Open Dylan Hacker's Guide Documentation

Release 0.1

Dylan Hackers

December 15, 2018

Contents:

CHAPTER

ONE

COPYRIGHT

CHAPTER

TWO

HOW TO CONTRIBUTE TO OPEN DYLAN

The first thing you'll need is a source checkout of the Git repository. The Open Dylan sources are hosted on GitHub, along with sources for the opendylan.org web site and many other repositories. If you don't yet have a GitHub account and ssh keys, now is a good time to get them.

To checkout the main "opendylan" repository:

```
git clone --recursive git@github.com:dylan-lang/opendylan
```

You'll want to fork this repository so you can push changes to your fork and then submit pull requests.

Aside from dylan-lang there are two other GitHub oranizations that may be of interest:

- dylan-foundry is Bruce Mitchener's large collection of Dylan libraries.
- dylan-hackers contains repositories of historical interest, such as old Harlequin Dylan documents and gwydion, the CMU implementation of Dylan which is no longer maintained.

Looking for Ideas?

If you're looking for ideas on how you can contribute to Open Dylan or what others might find useful, please see our list of projects in the Open Dylan wiki or check out the list of bugs labeled Easy. If you have something in mind that isn't there, feel free to talk with us or add it to the list.

Making Changes

- Fork the repository. You will need to have a GitHub account to do this.
- Create a topic branch with `git checkout -t -b your-contribution`.
- Commit your changes to your branch, putting each distinct fix in a separate commit.
- Push your changes to your fork on GitHub.
- Submit a pull request with your changes.

Guidelines

- You almost always want to branch from the master branch.
- Update documentation as necessary. Also, if appropriate, update the release notes, samples and other supporting materials.

- We suggest following this note about git commit messages.
- Follow the style guide for new code. When working in existing code, follow the existing style.
- Do not make a lot of style or whitespace fixes in the same commit as other changes.
- Use 2 spaces for indentation, **never** tabs. If you use emacs, dylan-mode does a decent job of indenting code.

Testing

In the near future, we will have our test suites working more reliably. We will also document the processes involved with updating and running tests. In the meantime, each library usually has a "tests" subdirectory with a test suite library named "*-tests".

Licensing

All materials in this repository are licensed under the terms expressed in the current License.txt file.

We are not interested in incorporating any source code using proprietary or copyleft-style licenses.

Open Dylan is under the collective ownership of the Dylan Hackers.

CHAPTER

THREE

WRITING DOCUMENTATION

We are working on a tool to automatically generate skeletal documentation from source code, but until then, we are documenting the Open Dylan libraries manually using Sphinx to build the HTML pages. Sphinx uses reStructuredText markup with some extensions of its own, and we have created additional extensions to document Dylan language entities.

The documentation — a number of RST files — is in the OpenDylan documentation directory. Consult the Sphinx web-site for details about reStructuredText markup and Sphinx extensions to it, and see the Dylan sphinx-extensions documentation repository for details about the Dylan language extensions. (You may use the `rst2html` tool to generate an HTML page from an `.rst` file.)

Guidelines

Gender

Try to avoid all gender-specific pronouns. In preference, use third person plural pronouns such as “they”, “their”, “them”.

Tense

Use simple present tense unless there is a clear need to do otherwise. Specifically, don’t use future tense to describe what a function **will** do, describe what it **does**.

Generating Documentation

`dylan-compiler` can output skeletal documentation for a library once it has been compiled. Prior to compiling the project, it will not output anything.

To generate documentation, once you’ve loaded and compiled your project, simply:

```
export -format rst interface-reference
```

The generated documention will need cleaning up and will not include anything from comments in the source.

We hope to improve this tool in the future.

Doctower

Doctower is a tool written by Dustin Voss and is available on GitHub.

This tool doesn't yet generate RST output, so the documentation from it will not integrate cleanly with Sphinx and our other documentation.

We don't currently use this tool for any of the Open Dylan documentation.

Example documentation

Here is an example of documentation to get a feel for how it works. Use the "View page source" link at the top of the page to see it in RST markup form.

Skip Lists

A skip-list is a data type equivalent to a balanced (binary) tree.

`skip-list` Library
The skip list library may be found in the "skip-list" repository.

`skip-list` Module
The skip-list module exports a number of symbols. Two new symbols are of interest:

- *<skip-list>*
- *element-sequence*

The skip-list module also adds to several generic methods. One of the new methods is:

- *element*

`<skip-list>` Open Primary Class
A skip-list is a data type equivalent to a balanced (binary) tree. All keys must be comparable by some kind of ordering function, e.g., <.

Superclasses <stretchy-collection>, <mutable-explicit-key-collection>

Init-Keywords

- **`key-test`** – The collection's key-test function; should return `#t` if two keys should be considered equal. Defaults to ==.
- **`key-order`** – A function that accepts two keys and returns `#t` if the first key sorts before the second. Defaults to <.
- **`size`** – Preallocates enough memory to hold this number of objects. Optional.
- **`capacity`** – Sets the maximum capacity of the skip list. Optional.
- **`probability`** – The probability to create a new level of the list. Equivalent to the fan-out of a tree. Defaults to 0.25.
- **`max-level`** – The list will not grow beyond this number of levels. Defaults to a value based on the `size` and `capacity` keywords.
- **`level`** – The list starts with this number of levels. Defaults to a value based on the `size` and `capacity` keywords.

In general, a skip list operates like a stretchy mutable key collection. But a skip list can also act as an *ordered* stretchy mutable key collection where the iteration order is the key order. To take advantage of this, the library defines `forward-by-key-iteration-protocol`, *`element-sequence`*, and `element-sequence-setter`.

`element-sequence` Generic function

Parameters

- **`list`** – A skip list.

Values

- **`sequence`** – An instance of `<sequence>`.

One of the useful features of skip lists is that they can be ordered. However, most of the useful operations that can be performed on ordered collections, such as sort, are only defined for sequences. To solve this problem, I add `element-sequence` and `element-sequence-setter`. The client may call the former to obtain a sequence, operate on it, and call the latter to fix the results in the skip list. The setter ensures that no elements have been added or removed from the skip list, only reordered.

`element(<skip-list>)` Method

A specialization of `element`.

Parameters

- **`collection`** – An instance of *`<skip-list>`*.
- **`key`** – The key of an element. An instance of `<object>`.
- **`default`** (*`#key`*) – A value to return if the element is not found. If omitted and element not found, signals an error.

Values

- **`object`** – The element associated with the key.

CHAPTER

FOUR

JAM-BASED BUILD SYSTEM

This document describes the Jam-based build system for Open Dylan.

Introduction

The purpose of the Open Dylan `build-system` is to coordinate the final stages in the building of a Dylan library project. The Open Dylan compiler takes a Dylan library in the form of Dylan source files, and directly generates x86 or PPC machine language object files (in either COFF or ELF format, depending on the target platform). These object files need to be linked together to construct either an executable object file, or a loadable dynamically-linked library (also called a shared library). This link needs to be performed by an external tool such as the Microsoft or GNU linker. The `build-system` component, controlled by a user-specified script file, directs the execution of these external tools.

Why Jam-based?

Use of the Jam scripting language increases the flexibility of Open Dylan for users, and makes it more maintainable by removing hard-coded assumptions spread throughout various layers of the compiler. Previous versions of the Open Dylan `build-system` component permitted users to choose either the Microsoft or GNU Binutils linker on the Win32 platform, and always used the GNU Binutils ELF linker on Linux platforms. New versions of these tools require modifications that go beyond the current flexibility of the `build-system` component, as will new Open Dylan target platforms.

Though the logic of `build-system` (and a former companion library, `linker-support`) was hard-coded, it did allow a limited amount of parameterization using script files. A typical parameterization, from the Microsoft linker script, looked like this:

```
#
linkdll
link /NODEFAULTLIB /INCREMENTAL:NO /PDB:NONE /NOLOGO \
  /ENTRY:$(mangled-dllname)Dll@12 -debug:full -debugtype:cv \
  /nologo /dll /out:$(full-dll-name) kernel32.lib \
  @$(dllname).link $(objects) $(c-libs) /base:$(base) \
  /version:$(image-version) $(linkopts)
```

Though these script files were poorly documented and insufficiently flexible, they did inspire the introduction of a real scripting language to direct the final stages of compilation in Open Dylan.

Jam is a build tool designed by Christopher Seiwald, founder of Perforce Software. It is similar in some ways to `make`, the traditional Unix build tool. However, instead of using only simple declarative rules to define build targets and the dependencies between them, Jam contains a full scripting language, allowing build script authors to define

high-level build instructions that match particular applications. The Jam program also includes Jambase, a library of rules (functions) for building executables and libraries from C, C++, and Fortran sources.

The original Jam tool is a standalone program written in C and YACC. Peter Housel re-implemented the Jam language interpreter and build logic as a reusable Dylan library for use in the Open Dylan `build-system`.

Choosing Build Scripts

Normally you can simply use the build script supplied with Open Dylan that corresponds to the external linker you will be using. The supplied build scripts include the following:

`x86-win32-vc6-build.jam` Build script for Microsoft Visual C++ 6.0.

`x86-win32-vc7-build.jam` Build script for Microsoft Visual C++ .NET.

`x86-linux-build.jam` Build script for x86 Linux systems using gcc.

The default build script is `platform-name-build.jam`. You can select a different build script from the Link page of the Environment Options dialog in the IDE, or using the `-build-script` option on the console compiler or console environment command-line.

Build scripts are written using the Jam script language, as described in the Jam manual page. Most Open Dylan build scripts `include` the `mini-jambase.jam` file, which contains excerpts from the Jambase file included with Perforce Jam and described in the Jambase Reference. They can also make use of additional built-in rules defined by the Open Dylan build system, as described in *Additional Built-In Jam Rules* and *Built-In Jam Variables*.

How the Compiler Uses the Build System

When you compile a library, the Open Dylan compiler constructs a new `build` directory and places the generated object files in it. It also constructs a text file called `dylanmakefile.mkf` to be read by the build system. This file contains information imported from the original LID or HDP project file, as well as information generated during compilation. Here is a sample `dylanmakefile.mkf`, in this case the one generated for the `build-system` component itself:

```
comment:        This build file is generated, please don't edit
library:        build-system
base-address:   0x63F20000
major-version:  0
minor-version:  0
library-pack:   0
compilation-mode:       tight
target-type:    executable
files:  library
        paths
        variables
        build
        jam-build
used-projects:  functional-dylan
        dummy
        ..\functional-dylan\
        io
        dummy
        ..\io\
        system
        dummy
        ..\system\
```

```
        file-source-records
        dummy
        ..\file-source-records\
        release-info
        dummy
        ..\release-info\
        dfmc-mangling
        dummy
        ..\dfmc-mangling\
        jam
        dummy
        ..\jam\
all-c-libraries: advapi32.lib
        shell32.lib
```

External files are used to communicate with the build system in order for the information to persist between invocations of the compiler. On the Win32 platform, `dylanmakefile.mkf` files are also copied into the `lib` directory on installation so that other libraries can link against the actual DLL (whose name might not be identical to the library name).

When Open Dylan needs to link a project, it calls the `build-system`, passing the name of the build directory and a list of targets to be built. The build system reads the `dylanmakefile.mkf` file and builds the targets accordingly.

The Open Dylan compiler's project manager expects the build script to define the following pseudo (`NotFile`) targets:

`exports` Describe exports.

`unify-dll` Describe unify-dll.

`dll` Link the project as a dynamically-linked library.

`unify-exe` Describe unify-exe.

`exe` Link the project as an executable program.

`release` Describe release.

`clean-all` Remove build products in the top-level project, and in all of the non-system libraries that it uses.

`clean` Remove build products in the top-level project.

Automatically-invoked Jam Rules

When the build system reads a `dylanmakefile.mkf` file, it invokes several of the Jam rules (functions) defined in the user's build script. These rules in turn register the necessary targets and their dependencies with the Jam build mechanism.

All of the rules described below take *image* as their first parameter; this is a list whose first element is the library name (from the `Library:` keyword of the `.mkf` file) and whose optional second component is the base name of the executable or shared library (from the `Executable:` keyword of the `.mkf` file).

`DylanLibrary *image* : *version* ;` Link a Dylan library as a shared library or executable image. This is always the first rule invoked for a given library, and it is usually charged with establishing the library target and setting global and target-specific variables.

The *version* argument normally contains two components, the first obtained from the `Major-version:` keyword of the `.mkf` file, and the second from the `Minor-version:` keyword.

DylanLibraryLinkerOptions *image* : *options* ; Add the given options to the link command line of the shared library and executable images. The link options provided in the `Linker-options:` keyword of the `.mkf` file are expanded using the usual Jam variable expansion rules before being passed to this rule. (This allows `Linker-options:` keywords in LID and HDP files to refer to platform-specific variables such as `$(guilflags)`).

DylanLibraryBaseAddress *image* : *address* ; Set the base address of the shared library. The compiler-computed base addresses are probably only usable on the Win32 platform.

DylanLibraryCLibraries *image* : *libraries* ; Link C (or other externally-derived) libraries into the shared library. The link options provided in the `C-libraries:` keyword of the `.mkf` file are expanded using the usual Jam variable expansion rules before being passed to this rule.

DylanLibraryCObjects *image* : *objects* ; Link C (or other externally-derived) object files into the shared library.

DylanLibraryCSources *image* : *sources* ; Link C source files into the shared library.

DylanLibraryCHeaders *image* : *headers* ; This rule normally does nothing. The `C-header-files:` HDP/LID file is normally used to ensure that files of various sorts (not just C header files) are copied into the build directory.

DylanLibraryC++Sources *image* : *sources* ; Link C++ source files into the shared library.

DylanLibraryRCFiles *image* : *rcfiles* ; Link Win32 resource files into the shared library and executable.

DylanLibraryJamIncludes *image* : *includes* ; Include other Jam files into the build definition. This is typically used via the `jam-includes:` keyword in the HDP/LID file. It is useful for setting up extensions to library or include search paths.

DylanLibraryUses *image* : *library* : *dir* ; Link other Dylan libraries into the shared library. The *library* argument gives the name of the other library, and the *dir* argument gives the name of the other library's build directory. If *dir* is `system`, then the library is an installed system library.

Additional Built-In Jam Rules

The build system defines the following additional built-in rules.

IncludeMKF *includes* ; Read each of the given `.mkf` files and invoke Jam rules as described in *Automatically-invoked Jam Rules*.

DFMCMangle *name* ; Mangle the given *name* according to the Open Dylan compiler's mangling rules. If *name* has a single component, it is considered to be a raw name; if there are three components they correspond to the variable-name, module-name, and library-name respectively.

Built-In Jam Variables

By default, the Jam build system is provided with some values. Some of these are derived from the base Jam implementation and are documented in the Jam manual page while others are Open Dylan extensions.

. The build directory.

Open Dylan extension.

COMPILER_BACKEND The name of the compiler back-end currently in use. Typically one `c`, `harp` or `llvm`.

Open Dylan extension.

JAMDATE The current date, in ISO-8601 format.

NT True on Windows.

OS The OS of the build host, not the target. This will typically be something like `linux`, `freebsd`, `darwin` or `win32`.

OSPLAT The CPU architecture of the build host, not the target. This will typically be something like `x86` or `x86_64`.

PERSONAL_ROOT The root of the destination build path, when the `-personal-root` compiler option or the `OPEN_DYLAN_USER_ROOT` environment variable is set.

Open Dylan extension.

SYSTEM_BUILD_SCRIPTS The path where the installed build scripts can be found.

Open Dylan extension.

SYSTEM_ROOT The path where the installation of Open Dylan can be found.

Open Dylan extension.

TARGET_PLATFORM The Open Dylan identifier for the target platform. This is something like `x86-linux` or `x86_64-darwin`.

Open Dylan extension.

UNIX True on non-Windows platforms, like Linux, FreeBSD and macOS.

Editing Jam Files

There is an Emacs major mode for editing Jam files.

CHAPTER

FIVE

DFMC, THE DYLAN FLOW MACHINE COMPILER

Notes, Warnings and Errors

The compiler provides a framework for handling notes, warnings and errors as they're encountered and created during the compilation process.

This code can be found in the dfmc-conditions library. Initial skeletal API documentation for this library can be found at *The DFMC-CONDITIONS API Reference*.

Conceptually, all notes, warnings and errors are subtypes of `<condition>`, so we will often refer to them collectively as *program conditions* or instances of `<program-condition>`.

Status of this Library

This library is interesting (to some) as it is in a half-completed state and many things are not yet fully implemented or used. We will try to note these things below when we discuss them.

Philosophy

The Open Dylan compiler tries hard not to abort compilation when an error is noted. This leads to many things that might be *fatal errors* elsewhere being represented as *serious warnings* here. For example, an undefined variable reference does not abort compilation, but if the code containing that reference is executed it causes a crash at run time.

This was part of a very exploratory development model under the Open Dylan IDE, where many errors could be corrected while an application was running.

This workflow isn't commonly used with Dylan today, so we may revise this philosophy and how it applies to the compiler's handling of program conditions, or at least make it configurable.

Similarly, a key element to how program conditions are currently used today is that when we store values on them, we store them as the raw object so that they're readily accessible from within the debugger, rather than only storing the string representations.

Program Condition Hierarchy

The root of the condition hierarchy is *<program-condition>*. This is an abstract class and one of the subclasses such as *<program-note>*, *<program-error>* or *<program-restart>* should be subclassed instead.

Reporting a Program Condition

The typical way to report that a program condition has arisen is to use *note*. There are other mechanisms, such as *raise*, *restart*, *simple-note* and *simple-raise*, but these are not in common usage.

