Design of CMU Common Lisp

Robert A. MacLachlan (ed)

January 15, 2003

Abstract

This report documents internal details of the CMU Common Lisp compiler and run-time system. CMU Common Lisp is a public domain implementation of Common Lisp that runs on various Unix workstations. This document is a work in progress: neither the contents nor the presentation are completed. Nevertheless, it provides some useful background information, in particular regarding the CMUCL compiler.

Contents

Part I

System Architecture

Chapter 1

Package and File Structure

1.1 Source Tree Structure

The CMUCL source tree has subdirectories for each major subsystem:

`assembly/` Holds the CMU CL source-file assembler, and has machine specific subdirectories holding assembly code for that architecture.

`clx/` The CLX interface to the X11 window system.

`code/` The Lisp code for the runtime system and standard CL utilities.

`compiler/` The Python compiler. Has architecture-specific subdirectories which hold backends for different machines. The `generic` subdirectory holds code that is shared across most backends.

`hemlock/` The Hemlock editor.

`lisp/` The C runtime system code and low-level Lisp debugger.

`pcl/` CMUCL version of the PCL implementation of CLOS.

`tools/` System building command files and source management tools.

1.2 Package structure

Goals: with the single exception of LISP, we want to be able to export from the package that the code lives in.

Mach, CLX... — These Implementation-dependent system-interface packages provide direct access to specific features available in the operating system environment, but hide details of how OS communication is done.

system contains code that must know about the operating system environment: I/O, etc. Hides the operating system environment. Provides OS interface extensions such as `print-directory`, etc.

kernel hides state and types used for system integration: package system, error system, streams (?), reader, printer. Also, hides the VM, in that we don't export anything that reveals the VM interface. Contains code that needs to use the VM and SYSTEM interface, but is independent of OS and VM details. This code shouldn't need to be changed in any port of CMU CL, but won't work when plopped into an arbitrary CL. Uses SYSTEM, VM, EXTENSIONS. We export "hidden" symbols related to implementation of CL: setf-inverses, possibly some global variables.

The boundary between KERNEL and VM is fuzzy, but this fuzziness reflects the fuzziness in the definition of the VM. We can make the VM large, and bring everything inside, or we can make it small. Obviously, we want the VM to be as small as possible, subject to efficiency constraints. Pretty much all of the code in KERNEL could be put in VM. The issue is more what VM hides from KERNEL: VM knows about everything.

lisp Originally, this package had all the system code in it. The current ideal is that this package should have *no* code in it, and only exist to export the standard interface. Note that the name has been changed by x3j13 to common-lisp.

extensions contains code that any random user could have written: list operations, syntactic sugar macros. Uses only LISP, so code in EXTENSIONS is pure CL. Exports everything defined within that is useful elsewhere. This package doesn't hide much, so it is relatively safe for users to use EXTENSIONS, since they aren't getting anything they couldn't have written themselves. Contrast this to KERNEL, which exports additional operations on CL's primitive data structures: PACKAGE-INTERNAL-SYMBOL-COUNT, etc. Although some of the functionality exported from KERNEL could have been defined in CL, the kernel implementation is much more efficient because it knows about implementation internals. Currently this package contains only extensions to CL, but in the ideal scheme of things, it should contain the implementations of all CL functions that are in KERNEL (the library.)

VM hides information about the hardware and data structure representations. Contains all code that knows about this sort of thing: parts of the compiler, GC, etc. The bulk of the code is the compiler back-end. Exports useful things that are meaningful across all implementations, such as operations for examining compiled functions, system constants. Uses COMPILER and whatever else it wants. Actually, there are different *machine*-`VM` packages for each target implementation. VM is a nickname for whatever implementation we are currently targeting for.

compiler hides the algorithms used to map Lisp semantics onto the operations supplied by the VM. Exports the mechanisms used for defining the VM. All the VM-independent code in the compiler, partially hiding the compiler intermediate representations. Uses KERNEL.

eval holds code that does direct execution of the compiler's ICR. Uses KERNEL, COMPILER. Exports debugger interface to interpreted code.

debug-internals presents a reasonable, unified interface to manipulation of the state of both compiled and interpreted code. (could be in KERNEL) Uses VM, INTERPRETER, EVAL, KERNEL.

debug holds the standard debugger, and exports the debugger

Chapter 2

System Building

It's actually rather easy to build a CMU CL core with exactly what you want in it. But to do this you need two things: the source and a working CMU CL.

