decl) next = do
    decl' <- lift $ lift (interpretDeclarations makeLetBinding decl)
    (rest, end) <- next
```

```
    return (decl' ++ rest, end)
interpretBlockItem item _ = lift $ lift (unimplemented item)
```

`scope` runs the translation steps in `m`, but then throws away any changes that `m` made to the environment. Any items added to the output are kept, though, as are any global state changes.

```
scope :: EnvMonad s a -> EnvMonad s a
scope m = do
    -- Save the current environment.
    old <- lift get
    a <- m
    -- Restore the environment to its state before running m.
    lift (modify (\ st -> old { globalState = globalState st }))
    return a
```

Smart constructors for blocks and statements

Sometimes we generate a block and then discover that we aren't going to use the final result from the block. In that case, if there is a final expression at the end of the block, we can turn it into a statement and append it to the rest of the statements.

```
blockToStatements :: Rust.Block -> [Rust.Stmt]
blockToStatements (Rust.Block stmts mexpr) = case mexpr of
    Just expr -> stmts ++ exprToStatements expr
    Nothing -> stmts
```

There's no point wrapping a new block around a list of statements if the only statement in the list is, itself, a block.

Use this function instead of the `Rust.Block` constructor when there's no final expression for the block.

```
statementsToBlock :: [Rust.Stmt] -> Rust.Block
statementsToBlock [Rust.Stmt (Rust.BlockExpr stmts)] = stmts
statementsToBlock stmts = Rust.Block stmts Nothing
```

We have to be particularly careful about how we generate `if` expressions when the result is not used. When most expressions are used as statements, it doesn't matter what type the expression has, because the result is ignored. But when a Rust `if`-expression is used as a statement, Rust requires its type to be `()`. We can ensure the type is correct by wrapping the true and false branches in statements rather than placing them as block-final expressions.

Also, assuming no two stack-allocated variables have the same name, using a block expression as a statement is never necessary.

Use this function instead of using the `Rust.Stmt` constructor directly to ensure the necessary invariants are maintained everywhere.

```
exprToStatements :: Rust.Expr -> [Rust.Stmt]
exprToStatements (Rust.IfThenElse c t f) =
    [Rust.Stmt (Rust.IfThenElse c (extractExpr t) (extractExpr f))]
    where
    extractExpr = statementsToBlock . blockToStatements
exprToStatements (Rust.BlockExpr b) = blockToStatements b
exprToStatements e = [Rust.Stmt e]
```

Expressions

Translating expressions works by a bottom-up traversal of the expression tree: we compute the type of a sub-expression, then use that to determine what the surrounding expression means.

We also need to record whether the expression is a legitimate "l-value", which determines whether we can assign to it or take mutable borrows of it. Here we abuse the `Rust.Mutable` type: If it is `Mutable`, then it's a legal l-value, and if it's `Immutable` then it is not.

Finally, of course, we record the Rust AST for the sub-expression so it can be combined into the larger expression that we're building up.

```
data Result = Result
    { resultType :: CType
    , resultMutable :: Rust.Mutable
    , result :: Rust.Expr
    }
```

`resultToStatements` is a convenience wrapper to lift `exprToStatements` to operate on `Results`.

```
resultToStatements :: Result -> [Rust.Stmt]
resultToStatements = exprToStatements . result
```

```
typeToResult :: IntermediateType -> Rust.Expr -> Result
typeToResult itype expr = Result
    { resultType = typeRep itype
    , resultMutable = typeMutable itype
    , result = expr
    }
```

`interpretExpr` translates a `CExpression` into a Rust expression, along with the other metadata in `Result`.

It also takes a boolean parameter, `demand`, indicating whether the caller intends to use the result of this expression. For some C expressions, we have to generate extra Rust code if the result is needed.

```
interpretExpr :: Bool -> CExpr -> EnvMonad s Result
```

C's comma operator evaluates its left-hand expressions for their side effects, throwing away their results, and then evaluates to the result of its right-hand expression.

Rust doesn't have a dedicated operator for this, but it works out the same as Rust's block expressions: `(e1, e2, e3)` can be translated to `{ e1; e2; e3 }`. If the result is not going to be used, we can translate it instead as `{ e1; e2; e3; }`, where the expressions are semicolon-terminated instead of semicolon-separated.

```
interpretExpr demand (CComma exprs _) = do
    let (effects, mfinal) = if demand then (init exprs, Just (last exprs)) else (exprs, Nothing)
    effects' <- mapM (fmap resultToStatements . interpretExpr False) effects
    mfinal' <- mapM (interpretExpr True) mfinal
    return Result
        { resultType = maybe IsVoid resultType mfinal'
        , resultMutable = maybe Rust.Immutable resultMutable mfinal'
        , result = Rust.BlockExpr (Rust.Block (concat effects') (fmap result mfinal'))
        }
```

C's assignment operator is complicated enough that, after translating the left-hand and right-hand sides recursively, we just delegate to the `compound` helper function defined below.

```
interpretExpr demand expr@(CAssign op lhs rhs _) = do
    lhs' <- interpretExpr True lhs
    rhs' <- interpretExpr True rhs
    compound expr False demand op lhs' rhs'
```

C's ternary conditional operator (`c ? t : f`) translates fairly directly to Rust's `if/else` expression.

```
interpretExpr demand expr@(CCond c (Just t) f _) = do
    c' <- fmap toBool (interpretExpr True c)
```

```
    t' <- interpretExpr demand t
    f' <- interpretExpr demand f
    if demand
        then promotePtr expr (mkIf c') t' f'
        else return Result
            { resultType = IsVoid
            , resultMutable = Rust.Immutable
            , result = mkIf c' (result t') (result f')
            }
    where
    mkIf c' t' f' = Rust.IfThenElse c' (Rust.Block [] (Just t')) (Rust.Block [] (Just f'))
```

C's binary operators are complicated enough that, after translating the left-hand and right-hand sides recursively, we just delegate to the `binop` helper function defined below.

```
interpretExpr _ expr@(CBinary op lhs rhs _) = do
    lhs' <- interpretExpr True lhs
    rhs' <- interpretExpr True rhs
    binop expr op lhs' rhs'
```

C's cast operator corresponds exactly to Rust's cast operator. The syntax is quite different but the semantics are the same. Note that the result of a cast is never a legal l-value.

In C, casting the result of an expression to `void` is an idiom for explicitly ignoring the result, meaning the expression is only intended to be evaluated for its side effects. It's commonly used in older code to suppress over-zealous compiler warnings, and continues to be used in certain special cases such as glibc's implementation of the assert macro. A naive translation of this idiom produces invalid Rust, so we special-case it to simply evaluate the subexpression without demanding its result.

```
interpretExpr _ (CCast decl expr _) = do
    (_mut, ty) <- typeName decl
    expr' <- interpretExpr (ty /= IsVoid) expr
    return Result
        { resultType = ty
        , resultMutable = Rust.Immutable
        , result = (if ty == IsVoid then result else castTo ty) expr'
        }
```

We de-sugar the pre/post-increment/decrement operators into compound assignment operators and call the common assignment helper.

```
interpretExpr demand node@(CUnary op expr _) = case op of
    CPreIncOp -> incdec False CAddAssOp
    CPreDecOp -> incdec False CSubAssOp
    CPostIncOp -> incdec True CAddAssOp
    CPostDecOp -> incdec True CSubAssOp
```

We translate C's address-of operator (unary prefix `&`) to either a mutable or immutable borrow operator, followed by a cast to the corresponding Rust raw-pointer type.

We re-use the l-value flag to decide whether this is a mutable or immutable borrow. If the sub-expression is a valid l-value, then it's always safe to take a mutable pointer to it. Otherwise, it may be OK to take an immutable pointer, or it may be nonsense.

This implementation does not check for the nonsense case. If you get weird results, make sure your input C compiles without errors using a real C compiler.