For proper error reporting, you will want to try to report as accurate a *source location* as you possibly can. This can be tricky at first, so look at other similar warnings if you need the assistance.

The actual code for noting a program condition is pretty straightforward, once you've identified the location to emit the program condition, and the type of program condition to emit.

```
note(<wrong-type-in-assignment>,
     variable-name: the-name,
     type: binding-type,
     rhs: rhs-value,
     source-location: fragment-source-location(fragment));
```

Source Locations

There are a couple of useful rules to follow for getting source locations for noting a program condition during compilation.

- If you're in C-FFI, you're probably working with fragments, and so `fragment-source-location` is the right function.
- If you're in `dfmc-definitions`, then you probably also want `fragment-source-location`.
- If you're in conversion, you may be dealing with either fragments or model objects. For fragments, you want `fragment-source-location`. For model objects, you want `model-source-location`.
- If you're in `dfmc-optimization`, then you likely want `dfm-source-location` if you're working with an object that is part of the control flow or data flow graphs, like any computation or temporary. However, in some cases, you'll still be working with model objects, so keep an eye out for when you need to use `model-source-location`.

Defining a new Program Condition

Depending on where you are defining your new program condition within the *Program Condition Hierarchy*, you will need to use the appropriate program condition definer:

- *performance-note-definer*
- *portability-note-definer*
- *program-condition-definer*
- *program-error-definer*
- *program-note-definer*
- *program-restart-definer*
- *program-warning-definer*
- *run-time-error-warning-definer*
- *serious-program-warning-definer*
- *style-warning-definer*

An example definition looks like:

```
define program-warning <ambiguous-copy-down-method>
  slot condition-method, required-init-keyword: meth:;
  slot condition-other-methods, required-init-keyword: other-methods:;
  format-string "Multiple applicable copy-down methods for %s, picking one at random";
  format-arguments meth;
end;
```

An interesting thing to note here is that the *other-methods* are being recorded by this *<program-note>* even though they are not used within the formatted output. This is because the additional values can be useful when viewing the condition within the debugger or by other programmatic processing such as filtering.

PPML, Pretty Print Markup Language

When conditions are stored, their slots are converted to PPML objects. Many objects within the compiler are already configured to be able to generate PPML via the many specializations of `as(class == <ppml>, ...)` that can be found within the `dfmc-debug-back-end` (see print-condition.dylan).

Slots are converted to PPML representations via code that is autogenerated by the various definer macros which create a specialization on *convert-condition-slots-to-ppml*.

Filtering of Program Conditions

This is functionality that has not been completed and is currently not entirely in use.

To be written.

How Warnings Are Displayed and Recorded

To be written.

Responding to a Program Condition

In Dylan, the condition system allows for responses to conditions and can restart a computation with new information. While parts of `dfmc-conditions` are designed to permit this, this functionality, has never been completed and is not yet working.

Future Work

Look at cleaning up unused API and things that are no longer necessary.

- *`obsolete-condition?`* is probably obsolete.
- *`format-condition`* and related code including *`<detail-level>`* are probably no longer necessary with the code in `dfmc-debug-back-end` and the specialization on `print-object` present there.
- The specialization on `print-object` can probably go away.
- *`simple-note`* and *`simple-raise`* can go away.
- There is a comment in `dfmc/conversion/convert.dylan` that the presence of `dfm-context-id` is a hack until true source locations are available. Should we remove `context-id` and the supporting code? (On a related note, does that implementation of `dfm-context-id` even work?

Complete other parts of the implementation:

- Program condition filtering.
- Program restarts.
- Make *`<program-error>`* distinct from a serious warning. This would also need a change to `dfmc-debug-back-end` and a specialization on `condition-classification`.
- Use more of the various subclasses of *`<program-note>`* like the style, performance and portability notes. This requires getting the filtering to work.
- The implementation doesn't use limited collection types where it can.

The DFMC-CONDITIONS API Reference

Definers for new Program Conditions

`program-condition-definer` Macro

Macro Call

```
define [modifier*] program-condition *class-name* (*superclasses*)
  *slot-spec*
  format-string *string*;
  format-arguments *slot*, ...;
  filter *filter*;
end program-note;
```

Parameters

- **`modifier`** – One or more class adjectives. *bnf*
- **`class-name`** – A valid Dylan class name. *bnf*
- **`superclasses`** – One or more Dylan class names to be used as the superclasses for the newly created program condition.

- **slot-spec** – A slot specification.
- **format-string** – A format string valid for use with `format`.
- **format-arguments** – One or more parameters which will be passed to `format` along with the *format-string*. The parameter values will be drawn from the corresponding slots.
- **filter** – A Dylan expression to be used as the value for *program-note-filter* on the new class. This should either be `#f` or an instance of `<function>` which returns a boolean value.

Discussion

This is not typically used outside of the `dfmc-conditions` library. It is used for creating a new direct subclass of *<program-condition>*. Most often, *program-note-definer* or a similar more specific definer macro would be used instead.

Any additional slot specifications will be modified slightly:

- The `constant` adjective will be removed if present.
- The type constraint for the slot will be a type union with *<ppml>*.

program-note-definer Macro

Macro Call

```
define [modifier*] program-note *class-name*
  *slot-spec*
  format-string *string*;
  format-arguments *slot*, ...;
  filter *filter*;
end program-note;

define [modifier*] program-note *class-name* (*superclasses*)
  *slot-spec*
  format-string *string*;
  format-arguments *slot*, ...;
  filter *filter*;
end program-note;
```

Discussion Create a new *<program-note>* subclass.

performance-note-definer Macro

Discussion Create a new *<performance-note>* subclass. See *program-note-definer* for details.

portability-note-definer Macro

Discussion Create a new *<portability-note>* subclass. See *program-note-definer* for details.

program-error-definer Macro

Discussion Create a new *<program-error>* subclass. See *program-note-definer* for details.

program-restart-definer Macro

Discussion Create a new *<program-restart>* subclass. See *program-note-definer* for details.

program-warning-definer Macro

Discussion Create a new *<program-warning>* subclass. See *program-note-definer* for details.

run-time-error-warning-definer Macro

Discussion Create a new *<run-time-error-warning>* subclass. See *program-note-definer* for details.

serious-program-warning-definer Macro

Discussion Create a new *<serious-program-warning>* subclass. See *program-note-definer* for details.

style-warning-definer Macro

Discussion Create a new <style-warning-note> subclass. See *program-note-definer* for details.

program-condition-definer-definer Macro

Discussion This is not commonly used outside of dfmc-conditions. It is creating new program-conditioner definer macros.

Program Conditions

<program-condition> Open Abstract Class

Superclasses <format-string-condition>

Init-Keywords

- **compilation-stage** – Defaults to the value of *current-stage*.
- **program-note-creator** – Defaults to the value of *current-dependent*.
- **source-location** – Defaults to #f. Every effort should be made to supply a valid value for this keyword.

Discussion

The root of the hierarchy is <program-condition>. All errors, warnings, etc, about code in a program being compiled should be reported as instances of this class.

This class should only be used for type declarations and as the superclass for mixin properties. For instantiable classes, it's best to subclass one of *<program-error>*, *<program-note>*, or *<program-restart>* instead.

<program-notes> Type

Supertypes <sequence>

<program-note> Open Primary Abstract Class

Superclasses <warning>, *<program-condition>*

Init-Keywords

- **context-id** – An instance of <string>.
- **subnotes** – A sequence of subnotes, allowing hierarchical explanations to be constructed. See *Subnotes*.

Discussion

When a *context-id* has been supplied, this is used to give an indication of the logical context of the source that the note is about, typically to give a concise textual hint, allowing for example (where `"process-foo"` is the *context-id*:

```
foo.dylan:180:Warning in process-foo: Bogus call to bar.
```

<program-error> Open Abstract Class

Superclasses *<program-note>*

Discussion A <program-error> is a language error. Examples would be (most) syntax errors, inconsistent direct superclasses, or a reference to an undefined name.

<program-restart> Open Primary Abstract Class

Superclasses *<program-condition>*, <restart>

Init-Keywords

- **default** –

Discussion A <program-restart> is a <restart> meant to be used as part of the recovery protocol for some *<program-condition>*.

<program-warning> Open Abstract Class

Superclasses *<program-note>*

Discussion A <program-warning> is a note about something that might be a mistake in program, but the compiler is able to compile it without intervention.

<run-time-error-warning> Open Abstract Class

Superclasses *<program-warning>*

Discussion Run-time-error warnings are given when the compiler can prove that executing the code will lead definitely lead to a run-time error, whether or not that error is handled. These warnings should be hard for the user to suppress. It should be possible for a user to treat these warnings as errors; that is, stop the compilation process because of one.

<serious-program-warning> Open Abstract Class

Superclasses *<program-warning>*

<style-warning> Open Abstract Class

Superclasses *<program-warning>*

Discussion Style warnings are given when the compiler detects code in a style that is legal (strictly speaking), but not desirable. The display of style warnings can be inhibited globally, or on a class-by-class basis.

<performance-note> Open Abstract Class

Superclasses *<program-note>*

Discussion Performance notes are given when the compiler is prevented from doing an optimization that should be reasonable or expected in the current context. Typical reasons would be that it has insufficient type, sealing, or program flow information.

<portability-note> Open Abstract Class

Superclasses *<program-note>*

Discussion

Portability notes are given when the compiler detects something that is valid in the Open Dylan compiler, but is not part of portable Dylan or could have undefined effects in Dylan.

It should be possible to turn these warnings into errors, to support a standards-conforming version of the compiler.

Program Condition Slots

`condition-compilation-stage` Generic function

Signature condition-compilation-stage (object) => (value)

Parameters

- **`object`** – An instance of *<program-condition>*.

Values

- **`value`** – An instance of <object>.

`condition-context-id` Generic function

Signature condition-context-id (object) => (value)

Parameters

- **`object`** – An instance of *<program-note>*.

Values

- **`value`** – An instance of <object>.

`condition-program-note-creator` Generic function

Signature condition-program-note-creator (object) => (value)

Parameters

- **`object`** – An instance of *<program-condition>*.

Values

- **`value`** – An instance of <object>.

`condition-source-location` Generic function

Signature condition-source-location (object) => (value)

Parameters

- **`object`** – An instance of *<program-condition>*.

Values

- **`value`** – An instance of <object>.

Signaling Program Conditions

`note` Open Generic function

Signature note (class #key #all-keys) => ()

Parameters

- **class** – An instance of `subclass(<program-condition>)`.

Discussion The primary program condition signaling interface is `note`, which calls `make` on the condition class and signals it, possibly returning. It can be used for any program condition, but is mainly oriented towards *<program-note>*.

Example

```
note(<inaccessible-open-definition>,
     binding: form-variable-binding(form),
     source-location: form-source-location(form));
```

`note(subclass(<program-condition>))` Method

`maybe-note` Macro

`raise` Open Generic function

Signature raise (class #key #all-keys) => ()

Parameters

- **class** – An instance of `subclass(<program-condition>)`.

Discussion This function is analogous to the standard Dylan `error` function and is guaranteed to not return.

`raise(subclass(<program-error>))` Method

`restart` Open Generic function

Signature restart (class #key #all-keys) => ()

Parameters

- **class** – An instance of `subclass(<program-restart>)`.

`restart(subclass(<program-restart>))` Method

Preserving Program Conditions

Program conditions are tracked in each library. They are stored in a table that is associated with each `<library-description>` via *library-conditions-table*. There are implementations of another generic function, `remove-dependent-program-conditions` which is commonly invoked during *retraction*. (What *retraction* is for isn't clear to me at this point.)

`add-program-condition` Generic function

Signature add-program-condition (condition) => ()

Parameters

- **condition** – An instance of `<condition>`.

Discussion Records a program condition. This does not usually need to be invoked directly outside of `dfmc-conditions` where it is usually invoked during the filtering of a program condition.

`add-program-condition(<condition>)` Method

Discussion Runtime errors that are not *<program-condition>* are not currently tracked. This method doesn't record them.

`add-program-condition(<program-condition>)` Method

Discussion

Preserves a program condition by storing it in the *library-conditions-table* for the current library being compiled.

library-conditions-table Generic function

Signature library-conditions-table (library) => (table)

Parameters

- **library** – An instance of <object>.

Values

- **table** – An instance of <table>.

remove-program-conditions-from! Generic function

Signature remove-program-conditions-from! (table key stages) => ()

Parameters

- **table** – An instance of <object>.
- **key** – An instance of <object>.
- **stages** – An instance of <object>.

Recovery and Restarting

condition-block Macro

***error-recovery-model* Variable**

Subnotes

This is a very rarely used capability within the program condition system and isn't currently well supported by the compiler output to standard out and standard error.

Any *<program-note>* can have additional notes attached to it. These notes are useful for attaching extra data to a note, like possible options or the sets of conflicting items.

An example usage of subnotes is:

```
note(<ambiguous-copy-down-method>,
     meth: m,
     other-methods: others,
     source-location: m.model-source-location,
     subnotes: map(method (m)
                     make(<ambiguous-copy-down-method-option>,
                          meth: m,
                          source-location: m.model-source-location)
                   end,
                   others));
```

Note: Subnotes are not displayed by the default printing of program conditions by the command line compiler. They can be found in the condition log file that is created during the build process. (_build/build/foo/foo.log)

subnotes Generic function

Signature subnotes (object) => (value)

Parameters

- **object** – An instance of *<program-note>*.

Values

- **value** – An instance of *<program-notes>*.

note-during Macro

accumulate-subnotes-during Macro

***subnotes-queue* Thread Variable**

Printing Program Conditions

***detail-level* Thread Variable**

Type *<detail-level>*

Discussion

> **Note:** This is currently ignored.

<detail-level> Type

Equivalent `one-of(#"terse", #"normal", #"verbose")`

Discussion

A simple, three-tiered approach to the amount of detail a condition presents.

> **Note:** This is currently ignored.

Operations *format-condition*

format-condition Generic function

Signature format-condition (stream condition detail-level) => ()

Parameters

- **stream** – An instance of `<stream>`.
- **condition** – An instance of *<program-condition>*.
- **detail-level** – An instance of *<detail-level>*.

Discussion This calls `format` to write to the `stream`. The format string and arguments come from the condition's `condition-format-string` and `condition-format-arguments` respectively.

print-object(<program-condition>, <stream>) Method

Signature print-object (condition, stream) => ()

Parameters

- **condition** – An instance of *<program-condition>*.
- **stream** – An instance of `<stream>`.

Discussion

This calls *format-condition* with a *detail-level* of #"terse".

This is provided for integrating program condition printing with the usual mechanisms for formatted output.

Note: This is not actually called often at all as there is a more specific specialization on *<program-note>* defined in dfmc-debug-back-end.

Unclassified API

$record-program-note Constant

$signal-program-error Function

Signature $signal-program-error (c) => ()

Parameters

- **c** – An instance of <condition>.

$signal-program-note Function

Signature $signal-program-note (c) => ()

Parameters

- **c** – An instance of <condition>.

<ignore-serious-note> Class

Superclasses *<program-restart>*

Init-Keywords

- **format-string** –
- **note** –

<program-note-filter> Constant

convert-condition-slots-to-ppml Generic function

Signature convert-condition-slots-to-ppml (condition) => ()

Parameters

- **condition** – An instance of <condition>.

Discussion Converts all slots on a condition to their PPML representation. This is typically autogenerated by the various program condition definer macros. It is called from *add-program-condition*.

convert-condition-slots-to-ppml(<condition>) Method

convert-condition-slots-to-ppml(type-union(<simple-condition>, <simple-error>, <simple-warn

convert-condition-slots-to-ppml(<program-note>) Method

convert-condition-slots-to-ppml(<program-restart>) Method

convert-condition-slots-to-ppml(<program-warning>) Method

convert-condition-slots-to-ppml(<serious-program-warning>) Method

convert-condition-slots-to-ppml(<program-error>) Method

convert-condition-slots-to-ppml(<run-time-error-warning>) Method

convert-condition-slots-to-ppml(<style-warning>) Method

convert-condition-slots-to-ppml(<performance-note>) Method

convert-condition-slots-to-ppml(<portability-note>) Method

convert-condition-slots-to-ppml(<ignore-serious-note>) Method

convert-slots-to-ppml Macro

dfmc-continue Thread Variable

dfmc-restart Thread Variable

do-with-program-conditions Function

Signature do-with-program-conditions (body) => (#rest results)

Parameters

- **body** – An instance of <object>.

Values

- **#rest results** – An instance of <object>.

interesting-note? Generic function

Signature interesting-note? (note) => (interesting?)

Parameters

- **note** – An instance of *<program-note>*.

Values

- **interesting?** – An instance of <boolean>.

Discussion True if the note is interesting to the user, according to the yet-to-be-defined compiler policy object. Uninteresting conditions are suppressed, either by not printing messages for them or not logging them at all. Because all errors and restarts are *serious*, they are also interesting.

interesting-note?(<program-note>) Method

Parameters

- **note** – An instance of *<program-note>*.

Values

- **interesting?** – Always returns #t.

interesting-note?(<performance-note>) Method

Parameters

- **note** – An instance of *<performance-note>*.

Values

- **interesting?** – Always returns #f.

make-program-note-filter Generic function

Signature make-program-note-filter (#key file-name from to in class action) => (filter)

Parameters

- **file-name** (*#key*) – An instance of <string>.
- **from** (*#key*) – An instance of <integer>.
- **to** (*#key*) – An instance of <integer>.
- **in** (*#key*) – An instance of <string>.
- **class** (*#key*) – An instance of subclass(<condition>).
- **action** (*#key*) – An instance of <function>.

