Derived Information Database

Roger Jarrett & Scott McKay

1. Introduction

The purpose of this paper is to describe the derived information database, that is, the database that describes all of the information we can determine by running a compiler (or some other compiler-like tool) over a source information database.

Note that it is not a requirement that the derived information database be implemented using a true, object-oriented database substrate In fact, the overhead of doing so (where the overhead includes size and performance costs as well as the cost of requiring all the clients of Dylan components to have a license for the database) may indicate that we should avoid using a true database for this. The current DOSS technology may prove a more appropriate format for storing all of this information.

2. Basic Design Goals

The basic goals of the derived database is to support the following:

- Storing information derived by the compiler in support of debugging, browsing, and so forth.
- Storing information derived by the compiler in support of incremental compilation.

3. The Derived Database Schema

The derived database for a project separately from the source database. In fact, the derived information may not be in a “real” database at all; it may reside in some combination of object files and other “DOSS” files. In any event, the derived “database” contains all of the information about a project that can be derived by running various tools (such as a compiler) over the source database. There are pointers from the derived database to the source database, but no pointers from the source database to the derived database.

3.1 The `section-name` entity

This entity provides a mapping from all language definition names (or “function specs”) to section handles. It is used to implement “meta-point” functionality. (See the Source Information Database document for a description of sections and section handles.)

As a convention, for sections that represent text in some programming language, the section name of each sections should be a “function spec” in that language. This convention makes it easier for users to see.

The section name to section handle mapping is created during the compilation process of a library, when the compiler has an accurate idea of what every section should be named. Note that a single source section handle might have several section names that map it. This can happen for several reasons. For example, the "section boundaries" in the source database might computed inaccurately by the editor's automatic sectionizer (due to the difficulties inherent in automatic sectionization). Or a section might contain a Dylan macro that expands into several top-level definitions.

3.2 The `pc-map` entity

This entity provides a mapping from program counters to source code fragments (such as section name and line number or expression index) in order to support things like source code steppers.

Since the PC to line number mapping cannot be resolved until very late in the compilation process, this entity will probably not be filled in until the first time the debugger is invoked on a program using this function. (For instance, the Dylan compiler in DylanWorks will use the native assembler to generate the object code, and so cannot fill in the PC map itself.) The debugger will compute this mapping by examining the object file and executable image.

The PCs will be stored as addresses relative to the beginning of the function, so that the function can be loaded at any address and the PC map will remain valid.

3.2.1 Debugging Inline Functions

Some entity, possibly related to the `pc-map`, will store information about macroexpansions and function inlining. The intent of this is to support easier debugging of code containing inlined functions.

3.3 The `call-frame` entity

For each function, this entity stores other information about what the state of the stack will be for calls to that function: where to find the frame's return address, the size of the frame, the value of the frame pointer, and where to find the non-volatile registers that were preserved by the local call frame, and so forth.

HP, MIPS, and M88k compilers all have good encodings of this information; our preference is to use the HP encoding. Unfortunately, the compilers for the Intel architectures do not seem to produce an encoding of the call-frame information. In this case, the debugger will need to disassemble the code while walking the stack in order to unwind the stack.

Note that the structure of the call-frame is available after the code generation phase, which may be done outside of our own compiler (by an assembler). Ideally, this entity will be entirely filled in by our own compiler, but since some of the information will not be available until after the assembler has been run, some parts of the entity (such as the PC range of the whole function) may be filled in the first time the debugger is used on the stack frame.

3.4 The `parameter-list` entity

The entity is used to store the “contract” of generic functions and methods, that is, the parameters of generic functions (or specializers of methods) and the returned-values declarations. For each parameter, this will record the name, type, default value, and “kind” (required argument, keyword argument, etc.). Returned-value declarations will record the name and type.

Tools such as the editor will use this to be able to display this information to programmers. The compiler will also use this during incremental compilation, in order to do type inferencing, generate good warnings, and so on.

3.5 The `local-variables` entity

This entity provides a mapping from program counters to a set of the active local variables active at that PC. The description of the local variables includes their names, types, and locations (such as, which register is the variable in or what location in a stack frame).

Variable names and lexical scope are available after the front-end pass of the compiler. Location (stack, register, heap) is available only after the code generation phase of the compiler. It is possible that some variables will be optimized away by the compiler, and thus will have no location.

The variable names entity will be used to evaluate and modify (assign to) variables.

3.6 The `cross-reference` entity

This entity provides, for each definition, a collection of its callers and callees. This is used to implement “who calls” and “calls who”. This will record *how* something is used as well as where it is used (called as a function, used as a variable reference, passed as the first argument to **make**, etc.).

DylanWorks will also provide a means to cause all the callers of a macro or an inlined function to be recompiled. This will be driven by the cross-reference information.

3.7 Class information entities

The following subsections describe the information that will be recorded for each Dylan class. Although we describe them as though they are separate entities, they might just as well be imlemented as a single “class-information” object.

Note that some of this information might be stored in the running application as well.

3.7.1 The `direct-subclasses` entity

For each class in the library, this entity stores its direct subclasses. Since Dylan does not require that an application retain this information for sealed classes, we store this so that we can do inspection of sealed classes in the development environment.

3.7.2 The `direct-superclasses` entity

For each class in the library, this entity stores its direct superclasses. Since Dylan does not require that an application retain this information for sealed classes, we store this so that we can do inspection of sealed classes in the development environment.

3.7.3 The `class-slots` entity

For each class in the library, this entity stores a description of each of its direct slots: its name, type, initial value, and so on. This information does not appear in the application, so it is retained here for use by the compiler and other tools.

3.7.4 The `class-name` entity

For each class in the library defined using `define class`, this entity stores its name. The debugger will use this to give more better information when displaying classes.

3.8 FFI

We intend to store information necessary to allow the user to debug programs containing FFI calls, at least at the simple level. This implies the ability to read other compiler's debug format from executable and object files.

3.9 Other Compilation Information

The derived information database also stores the following information:

- Macroexpanders. Allows the debugger to evaluate macros, and also allow the user to step through a macro 1 statement/line at a time. This would require assistance from the PC to source code location mapper.
- Information for doing inlining.
- Project and library information in order to do incremental/minimal compilation at a granularity smaller than the whole project or library.

4. Open Issues

How should these databases inherit from each other? That is, if you build an application from, say, the UI library and the 3d-graphics library, the derived information database for your application should be used to augment information gotten from the derived databases for the other two libraries. It is probably the case that simply loading multiple DOSS files into the development environment simply does the right thing (if you load them in the "right" order).

How do we manage multiple versions of DOSS files? In fact, how do we manage multiple versions of object files in general? Do we just jam them in different directories, or jam a version number in the name, or what?

Note that there are no direct links from the source information database into the derived information database. There are only links from the derived database back to the source database.

For entities like the PC map, variable names, class MOP, and so on, there may be different "versions" of these entities depending on various compiler settings (optimization level, target platform, etc.). Should we directly support the notion of multiple such entities identified by some "target id", or should we simply create a different derived database for each such target? It's easier to do the latter, but it makes it hard to answer such questions as "please show me all the callers of this function for every kind of platform".