```
    CAdrOp -> do
        expr' <- interpretExpr True expr
```

```
        let ty' = IsPtr (resultMutable expr') (resultType expr')
        return Result
            { resultType = ty'
            , resultMutable = Rust.Immutable
            , result = Rust.Cast (Rust.Borrow (resultMutable expr') (result expr')) (toRustType ty')
            }
```

C's indirection or dereferencing operator (unary prefix `*`) translates to the same operator in Rust. Whether the result is an l-value depends on whether the pointer was to a mutable value.

```
    CIndOp -> do
        expr' <- interpretExpr True expr
        case resultType expr' of
            IsPtr mut' ty' -> return Result
                { resultType = ty'
                , resultMutable = mut'
                , result = Rust.Deref (result expr')
                }
            IsFunc{} -> return expr'
            _ -> badSource node "dereference of non-pointer"
```

Ah, unary plus. My favorite. Rust does not have a unary plus operator, because this operator is almost entirely useless. It's the arithmetic opposite of unary minus: where that operator negates its argument, this operator returns its argument unchanged. . . . except that the "integer promotion" rules apply, so this operator may perform an implicit cast.

```
    CPlusOp -> do
        expr' <- interpretExpr demand expr
        let ty' = intPromote (resultType expr')
        return Result
            { resultType = ty'
            , resultMutable = Rust.Immutable
            , result = castTo ty' expr'
            }
```

C's unary minus operator translates to Rust's unary minus operator, after applying the integer promotions. . . except that if it's negating an unsigned type, then C defines integer overflow to wrap.

```
    CMinOp -> fmap wrapping $ simple Rust.Neg
```

Bitwise complement translates to Rust's `!` operator.

```
    CCompOp -> simple Rust.Not
```

Logical "not" also translates to Rust's `!` operator, but we have to force the operand to `bool` type first. We can avoid introducing silly extra "not" operators by creating a special-case `toNotBool` variant of `toBool` that returns the opposite value.

```
    CNegOp -> do
        expr' <- interpretExpr True expr
        return Result
            { resultType = IsBool
            , resultMutable = Rust.Immutable
            , result = toNotBool expr'
            }
```

Common helpers for the unary operators:

```
    where
    incdec returnOld assignop = do
```

```
        expr' <- interpretExpr True expr
        compound node returnOld demand assignop expr' Result
            { resultType = IsInt Signed (BitWidth 32)
            , resultMutable = Rust.Immutable
            , result = 1
            }
    simple f = do
        expr' <- interpretExpr True expr
        let ty' = intPromote (resultType expr')
        return Result
            { resultType = ty'
            , resultMutable = Rust.Immutable
            , result = f (castTo ty' expr')
            }
```

C's `sizeof` and `alignof` operators translate to calls to Rust's `size_of` and `align_of` functions in `std::mem`, respectively. Note that for an expression, we don't evaluate it but only determine its type.

> **TODO**: Variable length arrays have to be evaluated for `sizeof` (including possible side effects), using `std::mem::size_of_val` in Rust. What should `alignof` do for variable length arrays?

```
interpretExpr _ (CSizeofExpr e _) = do
    e' <- interpretExpr True e
    return (rustSizeOfType (toRustType (resultType e')))
interpretExpr _ (CSizeofType decl _) = do
    (_mut, ty) <- typeName decl
    return (rustSizeOfType (toRustType ty))
interpretExpr _ (CAlignofExpr e _) = do
    e' <- interpretExpr True e
    return (rustAlignOfType (toRustType (resultType e')))
interpretExpr _ (CAlignofType decl _) = do
    (_mut, ty) <- typeName decl
    return (rustAlignOfType (toRustType ty))
```

C's array index or array subscript operator, `e1[e2]`, works just like adding the two expressions and dereferencing the result. Note that in the C expression `e1[e2]`, although we usually expect `e1` to be a pointer and `e2` to be an integer, C permits them to be the other way around. Fortunately, C's pointer addition is also commutative so calling our `binop` helper here does the right thing.

```
interpretExpr _ expr@(CIndex lhs rhs _) = do
    lhs' <- interpretExpr True lhs
    rhs' <- interpretExpr True rhs
    case (resultType lhs', resultType rhs') of
        (IsArray mut _ el, _) -> return (subscript mut el (result lhs') rhs')
        (_, IsArray mut _ el) -> return (subscript mut el (result rhs') lhs')
        _ -> do
            ptr <- binop expr CAddOp lhs' rhs'
            case resultType ptr of
                IsPtr mut ty -> return Result
                    { resultType = ty
                    , resultMutable = mut
                    , result = Rust.Deref (result ptr)
                    }
                _ -> badSource expr "array subscript of non-pointer"
    where
    subscript mut el arr idx = Result
```

```
        { resultType = el
        , resultMutable = mut
        , result = Rust.Index arr (castTo (IsInt Unsigned WordWidth) idx)
        }
```

Function calls first translate the expression which identifies which function to call, and any argument expressions.

```
interpretExpr _ expr@(CCall func args _) = do
    func' <- interpretExpr True func
    case resultType func' of
        IsFunc retTy argTys variadic -> do
            args' <- castArgs variadic (map snd argTys) args
            return Result
                { resultType = retTy
                , resultMutable = Rust.Immutable
                , result = Rust.Call (result func') args'
                }
        _ -> badSource expr "function call to non-function"
    where
```

For each actual parameter, the expression given for that parameter is translated and then cast (if necessary) to the type of the corresponding formal parameter.

If we're calling a variadic function, then there can be any number of arguments after the ones explicitly given in the function's type, and those extra arguments can be of any type.

Otherwise, there must be exactly as many arguments as the function's type specified, or it's a syntax error.

```
    castArgs _ [] [] = return []
    castArgs variadic (ty : tys) (arg : rest) = do
        arg' <- interpretExpr True arg
        args' <- castArgs variadic tys rest
        return (castTo ty arg' : args')
    castArgs True [] rest = mapM (fmap promoteArg . interpretExpr True) rest
    castArgs False [] _ = badSource expr "arguments (too many)"
    castArgs _ _ [] = badSource expr "arguments (too few)"
```

In C, the "default argument promotions" (C99 6.5.2.2 paragraphs 6-7) are applied to any variable parameters after the last declared parameter. They would also be applied to arguments passed to a function declared with an empty argument list (`foo()`) or implicitly declared due to a lack of a prototype, except we don't allow either of those cases.

```
    promoteArg :: Result -> Rust.Expr
    promoteArg r = case resultType r of
        IsFloat _ -> castTo (IsFloat 64) r
        IsArray mut _ el -> castTo (IsPtr mut el) r
        ty -> castTo (intPromote ty) r
```

Structure member access has two forms in C (`.` and `->`), which only differ in whether the left-hand side is dereferenced first. We desugar `p->f` into `(*p).f`, then translate `o.f` into the same thing in Rust. The result's type is the type of the named field within the struct, and the result is a legal l-value if and only if the larger object was a legal l-value.

```
interpretExpr _ expr@(CMember obj ident deref node) = do
    obj' <- interpretExpr True $ if deref then CUnary CIndOp obj node else obj
    objTy <- completeType (resultType obj')
    fields <- case objTy of
        IsStruct _ fields -> return fields
```

```
        _ -> badSource expr "member access of non-struct"
    let field = applyRenames ident
    ty <- case lookup field fields of
        Just ty -> return ty
        Nothing -> badSource expr "request for non-existent field"
    return Result
        { resultType = ty
        , resultMutable = resultMutable obj'
        , result = Rust.Member (result obj') (Rust.VarName field)
        }
```

We preserve variable names during translation, so a C variable reference translates to an identical Rust variable reference. However, we do have to look up the variable in the environment in order to report its type and whether taking its address should produce a mutable or immutable pointer.

```
interpretExpr _ expr@(CVar ident _) = do
    sym <- getSymbolIdent ident
    maybe (badSource expr "undefined variable") return sym
```

C literals (integer, floating-point, character, and string) translate to similar tokens in Rust.

> **TODO**: Figure out what to do about floating-point hex literals, which as far as I can tell Rust doesn't support (yet?).

> **TODO**: Translate wide character and string literals.