Values

- **filter** – An instance of *<program-note-filter>*.

obsolete-condition? Open Generic function

Signature obsolete-condition? (condition) => (obsolete?)

Parameters

- **condition** – An instance of *<program-condition>*.

Values

- **obsolete?** – An instance of <boolean>.

obsolete-condition?(<program-condition>) Method

Parameters

- **condition** – An instance of *<program-condition>*.

Values

- **obsolete?** – Always returns #f.

Discussion

Note: This is never used.

present-program-error Generic function

Signature present-program-error (condition) => ()

Parameters

- **condition** – An instance of <condition>.

present-program-error(<condition>) Method

present-program-error(<program-note>) Method

present-program-note Generic function

Signature present-program-note (condition) => ()

Parameters

- **condition** – An instance of <condition>.

present-program-note(<condition>) Method

present-program-note(<program-note>) Method

program-note-class-= Function

Signature program-note-class-= (class) => (pred)

Parameters

- **class** – An instance of `subclass(<condition>)`.

Values

- **pred** – An instance of `<function>`.

program-note-file-name-= Function

Signature program-note-file-name-= (file-name) => (pred)

Parameters

- **file-name** – An instance of `<string>`.

Values

- **pred** – An instance of `<function>`.

program-note-filter Open Generic function

Signature program-note-filter (class) => (filter)

Parameters

- **class** – An instance of `subclass(<condition>)`.

Values

- **filter** – An instance of *<program-note-filter>*.

program-note-filter(subclass(<program-note>)) Method

program-note-filter(subclass(<condition>)) Method

program-note-filter(subclass(<program-warning>)) Method

program-note-filter(subclass(<serious-program-warning>)) Method

program-note-filter(subclass(<run-time-error-warning>)) Method

program-note-filter(subclass(<style-warning>)) Method

program-note-filter(subclass(<performance-note>)) Method

program-note-filter(subclass(<portability-note>)) Method

program-note-filter-setter Open Generic function

Signature program-note-filter-setter (filter class) => (filter)

Parameters

- **filter** – An instance of *<program-note-filter>*.
- **class** – An instance of `subclass(<program-condition>)`.

Values

- **filter** – An instance of *<program-note-filter>*.

program-note-filter-setter(<program-note-filter>, subclass(<program-condition>)) Method

program-note-in Function

Signature program-note-in (form) => (pred)

Parameters

- **form** – An instance of `<string>`.

Values

- **pred** – An instance of <function>.

program-note-location-between Function

Signature program-note-location-between (from to) => (pred)

Parameters

- **from** – An instance of <integer>.
- **to** – An instance of <integer>.

Values

- **pred** – An instance of <function>.

report-condition Open Generic function

Signature report-condition (condition) => ()

Parameters

- **condition** – An instance of <condition>.

serious-note? Generic function

Signature serious-note? (note) => (serious?)

Parameters

- **note** – An instance of *<program-note>*.

Values

- **serious?** – An instance of <boolean>.

Discussion

True if this note is serious – that is, requires terminating the current processing and picking a restart. The default behavior is that notes are not serious, but the policy object should allow upgrading them, with options like *"all warnings are errors"* for making *<program-warning>* serious, or *"strict Dylan"* for making *<portability-note>* serious.

Errors are always serious, by definition, because the compiler can't just skip them. Restarts are always serious, as much as such a definition make sense for them.

serious-note?(<program-note>) Method

Parameters

- **note** – An instance of *<program-note>*.

Values

- **serious?** – Always returns `#f`.

serious-note?(<program-error>) Method

Parameters

- **note** – An instance of *<program-error>*.

Values

- **serious?** – Always returns `#t`.

serious-note?(<serious-program-warning>) Method

Parameters

- **note** – An instance of *<serious-program-warning>*.

Values

- **serious?** – Always returns `#t`.

simple-note Generic function

Signature simple-note (class format-string #rest args) => ()

Parameters

- **class** – An instance of `subclass(<program-note>)`.
- **format-string** – An instance of `<string>`.
- **args** (*#rest*) – An instance of `<object>`.

simple-raise Generic function

Signature simple-raise (class format-string #rest args) => ()

Parameters

- **class** – An instance of `subclass(<program-error>)`.
- **format-string** – An instance of `<string>`.
- **args** (*#rest*) – An instance of `<object>`.

with-program-conditions Macro

with-simple-abort-retry-restart Macro

Not yet fully updated:

Compiler Design (Old)

Adding a DFM computation

What you have to do to add a new node class to the DFM:

- **Add it to `flow-graph/computation.dylan`, and ensure that you export** it from `flow-graph/flow-graph-library`.
- Create the converters to generate it. Likely in conversion, but some nodes are only created by optimizations.
- Make sure all the back ends handle it. This includes, at least:
  - c-back-end
  - debug-back-end – the printer
  - all native back ends
- In addition, it would be good to add any invariant checks to `flow-graph/checker.dylan`.

DFM block constructs

bind-exit

First, let's look at an example of bind-exit.

```
block (exit) exit(42); 13 end; =>
  [BIND]
  t2 := [BIND-EXIT entry-state: &t1 body: L1 exit-to: L0]
  L1:
  t4 := ^42
  t12 := exit entry-state: &t1 value: t4
  t6 := ^13
  end-exit-block entry-state: &t1
  L0:
  t7 := [MERGE t2 t6]
  return t7
```

(That's before register assignment, to make the difference in the temporaries used in the merge node clear.)

The <bind-exit> node establishes the place the exit jumps to, an <entry-state>. This is communicated to <exit> and <end-exit-block> through the temporary t1. The temporary returned by the <bind-exit> is set by the exit procedure.

(The printing code shows up one inconsistency: the temporary generated by the <bind-exit> node is actually not live after that point. It's live only if the exit procedure is taken. On the other hand, the entry-state is live after that point. Perhaps which temporary is the generated one from a <bind-exit> node should be exchanged.)

The merge node combines the two temporaries that could contain the result of the <merge> node – t2 by exiting, t6 by falling through. The <end-exit-block> node exists for at least two purposes: to possibly bash the exit procedure or entry state in order to prevent calls outside of its dynamic scope and to stop a thread in the execution engine. It references the entry state in order that it can be found from the <bind-exit> node.

Before we see the compiled code, here's the DFM code after register allocation:

```
block (exit) exit(42); 13 end; =>
  [BIND]
  t2 := [BIND-EXIT entry-state: &t0 body: L1 exit-to: L0]
  L1:
  t1 := ^42
  t3 := exit entry-state: &t0 value: t1
  t2 := ^13
  end-exit-block entry-state: &t0
  L0:
  t2 := [MERGE t2 t2]
  return t2
```

And this is the C code:

```
block (exit) exit(42); 13 end; =>
  D L4988I () {
    D T0;
    D T2;
    D T1;
    D T3;

    T0 = dNprimitive_make_bind_exit_frame();
    if (setjmp(dNprimitive_frame_destination(T0))) {
      T2 = dNprimitive_frame_return_value(T0);
      goto L0;
    }
```

```
  L1:
    T1 = I(42);
    dNprimitive_nlx(T0, T1);
  L0:
    return(T2);
  }
```

The only gotcha (other than how setjmp works in C) is that the emission engine knows that there's no point in generating code for the stuff that follows an <exit> node; it's a primitive form of dead code elimination. So that's why the `t2 := ^13` and `<end-exit-block>` nodes are not emitted.

The call to dNprimitive_nlx unwinds all <unwind-protect> frames on the way back to the entry state marked by T0. Eventually, (unless some cleanup calls another exit procedure) it will longjmp to the site of the setjmp. The second argument to dNprimitive_nlx is shoved into the dNprimitive_frame_return_value of the entry state.

On the other hand, if we omit the call to the exit procedure (or if there's some control flow path which falls through, or if it isn't inlined, as it was above), the generated code is:

```
block (exit) 13 end; =>
  D L1502I () {
    D T0;
    D T1;

    T0 = dNprimitive_make_bind_exit_frame();
    if (setjmp(dNprimitive_frame_destination(T0))) {
      T1 = dNprimitive_frame_return_value(T0);
      goto L0;
    }
  L1:
    T1 = I(13);
    /* invalidate T0 */
  L0:
    return(T1);
  }
```

Note that the call just falls through from the assignment to T1 to the return; no jump need take place.

The comment about invalidating reflects something I think we should do, but haven't done yet, which is ensure that the exit procedure is bashed when we leave the block. Bashing a single slot should be sufficient.

unwind-protect

Now, let's consider the DFM code for an unwind-protect:

```
block () xxx() cleanup yyy() end; =>
  [BIND]
  [UNWIND-PROTECT entry-state: t0 body: L1 cleanup: L2 next: L0]
  L1:
  t1 := ^xxx
  t2 := [CALLx t1()]
  end-protected-block entry-state: t0
  L0:
  return t2
  L2:
  t3 := ^yyy
  [CALLx t3()]
  end-cleanup-block entry-state: t0
```

I think this code is pretty straight-forward, at least in terms of the data flow graph. Note that t2 is live in the code outside the block statement.

```
block () xxx() cleanup yyy() end; =>
  D L2437I () {
    D T0;
    D T1;
    D T2;
    D T3;

    T0 = dNprimitive_make_unwind_protect_frame();
    if (setjmp(dNprimitive_frame_destination(T0)))
      goto L2;
  L1:
    T1 = dNxxx;
    T2 = CALL0(T1);
  L2:
    T3 = dNyyy;
    CALL0(T3);
    dNprimitive_continue_unwind();
  L0:
    return(T2);
  }
```

The `dNprimitive_continue_unwind` just returns in this case. If the cleanup clause were invoked by an exit procedure, it would have set a flag in the frame indicating that it continues non-local-exiting. The important thing to see is that the decision about whether to fall through from the cleanup clause into the code outside the block is made by `dNprimitive_continue_unwind`, based on dynamic information.

Final notes

Finally, note that a block with both an exit procedure (bind-exit) and a cleanup clause (unwind-protect) is simply a bind-exit wrapped around an unwind-protect.

Optimizations

Lots of optimizations can be done. Off the top of my head:

- Code following an `<exit>` is dead; it should be dead-code eliminated in the DFM.
- If an `<exit>` is inlined and there are no `<unwind-protect>` nodes between it and the `<bind-exit>`, it can be turned into a control transfer.
- If there are no `<exit>` nodes for a given `<entry-state>`, the `<bind-exit>` node can be removed.

An invalid optimization that had been suggested was to merge nested `<unwind-protect>` nodes without intervening `<bind-exit>` nodes with a test in the merged cleanup to determine whether the inner cleanup is still active. This isn't valid because then the inner cleanup is no longer protected by the outer cleanup.

DFM local assignment

We really want the DFM to be a *single assignment* form. That is, all temporaries should be defined and then never mutated. We want this because it makes many optimizations (common sub-expression elimination, inlining, etc) significantly easier. See the usual set of SSA papers for details; I can dig up references.

On the other hand, Dylan has assignment to locals, and we model locals with temporaries. Since the DFM doesn't have cycles (loops), we could replace assignments *to variables which aren't closed over* with new temporaries, in the

same way as SSA code is usually generated. But all the interesting cases in Dylan are when assigned variables are closed over, especially because they're assigned to in loop bodies.

Instead, based on Keith's suggestion, I map our Dylan-esque DFM into one that matches how ML, at the language level, with references (mutable variables): all temporaries which are assigned to are replaced with temporaries referring to boxed values.

The current approach:

I introduced three primitives:

```
make-box t => box              // create a box, containing t
get-box-value box => t         // return the value inside the box
set-box-value! box t => t      // set the value inside the box
```

There is a new compiler pass (eliminate-assignments) which traverses a DFM graph and does the rewriting.

Here's an example of what happens:

```
begin let a = 13; a := 42; a end; => // before
  [BIND]
  t0 := ^13
  t1 := ^42
  @a := t1
  return t0

begin let a = 13; a := 42; a end; => // after
  [BIND]
  t0 := ^13
  t1 := [PRIMOP primitive-make-box(t0)]
  t2 := ^42
  [PRIMOP primitive-set-box-value!(t1, t2)]
  t3 := [PRIMOP primitive-get-box-value(t1)] // tail call
  return t3
```

The eliminate-assignments pass should happen before any of the *interesting* optimizations, and should never need to be done twice on the same piece of code.

What remains to be done:

We probably want to turn these primitives into DFM computations before trying to do any optimizations on them.

make-box currently allocates the boxed cell in the heap. It should really allocate the cell either a closure or stack frame, depending on whether the box has dynamic extent. If the temporary the box is bound to (t1 in the example above) is only used as with get-box-value and set-box-value!, then we know that the box has the same extent as that temporary. I'm don't think that all optimizations will preserve that property, but it will probably be maintained most of the time.

When we have temporaries which aren't closed over, most of the time we should be able to do SSA-like elimination of assignments, rewriting them by introducing new temporaries. For example, assignment inside a conditional can produce something like this

```
begin let a = 1; if (p?) a := 2 else end; a end; =>
  [BIND]
  t2 := ^1
  t8 := [PRIMOP primitive-make-box(t2)]
  t9 := ^p?
  if (t9) goto L1 else goto L2
  L1:
  t13 := ^2
  t11 := [PRIMOP primitive-set-box-value!(t8, t13)]
  L0:
```

```
  [MERGE t11 t14]
  t10 := [PRIMOP primitive-get-box-value(t8)] // tail call
  return t10
  L2:
  t14 := ^&#f
  goto L0
```

but that should be easy to turn into

```
[BIND]
t1 := ^p?
if (t1) goto L1 else goto L2
L1:
t2 := ^2
L0:
t4 := [MERGE t2 t3]
return t4
L2:
t3 := ^1
goto L0
```

This sort of optimization, in the absence of cycles, is pretty easy. It may be more work making it happen for loops built up from tail calls, but still not as bad as SSA conversion in general.

DFM multiple values

To represent multiple values, there's a new temporary class in the DFM, <multiple-value-temporary>. Multiple values temporaries are not interchangeable with other temporaries; maybe we should introduce a <simple-temporary> class for non-multiple-value temporaries, but we can do that later. In the debugging print code, MV temporaries print with a * in front of them.

A multiple value temporary is the result of any computation which can produce multiple values, notably a call.

In order to produce efficient code, we have imposed the requirement that at most one MV temporary is live at a time (per thread). This allows us to allocate space for all MV temporaries ahead of time, as part of the calling convention, in the *multiple value area*. It is generally best to think of the multiple value area, which is used to pass multiple values across calls, as a single multiple valued register, which we allocate to the live MV temporary.

When there really is more than one live MV temporary, we must spill and unspill uses. One of the important optimizations is to reduce these spills when the number of values in a MV temporary is known, by extracting them into normal temporaries and repackaging them as an MV temporary when needed as one.

A multiple value temporary has slots which describe the number of required values and whether there are rest values. Types need to be incorporated here, just as with other temporaries. There's also a slot for a normal temporary, which is used when spilling the multiple value temporary.

To manipulate multiple values, there are five new computation classes:

<values>

super: <computation> slots: fixed-values, rest-value

Creates a `<multiple-value-temporary>` from a set of single value temporaries. For now, a `<values>` node comes from a converter for the *function macro* values; in the future, there should be only one `<values>` node created directly, and the rest created by inlining the function values from the Dylan library. (A similar change needs to be made for `<apply>`.)

```
values(1, 2, 3) =>
  [BIND]
```

```
  t0 := ^1
  t1 := ^2
  t2 := ^3
  *t3 := [VALUES t0 t1 t2]
  return *t3
```

<extract-single-value>

super: <computation> slots: multiple-values, index, rest-vector?

Produces a single-valued temporary from an MV temporary. The index is used to select which multiple value is extracted; the indices are numbered from 0. If rest-vector? is true, a vector of the values from index on is returned, rather than just the index. (Perhaps that should be a different <computation> class.)

These very commonly follow calls, extracting the single value. They should also appear based on optimizations of let bindings.

```
f(g()) =>
  [BIND]
  t0 := ^f
  t1 := ^g
  *t2 := [CALLx t1()]
  t3 := *t2 [0]
  *t4 := [CALLx t0(t3)] // tail call
  return *t4
```

<multiple-value-call>

super: <function-call>

Like an <apply> with no fixed arguments and a MV temporary as the single (last) argument. Constructed from `let` declarations which bind multiple values. (This could be used for all lets, but I wanted to wait with that until the multiple value optimizations were in place.)

The most important optimization with these nodes is to upgrade the calls to <simple-call> or <apply> with the shape of the MV temporary argument is know. If it's not known, the simplest code generation strategy is to extract all of the temporary values and transform the call into an <apply>.

```
begin let (a, b) = f(); g(a, b) end =>
  [BIND]
  t3 := ^[XEP lambda 741 [743] (a, b)
    [BIND]
    t0 := ^g
    *t1 := [CALLx t0(a, b)] // tail call
    return *t1
  end lambda]
  t0 := ^f
  *t1 := [CALLx t0()]
  *t2 := [MV-CALLx t3(*t1)] // tail call
  return *t2
```

<multiple-value-spill> <multiple-value-unspill>

super: <temporary-transfer>

These instructions turn an MV temporary into a single-value temporary and vice-versa, for the purpose of maintaining the property that a single MV temporary is live at a time. As much as possible, we should try to avoid these instructions in generated code, which can be done when we know we're dealing with a fixed number of values.

These computations are only generated by the mandatory compiler pass `spill-multiple-values`, which should run after all optimizations have happened. (The reason that it should run afterwards is the spill code can defeat other optimizations and other optimizations can get rid of the need to spill.)

```
block () f() afterwards g() end =>
  [BIND]
  t0 := ^f
  *t1 := [CALLx t0()]
  t3 := [MV-SPILL *t1]
  t2 := ^g
  [CALLx t2()]
  *t4 := [MV-UNSPILL t3]
  return *t4
```

The reason the spill is needed is that the call to g tramples over the multiple value area.