Basically, you use the working copy of CMU CL to compile the sources, then run a process call "genesis" which builds a "kernel" core. You then load whatever you want into this kernel core, and save it.

In the `tools/` directory in the sources there are several files that compile everything, and build cores, etc. The first step is to compile the C startup code.

Note: *the various scripts mentioned below have hard-wired paths in them set up for our directory layout here at CMU. Anyone anywhere else will have to edit them before they will work.*

2.1 Compiling the C Startup Code

There is a circular dependancy between lisp/internals.h and lisp/lisp.map that causes bootstrapping problems. The easiest way to get around this problem is to make a fake lisp.nm file that has nothing in it but a version number:

```
% echo "Map file for lisp version 0" > lisp.nm
```

and then run genesis with NIL for the list of files:

```
* (load ".../compiler/generic/new-genesis") ; compile before loading
* (lisp::genesis nil ".../lisp/lisp.nm" "/dev/null"
        ".../lisp/lisp.map" ".../lisp/lisp.h")
```

It will generate a whole bunch of warnings about things being undefined, but ignore that, because it will also generate a correct lisp.h. You can then compile lisp producing a correct lisp.map:

```
% make
```

and then use `tools/do-worldbuild` and `tools/mk-lisp` to build `kernel.core` and `lisp.core` (see section 2.3.)

2.2 Compiling the Lisp Code

The `tools` directory contains various lisp and C-shell utilities for building CMU CL:

compile-all* Will compile lisp files and build a kernel core. It has numerous command-line options to control what to compile and how. Try -help to see a description. It runs a separate Lisp process to compile each subsystem. Error output is generated in files with "`.log`" extension in the root of the build area.

setup.lisp Some lisp utilities used for compiling changed files in batch mode and collecting the error output. Sort of a crude defsystem. Loads into the "user" package. See `with-compiler-log-file` and `comf`.

*foo***com.lisp** Each system has a "`.lisp`" file in `tools/` which compiles that system.

2.3 Building Core Images

Both the kernel and final core build are normally done using shell script drivers:

do-worldbuild* Builds a kernel core for the current machine. The version to build is indicated by an optional argument, which defaults to "alpha". The `kernel.core` file is written either in the `lisp/` directory in the build area, or in `/usr/tmp/`. The directory which already contains `kernel.core` is chosen. You can create a dummy version with e.g. "touch" to select the initial build location.

mk-lisp* Builds a full core, with conditional loading of subsystems. The version is the first argument, which defaults to "alpha". Any additional arguments are added to the `*features*` list, which controls system loading (among other things.) The `lisp.core` file is written in the current working directory.

These scripts load Lisp command files. When `tools/worldbuild.lisp` is loaded, it calls genesis with the correct arguments to build a kernel core. Similarly, `worldload.lisp` builds a full core. Adding certain symbols to `*features*` before loading worldload.lisp suppresses loading of different parts of the system. These symbols are:

:no-compiler don't load the compiler.

:no-clx don't load CLX.

:no-clm don't load CLM.

:no-hemlock don't load Hemlock.

:no-pcl don't load PCL.

:runtime build a runtime code, implies all of the above, and then some.

Note: if you don't load the compiler, you can't (successfully) load the pretty-printer or pcl. And if you compiled hemlock with CLX loaded, you can't load it without CLX also being loaded.

These features are only used during the worldload process; they are not propagated to the generated `lisp.core` file.

Part II

Compiler Organization

Chapter 3

Compiler Overview

The structure of the compiler may be broadly characterized by describing the compilation phases and the data structures that they manipulate. The steps in the compilation are called phases rather than passes since they don't necessarily involve a full pass over the code. The data structure used to represent the code at some point is called an *intermediate representation.*

Two major intermediate representations are used in the compiler:

- The Implicit Continuation Representation (ICR) represents the lisp-level semantics of the source code during the initial phases. Partial evaluation and semantic analysis are done on this representation. ICR is roughly equivalent to a subset of Common Lisp, but is represented as a flow-graph rather than a syntax tree. Phases which only manipulate ICR comprise the "front end". It would be possible to use a different back end such as one that directly generated code for a stack machine.