```
interpretExpr _ expr@(CConst c) = case c of
```

In C, the type of an integer literal depends on which types its value will fit in, constrained by its suffixes (`U` or `L`) and whether its representation is decimal or another base. See C99 6.4.4.1 paragraph 5 and its subsequent table.

For the purposes of deciding whether a literal will fit within the bounds of a type, we choose to pretend that `long` is 32 bits, but the Rust type we give it is `isize`. If a constant does not fit in 32 bits, we always give it type `i64`.

```
    CIntConst (CInteger v repr flags) _ ->
        let allow_signed = not (testFlag FlagUnsigned flags)
            allow_unsigned = not allow_signed || repr /= DecRepr
            widths =
                [ (32 :: Int,
                    if any (`testFlag` flags) [FlagLongLong, FlagLong]
                    then WordWidth else BitWidth 32)
                , (64, BitWidth 64)
                ]
            allowed_types =
                [ IsInt s w
                | (bits, w) <- widths
                , (True, s) <- [(allow_signed, Signed), (allow_unsigned, Unsigned)]
                , v < 2 ^ (bits - if s == Signed then 1 else 0)
                ]
            repr' = case repr of
                DecRepr -> Rust.DecRepr
                OctalRepr -> Rust.OctalRepr
                HexRepr -> Rust.HexRepr
        in case allowed_types of
        [] -> badSource expr "integer (too big)"
        ty : _ -> return (literalNumber ty (Rust.LitInt v repr'))
```

```
    CFloatConst (CFloat str) _ -> case span (`notElem` "fF") str of
        (v, "") -> return (literalNumber (IsFloat 64) (Rust.LitFloat v))
        (v, [_]) -> return (literalNumber (IsFloat 32) (Rust.LitFloat v))
        _ -> badSource expr "float"
```

Rust's `char` type is for Unicode characters, so it's quite different from C's 8-bit `char` type. As a result, C character literals and strings need to be translated to Rust's "byte literals" (`b'?'`, `b"..."`) rather than the more familiar character and string literal syntax.

```
    CCharConst (CChar ch False) _ -> return Result
        { resultType = charType
        , resultMutable = Rust.Immutable
        , result = Rust.Lit (Rust.LitByteChar ch)
        }
```

In C, string literals get a terminating NUL character added at the end. Rust byte string literals do not, so we need to append one in the translation.

In Rust, the type of a byte string literal of length `n` is `&'static [u8; n]`. We need a raw pointer instead to match C's semantics. Conveniently, Rust slices have an `.as_ptr()` method which extracts a raw pointer for us. Note that since string literals have `'static` lifetime, the resulting raw pointer is always safe to use.

```
    CStrConst (CString str False) _ -> return Result
        { resultType = IsArray Rust.Immutable (length str + 1) charType
        , resultMutable = Rust.Immutable
        , result = Rust.Deref (Rust.Lit (Rust.LitByteStr (str ++ "\NUL")))
        }
    _ -> unimplemented expr
    where
```

A number like `42` gives no information about which type it should be. In Rust, we can suffix a numeric literal with its type name, so `42i8` has type `i8`, while `42f32` has type `f32`. `literalNumber` abuses the fact that our Rust AST representation of a type is just a string to get the right suffix for these literals.

Rust allows unsuffixed numeric literals, in which case it will try to infer from the surrounding context what type the number should have. However, we don't want to rely on Rust's inference rules here because we need to match C's rules instead.

```
    literalNumber ty lit = Result
        { resultType = ty
        , resultMutable = Rust.Immutable
        , result = Rust.Lit (lit (toRustType ty))
        }
```

C99 compound literals are really not much different than an initializer list with the type of the thing we are initializing included.

```
interpretExpr _ (CCompoundLit decl initials info) = do
    (mut, ty) <- typeName decl
    final <- interpretInitializer ty (CInitList initials info)
    return Result
        { resultType = ty
        , resultMutable = mut
        , result = final
        }
```

GCC's "statement expression" extension translates pretty directly to Rust block expressions.

```
interpretExpr demand stat@(CStatExpr (CCompound [] stmts _) _) = scope $ do
    let (effects, final) = case last stmts of
```

```
            CBlockStmt (CExpr expr _) | demand -> (init stmts, expr)
            _ -> (stmts, Nothing)
    effects' <- cfgToRust stat (foldr interpretBlockItem (return ([], Unreachable)) effects)
    final' <- mapM (interpretExpr True) final
    return Result
        { resultType = maybe IsVoid resultType final'
        , resultMutable = maybe Rust.Immutable resultMutable final'
        , result = Rust.BlockExpr (Rust.Block effects' (fmap result final'))
        }
```

Otherwise, we have not yet implemented this kind of expression.

```
interpretExpr _ expr = unimplemented expr
```

TODO: Document these expression helper functions.

```
wrapping :: Result -> Result
wrapping r@(Result { resultType = IsInt Unsigned _ }) = case result r of
    Rust.Add lhs rhs -> r { result = Rust.MethodCall lhs (Rust.VarName "wrapping_add") [rhs] }
    Rust.Sub lhs rhs -> r { result = Rust.MethodCall lhs (Rust.VarName "wrapping_sub") [rhs] }
    Rust.Mul lhs rhs -> r { result = Rust.MethodCall lhs (Rust.VarName "wrapping_mul") [rhs] }
    Rust.Div lhs rhs -> r { result = Rust.MethodCall lhs (Rust.VarName "wrapping_div") [rhs] }
    Rust.Mod lhs rhs -> r { result = Rust.MethodCall lhs (Rust.VarName "wrapping_rem") [rhs] }
    Rust.Neg e -> r { result = Rust.MethodCall e (Rust.VarName "wrapping_neg") [] }
    _ -> r
wrapping r = r

toPtr :: Result -> Maybe Result
toPtr ptr@(Result { resultType = IsArray mut _ el }) = Just ptr
    { resultType = IsPtr mut el
    , result = castTo (IsPtr mut el) ptr
    }
toPtr ptr@(Result { resultType = IsPtr{} }) = Just ptr
toPtr _ = Nothing

binop :: CExpr -> CBinaryOp -> Result -> Result -> EnvMonad s Result
binop expr op lhs rhs = fmap wrapping $ case op of
    CMulOp -> promote expr Rust.Mul lhs rhs
    CDivOp -> promote expr Rust.Div lhs rhs
    CRmdOp -> promote expr Rust.Mod lhs rhs
    CAddOp -> case (toPtr lhs, toPtr rhs) of
        (Just ptr, _) -> return (offset ptr rhs)
        (_, Just ptr) -> return (offset ptr lhs)
        _ -> promote expr Rust.Add lhs rhs
        where
        offset ptr idx = ptr
            { result = Rust.MethodCall (result ptr) (Rust.VarName "offset") [castTo (IsInt Signed WordW
            }
    CSubOp -> case (toPtr lhs, toPtr rhs) of
        (Just lhs', Just rhs') -> do
            ptrTo <- case compatiblePtr (resultType lhs') (resultType rhs') of
                IsPtr _ ptrTo -> return ptrTo
                _ -> badSource expr "pointer subtraction of incompatible pointers"
            let ty = IsInt Signed WordWidth
            let size = rustSizeOfType (toRustType ptrTo)
            return Result
```

```
                { resultType = ty
                , resultMutable = Rust.Immutable
                , result = (Rust.MethodCall (castTo ty lhs') (Rust.VarName "wrapping_sub") [castTo ty rh
                }
        (Just ptr, _) -> return ptr { result = Rust.MethodCall (result ptr) (Rust.VarName "offset") [Rus
        _ -> promote expr Rust.Sub lhs rhs
    CShlOp -> shift Rust.ShiftL
    CShrOp -> shift Rust.ShiftR
    CLeOp -> comparison Rust.CmpLT
    CGrOp -> comparison Rust.CmpGT
    CLeqOp -> comparison Rust.CmpLE
    CGeqOp -> comparison Rust.CmpGE
    CEqOp -> comparison Rust.CmpEQ
    CNeqOp -> comparison Rust.CmpNE
    CAndOp -> promote expr Rust.And lhs rhs
    CXorOp -> promote expr Rust.Xor lhs rhs
    COrOp -> promote expr Rust.Or lhs rhs
    CLndOp -> return Result { resultType = IsBool, resultMutable = Rust.Immutable, result = Rust.LAnd (
    CLorOp -> return Result { resultType = IsBool, resultMutable = Rust.Immutable, result = Rust.LOr (to
    where
    shift op' = return Result
        { resultType = lhsTy
        , resultMutable = Rust.Immutable
        , result = op' (castTo lhsTy lhs) (castTo rhsTy rhs)
        }
        where
        lhsTy = intPromote (resultType lhs)
        rhsTy = intPromote (resultType rhs)
    comparison op' = do
        res <- promotePtr expr op' lhs rhs
        return res { resultType = IsBool }
```

Assignment expressions, including the unary increment and decrement operators, have pretty complicated semantics in C. First, we reduce compound assignments (e.g. `x += ...`) into simple assignments with the corresponding binary operator (`x = x + ...`).