In the C run time, there's an extra data structure, MV, as follows:

```
typedef struct _mv {
  int count;
  D   value[VALUES_MAX];
} MV;
```

There's one global such thing (Preturn_values), and one per bind-exit or unwind-protect frame, used for the return value that's being passed around. The ones that live in those frames should probably be shortened to some small number of values (2? 4? 8?) and evacuate to the heap if more multiple values are stored; it's pretty rare, I expect, for a large number of values to appear in an unwind-protect frame, or to be passed back with an exit procedure.

The C code generated for all of these is pretty stupid right now, calling out to primitives in all cases, so I won't bother to present it. I want to get to the task of optimizing multiple values soon. I think that a little bit of optimization will go a long way here.

In the native run-time, we'll pass the first few multiple values and (if there is one) the count in registers. Tony can describe that far better than I can.

define compilation-pass macro

NOTE: this is currently not used at all - it had been dropped before going open source, but in general I (hannes) believe it is a good idea (and plan to revive it), thus I keep the documentation.

I've now replaced the old mechanism for specifying compilation passes in the DFM compiler (setting the vector *compilation-passes* in compile.dylan) with a declarative system, based around a macro, define compilation-pass.

The macro is exported by dfmc-common, so every module should have it. The basic idea is that you put a compilation-pass definition in the same place as you define the main entry point for a compiler-pass; the definition includes things about the pass, such as when its run, how it is called, and if it should cause other passes to run.

First, a simple example:

```
define compilation-pass eliminate-assignments,
  visit: functions,
  mandatory?: #t,
  before: analyze-calls;
```

This defines a pass named *eliminate-assignments*, which runs before analyze-calls is run; it is possible to use arbitrarily many before: options. The mandatory option declares that the pass is part of optimization level 0; that is, it's always run.

The *visit: functions* option says that the function is called for every function in the form being compiled. The default is *visit: top-level-forms*, which corresponds to the previous behavior.

```
define compilation-pass try-inlining,
  visit: computations,
  optimization: medium,
  after: analyze-calls,
  before: single-value-propagation,
  triggered-by: analyze-calls,
  trigger: analyze-calls;
```

The *visit: computations* option says that every computation (in the top-level and all nested lambdas) is passed to the pass's function. The after: option is like before: in reverse.

The trigger: option runs the named pass if the pass being defined reports that it changed anything. If the triggered pass has already run, then it is queued to run again; if the triggered pass is disabled or of a higher optimization level than currently being used, it's not run. Triggered-by: is trigger: in reverse.

A pass function reports that it changed something by returning any non-false value.

Full catalog of options:

visit: What things to pass to the pass's function: top-level-forms Just the top-level function. functions Every function. computations Every computation in every function.

optimization: What level of optimization to run this pass for? (Choices: mandatory, low, medium, high.)

mandatory?: Always run this pass; overrides optimization:.

before: Run this pass before the named one. after: Run this pass after the named one.

trigger: If this pass changed something, run the named pass. triggered-by: If the named pass changes something, run this pass.

print-before?: Print the DFM code before calling the pass. print-after?: Print the DFM code after the pass is done. print?: Same as print-before?: #t and print-after?: #t.

check-before?: Call ensure-invariants before calling the pass. check-after?: Call ensure-invariants after the pass is done. check?: Same as check-before?: #t and check-after?: #t.

back-end: Turn pass on for the named back end. (Default: all) exclude-back-end: Turn pass off for the named back end. (Default: none.)

disabled?: Turn pass off; overrides everything else.

Convenience functions:

trace-pass(pass-name) untrace-pass(pass-name)

Turns on (or off) printing and checking (both before and after) for the pass.

untrace-passes()

Calls untrace-pass for all traced passes.

Global state:

The thread-variable `*back-end*` is used with the options back-end: and exclude-back-end:.

The thread-variable `*trace-compilation-passes*` will print a message about each pass as it runs, and report when one pass triggers another.

Compiler Internals (Old)

Introduction

This chapter is an overview of the libraries involved during compilation, information was gathered while hacking on the compiler. It focuses only on DFMC, the Dylan Flow Machine Compiler, located in `sources/dfmc` of the opendylan repository.

But first look how to get there: let's consider the command-line compiler, invoked with

```
dylan-compiler -build hello-world
```

There is a fairly long call chain to perform the compilation, starting with command-line parsing in sources/environment/console/command-line.dylan, which looks like this:

```
do-execute-command(..., <build-project-command>)
  (defined in sources/environment/commands/build.dylan)
build-project
  (defined in sources/environment/dfmc/projects/projects.dylan)
compile-library
parse-and-compile
  (defined in sources/project-manager/projects/compilation.dylan)
parse-project-sources
  (defined in sources/dfmc/management/definitions-driver.dylan)
compile-project-definitions
  (defined in sources/dfmc/browser-support/glue-routines.dylan)
compile-library-from-definitions
  (defined in sources/dfmc/management/world.dylan called here
  under the name ``dfmc-compile-library-from-definitions`` due
  to a prefixed import.).
```

To explain these long call chains, we need some more understanding of the different libraries of the compiler:

- **environment** is the public API
- **project-manager** is a bunch of hacks for
  - finding the project source (via the registry)
  - calling the compiler and linker (to create a dll/so and executable) afterwards.
- **dfmc/browser-support** and **environment/dfmc** are the glue from environment to DFMC.

The big picture is pretty simple: **management** drives the different libraries, some are the front-end (**reader**, **macro-expander**) translating into definitions; some intermediate language (**conversion**, **optimization**, **typist**) which work on the flow-graph; others are back-end, including **linker**. There is some support needed for the actual runtime, which is sketched in **modeling** (parts of which are put into the dylan runtime library), **namespace** (which handles namespaces and defines the dylan library and its modules).

First we need to introduce some terminology and recapitulate some conventions:

- the unit of compilation is a single dylan library
- the metadata of a library is stored in DOOD, the dylan object-oriented database
- loose (development) vs tight (production): loose mode allows runtime updates of definitions, like adding generic function into a sealed domain, subclassing sealed classes - production mode has stricter checks
- batch compilation: when invoked from command line, or building a complete library
- interactive compilation: IDE feature to play around, adding a single definition to a library

DFMC is well structured, but sadly some libraries use each other, which they shouldn't (typist, conversion, optimization).

In the remainder of this guide, we will focus on a simple example, which prints `Hello` x times:

```
define method hello-world (x :: <integer>) => ()
  do(curry(format-out, "Hello %d\n"), range(to: x))
end
```

dfmc-management

The library dfmc-management drives the compilation process, prints general information about what is happening at the moment (progress, warnings) and takes care of some global settings like opening and closing source records, etc.

The main external entry point is `compile-library-from-definitions` in world.dylan. This requires that the source has already been parsed (really? but it calls compute-library-definitions itself!).

It then calls in sequence `compute-library-definitions`, `ensure-library-compiled`, `ensure-library-glue-linked`, `ensure-library-stripped` and `ensure-database-saved`, apart from console output (warnings, stats).

The very first method, `compute-library-definitions`, calls `ensure-library-definitions-installed`, which calls `update-compilation-record-definitions`, which mainly calls `compute-source-record-top-level-forms`. This passes the compilation record to `read-top-level-fragment` to get a fragment and then calls `top-level-convert-forms`.

The reader library defines the `read-top-level-fragment`, the definitions library the `top-level-convert-forms`. Thus, a fragment is read and converted into definitions.

The method `ensure-library-compiled` computes, finishes and checks the data models. Afterwards the code models are generated (the control flow graph), then type inference is done and the optimizer is run. Finally the heaps are generated. These are methods defined in `compilation-driver.dylan`, calling out to the modeling, conversion, flow-graph, typist and optimizer libraries.

Finally the glue is emitted (in `back-end-driver.dylan`) and the database is saved, which contains metadata of each library (like type information, code models, etc.).

Some global warnings for libraries are defined and checked for in the management library.

The unused file `interface.dylan` compares module and binding definitions, in order to judge whether the public API of a library/module has changed between two versions. Usage of this would allow lazy recompilation: only recompile if the API has changed of a linked library.

dfmc-reader

This library reads the Dylan source, tokenizes it, annotates source locations, and builds a parse tree. The parser is in `parser.dylgram`, which uses `app/parser-compiler` to generate `infix-parser.dylan`. The API used is `read-top-level-fragment` and `re-read-fragments`.

Error handling is on the token level, thus a mismatched `end` is noticed. Other sorts of errors are invalid token, integer too large, character too large, ratios not being supported, end of input (while more tokens were required).

Every `<fragment>`, the base class of the abstract syntax tree, has a compilation-record and a source-position.

So, for the above hello-world method, `read-top-level-fragment` returns the following parse tree:

```
<body-definition-fragment>:
  fragment-macro: <simple-variable-name-fragment>
  fragment-name: #"method-definer"
```

```
  fragment-modifiers: #()
  fragment-body-fragment:
    <simple-variable-name-fragment>:
      fragment-name: #"hello-world"
    <parens-fragment>:
      fragment-left-delimiter: <lparen-fragment>
      fragment-nested-fragments:
        <simple-variable-name-fragment>:
          fragment-name: #"x"
        <colon-colon-fragment>
        <simple-variable-name-fragment>:
          fragment-name: #"<integer>"
      fragment-right-delimiter: <rparen-fragment>
    <simple-variable-name-fragment>:
      fragment-name: #"do"
    <parens-fragment>:
      fragment-left-delimiter: <lparen-fragment>
      fragment-nested-fragments:
        <simple-variable-name-fragment>:
          fragment-name: #"curry"
        <parens-fragment>:
          fragment-left-delimiter: <lparen-fragment>
          fragment-nested-fragments:
            <simple-variable-name-fragment>:
              fragment-name: #"format-out"
            <comma-fragment>
            <string-fragment>:
              fragment-value: "Hello %d\n"
          fragment-right-delimiter: <rparen-fragment>
        <comma-fragment>
        <simple-variable-name-fragment>:
          fragment-name: #"range"
        <parens-fragment>:
          fragment-left-delimiter: <lparen-fragment>
          fragment-nested-fragments:
            <keyword-syntax-symbol-fragment>:
              fragment-value: #"to"
            <simple-variable-name-fragment>:
              fragment-name: #"x"
          fragment-right-delimiter: <rparen-fragment>
      fragment-right-delimiter: <rparen-fragment>
    <semicolon-fragment>
```

NB: the type hierarchy for <body-definition-fragment> is: <definition-fragment>, <macro-call-fragment>, <compound-fragment>, <fragment>, <object>

dfmc-definitions

Once the abstract syntax tree is generated (by the reader), it's time to convert this into definitions, which are the names in dylan. There are several top-level definitions in dylan, namely: binding, class, constant, (copy-down), domain, function, generic, macro, method, module, namespace (library) and variable. Every definition has its own class, inheriting from `<top-level-form>` (defined in common/top-level-forms.dylan). A top level form at least contains information about its compilation record, source location, parent form, sequence number and dependencies and referenced variables. Additional information available are adjectives, the word defined, its library, original library, top level methods.

As a side note, dependency tracking is also defined in `common/top-level-forms.dylan`.

The main entry point for the definition library is `top-level-convert(parent, fragment)`, defined in `top-level-convert.dylan`.

The building of definition objects relies heavily on the macro-expander, especially on procedural macros described in D-Expressions: Lisp Power, Dylan Style. Open Dylan extends the definitions with compiler, optimizer, primitive and shared-symbols, mainly used internally in the compiler.

Looking into `define-method.dylan`, we can see a class `<method-definition>`. This is built by the parser, more specifically there is a `define &definition method-definer`, which has two rules to match fragments, whereas the second rule is the error case. The first matches any `define method` syntax and calls `do-define-method` with the arguments. The method `do-define-method` defers the work to helper methods `parse-method-adjectives` and `parse-method-signature`, and instantiates a `<method-definition>` object.

For our hello-world example, `do-define-method` creates a single object:

```
<method-definition>
  private-form-body: <body-fragment>
    fragment-constituents: <prefix-call-fragment>
      fragment-arguments:
        <prefix-call-fragment>
          fragment-arguments:
            <simple-variable-name-fragment>
              fragment-name: #"format-out"
            <string-fragment>
              fragment-value: "Hello %d\n"
          fragment-function: <simple-variable-name-fragment>
            fragment-name: #"curry"
        <prefix-call-fragment>
          fragment-arguments:
            <keyword-syntax-symbol-fragment>
              fragment-value: #"to"
            <simple-variable-name-fragment>
              fragment-name: #"x"
          fragment-function: <simple-variable-name-fragment>
            fragment-name: #"range"
      fragment-function: <simple-variable-name-fragment>
        fragment-name: #"do"
  private-form-signature: <method-requires-signature-spec>
    private-spec-argument-next-variable-specs: <next-variable-spec>
      private-spec-variable-name: <simple-variable-name-fragment>
        fragment-name: #"next-method"
    private-spec-argument-required-variable-specs: <typed-required-variable-spec>
      private-spec-type-expression: <simple-variable-name-fragment>
        fragment-name: #"<integer>"
      private-spec-variable-name: <simple-variable-name-fragment>
        fragment-name: #"x"
  private-form-signature-and-body-fragment: <sequence-fragment>
    <parens-fragment>, <simple-variable-name-fragment>, <parens-fragment>, <semicolon-fragment>
  private-form-variable-name-or-names: <simple-variable-name-fragment>
    fragment-name: #"hello-world"
```

It is noteworthy that still no intra-library information is present, this is top-level Dylan code without any context. All macros are expanded.

Excursion into run-time and compile-time

Some objects are defined in the compiler, but are injected into the Dylan world. How does this happen?

In the Dylan library you see `// BOOTED:` comments here and there. The source location of well-known basic types and functions is `dylan:dylan-user:boot-dylan-definitions`.

There is no definition of this specific method.

The method `boot-definitions-form?` (in dfmc-definitions) checks exactly for this name. The method `top-level-convert-forms` behaves differently if `boot-definitions-form?` returns true, namely it calls `booted-source-sequence` (in boot-definitions.dylan) which grabs the boot-record and returns it sorted as a vector.

But what is a boot-record after all? Well, its definition is all in boot-definitions.dylan, with the explanation "records the set of things that must be inserted into a Dylan world at the very start. Some things are core definitions, such as converters and macros, and these are booted at the definition level. The rest are expressed as source to be fed to the compiler."

The constant `*boot-record*` is filled by do-define-core-*. These are called by **dfmc-modeling**. Namely, primitives (which names and signatures are installed), macros, modules, libraries, classes.

Be aware that the actual implementation of the primitives is in the runtime (either `sources/lib/run-time/run-time.c` or the runtime-generator generates a runtime.o containing those definitions), but some crucial bits, like the adjectives (`side-effect-free`, `dynamic-extent`, `stateless` and `opposited`) are in **dfmc-modeling** and are used in the optimization!

The core classes are emitted from modeling with actual constructors. Be aware that the runtime layout is also recorded in run-time.h.

The dylan library and module definitions are in modeling/namespaces.dylan.

A noteworthy comment is that a compiler (comp-0, generation 0) loads the Dylan library (dylan-0), which contains the definitions (defs-0). When compiling itself (comp-1), first a fresh Dylan library (dylan-1) is built, which contains still the old booted definitions (defs-0). It emits new definitions (defs-1) and a new boot-record when dumping dfmc-definitions. Now the next generation compiler (comp-1) will use these new definitions in the next Dylan (dylan-2) library. Beware of dragons.

dfmc-macro-expander

The deep magic happens here.

dfmc-convert

Converts definition objects to model objects. In order to fulfill this task, it looks up bindings to objects from other libraries. Also converts the bodies of definitions to a flow graph. Does some initial evaluation, for example `limited(<vector>, of: <string>)` gets converted to a `<&limited-vector-type>` instance. Thus, it contains a poor-mans eval.

Also, creates init-expressions, which may be needed for the runtime. Since everything can be dynamic, each top-level form may need initializing. This happens when the library is loaded.

Also sets up a lexical environment for the definitions, and checks bindings.

Here, type variables are now recorded into the lexical environment, the type variables are passed around while the signature is checked.

After Dylan code is converted, it is in a representation which can be passed to a backend to generate code. Modeling objects have corresponding compile and run time objects, and are prefixed with an ampersand, e.g., `<&object>`.

dfmc-modeling

Contains modeling of runtime and compile time objects. Since some calls are attempted at compile time rather than at runtime, it provides these compile time methods with a mechanism to override the runtime methods (`define &override-function`). An example for this is `^instance?`, compile time methods are prefixed with a `^`, while compile and runtime class definitions are prefixed with `&`, like `define &class <type>`.

Also, DOOD (a persistent object store) models and proxies for compile time definitions are available in this library, in order to load definitions of dependent libraries.

dfmc-flow-graph

The flow graph consists of instances of the `<computation>` class, like `<if>`, `<loop-call>`, `<assignment>`, `<merge>`. The flow graph is in a (pseudo) static single assignment (SSA) form. Every time any algorithm alters the flow graph, it disconnects the deprecated computation and inserts new computations. New temporaries are introduced if a binding is assigned to a new value. Subclasses of `<computation>` model control flow, `<temporary>` (as well as `<referenced-object>`) model data flow.

