Portable Universal Representation Program Language (PURPLe)

Thoughts on a High-Level Virtual Machine Language

This is a rough sketch of a file format for storing semi-compiled programs, similar in concept to Java Byte Code or CIL. The basic idea is to define a mostly-source-language-agnostic and mostly-platform-agnostic representation of a compiled program which can then be lowered into platform-optimized machine code at runtime.

One of the goals is to preserve enough high-level information to allow the VM to make significant optimizations which would not be feasible with a purely low-level representation.

Module Structure

A module is a container for a set of entities such as types, globals, constants, functions and so on. Each type of entity has an associated table, which is conceptually just a flat array of items of that type.

Most tables are sorted by descending frequency of reference - so the most popular items (in terms of the number of references to that item) are stored at the beginning of the table. Because VARINTs are used to encode references to table entries, this allows the use of fewer bits to store references to the most commonly referred entries.

- Tables
    - Names
        - Name tables consist of two tables, one for simple names and one for compound names.
        - Simple names correspond to an identifier in the source language. There are no restrictions on what characters are allowed in a simple name (except that they have to be printable.)
        - The simple names table is encoded as follows:
            - VARINT indicating the number of names in the table.
            - For each name:
                - VARINT of the number of bytes of the identifier
                - The actual bytes of the identifier
        - Compound names are made up of simple names, and are used to represent package names, path names, and references to symbols within another scope.
        - Normally compound names are printed by separating the name parts with

a dot (.), however in some languages a different separator string may be used - so for example to represent C++ scopes, a compound name could be printed using "::" as the separator. Such separators are only used for display purposes, and do not affect the meaning of the compound name. This means that A.B.C and A::B::C are equivalent.
        - Every compound name is stored as a 2-tuple, where the first part is either a simple or compound name, and the second part is always a simple name. So A.B.C would actually be stored as ((A.B).C). In a typical module, there will be many compound names that have the same prefix, so this representation allows all of those identifiers to share the tuple representing that prefix.
        - References to names are VARINTS, where the lowest bit indicates whether it is a simple (0) or compound (1) name. The rest of the bits contain the index of the name in the corresponding name table.
        - Compound names are always fully-qualified, meaning that they are absolute. However, not all compound names are within the same namespace, so the "root" will mean different things depending on the context of where the compound name is used.
    - Types
        - The types table contains the definition of all non-primitive types used in the module.
        - The types table only contains structural information about the type - it does not list information about methods or static variables, those are stored in the scope hierarchy.
        - The format for representing types is similar in concept to LLVM IR, with the addition of the following:
            - Explicit inheritance declarations.
            - Type members have names (indices into the names table)
            - Visibility (public / private / protected / internal) information.
            - Information about member mutability (const / mutable).
            - A "final" modifier for classes.
    - Constants
        - The constants table contains all of the unnamed constants used by the modules. (Names constants are stored in the scope hierarchy).
    - Scope Hierarchy
        - The Scope hierarchy consists of a Module scope, which can have any number of child scopes representing namespaces or classes.
        - Within each scope can exist globals, constants, or functions.

Functions

A function consists of a nested series of "scope blocks". These are similar to basic blocks, except that they are explicitly hierarchical. Each scope block defines a lexical scope, and may

contain an arbitrary number of child scope blocks intermixed with instructions. Each scope block must have a valid terminator. Local variables declared within a scope block are assumed to be destroyed when control leaves the block.

Block terminators include unconditional and conditional branch instructions, return and throw instructions, computed gotos, and other block transitions that are needed to implement the higher-level control structures of a given language.

It is assumed that all languages which compile into PURPLe code have compatible exceptions - that is, regardless of the source language, it will be possible to do an isSubclass() test on the value of an exception object. This means that the ‘catch’ terminator instruction can be expressed in terms of high-level types instead of runtime IDs and personality functions.

Debugging Info

The VM will generate DWARF or other host-specific debugging information when needed.

There are two types of debugging info: Those associated with a table entry (global, constant, etc), and those associated with an instruction.

Generally speaking, the table entry will contain enough information to completely create the DWARF DIEs for that entry. So for example for a field of a struct, we have all of the information needed - the field name, the type, visibility, modifiers, and so on. The field will also contain information about the source file and source line, in a compressed representation.

Each scope block will contain a reference to the original source file and source line of that block. Individual instructions will just store line-number deltas from the previous instruction - if there are large discontinuities in the sequence of source line numbers, then an auxiliary scope block can be inserted as needed to restart the sequence at a new base number.

Closures

Closures are simply objects that have a special “call” method, and are otherwise treated like any other object.

Method Dispatch

Method dispatch is handled by the VM, not by the compiler. The compiler does not have knowledge of the format of dispatch tables or the algorithm used for dispatch.

The goal is to allow the VM to make optimizations based on its knowledge of the type hierarchy - so for example, if the compiler knows that class A is never subclassed, then it can in many cases call methods of A directly, avoiding a vtable lookup.

Reflection

The representation used for types and functions contains all of the information needed to generate reflection information for a given type.

Module Dependencies

Every module has a table which lists what other modules were used in compiling this module. This can be used to construct makefile dependencies, or otherwise used in computing which modules need to be rebuilt when a given module has changed.

Templates

In some cases, a module may contain code templates, which aren't actually meant to be runnable, but rather instantiated with a given set of type parameters.

There are two different approaches to handling templates: Instantiation by the compiler, and instantiation by the VM.

Languages which use 'ad-hoc' polymorphism (like C++) require compile-time instantiation. Ad-hoc polymorphism means that the template instantiation process can be modeled as a textual transformation (regardless of whether or not it is actually implemented that way.) An example would be a template that takes a parameter T, and calls T::next() or some other static method of T, without actually knowing whether T has such a method or not, nor what it's type is. Obviously it is an error to attempt to instantiate that template with a T parameter that has no such method, but even if the type that is bound to T does have a next() method, the generated code can vary wildly depending on the type signature of T.

A different approach can be taken for languages which use "parametric" polymorphism, meaning that templates can only use features of a type that they know about in advance. So in the case of calling T::next(), the T parameter would have to be declared in such a way as to tell the compiler that there is in fact a next() method, and what it's type signature is.

Languages like these allow the compiler to compile the template right down to PURPLe code, with the template parameters serving as "placeholders" in the generated code. The VM can then clone the generated code, replacing the template parameters, and then optimize and translate to machine language.