```
compound :: CExpr -> Bool -> Bool -> CAssignOp -> Result -> Result -> EnvMonad s Result
compound expr returnOld demand op lhs rhs = do
    let op' = case op of
            CAssignOp -> Nothing
            CMulAssOp -> Just CMulOp
            CDivAssOp -> Just CDivOp
            CRmdAssOp -> Just CRmdOp
            CAddAssOp -> Just CAddOp
            CSubAssOp -> Just CSubOp
            CShlAssOp -> Just CShlOp
            CShrAssOp -> Just CShrOp
            CAndAssOp -> Just CAndOp
            CXorAssOp -> Just CXorOp
            COrAssOp  -> Just COrOp
```

Some C assignments need to be translated to multiple Rust statements where we have to use the left-hand expression multiple times. We'll duplicate the left-hand side if either:

1. the assignment is a compound assignment, because the Rust compound assignment operators don't support many of the operand type combinations that the C compound assignment operators do;

2. or the surrounding expression uses the result of the assignment, in which case we have to both read and write the l-value in separate statements, because Rust assignment operators always evaluate to `()`.

```
let duplicateLHS = isJust op' || demand
```

However, if the left-hand expression might have side effects, then we must not duplicate those effects. In that case we need to use a more complicated translation:

- We let-bind a mutable borrow of the result of evaluating the left-hand side. That way we can dereference the borrow multiple times without evaluating the expression again.
- But the right-hand side expression might use variables that we'll borrow when capturing the left-hand side, so we have to make sure we evaluate the right-hand side *first* or Rust's borrow-checker may complain. So we generate a let-binding for the right-hand side as well.

To avoid using the complicated translation for the common cases, we use `hasNoSideEffects` (see below) to check whether an expression is guaranteed not to have side effects.

```
let (bindings1, dereflhs, boundrhs) =
        if not duplicateLHS || hasNoSideEffects (result lhs)
        then ([], lhs, rhs)
        else
            let lhsvar = Rust.VarName "_lhs"
                rhsvar = Rust.VarName "_rhs"
            in ([ Rust.Let Rust.Immutable rhsvar Nothing (Just (result rhs))
                , Rust.Let Rust.Immutable lhsvar Nothing (Just (Rust.Borrow Rust.Mutable (result lhs
                ], lhs { result = Rust.Deref (Rust.Var lhsvar) }, rhs { result = Rust.Var rhsvar })
rhs' <- case op' of
    Just o -> binop expr o dereflhs boundrhs
    Nothing -> return boundrhs
let assignment = Rust.Assign (result dereflhs) (Rust.:=) (castTo (resultType lhs) rhs')
```

Next we have to figure out what this assignment was supposed to evaluate to, unless the surrounding expression doesn't use the result, in which case we don't bother. In practice, most assignments in C programs are evaluated only for their side effects, not their result, so we can substantially simplify the code we generate by detecting that case.

However, for those few assignments where the result is needed, we have to determine whether the result is the newly assigned value (as in assignments and pre-increment/decrement operators) or the old value (as in post-increment/decrement operators). In the latter case, we let-bind a copy of the old value before performing the assignment.

```
let (bindings2, ret) =
        if not demand
        then ([], Nothing)
        else if not returnOld
        then ([], Just (result dereflhs))
        else
            let oldvar = Rust.VarName "_old"
            in ([Rust.Let Rust.Immutable oldvar Nothing (Just (result dereflhs))], Just (Rust.Var ol
```

Now we put together the generated let-bindings, assignment statement, and result expression, and check whether we need a block expression to wrap them all up.

If there's no result expression (because the result isn't used), then we generate a `void`-typed result. If, in addition, there are no let-bindings, then we don't need a block expression at all and can just return the assignment expression by itself.

```
return $ case Rust.Block (bindings1 ++ bindings2 ++ exprToStatements assignment) ret of
    b@(Rust.Block body Nothing) -> Result
```

```
            { resultType = IsVoid
            , resultMutable = Rust.Immutable
            , result = case body of
                [Rust.Stmt e] -> e
                _ -> Rust.BlockExpr b
            }
        b -> lhs { result = Rust.BlockExpr b }
    where
```

We compute a conservative approximation to the question of whether an expression has side effects. `hasNoSideEffects` only returns `True` if we can be absolutely sure that the expression does not have side effects, but may return `False` even for expressions that further inspection would show are safe to duplicate.

This is not a general-purpose side-effect checker either. It only processes expressions that might appear in an l-value, because that's all we currently use it for. For anything else, it would return a conservative guess of `False`.

```
    hasNoSideEffects (Rust.Var{}) = True
    hasNoSideEffects (Rust.Path{}) = True
    hasNoSideEffects (Rust.Member e _) = hasNoSideEffects e
    hasNoSideEffects (Rust.Deref p) = hasNoSideEffects p
    hasNoSideEffects _ = False

rustSizeOfType :: Rust.Type -> Result
rustSizeOfType (Rust.TypeName ty) = Result
        { resultType = IsInt Unsigned WordWidth
        , resultMutable = Rust.Immutable
        , result = Rust.Call (Rust.Var (Rust.VarName ("::std::mem::size_of::<" ++ ty ++ ">"))) []
        }

rustAlignOfType :: Rust.Type -> Result
rustAlignOfType (Rust.TypeName ty) = Result
        { resultType = IsInt Unsigned WordWidth
        , resultMutable = Rust.Immutable
        , result = Rust.Call (Rust.Var (Rust.VarName ("::std::mem::align_of::<" ++ ty ++ ">"))) []
        }
```

Constant expressions

C defines a subset of the expression syntax that can be evaluated at compile-time, and then allows such constant expressions to be used for various things, such as array size expressions.

In this section we define a constant expression evaluator that we can use when we need to know the actual value of a constant expression in order to translate it to Rust. This is not always necessary: often, equivalent constant expressions can be used in Rust, and doing so is better because it preserves more of the programmer's intent.

```
interpretConstExpr :: CExpr -> EnvMonad s Integer
interpretConstExpr (CConst (CIntConst (CInteger v _ _) _)) = return v
interpretConstExpr expr = unimplemented expr
```

Implicit type coercions

C implicitly performs a variety of type conversions. Rust has far fewer cases where it will convert between types without you telling it to. So we have to encode C's implicit coercions as explicit casts in Rust.

On the other hand, if it happens that a translated expression already has the desired type, we don't want to emit a cast expression that will just clutter up the generated source. So `castTo` is a smart constructor that inserts a cast if and only if we need one.

```
castTo :: CType -> Result -> Rust.Expr
castTo target source | resultType source == target = result source
```

Before we can generate any C-compatible cast from an array type, we need to turn it into a raw pointer first. Then we can try again to cast it to the desired type.