Computations are a doubly-linked list, with special cases for merge nodes, loops, if, bind-exit and unwind-protect. Every computation may have a computation-type field, which is bound to a `<type-variable>`. It also may have a temporary slot, which is its return value. Several cases, single and multiple return values, are supported. The temporary has a link to its generator, a list of users and a reference to its value.

Additional (data flow) information is kept in special slots, test in `<if>`, arguments of a `<call>`, etc. These are all `<referenced-object>`, or more specially `<value-reference>`, `<object-reference>`, etc. `<object-reference>` contains a binding to its actual value.

`<temporary>` and `<environment>` classes are defined in this library.

`join-2x1` etc. are the operations on the flow graph.

dfmc-typist

This library contains runtime type algebra as well as a type inference algorithm.

Main entry point is `type-estimate`, which calls `type-estimate-in-cache`. Each library contains a type-cache, mapping from method definitions, etc. to type-variables.

Type variables contain an actual type estimate as well as justifications (supporters and supportees), used for propagation of types.

converts types to `<type-estimate>` objects

`type-estimate-function-from-signature` calls `type-estimate-body` if available (instead of using types of the signature), call chain is `type-estimate-call-from-site` -> `type-estimate-call-stupidly-from-fn` -> `function-valtype`

contains hard-coded hacks for `make`, `element`, `element-setter` (in `type-estimate-call-from-site`)

typist/typist-inference.dylan:poor-mans-check-type-intersection if #f (the temp), optimizer has determined that type check is superfluous

dfmc/typist-protocol.dylan:151 - does not look sane!

```
define function type-estimate=?(te1 :: <type-estimate>, te2 :: <type-estimate>)
 => (e? :: <boolean>, known? :: <boolean>)
  // Dylan Torah, p. 48: te1 = te2 iff te1 <= te2 & te2 <= te1
  let (sub?-1, known?-1) = type-estimate-subtype?(te1, te2);
  let (sub?-2, known?-2) = type-estimate-subtype?(te1, te2);
```

dfmc-optimization

This library contains several optimizations: dead code removal, constant folding, common subexpression elimination, inlining, dispatch upgrading and tail call analysis.

Main entry point from management is `really-run-compilation-passes`. This loops over all lambdas in the given code fragment, converts assigned variables to a `<cell>` representation, renames temporaries in conditionals, then runs the "optimizer". This builds an optimization queue, initially containing all computations. It calls do-optimize on each element of the optimization-queue, as long as it returns `#f`. (The protocol is that if an optimization was successful, it returns `#t`, if it was not successful, `#f`). For different types of computations different optimizations are run. Default optimizations are deletion of useless computations and constant folding. `<bind>` is skipped, for `<function-call>` additionally upgrade (analyzes the call, tries to get rid of gf dispatch) and inlining is done. `<primitive-call>` are optimized by `analyze-calls`.

constant folds (constant-folding.dylan):

```
   // The following is because we seem to have a bogus class hierarchy
   // here 8(
   // We mustn't propagate a constraint type above its station, since
   // the constraint is typically local (true within a particular
   // branch, say).
   & ~instance?(c, <constrain-type>)

optimization/dispatch.dylan: gf dispatch optimization
```

optimization/assignment: here happens the "occurrence typing" (type inference for instance?)... `<constrain-type>` is only for the instance? and conditionals hack

CHAPTER

SIX

THE RUNTIME

Originally written by:

Tony Mann, Jonathan Bachrach at Harlequin, Ltd.

4 December 1995

Object Representation

Harlequin's implementation achieves dynamic typing of Dylan objects by associating the type with an object based on tagging.

In many circumstances, the Dylan compiler can statically determine the type of an object. This knowledge can be used to select an alternative representation which is more efficient than the canonical representation. For example, the canonical representation of a double float object in Dylan is as a pointer to heap-allocated storage which contains the IEEE bit pattern of the double float in addition to a reference to the Dylan class object `<double-float>`. The compiler may choose to represent the value as a direct bit pattern, wherever this does not violate the semantics of the program.

Tagging Scheme

All Dylan values are represented as data of the same size, the size of a pointer. The bit pattern of these values contains tag bits which indicate whether the value is actually a pointer, or whether it is a direct value. Since there are three major groups (integers, characters and everything else), the representation for all platforms is to use two bits.

Tag Bits	Type
00	Heap Allocated
01	Integers
10	Characters
11	Unused

Integers and Characters

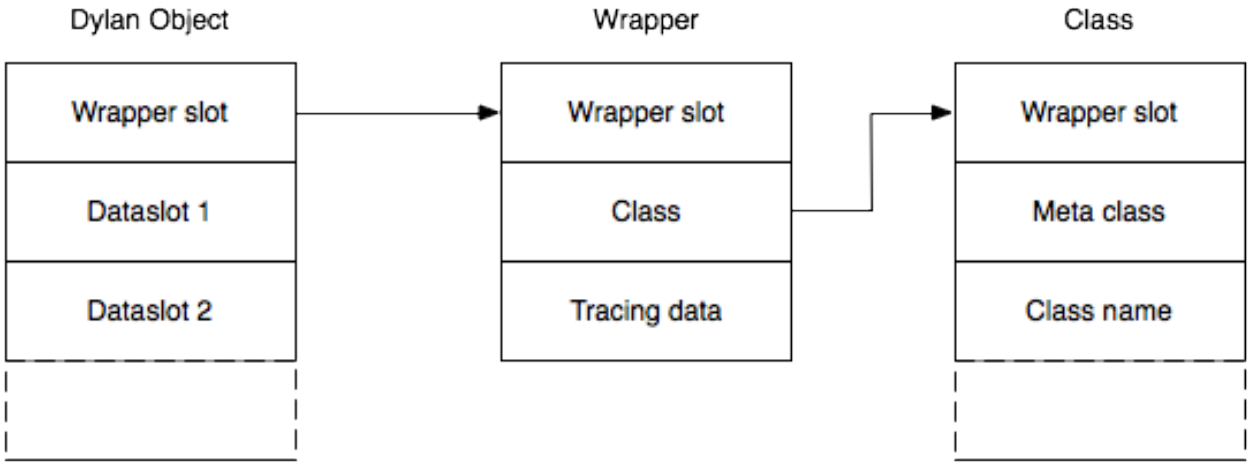
Integers and characters are represented as direct values, using the tag bits as the only indication of type. The tagging scheme uses the least significant two bits. With this scheme, a character or integer is converted to its untagged representation by arithmetic right shifting by two bits. Similarly the conversion from an untagged to a tagged representation is to shift left and add in the tag bits.

Operations on these values (e.g., addition, or other arithmetic operations) are always performed on the untagged representation. This is sub-optimal, because it is possible to perform arithmetic operations directly on the tagged values. It is planned to improve this mechanism at a later date, along with a revision of the tagging scheme.

Boxed Objects

Apart from integers and characters, all Dylan objects are indirectly represented as *boxed* values (that is, they are pointers to heap allocated boxes). The runtime system is responsible for ensuring that these boxed values are appropriately tagged, because the runtime system provides the allocation service, and must ensure appropriate alignment.

Boxed objects are dynamically identified by their first slot, which is an identification wrapper. This identification wrapper (itself a boxed Dylan object) contains a pointer to the class of the object it is wrapping, as well as some encoded information for the garbage collector about which slots should be traced.



Boxed Objects

Note that two indirections are necessary to find the class of an object. In practice, this is a rare operation, because almost all dynamic class testing within Dylan is implicit, and the implementation can use the wrapper for these implicit tests. Note that there is potentially a many-to-one correspondence between wrapper objects and class objects.

The Dylan compiler builds literal boxed objects statically whenever it can. In practice, this will include most function objects apart from closures, virtually all wrappers, and most class objects, as well as strings, symbols, and literal vectors and lists.

Variably Sized Objects

Variably sized objects, such as strings, vectors and arrays, are boxed objects which contain a *repeated slot*. The repeated slot is implemented as a variably sized data area preceded by a normal slot containing the size of the variably sized data represented as a tagged integer. The size slot is used at the Dylan language level to determine the size of the array. There is also a special encoding for it in the tracing data of the wrapper so that the memory manager knows how to trace the repeated data. For example, an instance of the `<byte-string>` *"foo"* is represented as in:.

Function Objects

Dylan provides two built-in classes of functions: `<generic-function>` and `<method>`. These both obey the same general purpose calling convention, but also support specialized calling conventions (described below) which the compiler may use depending on the detail of its knowledge about the function being called, and the circumstances. Slots in the function object point to the code which implements each convention.

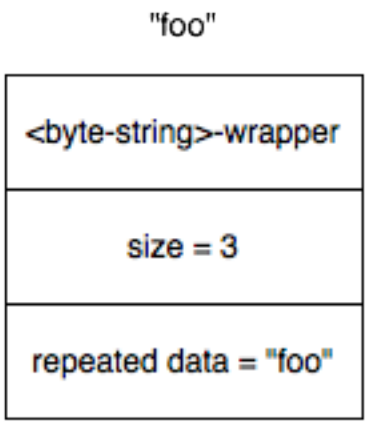


Fig. 6.1: An Instance of `<byte-string>`

All functions also have a slot which encodes the number of required parameters the function accepts, and whether the function accepts optional or keyword parameters. Another slot in each function object contains a vector of the types which are acceptable for each required parameter. These slots are used for consistency checking of the arguments.

Generic functions have further slots which support the method dispatching process — including a slot which contains a vector of all the methods belonging to the generic function, and a slot which contains a cache of sorted applicable methods for combinations of arguments which have been processed before.

Methods may be closures, in which case a slot in the method object contains the environment for the method, which is represented as a vector of closed-over variables. If the variable is known actually to be constant, then the constant value is stored directly in the vector. Alternatively, if there is any possibility of an assignment to the variable, then the value is stored with an extra indirection to a *value cell*, which may be shared between many closures with related environments.

Calling Convention

Some terminology

Arguments passed to a function at the implementation level fall into 2 different groups. *Language parameters* correspond to the explicit arguments in the source code. *Implementation parameters* correspond to the house-keeping information used by the implementation.

The overall calling convention consists of several specific conventions with different properties, described below. Each convention is implemented by a separate *entry point*. There are partial orderings between the entry points for these conventions, depending on how specific each one is. The code which implements a control flow from one entry point to the next may be obliged to rearrange parameters (e.g. on the stack). This process is called *stack fixing*.

The register model

Three registers are used within the calling convention to support the passing of *implementation parameters*: Note that for the C backend, global (or thread-local) variables might be used instead of real registers to pass these parameters.

Register	Purpose
arg-count	number of args passed
function	the `<function>` object being called
mlist	the next-method list (`#f` for direct-entry)

The argument passing conventions

For each of the conventions, arguments are pushed onto the stack in reverse order (i.e. the rightmost argument is pushed first). The leftmost (or leftmost few) arguments are passed in registers. This has a possible disadvantage from the opposite ordering in terms of the need for temporary variables to hold interim results for order-of-evaluation reasons. In practice, the disadvantages will be small because:

- Many arguments to functions are expected to be simple expressions (like constants or variable references) - so order of evaluation does not normally matter.
- On a RISC implementation, we won't want to push each argument anyway - instead it will be more efficient to allocate enough stack space for the call, and store each argument when it's available. This works well with a conservative GC - but it might be poor with a total GC.

This calling convention has the following advantages:

- required arguments can always be found at a known offset from a stack or frame pointer for any of the calling conventions
- optional arguments appear in the same order in memory as they would if vectored up as `#rest` parameters
- Stack allocating the optional arguments as vectors is almost trivial.

For the native code implementation, the callee is responsible for popping any arguments from the stack. This is always possible (even with dynamically sized optional args), because the argcount is available to say how many arguments were passed. This is not possible for the C backend - and this is the only substantial difference from the C arg passing convention.

Calling Convention Goals

Internal entry points should be as efficient as possible. I.e. there should not be any constraints on them because Dylan is a dynamic language.

1. There must be a consistent convention for all functions at the *external entry point*, so that functions can be called without the caller having any knowledge of what they are.
2. The code which is executed at external entry points should be shared by all functions with similar properties / lambda-lists.
3. The design should make the path from the external entry point to the internal entry point as simple as is reasonably possible.

The External Entry Point Convention

All Dylan function objects support the *external* convention. Each function object has an *XEP* slot containing the code to support this convention. External entry points are used for all unoptimized, normal calls to functions. This includes direct calls to methods and generic functions. Of course, whenever the compiler can use a more efficient entry point instead, then it will.

The registers are used as follows:

Register	Purpose
argcount	number of arguments
function	the function object
mlist	not used

If the function has a complex lambda list (with `#rest` or `#key`), then the external entry code will be one of a standard set of stack fixing functions. This stack fixer will make use of information in the function register to determine which keys to look for, whether the arg-count is legal, whether the arguments have appropriate types etc. The stack fixer will

then tail jump to the internal entry point (again, found from the function object). This mechanism requires 2 transfers of control (caller -> stack-fixer -> callee).

For example, consider the following Dylan code:

```
define method func1 (a, b, #rest optionals, #key key1, key2)
end method;
func1(1, 2, key2: 99);
```

For the call to `func1`, above, the parameters are described in the following table:

XEP Parameters for the Call to *func1*

XEP Parameters	Values
language parameters	`1`, `2`, `#"key2"`, `99`
argcount	4
function	generic function *func1*‘

Internal Entry Point Convention

The IEP convention uses a fixed number of language parameters, corresponding to each of the parameters of the function (5 in the case of *func1*, above, corresponding to a, b, optionals, key1, key2). In addition, there are two implementation parameters:

- *mlist*, a list of the next applicable methods to call if the function is a method called from a generic function (this parameter is used to support calls to *next-method*). If the function is not being called from a generic function, the value is *#f* (false).
- *function*, the Dylan function object being called (as for the XEP).

The implementation parameters are not obligatory for all IEP code. It is only necessary to pass *mlist* if the function contains a call to *next-method*. It is only necessary to pass *function* if the function is a closure (because the value is used by the IEP code to locate the environment of the closure). If the IEP is called from the XEP code, both the implementation parameters will always be set, even though they may not be necessary. For the same call to `func1`, above, the parameters are described in ‘ <runtime.htm#12946>‘_.

IEP Parameters for the Call to *func1*

IEP Parameters	Values
language parameters	`1`, `2`, `optionals`, `#f`, `99`
mlist	`#f`
function	generic function `func1`

Note that the language parameters now correspond to the formal parameters of the function, whereas, for the XEP, they corresponded to the supplied arguments.

The value of *optionals* in the set of language parameters is the Dylan vector `#[#"key2", 99]` which corresponds to all the optional arguments. The language parameter corresponding to `key1` is `#f`, because the keyword `#"key1"` was not supplied. However, the language parameter corresponding to `key2` is `99`, because `#"key2"` was supplied with that value.

The Method Entry Point Convention

All `<method>` objects support the *method entry point* convention. Each method object has an *MEP* slot containing the code to support this convention. When a method is called by a generic function (or via next method), the caller uses a dedicated entry point (available from the function object). If the method accepts `#key` or `#rest` parameters, then the method is called with a (possibly stack-allocated) vector representing the optional args. This vector appears as a single extra required argument.

If the method accepts `#key` parameters, then the method entry point will process the supplied keywords - stack fixing them so that they appear as required arguments. It will then tail-call the internal entry point.

If the method does not accept #key, then the method entry point is the same as the internal entry point.

Special Features

Introduction to bind-exit and unwind-protect

The following sections describe the implementation for the native code compiler, only.

Bind-exit and unwind-protect are represented on the stack as frames which contain information about how to invoke the relevant continuation. Unwind-protect frames are also chained together, and the current environment of existing unwind-protects is available in `%current-unwind-protect-frame`.

There are primitives to build each type of frame, and also to remove unwind-protect frames (bind-exit frames just have to be popped - so that is done inline). The primitive which removes unwind-protect frames in the fall-through case is also responsible for invoking the cleanup code (which is called as a sub-function in the same function frame as its parent).

There are also primitives to do non-local exits (*NLX*). These are passed the address of the bind-exit frame for the destination, and also the multiple values to be returned. As part of the NLX, any intervening unwind-protects are invoked and their frames are removed. Multiple-values are saved around the unwind-protects in the bind-exit frame of the destination.

unwind-protect

An *unwind-protect* frame (*UPF*) looks as follows:

Offset	Value
8	address of start of cleanup code
4	frame pointer
0	previous unwind-protect frame

The compiler compiles unwind-protect as follows:

```
let frame = primitive-build-unwind-protect-frame(tag1);
do-the-protected-forms-setting-results-as-for-a-return();
primitive-unwind-protect-cleanup();
goto(tag-finished);
tag1:
  do-the-cleanup-forms();
  end-cleanup(); // inlined as a return instruction
tag-finished:
```

If the protected body exits normally, then *primitive-unwind-protect-cleanup* is called (in the runtime system). This causes the unwind-protect frame to be unlinked from the chain, and the cleanup code to be invoked, as a subroutine call within the same function frame as the protected body. The cleanup code finishes by executing a return instruction. The runtime system ensures that any multiple values are restored, and returns control to the compiled code, which then executes the code following the unwind-protect.

If the cleanup code is invoked because of an NLX, then the runtime function finds the ultimate destination *bind exit frame* (*BEF*) from the UPF. The runtime function then passes this BEF to another runtime function (as for *bind-exit)* to test whether there are any further intervening cleanups, or to transfer control to the ultimate destination if not.

bind-exit

A *bind-exit* frame (*BEF*) looks as follows:

Offset	Value
52	continuation address
48	frame pointer
44	current unwind-protect frame
4	space for stack-allocated vector for up to 8 multiple values
0	pointer to saved multiple values as a vector

The compiler compiles bind-exit as follows:

```
let frame = primitive-build-bind-exit-frame(tag1);
let closure = make-bind-exit-closure(frame);
do-the-bind-exit-body-setting-results-as-for-a-return();
tag1:
```

During an NLX, multiple-values will be saved in the frame if an intervening unwind-protect is active. The frame itself contains space for 8 values. If more values are present, then they will be heap allocated.

