Debug Info Module

This module serves the purpose of generating debug symbols. We use LLVM's source level debugging features for generating the debug information. The general principle is this:

Given the right metadata in the LLVM IR, the LLVM code generator is able to create DWARF debug symbols for the given code. The metadata is structured much like DWARF *debugging information entries* (DIE), representing type information such as datatype layout, function signatures, block layout, variable location and scope information, etc. It is the purpose of this module to generate correct metadata and insert it into the LLVM IR.

As the exact format of metadata trees may change between different LLVM versions, we now use LLVM DIBuilder to create metadata where possible. This will hopefully ease the adaption of this module to future LLVM versions.

The public API of the module is a set of functions that will insert the correct metadata into the LLVM IR when called with the right parameters. The module is thus driven from an outside client with functions like `debuginfo::create_local_var_metadata(bx: block, local: &ast::local)`.

Internally the module will try to reuse already created metadata by utilizing a cache. The way to get a shared metadata node when needed is thus to just call the corresponding function in this module:

```
let file_metadata = file_metadata(cx, file);
```

The function will take care of probing the cache for an existing node for that exact file path.

All private state used by the module is stored within either the CodegenUnitDebugContext struct (owned by the CodegenCx) or the FunctionDebugContext (owned by the FunctionCx).

This file consists of three conceptual sections: 1. The public interface of the module 2. Module-internal metadata creation functions 3. Minor utility functions

Recursive Types

Some kinds of types, such as structs and enums can be recursive. That means that the type definition of some type X refers to some other type which in turn (transitively) refers to X. This introduces cycles into the type referral graph. A naive algorithm doing an on-demand, depth-first traversal of this graph when describing types, can get trapped in an endless loop when it reaches such a cycle.

For example, the following simple type for a singly-linked list…

```
struct List {
    value: i32,
    tail: Option<Box<List>>,
}
```

will generate the following callstack with a naive DFS algorithm:

```
describe(t = List)
  describe(t = i32)
  describe(t = Option<Box<List>>)
    describe(t = Box<List>)
      describe(t = List) // at the beginning again...
      ...
```

To break cycles like these, we use “stubs”. That is, when the algorithm encounters a possibly recursive type (any struct or enum), it immediately creates a type description node and inserts it into the cache *before* describing the members of the type. This type description is just a stub (as type members are not described and added to it yet) but it allows the algorithm to already refer to the type. After the stub is inserted into the cache, the algorithm continues as before. If it now encounters a recursive reference, it will hit the cache and does not try to describe the type anew. This behavior is encapsulated in the `type_map::build_type_with_children()` function.

Source Locations and Line Information

In addition to data type descriptions the debugging information must also allow to map machine code locations back to source code locations in order to be useful. This functionality is also handled in this module. The following functions allow to control source mappings:

- `set_source_location()`
- `clear_source_location()`
- `start_emitting_source_locations()`

`set_source_location()` allows to set the current source location. All IR instructions created after a call to this function will be linked to the given source location, until another location is specified with `set_source_location()` or the source location is cleared with `clear_source_location()`. In the later case, subsequent IR instruction will not be linked to any source location. As you can see, this is a stateful API (mimicking the one in LLVM), so be careful with source locations set by previous calls. It’s probably best to not rely on any specific state being present at a given point in code.

One topic that deserves some extra attention is *function prologues*. At the beginning of a function’s machine code there are typically a few instructions for loading argument values into allocas and checking if there’s enough stack space for the function to execute. This *prologue* is not visible in the source code and LLVM puts a special PROLOGUE END marker into the line table at the first non-prologue instruction of the function. In order to find out where the prologue ends, LLVM looks for the first instruction in the function body that is linked to a source location. So, when generating prologue instructions we have to make sure that we don't emit source location information until the 'real' function body begins. For this reason, source location emission is disabled by default for any new function being codegened and is only activated after a call to the third function from the list above, `start_emitting_source_locations()`. This function should be called right before regularly starting to codegen the top-level block of the given function.

There is one exception to the above rule: `llvm.dbg.declare` instruction must be linked to the source location of the variable being declared. For function parameters these `llvm.dbg.declare` instructions typically occur in the middle of the prologue, however, they are ignored by LLVM's prologue detection. The `create_argument_metadata()` and related functions take care of linking the `llvm.dbg.declare` instructions to the correct source locations even while source location emission is still disabled, so there is no need to do anything special with source location handling here.