```
castTo target (Result { resultType = IsArray mut _ el, result = source }) =
    castTo target Result
        { resultType = IsPtr mut el
        , resultMutable = Rust.Immutable
        , result = Rust.MethodCall source (Rust.VarName method) []
        }
    where
    method = case mut of
        Rust.Immutable -> "as_ptr"
        Rust.Mutable -> "as_mut_ptr"
```

Rust doesn't allow casting any other type directly to `bool`, so use the special-case boolean conversions defined below instead.

```
castTo IsBool source = toBool source
```

Because of the way C integer literals are defined, it's very common for us to translate a literal as an `i32` and then discover that we actually want it to be a `u8`, or whatever. Simplifying `42i32 as (u8)` into just `42u8` not only makes the generated code easier to read and maintain, but it also enables the Rust compiler to give useful warnings about literals that are outside the legal range for their type.

There is one caveat to this, however. Rust does not allow negative integer literals for unsigned types. It doesn't just warn about them; attempting it results in a compile-time error. The result we want is to wrap such negative values around as large positive values. Emitting a signed integer literal and then casting it to unsigned produces the right result, which is what we'd have done anyway without this simplification, so we can just suppress this rule in that case.

```
castTo target@(IsInt{}) (Result { result = Rust.Lit (Rust.LitInt n repr _) })
    = Rust.Lit (Rust.LitInt n repr (toRustType target))
castTo (IsInt Signed w) (Result { result = Rust.Neg (Rust.Lit (Rust.LitInt n repr _)) })
    = Rust.Neg (Rust.Lit (Rust.LitInt n repr (toRustType (IsInt Signed w))))
```

If none of the special cases apply, then emit a Rust cast expression.

```
castTo target source = Rust.Cast (result source) (toRustType target)
```

Similarly, when Rust requires an expression of type `bool`, or C requires an expression to be interpreted as either "0" or "not 0" representing "false" and "true", respectively, then we need to figure out how to turn the arbitrary scalar expression we have into a `bool`.

```
toBool :: Result -> Rust.Expr
```

To convert an integer literal expression to `bool`, we don't need to look at the type, just the value. We only convert the specific values 1 and 0, because those are idiomatic representations of `true` and `false` in C. Any other values would be surprising if used in a boolean context and we choose to translate them more verbosely to call the developer's attention to them.

```
toBool (Result { result = Rust.Lit (Rust.LitInt 0 _ _) })
    = Rust.Lit (Rust.LitBool False)
toBool (Result { result = Rust.Lit (Rust.LitInt 1 _ _) })
```

```
    = Rust.Lit (Rust.LitBool True)
toBool (Result { resultType = t, result = v }) = case t of
```

The expression may already have boolean type, in which case we can return it unchanged.

```
    IsBool -> v
```

Or it may be a pointer, in which case we need to generate a test for whether it is not a null pointer.

```
    IsPtr _ _ -> Rust.Not (Rust.MethodCall v (Rust.VarName "is_null") [])
```

Otherwise, it should be numeric and we just need to test that it is not equal to 0.

```
    _ -> Rust.CmpNE v 0
```

`toNotBool` works just like `Rust.Not . toBool`, except that it can generate simpler expressions. Instead of `!!p.is_null()`, for example, it simply generates `p.is_null()`. This approach satisfies our design goal of generating Rust which is as structurally close to the input as possible.

```
toNotBool :: Result -> Rust.Expr
toNotBool (Result { result = Rust.Lit (Rust.LitInt 0 _ _) })
    = Rust.Lit (Rust.LitBool True)
toNotBool (Result { result = Rust.Lit (Rust.LitInt 1 _ _) })
    = Rust.Lit (Rust.LitBool False)
toNotBool (Result { resultType = t, result = v }) = case t of
    IsBool -> Rust.Not v
    IsPtr _ _ -> Rust.MethodCall v (Rust.VarName "is_null") []
    _ -> Rust.CmpEQ v 0
```

C defines a set of rules called the "integer promotions" (C99 section 6.3.1.1 paragraph 2). `intPromote` encodes those rules as a type-to-type transformation. To convert an expression to its integer-promoted type, use with `castTo`.

```
intPromote :: CType -> CType
```

From the spec: "If an int can represent all values of the original type, the value is converted to an int,"

Here we pretend that `int` is 32-bits wide. This assumption is true on the major modern CPU architectures. Also, we're trying to create a standard-conforming implementation of C, not clone the behavior of a particular compiler, and the C standard allows us to define `int` to be whatever size we like. That said, this assumption may break non-portable C programs.

Conveniently, Rust allows `bool` and `enum` typed expressions to be cast to any integer type, and it converts `true` to 1 and `false` to 0 just like C requires, so we don't need any special magic to handle booleans or enumerated types here.

```
intPromote IsBool = IsInt Signed (BitWidth 32)
intPromote (IsEnum _) = enumReprType
intPromote (IsInt _ (BitWidth w)) | w < 32 = IsInt Signed (BitWidth 32)
```

"otherwise, it is converted to an unsigned int. ... All other types are unchanged by the integer promotions."

```
intPromote x = x
```

C also defines a set of rules called the "usual arithmetic conversions" (C99 section 6.3.1.8) to determine what type binary operators should be evaluated at.

```
usual :: CType -> CType -> Maybe CType
usual (IsFloat aw) (IsFloat bw) = Just (IsFloat (max aw bw))
usual a@(IsFloat _) _ = Just a
usual _ b@(IsFloat _) = Just b
```

"Otherwise, the integer promotions are performed on both operands."

```
usual origA origB = case (intPromote origA, intPromote origB) of
```

"Then the following rules are applied to the promoted operands:"

Note that we refuse to translate this expression if either promoted type is not an integer type, or if the integer conversion ranks are incomparable according to `integerConversionRank`. The latter is not due to anything in the C standard, which describes a total order for integer conversion ranks. Rather, we refuse to translate these programs due to non-portable code complicating the translation.

- "If both operands have the same type, then no further conversion is needed."

  ```
  (a, b) | a == b -> Just a
  ```

- "Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank."

  ```
  (IsInt Signed sw, IsInt Unsigned uw) -> mixedSign sw uw
  (IsInt Unsigned uw, IsInt Signed sw) -> mixedSign sw uw
  (IsInt as aw, IsInt _bs bw) -> do
      rank <- integerConversionRank aw bw
      Just (IsInt as (if rank == GT then aw else bw))
  _ -> Nothing
  where
  ```

At this point we have one signed and one unsigned operand. The usual arithmetic conversions don't care which order the operands are in, so `mixedSign` is a helper function where the signed width is always the first argument and the unsigned width is always the second.

- "Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type."

  ```
  mixedSign sw uw = do
      rank <- integerConversionRank uw sw
      Just $ case rank of
          GT -> IsInt Unsigned uw
          EQ -> IsInt Unsigned uw
  ```

- "Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type."

  A signed type can represent all the values of an unsigned type if the unsigned type's bit-width is strictly smaller than the signed type's bit-width.

  For our purposes, `long` can only represent the values of `unsigned` types smaller than `int` (because `long` and `int` might be the same size). But `unsigned long` values can only be represented by signed types bigger than 64 bits (because `long` might be 64 bits instead).

  ```
  _ | bitWidth 64 uw < bitWidth 32 sw -> IsInt Signed sw
  ```

- "Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type."

  Given the above definitions, this only happens for `unsigned long` and `int64_t`, where we want to choose `uint64_t`.

  ```
  _ -> IsInt Unsigned sw
  ```

The usual arithmetic conversions refer to a definition called "integer conversion rank" from C99 6.3.1.1. We want these widths to have strictly increasing integer conversion rank:

- `BitWidth 32` (our representation of `int`)
- `WordWidth` (our representation of `long`)
- `BitWidth 64`

The first two come from C99 6.3.1.1 which says `long` has a higher rank than `int`. We add the last as an implementation choice. Since `isize` could be either `i32` or `i64`, when combining `isize` with another type we'll bump up to whichever type is definitely bigger.

We disallow comparing bit-widths between 32 and 64 bits, exclusive, against word-size. Since word-size is platform-dependent, we can't be sure which is larger.