When an NLX occurs, the transfer of control is implemented by a call into the runtime system, passing the pointer to the BEF as a parameter. The runtime function first checks whether there is an intervening cleanup, by testing whether the target dynamic environment in the BEF matches the current global dynamic environment. If there is no intervening cleanup, then control is transferred to the destination of the BEF. Alternatively, if there is an intervening cleanup, then the ultimate destination field of the current UPF is set to the destination BEF, and the cleanup code is invoked within a loop which repeatedly tests for further intervening unwind-protect frames until no more are found.

Multiple Values

The current implementation of multiple values supports Common Lisp semantics. It is about to be replaced by a new version which supports the new Dylan semantics.

Harlequin's current implementation uses a register to return a single Dylan value, as this is the only value that is used by almost all callers. In addition, each function returns a count of the number of values being returned. This count can be examined by the caller, if required, to determine how many values were returned. If a function is returning more than one value, the additional values are stored in a global (thread-local) area, where the caller may retrieve then, if desired. On RISC architectures, the multiple value count is returned in a register. For the x86 architecture, the *direction flag* register is set / unset to specify whether single / multiple values are being returned, respectively. If multiple values are being returned, then the count of the values is stored in a global (thread-local) location.

Documentation for the new version will be available shortly. Until then, here's an overview:

Functions which return a fixed number of return values just return those values, without returning a count. The first few values will be returned in registers (an architecture-specific number), and remaining values will be returned in a thread-local overspill area. If a function always returns zero values, then no code need be executed to indicate this fact.

Functions which return a dynamically-sized number of values return their values as above, but also return a count of the number being returned in a register. If a function dynamically happens to return zero values, then the return count will be set to zero, but the value *#f* will be returned as if it were the first return value.

If the caller of a function can statically determine the number of return values (i.e. at compile-time), then it need perform no checks. However, if the caller has no knowledge of the function being called, then it must check the properties of the callee function object to determine whether the static or dynamic convention is being used, and may then need to read either the dynamic return value count, or the static count in the properties of the function object.

This design has some interesting implications for tail-call optimization. A function can simply tail another function only if both the following rules apply:

1. The callee is known to return at least as many values as the caller, and they have appropriate types.
2. If the caller returns a dynamically-sized number of values, then the callee must too.

Name Mangling

In Dylan, unlike C, identifier names are case insensitive. Dylan also permits additional characters to appear in names. As a further complication, Dylan provides multiple namespaces, and the namespaces are controlled within a two-tier hierarchy of modules and libraries.

In order to make it possible to link Dylan code with tools designed to support more traditional languages, the Dylan compiler transforms the names which appear in Dylan programs to C compatible names, via a process called *mangling*.

The library, module and identifier names are each processed, according to the following rules:

1. All uppercase characters are converted to lowercase.
2. Any character which appears on the left-hand side of the table is mapped to the new character sequence accordingly.

Old	New	Comment
-	_	dash
!	*X*	exclamation
$	*D*	dollar
%	*P*	percent
*	*T*	times
/	*S*	slash
<	*L*	less
>	*G*	greater
?	*Q*	question mark
+	*A*	plus
&	*B*	
^	*C*	caret
_	*U*	underscore
@	*O*	
=	*E*	
~	*N*	

Finally, the fully mangled name is created by concatenating the mangled names together, along with other special markers.

Example 1:

```
Kexecute_componentQYPtestworksVtestworks

K                  = constant (methods are constants)
execute_componentQ = method name (Q = ?)
Y                  = module separator
Ptestworks         = module name (P = %)
V                  = library separator
testworks          = library name
```

Example 2:

```
Kstream_sizeYstreams_protocolVcommon_dylanMioM0I

K                    = constant
stream_size          = method name
Y                    = module separator
streams_protocol     = module name
V                    = library separator
common_dylan         = library name
M                    = method separator
io                   = library which defines this method
M                    = method separator
0                    = 0th method in stream_size generic
I                    = Internal Entry Point (IEP)
```

The Dylan mangling scheme takes care to omit some elements of the name when possible to help abbreviate the names. The usual things that are omitted are:

- The module marker and name when the module name is the same as the library name.
- The name of the library where the method is defined (between the method separators). This is omitted when it is the same as the library which defines the generic function.

We'll see this in our next example:

```
Krun_test_applicationVtestworksMM0I

K                    = constant
run_test_application = method name
(module marker and name omitted when same as library name)
V                    = library separator
testworks            = library name
M                    = method separator
(library name omitted as this is defined in the same library as the generic)
M                    = method separator
0                    = 0th method in run_test_application generic
I                    = Internal Entry Point (IEP)
```

Bindings within the Dylan library are given special shortened forms. In this case, instead of the library separator `V`, the library name and then the `Y` module separator, we would see the library separator `V` and then then special `dylan` module separator `K`. Some modules within the `dylan` library are given special single letter abbreviations. For example the `dylan` module's name is just `d` and the `internal` module is `i`.

Examples:

- `KLempty_listGVKd`
- `Kcondition_format_arguments_vectorVKiI`

See the source for more detail.

Compiler Support for Threads

Dylan Portability Interface

The Threads Library is designed for implementation using different threads APIs from common operating systems, including Unix and Windows. Harlequin's implementation of the library is designed to be directly portable onto these operating systems. This portability is achieved by using primitive operations defined within our runtime system. Each primitive operation must be implemented specially for each operating system.

The set of portable primitive operations is collectively called the *portability layer*. The Dylan compiler has special knowledge of the portability layer via primitive function definitions and some specialized emit methods for flow-graph node types which are specific to threads.

Portability and Runtime Layers

The design assumes that each of the concrete classes of the Threads Library (`<thread>`, `<simple-lock>`, `<recursive-lock>`, `<semaphore>` and `<notification>`) corresponds with an equivalent lower-level feature provided directly by either the operating system or the runtime system. The Dylan objects which are instances of these classes are implemented as *containers* for handles corresponding to low-level (non-Dylan) objects. The Dylan objects contain normal Dylan slots too, and these are directly manipulated by the Dylan library. However, the slots containing the low-level handles may only be manipulated via primitive function calls. For each of the classes, primitive functions are defined to both create and destroy the low-level handles, as well as to perform the basic functions of the class, such as *wait-for* and *release*. The platform-specific implementation of these primitive functions is free to choose any representation for these handles, provided that it is the same shape as a Dylan slot (which is equivalent to C's *void **).

As with all Dylan objects, the container objects defined by the threads library are subject to automatic memory management, and possible relocation by the garbage collector. The contents of the container slots will be copied during such a relocation — but the values they contain will not be subject to garbage collection or relocation themselves.

The portability layer provides no direct support for the *dynamic-bind* operation. The library implements a *dynamic variable* as a thread-local variable via the high-level Dylan constructs *define thread variable* and *block ... cleanup* to manage the creation and deletion of new bindings.

The portability layer includes support for conditional update of atomic variables, as well as assignment. The implementation mechanism for these is not defined, but it is hoped that many platforms will provide direct hardware support for this operation. Where hardware support is not available, the low-level implementation may choose to use a lock to protect conditional updates and assignments, as a fall back option. It is assumed that atomic variables may always be read as normal variables.

Implementations of Dylan Thread Interfaces shows the expected mapping between the concrete Dylan classes and low-level operating system features, for three of the most popular general-purpose operating systems.

Implementations of Dylan Thread Interfaces

Dylan Interface	Unix Implementation	Win32 Implementation
`<thread>`	thread	thread
`<simple-lock>`	mutex	critical region
`<recursive-lock>`	mutex	critical region
`<semaphore>`	semaphore	semaphore
`<notification>`	condition variable	event
`dynamic variable`	thread-local variable	thread-local variable
`conditional-update!`	mutex	exchange instruction (using a guard value as a lock);

Dylan Types for Threads Portability

Three Dylan types merit discussion for their use with portability primitives: `<thread>`, `<portable-container>`. Objects that are instances of the `<thread>` and `<portable-container>` classes have slots which contain lower-level objects that are specific to the Dylan runtime or operating system.

<thread>

[Class]

A Dylan object of class <thread> contains two OS handles. One of these represents the underlying OS thread, and the other may be used by implementations to contain the current status of the thread, as an aid to the implementation of the join state.

<portable-container>

[Class]

The <portable-container> class is used by the implementation as a superclass for all the concrete synchronization classes (<simple-lock>, <recursive-lock>, <semaphore>, and <notification>). Each <portable-container> object contains an OS handle, which is available to the runtime for storing any OS-specific data. Subclasses may provide additional slots.

Various classes of Dylan objects are passed through the portability interface, and hence require description in terms of lower level languages. *Correspondence Between Dylan Types and C Types* maps the layout of these Dylan objects onto their C equivalents, which are used by runtime-specific implementations of the portability layer.

In general, all Dylan types can be thought of as equivalent to the C type D, which is in turn equivalent to the C type void*. Of course, runtime-specific implementations of the portability layer must have access to relevant fields of the Dylan objects on which they operate. The type definitions in *Correspondence Between Dylan Types and C Types* give implementations access to fields needed for specific types. These definitions are not necessarily complete descriptions of the Dylan objects, however. The objects may contain additional fields that are not of interest to the portability layer, and subclasses may add additional fields of their own.

Correspondence Between Dylan Types and C Types

Dylan Type	C Type	C Type Definition
<object>	*D*	*typedef void* D;*
<integer>	*DINT*	*platform specific (size of void*)*
<function>	*DFN*	*typedef D(*DFN)(D, int, . . .);*
<simple-object-vector>	*SOV**	*typedef struct _sov { * *D class; * *DINT size; D data[ ]; } SOV;*
<byte-string>	*B_STRING**	*typedef struct _bst { * *D class; * *DINT size; char data[ ]; } B_STRING;*
false-or(<byte-string>)	*D_NAME*	*typedef void* D_NAME;*
<portable-container>	*CONTAINER**	*typedef struct _ctr { * *D class; * *void* handle; } CONTAINER;*
<thread>	*D_THREAD**	*typedef struct _dth { * *D class; * *void* handle1; void* handle2; } D_THREAD;*

Compiler Support for the Portability Interface

The Compiler Flow Graph

The front end of the compiler parses Dylan source code and produces an intermediate representation, the Implicit Continuation Representation (ICR). The ICR is a directed acyclic graph (DAG) of Dylan objects. A *leaf* in the ICR represents a basic computational object, such as a variable (of class <variable-leaf>) or a function (of class <function-leaf>). A *node* in the ICR represents an operation such as assignment (class <assignment>), conditional execution (class <if>), or a reference to a leaf (class <reference>).

In mapping Dylan code to the ICR, the compiler uses a set of *converters*, which perform syntactic pattern matching against fragments of Dylan code and generate the ICR corresponding to the matched code. For example, when the compiler encounters a top-level variable definition (introduced by the Dylan *define variable* construct), the converter for *define variable* creates a new instance of <global-variable-leaf> in the ICR to represent this variable and to record data such as its name, initial value, and typing information.

The back end of the compiler traverses the flow graph and emits code in the target language for compiler output. Methods in the back end specialize on node and leaf classes to enable them to produce the appropriate output.

Compiler Support for Atomic and Fluid Variables

The portability layer provides support for atomic variable access and for Dylan fluid variables (implemented as thread-local variables). Atomic variables and thread variables are directly represented in the flow graph, where they are subject to dataflow analysis. The variables themselves appear as leaves in the graph.

Because both atomic and fluid variables need special treatment when they are accessed, the back end must emit output that is different from that for accessing other kinds of variables. The compiler defines two specialized classes of leaf for the ICR, `<atomic-global-variable-leaf>` (corresponding to atomic variables) and `<fluid-global-variable-leaf>` (corresponding to fluid variables). These are subclasses of `<global-variable-leaf>` and therefore inherit general characteristics of leaves that represent variables.

ICR leaves representing both atomic and fluid variables are created by the converter for `define variable`. When the compiler encounters a definition of an atomic variable (introduced by the `define atomic-variable` construct), the converter for `define variable` creates an instance of `<atomic-global-variable-leaf>` in the ICR. When the compiler encounters a definition of a fluid variable (introduced by the `define fluid-variable` construct), the converter creates an instance of `<fluid-global-variable-leaf>`.

The operations of reading, writing, and conditionally updating atomic variables and of reading and writing fluid variables are not represented by primitive functions. Instead, they are represented directly in the flow graph. They are implemented by specializing methods on the leaf classes that represent atomic and fluid variables.

Compiler Support for Primitives

When the compiler constructs the flow graph, it represents a function call as a node in the ICR. Just as the compiler distinguishes atomic and fluid variables by means of specialized leaf classes, so it distinguishes calls to primitive functions of the portability interface by means of a specialized node class.

A function call is an operation on several components: the function object, the arguments, and the destination for returned values. When the compiler encounters a regular Dylan call, which typically appears as a call to a generic function, it represents the call in the ICR as a node of class `<combination>`.

However, the compiler contains a table of the primitive functions in the portability interface. Before creating an ICR node to represent a function call, the compiler looks up the function being called in the table of primitives. If the function appears in the table, the compiler creates an ICR node of class `<primitive-combination>`.

When the back end traverses the flow graph, methods specialized on the node class `<primitive-combination>` emit calls to primitive functions.

Support for Dylan Language Features

Interfacing to Foreign Code

It is intended that threads created by the Dylan library may inter-operate with code written in other languages with no special constraints. Dylan is interfaced with other languages via a Foreign Language Interface (*FLI*), which acts as a barrier between Dylan conventions and the *neutral* conventions of the platform. The FLI is responsible for:

1. mapping between Dylan and foreign data types,
2. converting between Dylan and foreign calling conventions
3. maintaining the Dylan dynamic environment

4. maintaining any support necessary for garbage collection (such as ensuring that all Dylan values can be traced).

The first and second of these require no significant extensions to support multiple threads, since these are inherently computations which have no effect on any thread other than the one performing the computation.

There is a requirement that the dynamic environment for each thread is stored in a thread-local variable. Since the environment is stored in this way, its value is preserved across calls into foreign code, and it will still be valid if the foreign code calls back into Dylan. The techniques described in [MG95] for maintaining the dynamic environment across foreign calls are therefore directly appropriate to a multi-threaded implementation too.

If an object is passed to foreign code with dynamic extent, then it is sufficient to ensure that the object is referenced from the current stack, which the garbage collector will scan conservatively. In a multi-threaded implementation, the garbage collector will scan all the stacks conservatively, so there is no requirement to maintain a thread-global data structure.

If an object is passed with indefinite extent, then it must be recorded in a table. The table may be maintained by the runtime system, by means of suitable primitive functions to add and remove references. There are potentially synchronization problems associated with multiple threads manipulating a global data structure — but the runtime system implementation is free to choose whether to have separate tables for each thread, or whether to have a global table with an associated lock to guard accesses. Either technique is possible — but Harlequin have not yet implemented this feature.

One further consideration is the interaction of the Dylan threads library itself with foreign components:

If foreign code is not designed for multiple threads (for instance, because it uses global data structures, and doesn't synchronize updates), then the code may fail if it is invoked from multiple Dylan threads. However, this problem is not related to the Dylan implementation, since it would fail if called from multiple threads created by any means. The solution is to modify the foreign component to make it thread safe.

If foreign code is designed for use with multiple threads, then it is valid for it to use the synchronization facilities of the Dylan library (by calling back into Dylan, to invoke the Threads Library synchronization functions). Alternatively, it may use its own methods for synchronization, provided that these are not incompatible with the methods provided by the operating system. This is valid whenever it has been possible to implement the runtime system support for threads directly in terms of operating system features, and it is anticipated that this will always be true if the operating system supports threads. Typically, foreign code is expected to make direct use of operating system threads facilities.

However, a problem may arise if a thread is created in foreign code, and the new thread then calls back into Dylan. In this case, the Dylan thread library itself will not be able to find an existing `<thread>` object corresponding to the current thread, and the fluid variables for the current thread will not have been correctly initialized. Worse still, the garbage collector may not have enough information to locate the roots of the thread. Harlequin have not yet allowed for this in their implementation, but they have an anticipated solution.

It is possible to detect that a thread has never been executing on the Dylan side of the FLI before because it will have an uninitialized (zero) value for its thread-local dynamic environment variable. This can be checked at a call-in in the stub function which implements the FLI. Once such a thread has been detected, appropriate initialization steps can be taken. A function in the runtime system can be called to register the stack of the thread for root tracing; the dynamic environment can be set to point to a suitable value on the stack; finally a new Dylan `<thread>` object can be allocated and initialized with `primitive-initialize-current-thread` (as for the first thread).

Finalization

As has been discussed, the Dylan synchronization objects are implemented as wrappers around lower-level operating system structures. The Dylan objects are subject to garbage collection, and their memory will be automatically freed by the garbage collector at an undefined point in the program. But the low-level structures are not Dylan objects and must be explicitly freed when the Dylan container is collected (primitive functions are provided for this purpose). However, the core language of Dylan provides no *finalization* mechanism to invoke cleanup code when objects are reclaimed. Harlequin's implementation of the Threads Library strictly requires this, but it is not yet implemented. It is intended to provide finalization support for Dylan with a new garbage collector which is currently under development.