- The Virtual Machine Representation (VMR) represents the implementation of the source code on a virtual machine. The virtual machine may vary depending on the the target hardware, but VMR is sufficiently stylized that most of the phases which manipulate it are portable.

Each phase is briefly described here. The phases from "local call analysis" to "constraint propagation" all interact; for maximum optimization, they are generally repeated until nothing new is discovered. The source files which primarily contain each phase are listed after "Files: ".

ICR conversion Convert the source into ICR, doing macroexpansion and simple source-to-source transformation. All names are resolved at this time, so we don't have to worry about name conflicts later on. Files: `ir1tran, srctran, typetran`

Local call analysis Find calls to local functions and convert them to local calls to the correct entry point, doing keyword parsing, etc. Recognize once-called functions as lets. Create *external entry points* for entry-point functions. Files: `locall`

Find components Find flow graph components and compute depth-first ordering. Separate top-level code from run-time code, and determine which components are top-level components. Files: `dfo`

ICR optimize A grab-bag of all the non-flow ICR optimizations. Fold constant functions, propagate types and eliminate code that computes unused values. Special-case calls to some known global functions by replacing them with a computed function. Merge blocks and eliminate IF-IFs. Substitute let variables. Files: `ir1opt, ir1tran, typetran, seqtran, vm/vm-tran`

Type constraint propagation Use global flow analysis to propagate information about lexical variable types. Eliminate unnecessary type checks and tests. Files: `constraint`

Type check generation Emit explicit ICR code for any necessary type checks that are too complex to be easily generated on the fly by the back end. Files: `checkgen`

Event driven operations Various parts of ICR are incrementally recomputed, either eagerly on modification of the ICR, or lazily, when the relevant information is needed.

- Check that type assertions are satisfied, marking places where type checks need to be done.

- Locate let calls.
- Delete functions and variables with no references

Files: `ir1util`, `ir1opt`

ICR finalize This phase is run after all components have been compiled. It scans the global variable references, looking for references to undefined variables and incompatible function redefinitions. Files: `ir1final`, `main`.

Environment analysis Determine which distinct environments need to be allocated, and what context needed to be closed over by each environment. We detect non-local exits and set closure variables. We also emit cleanup code as funny function calls. This is the last pure ICR pass. Files: `envanal`

Global TN allocation (GTN) Iterate over all defined functions, determining calling conventions and assigning TNs to local variables. Files: `gtn`

Local TN allocation (LTN) Use type and policy information to determine which VMR translation to use for known functions, and then create TNs for expression evaluation temporaries. We also accumulate some random information needed by VMR conversion. Files: `ltn`

Control analysis Linearize the flow graph in a way that minimizes the number of branches. The block-level structure of the flow graph is basically frozen at this point. Files: `control`

Stack analysis Maintain stack discipline for unknown-values continuation in the presence of local exits. Files: `stack`

Entry analysis Collect some back-end information for each externally callable function.

VMR conversion Convert ICR into VMR by translating nodes into VOPs. Emit type checks. Files: `ir2tran, vmdef`

Copy propagation Use flow analysis to eliminate unnecessary copying of TN values. Files: `copyprop`

Representation selection Look at all references to each TN to determine which representation has the lowest cost. Emit appropriate move and coerce VOPS for that representation.

Lifetime analysis Do flow analysis to find the set of TNs whose lifetimes overlap with the lifetimes of each TN being packed. Annotate call VOPs with the TNs that need to be saved. Files: `life`

Pack Find a legal register allocation, attempting to minimize unnecessary moves. Files: `pack`

Code generation Call the VOP generators to emit assembly code. Files: `codegen`

Pipeline reorganization On some machines, move memory references backward in the code so that they can overlap with computation. On machines with delayed branch instructions, locate instructions that can be moved into delay slots. Files: `assem-opt`

Assembly Resolve branches and convert into object code and fixup information. Files: `assembler`

Dumping Convert the compiled code into an object file or in-core function. Files: `debug-dump`, `dump`, `vm/core`

Chapter 4

The Implicit Continuation Representation

The set of special forms recognized is exactly that specified in the Common Lisp manual. Everything that is described as a macro in CLTL is a macro.