```
integerConversionRank :: IntWidth -> IntWidth -> Maybe Ordering
integerConversionRank (BitWidth a) (BitWidth b) = Just (compare a b)
integerConversionRank WordWidth WordWidth = Just EQ
integerConversionRank (BitWidth a) WordWidth
    | a <= 32 = Just LT
    | a >= 64 = Just GT
integerConversionRank WordWidth (BitWidth b)
    | b <= 32 = Just GT
    | b >= 64 = Just LT
integerConversionRank _ _ = Nothing
```

Here's a helper function to apply the usual arithmetic conversions to both operands of a binary operator, cast as needed, and then combine the operands using an arbitrary Rust binary operator.

```
promote
    :: (Pretty node, Pos node)
    => node
    -> (Rust.Expr -> Rust.Expr -> Rust.Expr)
    -> Result -> Result -> EnvMonad s Result
promote node op a b = case usual (resultType a) (resultType b) of
    Just rt -> return Result
        { resultType = rt
        , resultMutable = Rust.Immutable
        , result = op (castTo rt a) (castTo rt b)
        }
    Nothing -> badSource node $ concat
        [ "arithmetic combination for "
        , show (resultType a)
        , " and "
        , show (resultType b)
        ]
```

`compatiblePtr` implements the equivalent of the "usual arithmetic conversions" but for pointer types.

FIXME: Citation to C99 needed.

```
compatiblePtr :: CType -> CType -> CType
compatiblePtr (IsPtr _ IsVoid) b = b
compatiblePtr (IsArray mut _ el) b = compatiblePtr (IsPtr mut el) b
compatiblePtr a (IsPtr _ IsVoid) = a
compatiblePtr a (IsArray mut _ el) = compatiblePtr a (IsPtr mut el)
compatiblePtr (IsPtr m1 a) (IsPtr m2 b) = IsPtr (leastMutable m1 m2) (compatiblePtr a b)
    where
    leastMutable Rust.Mutable Rust.Mutable = Rust.Mutable
    leastMutable _ _ = Rust.Immutable
compatiblePtr a b | a == b = a
```

If we got here, the pointer types are not compatible, which as far as I can tell is not allowed by C99. But GCC only treats it as a warning, so we cast both sides to a void pointer, which should work on the usual architectures.

```
compatiblePtr _ _ = IsVoid
```

Finally, `promotePtr` is like `promote` but for operators that allow pointers as operands, not just arithmetic types. Since integer literals may be implicitly used as pointers, if either operand is a pointer, we pretend the other one is a void pointer and let `compatiblePtr` figure out what type it should really be converted to.

```
promotePtr
    :: (Pretty node, Pos node)
    => node
    -> (Rust.Expr -> Rust.Expr -> Rust.Expr)
    -> Result -> Result -> EnvMonad s Result
promotePtr node op a b = case (resultType a, resultType b) of
    (IsArray _ _ _, _) -> ptrs
    (IsPtr _ _, _) -> ptrs
    (_, IsArray _ _ _) -> ptrs
    (_, IsPtr _ _) -> ptrs
    _ -> promote node op a b
    where
    ptrOrVoid r = case resultType r of
        t@(IsArray _ _ _) -> t
        t@(IsPtr _ _) -> t
        _ -> IsPtr Rust.Mutable IsVoid
    ty = compatiblePtr (ptrOrVoid a) (ptrOrVoid b)
    ptrs = return Result
        { resultType = ty
        , resultMutable = Rust.Immutable
        , result = op (castTo ty a) (castTo ty b)
        }
```

Internal representation of C types

TODO: Document the type representation we use here.

```
data Signed = Signed | Unsigned
    deriving (Show, Eq)

data IntWidth = BitWidth Int | WordWidth
    deriving (Show, Eq)
```

Sometimes we want to treat `WordWidth` as being equivalent to a particular number of bits, but the choice depends on whether we want it to be its smallest width or its largest. (We choose to define the machine's word size as always either 32 or 64, because those are the only sizes Rust currently targets.)

```
bitWidth :: Int -> IntWidth -> Int
bitWidth wordWidth WordWidth = wordWidth
bitWidth _ (BitWidth w) = w

data CType
    = IsBool
    | IsInt Signed IntWidth
    | IsFloat Int
    | IsVoid
    | IsFunc CType [(Maybe (Rust.Mutable, Ident), CType)] Bool
```

```
    | IsPtr Rust.Mutable CType
    | IsArray Rust.Mutable Int CType
    | IsStruct String [(String, CType)]
    | IsEnum String
    | IsIncomplete Ident
    deriving Show
```

Deriving a default implementation of the `Eq` typeclass for equality almost works, except for our representation of function types, where argument names and `const`-ness may differ without the types being different.

```
instance Eq CType where
    IsBool == IsBool = True
    IsInt as aw == IsInt bs bw = as == bs && aw == bw
    IsFloat aw == IsFloat bw = aw == bw
    IsVoid == IsVoid = True
    IsFunc aRetTy aFormals aVariadic == IsFunc bRetTy bFormals bVariadic =
        aRetTy == bRetTy && aVariadic == bVariadic &&
        map snd aFormals == map snd bFormals
    IsPtr aMut aTy == IsPtr bMut bTy = aMut == bMut && aTy == bTy
    IsArray aMut _ aTy == IsArray bMut _ bTy = aMut == bMut && aTy == bTy
    IsStruct aName aFields == IsStruct bName bFields =
        aName == bName && aFields == bFields
    IsEnum aName == IsEnum bName = aName == bName
    IsIncomplete aName == IsIncomplete bName = aName == bName
    _ == _ = False

toRustType :: CType -> Rust.Type
toRustType IsBool = Rust.TypeName "bool"
toRustType (IsInt s w) = Rust.TypeName ((case s of Signed -> 'i'; Unsigned -> 'u') : (case w of BitWidt
toRustType (IsFloat w) = Rust.TypeName ('f' : show w)
toRustType IsVoid = Rust.TypeName "::std::os::raw::c_void"
toRustType (IsFunc retTy args variadic) = Rust.TypeName $ concat
    [ "unsafe extern fn("
    , args'
    , ")"
    , if retTy /= IsVoid then " -> " ++ typename retTy else ""
    ]
    where
    typename (toRustType -> Rust.TypeName t) = t
    args' = intercalate ", " (
            map (typename . snd) args ++ if variadic then ["..."] else []
        )
toRustType (IsPtr mut to) = let Rust.TypeName to' = toRustType to in Rust.TypeName (rustMut mut ++ to')
    where
    rustMut Rust.Mutable = "*mut "
    rustMut Rust.Immutable = "*const "
toRustType (IsArray _ size el) = Rust.TypeName ("[" ++ typename el ++ "; " ++ show size ++ "]")
    where
    typename (toRustType -> Rust.TypeName t) = t
toRustType (IsStruct name _fields) = Rust.TypeName name
toRustType (IsEnum name) = Rust.TypeName name
toRustType (IsIncomplete ident) = Rust.TypeName (identToString ident)
```

Functions that don't return anything have a return type in Rust that is different than the representation of C's `void` type elsewhere.

```
toRustRetType :: CType -> Rust.Type
toRustRetType IsVoid = Rust.TypeName "()"
toRustRetType ty = toRustType ty
```

C leaves it up to the implementation to decide whether the base `char` type is signed or unsigned. We choose unsigned, because that's the type of Rust's byte literals.

```
charType :: CType
charType = IsInt Unsigned (BitWidth 8)
```

C permits implementations to represent enum values using integer types narrower than `int`, but for the moment we choose not to. This may be incompatible with some ABIs.

```
enumReprType :: CType
enumReprType = IsInt Signed (BitWidth 32)
```

`CType` is a representation of C types, simplified so that it's easy to translate to a corresponding Rust type–but during the process of constructing a `CType` from the raw C AST, here's some extra state we need to keep track of.