Startup

When a Dylan library (either a shared library/DLL or executable) is loaded, a number of initialization tasks need to take place:

- The Dylan run-time facilities, including the garbage collector and the main thread's thread environment block (TEB), need to be initialized if they haven't already been.
- The final addresses of any `<symbol>` objects referenced by the library need to be resolved, and data locations that reference symbols need to be patched to point to the resolved addresses. This is part of the "for-system" initialization.
- Top-level forms in each of the library's source files need to be evaluated. This is the "for-user" initialization.

The following describes how Dylan libraries are initialized under the different compiler back-ends.

LLVM

Initialization for programs compiled by the LLVM back-end relies on constructor functions, provided on most platforms to support (among other things) C++ constructors. The LLVM linker makes use of two different constructor priorities, system (priority 0) and user (priority 65535).

The first constructor generated at system priority within the `_glue.bc` file generated for the `dylan` library is a call to `_Init_Run_Time`, a C function defined in `sources/lib/run-time/llvm-runtime-init.c` to initialize the garbage collector and other system facilities needed by the Dylan run-time.

For each compile unit that requires it, a constructor at system priority is generated to do for-system initialization:

- When `<symbol>` resolution is required, a call to `primitive-symbol-fixup` is generated, passing a table of symbols to resolve and reference addresses to patch.
- If there are any system initializers in the compile unit, calls to these are also generated. These normally only occur when a class definition depends on non-static values.

The remaining for-user initialization is done in the per-library startup glue function, named `_Init_`*mangled-library-name*, where *mangled-library-name* is the Dylan library name transformed by name mangling. This function, generated in `_glue.bc` by `emit-gluefile` in `sources/dfmc/llvm-linker/llvm-gluefile.dylan`, does the following:

1. Checks a flag to see if the library initialization has already been done, and exits if it has
2. Sets the initialization flag
3. Calls the library startup glue functions of all libraries (except for the Dylan library) referenced by this one.
4. Calls the library's self-init function, named `_Init_`*mangled-library-name*`_X`, which in turn calls each of the for-user init functions for each compile-unit in the library. These contain the code generated for all of the top-level forms.
5. For the Dylan library only, the self-init function also calls the `%install-boot-symbols` function defined in `sources/dylan/symbol-table.dylan`. ()

The startup glue function for the Dylan library is called from a global constructor at user priority. This ensures that the `%install-boot-symbols` is performed before other libraries' symbol resolution begins.

For executables, execution begins at the usual `main` entry point. The `main` function, generated in `_main.bc` by `emit-gluefile` in `sources/dfmc/llvm-linker/llvm-gluefile.dylan`, simply stores its `argc` and `argv` arguments in `*argc*` and `*argv*` runtime variables and calls the library startup glue function.

HARP (Win32)

When a Win32 DLL compiled by the Open Dylan HARP back-end is loaded, execution starts at *mangled-library-name*`Dll@12`, where *mangled-library-name* is the Dylan library name transformed by name mangling. This entry point, generated for each library by `emit-executable-entry-points` in `sources/dfmc/harp-native-cg/linker.dylan`, stores the address of the library's glue function, named `_Init_`*mangled-library-name*, into the `_init_dylan_library` variable and branches to `_DylanDllEntry@12`.

A DllMain entry point receives three arguments: the module handle of the DLL, a reason code, and a flag that indicates whether the DLL was loaded statically or dynamically. The `_DylanDllEntry@12` routine, generated in `sources/harp/x86-windows-rtg/ffi-barrier.dylan`, looks at the reason code to determine what to do:

- For process attach, the routine:
  1. Stores the module handle in a DLL-local `_module_hInstance` variable.
  2. Allocates and initializes the TEB and GC-TEB, and initializes the garbage collector by calling `dylan_init_memory_manager` and `dylan_mm_register_thread` (`DxDYLAN.dll` only).
  3. Calls `_dylan_initialize` (described below)
- For thread attach, the routine clears the TLV vector slot of the TEB (`DxDYLAN.dll` only).
- For thread detach, the routine:
  1. Deregisters the thread by calling `dylan_mm_deregister_thread_from_teb` (`DxDYLAN.dll` only).
  2. Dealocates the TEB (`DxDYLAN.dll` only)
- For process detach,
  1. Deregisters the main thread by calling `dylan_mm_deregister_thread_from_teb` (`DxDYLAN.dll` only).
  2. Deinitializes the garbage collector by calling `dylan_shut_down_memory_manager` (`DxDYLAN.dll` only).

The `_dylan_initialize` function (generated in `sources/harp/native-rtg/ffi-barrier.dylan`):

1. Calls the runtime's `init_dylan_data` function, which:
   (a) Calls `primitive_fixup_unimported_dylan_data`
   (b) Calls `primitive_fixup_imported_dylan_data`
   (c) Calls `primitive_register_traced_roots`
2. Sets the FFI barrier state to `$inside-dylan`.
3. Calls `dylan_init_thread_local` (which tail-calls the `dylan_init_thread` C function defined in `sources/lib/run-time/exceptions.c`) passing a pointer to the `call_init_dylan` function. This function in turn is called as an MPS trampoline.
4. Resets the FFI barrier state to `$outside-dylan`.

The `call_init_dylan` function retrieves the value of the `_init_dylan_library` variable (pointing to a library startup glue function) and performs an indirect call.

Similarly, an Win32 executable starts execution at *mangled-library-name*`Exe`. This routine, which is also generated by `emit-executable-entry-points`, stores the address of `_Init_`*mangled-library-name* in the `_init_dylan_library` variable and branches to `dylan_main`.

CHAPTER

SEVEN

DUIM - DYLAN USER INTERFACE MANAGER

We have a lot to learn about hacking on DUIM, so there isn't much to document yet as we haven't learned yet.

GTK Back-end

The GTK+ back-end is usable on both macOS and Linux. It is based on GTK+ 3.x. On macOS, it assumes that the Quartz back-end is being used rather than X11.

The bindings for GTK+ are largely generated from gobject-introspection meta-data. Some portions are hand-edited however or entirely done by hand, like the Cairo bindings. Most of the special cases for hand-editing are put into the binding generator (gir-dylan), but not all.

Getting GTK+ 3.x with Quartz on macOS

Building GTK+ 3.x with Quartz on macOS must be done in a specific way. This currently can *not* be done with Homebrew.

Follow the directions on the GTK+ wiki, but make sure to build GTK+ 3.x rather than 2.x. To do this, you'll want to use the moduleset `meta-gtk-osx-gtk3` rather than `meta-gtk-osx-core`.

Once you have a build, be sure to add the appropriate directories to your path.

We have not yet integrated GTK+'s macOS integration module, so the menu bar will not (yet) be integrated with the system menu bar when running a DUIM application.

Note: GTK+ will be built as a 64 bit binary. This means that your Dylan code will also have to be built as a 64 bit binary. Doing this requires a build of Open Dylan from the master branch. Once you have a current build of Open Dylan, set the `OPEN_DYLAN_TARGET_PLATFORM` environment variable to `x86_64-darwin` and your builds will be 64 bit.

Debugging Information

To enable more debugging information, uncomment the following line in `sources/duim/gtk/gtk-debug.dylan`:

```
*debug-duim-function* := dbg;
```

You can then output debugging information with either of the following functions:

- `ignoring(msg)`

- `duim-debug-message(format, args)`

Additional debug information can be obtained via the GTK_DEBUG and GDK_DEBUG environment variables.

CHAPTER

EIGHT

TOPICS

Method Dispatch

Method dispatch is a critical area for compile time and runtime optimization in Dylan. Since nearly everything appears to involve a method dispatch including slot access, it is very important that the compiler be able to optimize much of this away.

Method dispatch has support in both the compiler and the runtime. The runtime portion also has data structures that are defined in the compiler (leading to dependencies between the compiler and the runtime).

Runtime portions, including compiler support, can be found in these files:

- sources/dylan/discrimination.dylan
- sources/dylan/dispatch-caches.dylan
- sources/dylan/dispatch-prologue.dylan
- sources/dylan/dispatch.dylan
- sources/dylan/new-dispatch.dylan
- sources/dylan/slot-dispatch.dylan
- sources/dfmc/modeling/functions.dylan

Compiler support for method dispatch optimization can be found in:

- sources/dfmc/optimization/dispatch.dylan

Publications

One paper has been written about the implementation of dispatch in Open Dylan:

- **Partial Dispatch: Optimizing Dynamically-Dispatched Multimethod Calls with Compile-Time Types and Runtime Feedback** (by Jonathan Bachrach and Glenn Burke - Technical Report 2000 pdf)

Generic Function Representation

In Dylan, a generic function looks like (from sources/dfmc/modeling/functions.dylan):

```
define abstract primary &class <generic-function> (<function>)
  lazy &slot function-signature :: false-or-compile-time-or(<signature>),
    init-value: #f,
    init-keyword: signature:;
  &slot %gf-cache, init-value: #f;
```

```
  lazy &slot debug-name :: <object>,
    init-value:   #f,
    init-keyword: debug-name:;
  lazy &computed-slot generic-function-methods :: <list>,
    init-value: #();
  // If we start using this it should probably be made lazy, as it would
  // only be used for creating the runtime object, not compilation.
  &slot discriminator, init-value: #f;

  // Compile-time slots.
  slot ^generic-function-properties :: <integer>, init-value: 0;
  lazy slot signature-spec :: <signature-spec>,
    required-init-keyword: signature-spec:;
  lazy slot %generic-function-domains :: <list> = #();
  slot parameters-dynamic-extent,
       init-value: #f,
    init-keyword: dynamic-extent:;
  slot ^generic-function-cache-info = #f;
  metaclass <function-class>;
end &class <generic-function>;
```

There are 2 subclasses of `<generic-function>`: `<sealed-generic-function>` and `<incremental-generic-function>`. A sealed generic function adds no slots, while an incremental generic function maintains some extra data to support adding further methods.

At the moment, we'll focus on sealed generic functions.

And in the generated C, a sealed generic function looks like:

```
typedef struct {
  D wrapper;
  D xep_;
  D function_signature_;
  D Pgf_cache_;
  D debug_name_;
  D generic_function_methods_;
  D discriminator_;
} _KLsealed_generic_functionGVKe;

_KLsealed_generic_functionGVKe Ksize_in_wordsVKi = {
  &KLsealed_generic_functionGVKeW,
  &gf_xep_1,
  &KDsignature_LobjectG_object_rest_value_1VKi,
  &KPfalseVKi,
  &K342,
  &K632,
  &RSINGULAR_Labsent_engine_nodeG
};
```

Incremental generic functions look similar, but contain some additional data after the discriminator.

The types in the generated C are just `D` which is what the Dylan compiler's C back-end likes to generate. More specific types for some values are available but not emitted by the C back-end. (Improvements in this area are worth considering as they would improve the debugging experience.)

Runtime Dispatch

Much of the technical report by Bachrach and Burke remains accurate with respect to the basics of dispatch.

Discriminators at Runtime

The initial discriminator of a generic function is `$absent-engine-node` (or in C, `RSINGULAR_absent_engine_node`). When this is encountered when performing a dispatch, `gf-dispatch-absent` is invoked, which calls `handle-missed-dispatch`. The initial dispatch engine state will then be calculated in `calculate-dispatch-engine` and dispatch will proceed.

In this way, dispatch data is built incrementally at runtime as it is needed and can take advantage of data available at runtime. In fact, dispatch can start out being monomorphic and grow to linear and then hash-based discriminators as the number of relevant methods changes at runtime.

For example, when growing a linear discriminator (`grow-linear-class-keyed-discriminator`), it can be upgraded to become a hashed discriminator.

The logic for creating a new discriminator starts in `compute-discriminator-for-arg` (defined in sources/dylan/discrimination.dylan).

Discriminator Structure

The classes that dictate the in-memory layout of the discriminators are defined within the compiler in sources/dfmc/modeling/functions.dylan.

Of particular interest are the `<linear-by-class-discriminator>` and `<hashed-by-class-discriminator>`. These, along with some variants for dealing with singleton dispatch, define a repeated slot for storing their data:

```
repeated &slot class-keyed-discriminator-table-element,
  init-value:         #f,
  size-getter:        class-keyed-discriminator-table-size,
  size-init-keyword: size:,
  size-init-value:    0;
```

For these discriminators, the keys and values are stored in alternating sequence:

```
key1, value1, key2, value2
```

This allows for a compact representation within memory without extra allocations for pairs of values, a hash table, etc.

The code for iterating over this data can be found in the functions `linear-class-key-lookup` and `hashed-class-key-lookup` as found within sources/dylan/new-dispatch.dylan. That file also contains the code for adding new methods to the discriminator.

Compile Time Optimization

Discuss the impact of sealing and other things here.

Analysis

Performance Highlighting

The compiler records dispatch decisions as they're made within the optimizer. This work is performed within sources/dfmc/optimization/dispatch.dylan (look for calls to `color-dispatch`). It is worth noting that the dispatch decisions are compacted by `compact-coloring-info` in sources/dfmc/management/compilation-driver.dylan.

In the IDE, Open Dylan supports performance highlighting to indicate how much optimization the compiler was able to apply. This is performed within sources/environment/deuce/dylanworks-mode.dylan by examining the results from `source-record-colorization-info`.

This information is also available in `.el` files within the build directory that can be used with the `dylan-mode` in emacs. The generation of the `.el` files is performed by `project-dump-emacs-dispatch-colors` in sources/project-manager/projects/implementation.dylan.

The available dispatch decisions that are recorded for highlighting are:

- `#"not-all-methods-known"`
- `#"failed-to-select-where-all-known"`
- `#"lambda-call"`
- `#"inlining"`
- `#"slot-accessor-fixed-offset"`
- `#"eliminated"`
- `#"dynamic-extent"`
- `#"bogus-upgrade"`

Link to documentation on both of these features, perhaps embed some screenshots.

Dispatch Profiler

There is a dispatch profiler in sources/lib/dispatch-profiler but no one knows how to use it.

Future Work

- Learn more about partial dispatch and possibly enable it.
- Look at the effectiveness of call site caching.
- Can the hashing in the megamorphic hashed by-class discriminator be tuned better?
- Learn more about and document things mentioned in this document but that aren't understood well (like dispatch profiling).
- Much more documentation.

Debugging

Debugging a Compiler Crash or Internal Error

When the compiler generates an internal error or directly crashes, using gdb or lldb is a great way to find out more about what has gone wrong.

See Debugging with GDB or LLDB for general information on debugging applications written with Open Dylan.

An internal error from the compiler will often look something like this:

```
Internal error: ELEMENT outside of range: -5
```

To find out what is happening, run the compiler under gdb or lldb, and set a breakpoint on `Kdisplay_conditionYcommand_linesVenvironment_commandsI`. Doing so will allow you to see the stack trace at the time of the internal error.

Dumping DFM Output

DFM is a relatively readable intermediate language used by the compiler. It can be useful for debugging the compiler, but also for optimizing Dylan code in general since it shows (for example) where the compiler wasn't able to optimize method dispatch.

The main point you need to know is to add the `-dfm` flag when you invoke `dylan-compiler`. This will generate DFM output files in your `_build/build/` directory.

Porting to a New Target Platform

Warning: This is a work in progress, as we work through doing a new port.

Naming Your Target

A target name typically consists of the CPU type and the OS name, separated by a hyphen. Due to using the hyphen as a separator, the CPU type and OS name should not include a hyphen.

Example target names are:

- `arm-linux`
- `ppc-darwin`
- `x86-win32`
- `x86_64-linux`

CPU types are typically:

- `#"arm"`
- `#"ppc"`
- `#"x86"`
- `#"x86_64"`

OS names are typically:

- `#"linux"`
- `#"darwin"`
- `#"freebsd"`
- `#"win32"`

Warning: Some existing targets are poorly named. We may attempt to rename `#"win32"` to `#"windows"` in the future.

> **Warning:** We do not yet have a strategy in place for targets which are not for a particular CPU and OS type. Examples of these would be emscripten, the JVM, and Google's Native Client.

New Registry

The registry is how we map library names to the actual project files that should be built to provide that library. This lets us provide per-platform implementations of libraries as required.

The registry can be found within `sources/registry`.

The easiest way to do this is to copy an existing directory and modify it as needed. If you are doing a port to a Unix-like platform, this should be pretty straightforward to copy from one of the Linux definitions.

System Library

New LID and definitions

You will need to create a new `.lid` file to point to the per-platform files for the new target. You will also need a `*-operating-system.dylan` file to provide a couple of definitions.

There are plenty of examples of this within `sources/system` for the currently supported target platforms.

Magic Numbers

The system library requires some magic numbers for accessing struct members defined within the standard C library.

To do this, compile `sources/system/dump-magic-numbers.c` for your target platform and execute it on the target. This will generate a Dylan file that you can add to the system library.

C Back-End

If the new processor type is a 64 bit processor, you will want to update the C back-end appropriately:

- Edit `sources/dfmc/c-back-end/c-back-end.dylan`
- Update the definition of `back-end-word-size`

You will also need to do a couple of edits to the C run-time:

- Edit `sources/lib/run-time/run-time.*`, looking for usages of `__x86_64__`

> **Warning:** We should improve this code within the C run-time.

Additionally, there are some mappings involving `OPEN_DYLAN_TARGET_PLATFORM` in `sources/lib/run-time/Makefile.in` that will need to be updated.

LLVM Back-End

The per-platform configurations for the LLVM back-end can be found within `sources/dfmc/llvm-back-end/llvm-targets.dylan`.

When adding a new CPU type, add a new abstract back-end for the CPU with the appropriate data layout and word size.

When adding a new OS type, add a new abstract back-end for the OS.

After that, you can create the new target back-end that inherits from the appropriate CPU and OS specific abstract back-ends. You may need to override the data layout if a different ABI is required.