Large amounts of syntactic information are thrown away by the conversion to an anonymous flow graph representation. The elimination of names eliminates the need to represent most environment manipulation special forms. The explicit representation of control eliminates the need to represent BLOCK and GO, and makes flow analysis easy. The full Common Lisp LAMBDA is implemented with a simple fixed-arg lambda, which greatly simplifies later code.

The elimination of syntactic information eliminates the need for most of the "beta transformation" optimizations in Rabbit. There are no progns, no tagbodys and no returns. There are no "close parens" which get in the way of determining which node receives a given value.

In ICR, computation is represented by Nodes. These are the node types:

if Represents all conditionals.

set Represents a `setq`.

ref Represents a constant or variable reference.

combination Represents a normal function call.

MV-combination Represents a `multiple-value-call`. This is used to implement all multiple value receiving forms except for `multiple-value-prog1`, which is implicit.

bind This represents the allocation and initialization of the variables in a lambda.

return This collects the return value from a lambda and represents the control transfer on return.

entry Marks the start of a dynamic extent that can have non-local exits to it. Dynamic state can be saved at this point for restoration on re-entry.

exit Marks a potentially non-local exit. This node is interposed between the non-local uses of a continuation and the `dest` so that code to do a non-local exit can be inserted if necessary.

Some slots are shared between all node types (via defstruct inheritance.) This information held in common between all nodes often makes it possible to avoid special-casing nodes on the basis of type. This shared information is primarily concerned with the order of evaluation and destinations and properties of results. This control and value flow is indicated in the node primarily by pointing to continuations.

The `continuation` structure represents information sufficiently related to the normal notion of a continuation that naming it so seems sensible. Basically, a continuation represents a place in the code, or alternatively the destination of an expression result and a transfer of control. These two notions are bound together for the same reasons that they are related in the standard functional continuation interpretation.

A continuation may be deprived of either or both of its value or control significance. If the value of a continuation is unused due to evaluation for effect, then the continuation will have a null `dest`. If the `next` node for a continuation is deleted by some optimization, then `next` will be `:none`.

[### Continuation kinds...]

The `block` structure represents a basic block, in the the normal sense. Control transfers other than simple sequencing are represented by information in the block structure. The continuation for the last node in a block represents only the destination for the result.

It is very difficult to reconstruct anything resembling the original source from ICR, so we record the original source form in each node. The location of the source form within the input is also recorded, allowing for interfaces such as "Edit Compiler Warnings". See section 25.2.

Forms such as special-bind and catch need to have cleanup code executed at all exit points from the form. We represent this constraint in ICR by annotating the code syntactically within the form with a Cleanup structure describing what needs to be cleaned up. Environment analysis determines the cleanup locations by watching for a change in the cleanup between two continuations. We can't emit cleanup code during ICR conversion, since we don't know which exits will be local until after ICR optimizations are done.

Special binding is represented by a call to the funny function %Special-Bind. The first argument is the Global-Var structure for the variable bound and the second argument is the value to bind it to.

Some subprimitives are implemented using a macro-like mechanism for translating %PRIMITIVE forms into arbitrary lisp code. Subprimitives special-cased by VMR conversion are represented by a call to the funny function %%Primitive. The corresponding Template structure is passed as the first argument.

We check global function calls for syntactic legality with respect to any defined function type function. If the call is illegal or we are unable to tell if it is legal due to non-constant keywords, then we give a warning and mark the function reference as :notinline to force a full call and cause subsequent phases to ignore the call. If the call is legal and is to a known function, then we annotate the Combination node with the Function-Info structure that contains the compiler information for the function.

4.1 Tail sets

#— Probably want to have a GTN-like function result equivalence class mechanism for ICR type inference. This would be like the return value propagation being done by Propagate-From-Calls, but more powerful, less hackish, and known to terminate. The ICR equivalence classes could probably be used by GTN, as well.