```
data IntermediateType = IntermediateType
    { typeMutable :: Rust.Mutable
    , typeIsFunc :: Bool
    , typeRep :: CType
    }
```

When translating certain C constructs, such as types or `extern` declarations, we defer translation until we see that the declaration is actually used. But we don't want to translate things multiple times. So `runOnce`, given some action to maybe run later, wraps it up in a mutable reference cell (somewhat like Rust's `Cell` or `RefCell` types). Then it returns a new action, which reads the current contents of the reference cell, and runs the original action if necessary, caching the result in the reference cell.

```
runOnce :: EnvMonad s a -> EnvMonad s (EnvMonad s a)
runOnce action = do
    cacheRef <- lift $ lift $ newSTRef (Left action)
    return $ do
        cache <- lift $ lift $ readSTRef cacheRef
        case cache of
            Left todo -> do
                lift $ lift $ writeSTRef cacheRef $ Left $
                    fail "internal error: runOnce action depends on itself, leading to an infinite loop
                val <- todo
                lift $ lift $ writeSTRef cacheRef (Right val)
                return val
            Right val -> return val
```

Given a bag of declaration specifiers like `static`, `const`, or `int`, we construct our own representation of the described type. In the process we add any nested declarations of new `struct`, `union`, or `enum` types to the environment.

```
baseTypeOf :: [CDeclSpec] -> EnvMonad s (Maybe CStorageSpec, EnvMonad s IntermediateType)
baseTypeOf specs = do
    -- TODO: process attributes and the `inline` keyword
    let (storage, _attributes, basequals, basespecs, _inlineNoReturn, _align) = partitionDeclSpecs specs
    mstorage <- case storage of
        [] -> return Nothing
        [spec] -> return (Just spec)
        _ : excess : _ -> badSource excess "extra storage class specifier"
    base <- typedef (mutable basequals) basespecs
```

```
    return (mstorage, base)
    where
```

If the type specifiers include a reference to a `typedef`, that must be the only type specifier.

```
    typedef mut [spec@(CTypeDef ident _)] = do
```

Using a `typedef` is tricky, since they can have `const` baked in, or represent a function type rather than a function pointer. The resulting type is `const` if at least one of the `typedef` or this specifier list includes `const`. Other type information is just copied from the `typedef`.

```
        (name, mty) <- getTypedefIdent ident
        case mty of
            Just deferred | mut == Rust.Immutable ->
                return (fmap (\ itype -> itype { typeMutable = Rust.Immutable }) deferred)
            Just deferred -> return deferred
```

GCC headers use `__builtin_va_list` as the type implementing `va_list`. On all the platforms I checked, this type's ABI is compatible with a pointer, so we translate it to a pointer to a unique incomplete type.

```
            Nothing | name == "__builtin_va_list" -> runOnce $ do
                ty <- emitIncomplete Type ident
                return IntermediateType
                    { typeMutable = mut
                    , typeIsFunc = False
                    , typeRep = IsPtr Rust.Mutable ty
                    }
            Nothing -> badSource spec "undefined type"
```

None of the other type specifiers can make a type `const`, so let the remaining cases return a `CType` and we'll wrap it up with the common extra fields.

```
    typedef mut other = do
        deferred <- singleSpec other
        return (fmap (simple mut) deferred)

    simple mut ty = IntermediateType
        { typeMutable = mut
        , typeIsFunc = False
        , typeRep = ty
        }
```

Like `typedef` types, none of this next group of type specifiers can be used with any other type specifier.

```
    singleSpec [CVoidType _] = return (return IsVoid)
    singleSpec [CBoolType _] = return (return IsBool)
```

A `struct` with no fields must have a tag, and represents a reference to a `struct` whose fields are given elsewhere. If this is a forward declaration, then values of this type can't be constructed or accessed until after the actual declaration is processed, so we mark this type incomplete until then.

```
    singleSpec [CSUType (CStruct CStructTag (Just ident) Nothing _ _) _] = do
        mty <- getTagIdent ident
        return $ fromMaybe (emitIncomplete Struct ident) mty
```

Translating a `struct` declaration begins by recursively translating the type of each field. We need to look up these types now, before later definitions might shadow the ones that are currently in scope—but we must be careful not to treat these types as used unless this struct itself is used. So we save the deferred type declarations for later.

Bitfields are not translated yet, but we can be lazy about them too, only reporting an error if this struct is actually used.

```
    singleSpec [CSUType (CStruct CStructTag mident (Just declarations) _ _) _] = do
        deferredFields <- fmap concat $ forM declarations $ \ declaration -> case declaration of
          CStaticAssert {} -> return []
          CDecl spec decls _ -> do
            (storage, base) <- baseTypeOf spec
            case storage of
                Just s -> badSource s "storage class specifier in struct"
                Nothing -> return ()
            forM decls $ \ decl -> case decl of
                (Just declr@(CDeclr (Just field) _ _ _ _), Nothing, Nothing) -> do
                    deferred <- derivedDeferredTypeOf base declr []
                    return (applyRenames field, deferred)
                (_, Nothing, Just _size) -> do
                    return ("<bitfield>", unimplemented declaration)
                _ -> badSource declaration "field in struct"
```

The next steps should only happen if this `struct` is used, and should happen at most once.

```
        deferred <- runOnce $ do
```

In C, a `struct` is allowed to be anonymous. Not so in Rust. If we encounter an anonymous `struct`, we need to construct a unique name for it and use that.

```
            (shouldEmit, name) <- case mident of
                Just ident -> do
                    rewrites <- lift (asks itemRewrites)
                    case Map.lookup (Struct, identToString ident) rewrites of
                        Just renamed -> return (False, concatMap ("::" ++) renamed)
                        Nothing -> return (True, identToString ident)
                Nothing -> do
                    name <- uniqueName "Struct"
                    return (True, name)
```

Now it's time to translate all the types we need for the fields of this `struct`.

```
            fields <- forM deferredFields $ \ (fieldName, deferred) -> do
                itype <- deferred
                return (fieldName, typeRep itype)
```

C allows assigning `struct` variables to each other and passing copies of them by value to function calls. To allow the same behavior in Rust, we need an implementation of the `Copy` trait, and that requires `Clone` as well. Usually we would auto-derive both of these traits but unfortunately arrays with a size larger than 32 do not implement `Clone`. They do however always implement `Copy` (as long as their elements can be copied) so we generate an explicit `Clone` implementation which simply copies the struct.

We also request that the Rust compiler lay out the fields of this `struct` using the same rules as the C ABI for the target platform.

```
            let attrs = [Rust.Attribute "derive(Copy)", Rust.Attribute "repr(C)"]
            when shouldEmit $ emitItems
                [ Rust.Item attrs Rust.Public (Rust.Struct name [ (field, toRustType fieldTy) | (field,
                , Rust.Item [] Rust.Private (Rust.CloneImpl (Rust.TypeName name))
                ]
            return (IsStruct name fields)
```

At this point we've set aside all the actions we'll want to do later if this type is used. Now we just need to

ensure those actions will happen if either: this `struct` is referenced by name; or the declaration in which this type appeared also declares some symbols.

```
        case mident of
            Just ident -> addTagIdent ident deferred
            Nothing -> return ()
        return deferred
```

As of this writing, Rust support for C-style `union` types has just recently landed in rustc nightly. Until that stabilizes, we can't fully translate C `union` types. What we can do, though, is allow pointers to such unions to be passed around anywhere. So for now, this creates an incomplete type for each `union`.

```
    singleSpec [CSUType (CStruct CUnionTag mident _ _ _) node] = runOnce $ do
        ident <- case mident of
            Just ident -> return ident
            Nothing -> do
                name <- uniqueName "Union"
                return (internalIdentAt (posOfNode node) name)
        emitIncomplete Union ident
```

Unlike `struct` references, an `enum` reference must name an `enum` that has already been declared.

```
    singleSpec [spec@(CEnumType (CEnum (Just ident) Nothing _ _) _)] = do
        mty <- getTagIdent ident
        case mty of
            Just ty -> return ty
            Nothing -> badSource spec "undefined enum"
```

In C, an `enum` declaration not only creates a new type, but also declares a collection of constants, which we add to the symbol environment.