> **Warning:** Someone should document how to correctly determine the appropriate data layout to use.

Be sure to invoke `register-back-end` correctly with your new back-end class.

Additionally, as with the C back-end, there are some mappings involving `OPEN_DYLAN_TARGET_PLATFORM` in `sources/lib/run-time/Makefile.in` that will need to be updated.

This isn't completely written yet.

Build Scripts

Build scripts are partially documented within Jam-based Build System. The existing build scripts can be found within `sources/jamfiles`.

You will want to copy an existing one and make whatever changes are required. When targeting a Unix-like platform, much of the logic is already shared within `sources/jamfiles/posix-build.jam`.

You should also add your new build script to `sources/jamfiles/Makefile.in` so that it gets installed.

Autoconf

The `configure.ac` script handles detecting a target platform and setting some appropriate variables within the build system. There is a large block that deals with checking the `$host` (set up by `AC_CANONICAL_TARGET`) and configuring things appropriately.

After updating `configure.ac`, be sure to re-run `autogen.sh` to create an updated `configure` script before re-running `configure`.

Performing a Cross-Build

In the examples below, we will use `arm-linux` as the example.

Preparing the Garbage Collector

Currently, most ports of Open Dylan will probably be using the Boehm garbage collector rather than MPS. An easy way to get the required files is to install the Boehm GC on the target platform and then copy the include and library files back to the build machine being used to perform cross-compilation.

If you intend to start with MPS, you will need to ensure that the MPS has been ported to the target platform first. This is an undertaking that is outside the scope of this document.

Building the Run-Time

You can cross-compile the run-time by going to `sources/lib/run-time` and running `make install`, however, you will need to pass some special flags to `make`:

CC This should point to your cross-compiler, along with any special compilation flags that are required, such as the include paths for the garbage collector, or target CPU flags.

OPEN_DYLAN_TARGET_PLATFORM This should be the name of the target platform for which you are cross-compiling.

OPEN_DYLAN_USER_INSTALL This points to the path to the installation of Open Dylan which you will be using to perform cross-compilation.

An example command line might look like:

```
make CC="arm-linux-gnueabihf-gcc -I/path/to/gc/include" \
  OPEN_DYLAN_TARGET_PLATFORM=arm-linux \
  OPEN_DYLAN_USER_INSTALL=/opt/opendylan-current \
  clean install
```

Creating a Custom Build Script

When cross-compiling, it is best to set up a custom Jam build script which can be passed to the `dylan-compiler`. This will allow you to customize important parts of the build configuration.

An example custom script might look like:

```
CC = /opt/arm-linux/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin/arm-linux-gnueabihi
GC_CFLAGS = -I/opt/arm-linux/gc/include -DGC_USE_BOEHM -DGC_THREADS ;
GC_LFLAGS = -L/opt/arm-linux/gc/lib -lgc ;

include $(SYSTEM_BUILD_SCRIPTS)/arm-linux-build.jam ;
```

This just overrides the default values for some variables and then includes the system-provided build script for `arm-linux`.

Cross-Building a Test Application

Assuming that you've cross-compiled and installed a copy of the run-time into the version of Open Dylan that you're using for cross-compilation, this is an easy step:

```
OPEN_DYLAN_TARGET_PLATFORM=arm-linux \
  dylan-compiler -build-script path/to/custom-build.jam \
    -build hello-world
```

This should create a build of the `hello-world` application in the `_build` directory. This directory can be copied to the target machine and executed. Hopefully it runs correctly. If not, now is the time to start debugging.

Cross-Building the Dylan Compiler

This is no different from building the `hello-world` application except that now you are building `dylan-compiler`:

```
OPEN_DYLAN_TARGET_PLATFORM=arm-linux \
  dylan-compiler -build-script path/to/custom-build.jam \
    -build dylan-compiler
```

Building without a Cross-Compiler

When you don't have an actual cross-compiler when porting to a new platform, the recommended solution is to use a shared file system and custom compiler scripts.

- Set up a file system that is shared between the build host and the target host.

- On the build host, create a script for running the C compiler via ssh on the target host.
- Use this custom script to build the run-time library.
- Set up a custom build script as described above and use it to compile a test application and the Dylan compiler.

The PPML library

Program notes need to be stored and later displayed in browsers. This presents us with two problems. At the point of creation we have no way of knowing the column width that will be used when the note is displayed. There may even be more than one width if we want to be smart when a browser window is resized. A second problem arises if we store a program note in the form of a condition string + arguments and the arguments are context sensitive. We could just save everything as a string, but then the logical structure of the message is lost. An alternative is to store the text in a more structured form. This is the purpose of the *<ppml>* class and its derivatives. The interface is based on Oppen's 1980 TOPLAS paper.

A PPML document, represented as *<ppml>* is a tree of PPML tokens. The tokens available and their associated constructor functions are:

- *<ppml-block>* (*ppml-block*)
- *<ppml-break>* (*ppml-break*)
- *<ppml-browser-aware-object>* (*ppml-browser-aware-object*)
- *<ppml-separator-block>* (*ppml-separator-block*)
- *<ppml-string>* (*ppml-string*)
- *<ppml-suspension>* (*ppml-suspension*)

Structure is provided through instances of *<ppml-block>* and *<ppml-separator-block>* as they can contain subnodes of PPML.

Constructing a PPML Document

Constructing a PPML document is typically done by using the various constructor functions:

```
define compiler-sideways method as
    (class == <ppml>, o :: <&generic-function>)
 => (ppml :: <ppml>)
  let sig = model-signature(o);
  if (sig)
    ppml-block(vector(ppml-string(o.^function-name),
                      ppml-break(),
                      as(<ppml>, sig)))
  else
    ppml-string(o.^function-name)
  end;
end method;
```

Printing a PPML Document

Given a PPML document, the best way to print it is via *ppml-print*:

```
ppml-print(ppml, make(<ppml-printer>, margin: 100));
```

The PPML module

PPML Tokens and Constructors

<ppml> Abstract Class

Superclasses <object>

Discussion The abstract base class for all PPML tokens.

<ppml-block> Class

Superclasses <ppml>

Init-Keywords

- **break-type** – An instance of <ppml-break-type>.
- **constituents** – An instance of <ppml-sequence>.
- **offset** – An instance of <nat>.

Discussion

To add structure to the output, we can package up a sequence of tokens into a block. There are a couple of attributes associated with a block. The *offset* indicates how much to indent subsequent lines of the block if a break is necessary. When a block is longer than a line then we have a number of options. We can display as much on each line as possible, only breaking when really necessary. Alternatively, we can break the block at each break point, e.g.:

```
aaa             aaa bbb
bbb             ccc ddd
ccc
ddd
```

The choice of layout depends on whether the *break-type* attribute of the block is `#"consistent"` or `#"inconsistent"`. The third alternative is `#"fit"`. This suppresses all breaks and truncates the output if it won't fit on a line.

The size of the block is cached in the block for efficiency.

ppml-block Function

Signature ppml-block (constituents #key offset type) => (ppml)

Parameters

- **constituents** – An instance of <ppml-sequence>.
- **offset** (*#key*) – An instance of <nat>.
- **type** (*#key*) – An instance of <ppml-break-type>.

Values

- **ppml** – An instance of <ppml-block>.

Discussion Construct a <ppml-block>.

<ppml-break> Class

Superclasses <ppml>

Init-Keywords

- **blank-space** – An instance of <nat>.

- **offset** – An instance of *<nat>*.

Discussion A <ppml-break> indicates a position in the output where it is permissible to break the output if it won't fit on a single line. If we don't need to break the line then we output blank-space spaces. If we do need to break then we indent offset spaces relative to the current line indent.

ppml-break Function

Signature ppml-break (#key space offset) => (ppml)

Parameters

- **space** (*#key*) – An instance of *<nat>*.
- **offset** (*#key*) – An instance of *<nat>*.

Values

- **ppml** – An instance of *<ppml-break>*.

Discussion Construct a *<ppml-break>*.

<ppml-browser-aware-object> Class

Superclasses *<ppml>*

Init-Keywords

- **object** – An instance of <object>.

Discussion The browser "knows" about some of the objects manipulated by the compiler, e.g. the various kinds of definition, and so we store these directly. Furthermore we recompute the ppml representation of the object every time token-size is called as the representation may depend on browser settings.

ppml-browser-aware-object Function

Signature ppml-browser-aware-object (o) => (ppml)

Parameters

- **o** – An instance of <object>.

Values

- **ppml** – An instance of *<ppml-browser-aware-object>*.

Discussion Construct a *<ppml-browser-aware-object>*.

<ppml-separator-block> Class

Superclasses *<ppml-block>*

Init-Keywords

- **separator** – An instance of *<ppml-sequence>*.

Discussion

When constructing blocks representing collections it is wasteful to explicitly store the separators between elements. The *<ppml-separator-block>* class captures this common case.

The default value for the *separator* is `vector(ppml-string(","), ppml-break(space: 1))`.

ppml-separator-block Function

Signature ppml-separator-block (constituents #key separator offset type left-bracket right-bracket) => (ppml)

Parameters

- **constituents** – An instance of *<ppml-sequence>*.
- **separator** (*#key*) – An instance of *<ppml-sequence>*.
- **offset** (*#key*) – An instance of *<nat>*.
- **type** (*#key*) – An instance of *<ppml-break-type>*. The default value is `#"inconsistent"`.
- **left-bracket** (*#key*) – An instance of `false-or(<ppml>)`.
- **right-bracket** (*#key*) – An instance of `false-or(<ppml>)`.

Values

- **ppml** – An instance of *<ppml>*.

Discussion Construct a *<ppml-separator-block>*.

<ppml-string> Class

Superclasses *<ppml>*

Init-Keywords

- **string** – An instance of `<byte-string>`.

Discussion The simplest ppml token is just a string.

ppml-string Function

Signature ppml-string (str) => (ppml)

Parameters

- **str** – An instance of `<byte-string>`.

Values

- **ppml** – An instance of *<ppml-string>*.

Discussion Construct a *<ppml-string>*.

<ppml-suspension> Class

Superclasses *<ppml>*

Init-Keywords

- **cache-token?** – An instance of `<boolean>`. The default value is `#t`.
- **pair** – Either an instance of *<ppml>* or a `<pair>` of `<function>` and its arguments.

Discussion Sometimes it is more space efficient to delay the construction of the *<ppml>* equivalent of an object until we need to print it. The *<ppml-suspension>* class supports this. It contains either a *<ppml>* token, or a pair of a function and its arguments. When we need a token and encounter the pair we apply the function to its arguments. This should return an instance of *<ppml>*. Optionally we can overwrite the pair by the result.

ppml-suspension Function

Signature ppml-suspension (fun #rest args) => (ppml)

Parameters

- **fun** – An instance of <function>.
- **args** (*#rest*) – An instance of <object>.

Values

- **ppml** – An instance of *<ppml-suspension>*.

Discussion Construct a *<ppml-suspension>*.

Conversion to PPML

as(class == <ppml>, <object>) Method

Parameters

- **class** – The class *<ppml>*.
- **object** – An instance of <object>.

Values

- **ppml** – An instance of *<ppml>*.

Discussion Returns the result of calling `print-object` on the object as PPML. (It does this by using `"%="` along with `format-to-string`.

as(class == <ppml>, <byte-string>) Method

Parameters

- **class** – The class *<ppml>*.
- **object** – An instance of <byte-string>.

Values

- **ppml** – An instance of *<ppml>*.

Discussion Returns the quoted string value as PPML.

as(class == <ppml>, <symbol>) Method

Parameters

- **class** – The class *<ppml>*.
- **object** – An instance of <symbol>.

Values

- **ppml** – An instance of *<ppml>*.

Discussion Returns the string value of the symbol as PPML.

as(class == <ppml>, <collection>) Method

Parameters

- **class** – The class *<ppml>*.
- **object** – An instance of <collection>.

Values

- **ppml** – An instance of *<ppml>*.

Discussion Returns PPML representing the collection as a comma-separated list surrounded by `#(...)`.

as(class == <ppml>, <explicit-key-collection>) Method

Parameters

- **class** – The class *<ppml>*.
- **object** – An instance of <explicit-key-collection>.

Values

- **ppml** – An instance of *<ppml>*.

Discussion Returns PPML representing the collection.

as(class == <ppml>, <vector>) Method

Parameters

- **class** – The class *<ppml>*.
- **object** – An instance of <vector>.

Values

- **ppml** – An instance of *<ppml>*.

Discussion Returns PPML representing the vector as a comma-separated list surrounded by `#[...]`.

as(class == <ppml>, <list>) Method

Parameters

- **class** – The class *<ppml>*.
- **object** – An instance of <list>.

Values

- **ppml** – An instance of *<ppml>*.

Discussion Returns PPML representing the list as a comma-separated list surrounded by `#(...)`.

Printing / Formatting

format-to-ppml Generic function

Signature format-to-ppml (string #rest args) => (ppml)

Parameters

- **string** – An instance of <byte-string>.
- **args** (*#rest*) – An instance of <object>.

Values

- **ppml** – An instance of *<ppml>*.

Discussion We insert breaks at the places where arguments are inserted in the format string. This will hopefully give us reasonable output, but not always as good as we could do by hand. We separate out the processing of the format string so that we can share the constant components of the resulting ppml-block if the same format expression is used multiple times.

ppml-format-string Generic function

Signature ppml-format-string (string) => (f)

Parameters

- **string** – An instance of <byte-string>.

Values

- **f** – An instance of <function>.

Discussion Used by *format-to-ppml*.

ppml-print Generic function

Signature ppml-print (t pp) => ()

Parameters

- **t** – An instance of *<ppml>*.
- **pp** – An instance of *<ppml-printer>*.

Discussion This is the best way to display PPML.

ppml-print-one-line Generic function

Signature ppml-print-one-line (t pp) => ()

Parameters

- **t** – An instance of *<ppml>*.
- **pp** – An instance of *<ppml-printer>*.

Discussion This prints in the same way as *ppml-print*, but will limit the output to a single line. It does this by internally using a line-break-type of #"fit".

<ppml-printer> Class

Superclasses <object>

Init-Keywords

- **margin** – An instance of *<nat>*.
- **newline-function** – An instance of <function>, taking no arguments. The default value writes a newline to *standard-output*.
- **output-function** – An instance of <function>, taking a single <string> argument. The default value writes to *standard-output*.
- **terse-depth** – An instance of <integer>. The default value is 100.

Discussion

When outputting ppml we need to keep track of the space left on the current line and the current margin. We store these values in a *<ppml-printer>* object, along with the functions used to display text and line breaks.

The *terse-depth* is used to limit recursion amongst *<ppml-block>* instances. Once printing has recursed through *terse-depth* blocks, it will change the *break-type* to #"fit" to abbreviate things.

Type Aliases and Constants

<nat> Type

Equivalent limited(<integer>, min: 0)

<ppml-break-type> Type

Equivalent `one-of(#"consistent", #"inconsistent", #"fit")`

<ppml-sequence> Type

Equivalent `<sequence>`

Discussion

> **Note:** This should be `limited(<sequence>, of: <ppml>)`.

$line-break Constant

Value `ppml-break(space: 999)`

Discussion A way to force a line break by making a break with a space larger than the column width.

CHAPTER

NINE

GLOSSARY

canonical sources only meaningful in reference to a particular compiler database or compilation context. The set of sources from which the information in the database was derived.

compilation context the in-memory representation of an open compiler database. Consists of a connection to a disk database, a reference to the owning project, and a collection of caches for pre-loaded/pre-computed information.

compiler database a file or set of files containing information derived from compiling a project. A project can have multiple compiler databases, corresponding to different target machines, compiler settings, or even different versions of the project sources. The project manager is responsible for managing the collection of project databases and telling the compilation system which database to use.

component a native DLL or EXE file.

DFMC the Dylan Flow Machine Compiler, the intermediate representation on which type inference and optimization (dispatch, inline, dynamic extent, common subexpression elimination, constant folding, dead code removal) is done. Source of data and control flow elements is in *dfmc/flow-graph*.

execution context the compiler-derived information about a process, such as the namespaces of known runtime components, installed definitions, etc. Initialized from the compilation contexts of the preloaded runtime components and subsequently updated by interactive execution.

HARP Harlequin Abstract RISC Machine, the native back-end of Open Dylan. On this representation register allocation etc is done: input DFM, output: assembly. Source is in *harp* subdirectory.

interactive execution a mechanism for exploratory programming by which the user can execute Dylan forms in an existing process. May allow "out of language" operations such as addition of new variables to existing modules, redefinition of classes, constants, overriding sealing restrictions, etc. Forms to be executed can come from a project or from an interactor.

project a development environment object representing a Dylan program under development. It corresponds to a Dylan library, but it exists before the compiler is ever invoked. It is used by the compiler only to identify a library in interactions with the project manager.

project manager the part of the development environment charged with managing projects. The project manager is a client of the compilation system, i.e. it is the project manager which is expected to invoke many of the functions in this API.

runtime component a runtime manager object representing a component loaded into a tethered process.

interactor a mechanism by which a user can type in source records for interactive execution without modifying project sources.

runtime manager the part of the development environment charged with controlling the runtime. The compilation system is a client of the runtime manager, i.e. the compilation system will invoke runtime manager functions as needed to effect interactive execution.

source record smallest unit of source suitable for compilation. Consists of a stream of characters and a module name. Contains complete top-level forms (i.e. top-level forms may not be split across source records).

tether a runtime manager object representing a debuggable process on the runtime. Sometimes referred to as an "access-path", but I'm staying away from that term because it seems to be used differently in different documents.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- search

A

C

D

E

F

I

R

S

W

Symbols

A

C

D