```
    singleSpec [CEnumType (CEnum mident (Just items) _ _) _] = do
        deferred <- runOnce $ do
            (shouldEmit, name) <- case mident of
                Just ident -> do
                    rewrites <- lift (asks itemRewrites)
                    case Map.lookup (Enum, identToString ident) rewrites of
                        Just renamed -> return (False, concatMap ("::" ++) renamed)
                        Nothing -> return (True, identToString ident)
                Nothing -> do
                    name <- uniqueName "Enum"
                    return (True, name)

            -- FIXME: these expressions should be evaluated in the
            -- environment from declaration time, not at first use.
            enums <- forM items $ \ (ident, mexpr) -> do
                let enumName = applyRenames ident
                case mexpr of
                    Nothing -> return (Rust.EnumeratorAuto enumName)
                    Just expr -> do
                        expr' <- interpretExpr True expr
                        return (Rust.EnumeratorExpr enumName (castTo enumReprType expr'))
```

Like `struct` above, `enum` needs both `Copy` and `Clone`. But we also force each `enum` to be represented just like `enumReprType`, defined above.

```
            let Rust.TypeName repr = toRustType enumReprType
            let attrs = [ Rust.Attribute "derive(Clone, Copy)"
```

```
                    , Rust.Attribute (concat [ "repr(", repr, ")" ])
                    ]
            when shouldEmit $
                emitItems [Rust.Item attrs Rust.Public (Rust.Enum name enums)]
            return (IsEnum name)

        forM_ items $ \ (ident, _mexpr) -> addSymbolIdentAction ident $ do
            IsEnum name <- deferred
            return Result
                { resultType = IsEnum name
                , resultMutable = Rust.Immutable
                , result = Rust.Path (Rust.PathSegments [name, applyRenames ident])
                }

        case mident of
            Just ident -> addTagIdent ident deferred
            Nothing -> return ()
        return deferred
```

If none of those matched the type specifier list, then hopefully it's an arithmetic type. The default type in C is `signed int`, so we start there and modify it according to whatever type specifiers are present.

```
    singleSpec other = return (foldrM arithmetic (IsInt Signed (BitWidth 32)) other)

    arithmetic :: CTypeSpec -> CType -> EnvMonad s CType
    arithmetic (CSignedType _) (IsInt _ width) = return (IsInt Signed width)
    arithmetic (CUnsigType _) (IsInt _ width) = return (IsInt Unsigned width)
    arithmetic (CCharType _) _ = return charType
    arithmetic (CShortType _) (IsInt s _) = return (IsInt s (BitWidth 16))
    arithmetic (CIntType _) (IsInt s _) = return (IsInt s (BitWidth 32))
    arithmetic (CLongType _) (IsInt s _) = return (IsInt s WordWidth)
    arithmetic (CLongType _) (IsFloat w) = return (IsFloat w)
    arithmetic (CFloatType _) _ = return (IsFloat 32)
    arithmetic (CDoubleType _) _ = return (IsFloat 64)
    arithmetic spec _ = unimplemented spec
```

We've just finished covering how to compute a simple type, but C declarators add to the complexity. A declarator can inductively wrap a simple type in zero or more derived types, each of which can be a pointer, an array, or a function type.

```
derivedTypeOf :: EnvMonad s IntermediateType -> CDeclr -> EnvMonad s IntermediateType
derivedTypeOf deferred declr = join (derivedDeferredTypeOf deferred declr [])

derivedDeferredTypeOf
    :: EnvMonad s IntermediateType
    -> CDeclr
    -> [CDecl]
    -> EnvMonad s (EnvMonad s IntermediateType)
derivedDeferredTypeOf deferred declr@(CDeclr _ derived _ _ _) argtypes = do
    derived' <- mapM derive derived
    return $ do
        basetype <- deferred
        foldrM ($) basetype derived'
    where
```

In our internal type representation, `IsFunc` is a function *pointer*. So if we see a `CPtrDeclr` followed by a

`CFunDeclr`, we should eat the pointer.

If we see a `CArrDeclr` or `CFunDeclr` before a `CFunDeclr`, that's an error; there must be an intervening pointer.

```
    derive (CPtrDeclr quals _) = return $ \ itype ->
        if typeIsFunc itype
        then return itype { typeIsFunc = False }
        else return itype
            { typeMutable = mutable quals
            , typeRep = IsPtr (typeMutable itype) (typeRep itype)
            }
    derive (CArrDeclr quals arraySize _) = return $ \ itype ->
        if typeIsFunc itype
        then badSource declr "function as array element type"
        else do
            sizeExpr <- case arraySize of
                CArrSize _ sizeExpr -> return sizeExpr
                CNoArrSize _ -> unimplemented declr
            size <- interpretConstExpr sizeExpr
            return itype
                { typeMutable = mutable quals
                , typeRep = IsArray (typeMutable itype) (fromInteger size) (typeRep itype)
                }

    derive (CFunDeclr foo _ _) = do
        let preAnsiArgs = Map.fromList
                [ (argname, CDecl argspecs [(Just declr', Nothing, Nothing)] pos)
                | CDecl argspecs declrs _ <- argtypes
                , (Just declr'@(CDeclr (Just argname) _ _ _ pos), Nothing, Nothing) <- declrs
                ]
        (args, variadic) <- case foo of
            Right (args, variadic) -> return (args, variadic)
            Left argnames -> do
                argdecls <- forM argnames $ \ argname ->
                    case Map.lookup argname preAnsiArgs of
                    Nothing -> badSource declr ("undeclared argument " ++ show (identToString argname))
                    Just arg -> return arg
                return (argdecls, False)
        args' <- sequence
            [ do
                (storage, base') <- baseTypeOf argspecs
                case storage of
                    Nothing -> return ()
                    Just (CRegister _) -> return ()
                    Just s -> badSource s "storage class specifier on argument"
                (argname, argTy) <- case declr' of
                    [] -> return (Nothing, base')
                    [(Just argdeclr@(CDeclr argname _ _ _ _), Nothing, Nothing)] -> do
                        argTy <- derivedDeferredTypeOf base' argdeclr []
                        return (argname, argTy)
                    _ -> badSource arg "function argument"
                return $ do
                    itype <- argTy
                    when (typeIsFunc itype) (badSource arg "function as function argument")
```

```
                let ty = case typeRep itype of
                        IsArray mut _ el -> IsPtr mut el
                        orig -> orig
                return (fmap ((,) (typeMutable itype)) argname, ty)
        | arg@(CDecl argspecs declr' _) <-
```

Treat argument lists `(void)` and `()` the same: we'll pretend that both mean the function takes no arguments.

```
            case args of
            [CDecl [CTypeSpec (CVoidType _)] [] _] -> []
            _ -> args
        ]
    return $ \ itype -> do
        when (typeIsFunc itype) (badSource declr "function as function return type")
        args'' <- sequence args'
        return itype
            { typeIsFunc = True
            , typeRep = IsFunc (typeRep itype) args'' variadic
            }
```

Several places above need to check whether any type qualifiers are `const`, so this helper function answers that question.

```
mutable :: [CTypeQualifier a] -> Rust.Mutable
mutable quals = if any (\ q -> case q of CConstQual _ -> True; _ -> False) quals then Rust.Immutable els

typeName :: CDecl -> EnvMonad s (Rust.Mutable, CType)
typeName decl@(CStaticAssert {}) = badSource decl "static assert in type name "
typeName decl@(CDecl spec declarators _) = do
    (storage, base) <- baseTypeOf spec
    case storage of
        Just s -> badSource s "storage class specifier in type name"
        Nothing -> return ()
    itype <- case declarators of
        [] -> base
        [(Just declr@(CDeclr Nothing _ _ _ _), Nothing, Nothing)] ->
            derivedTypeOf base declr
        _ -> badSource decl "type name"
    when (typeIsFunc itype) (badSource decl "use of function type")
    return (typeMutable itype, typeRep itype)
```