What we do is have local call analysis eagerly maintain the equivalence classes of functions that return the same way by annotating functions with a Tail-Info structure shared between all functions whose value could be the value of this function. We don't require that the calls actually be tail-recursive, only that the call deliver its value to the result continuation. [### Actually now done by ICR-OPTIMIZE-RETURN, which is currently making ICR optimize mandatory.]

We can then use the Tail-Set during ICR type inference. It would have a type that is the union across all equivalent functions of the types of all the uses other than in local calls. This type would be recomputed during optimization of return nodes. When the type changes, we would propagate it to all calls to any of the equivalent functions. How do we know when and how to recompute the type for a tail-set? Recomputation is driven by type propagation on the result continuation.

This is really special-casing of RETURN nodes. The return node has the type which is the union of all the non-call uses of the result. The tail-set is found though the lambda. We can then recompute the overall union by taking the union of the type per return node, rather than per-use.

How do result type assertions work? We can't intersect the assertions across all functions in the equivalence class, since some of the call combinations may not happen (or even be possible). We can intersect the assertion of the result with the derived types for non-call uses.

When we do a tail call, we obviously can't check that the returned value matches our assertion. Although in principle, we would like to be able to check all assertions, to preserve system integrity, we only need to check assertions that we depend on. We can afford to lose some assertion information as long as we entirely lose it, ignoring it for type inference as well as for type checking.

Things will work out, since the caller will see the tail-info type as the derived type for the call, and will emit a type check if it needs a stronger result.

A remaining question is whether we should intersect the assertion with per-RETURN derived types from the very beginning (i.e. before the type check pass). I think the answer is yes. We delay the type check pass so that we can get our best guess for the derived type before we decide whether a check is necessary. But with the function return type, we aren't committing to doing any type check when we intersect with the type assertion; the need to type check is still determined in the type check pass by examination of the result continuation.

What is the relationship between the per-RETURN types and the types in the result continuation? The assertion is exactly the Continuation-Asserted-Type (note that the asserted type of result continuations will never change after ICR conversion). The per-RETURN derived type is different than the Continuation-Derived-Type, since it is intersected with the asserted type even before Type Check runs. Ignoring the Continuation-Derived-Type probably makes life simpler anyway, since this breaks the potential circularity of the Tail-Info-Type will affecting the Continuation-Derived-Type, which affects...

When a given return has no non-call uses, we represent this by using *empty-type*. This is consistent with the interpretation that a return type of NIL means the function can't return.

4.2 Hairy function representation

Non-fixed-arg functions are represented using Optional-Dispatch. An Optional-Dispatch has an entry-point function for each legal number of optionals, and one for when extra args are present. Each entry point function is a simple lambda. The entry point function for an optional is passed the arguments which were actually supplied; the entry point function is expected to default any remaining parameters and evaluate the actual function body.

If no supplied-p arg is present, then we can do this fairly easily by having each entry point supply its default and call the next entry point, with the last entry point containing the body. If there are supplied-p args, then entry point function is replaced with a function that calls the original entry function with T's inserted at the position of all the supplied args with supplied-p parameters.

We want to be a bit clever about how we handle arguments declared special when doing optional defaulting, or we will emit really gross code for special optionals. If we bound the arg specially over the entire entry-point function, then the entry point function would be caused to be non-tail-recursive. What we can do is only bind the variable specially around the evaluation of the default, and then read the special and store the final value of the special into a lexical variable which we then pass as the argument. In the common case where the default is a constant, we don't have to special-bind at all, since the computation of the default is not affected by and cannot affect any special bindings.

Keyword and rest args are both implemented using a LEXPR-like "more args" convention. The More-Entry takes two arguments in addition to the fixed and optional arguments: the argument context and count. `(ARG <context> <n>)` accesses the N'th additional argument. Keyword args are implemented directly using this mechanism. Rest args are created by calling %Listify-Rest-Args with the context and count.

The More-Entry parses the keyword arguments and passes the values to the main function as positional arguments. If a keyword default is not constant, then we pass a supplied-p parameter into the main entry and let it worry about defaulting the argument. Since the main entry accepts keywords in parsed form, we can parse keywords at compile time for calls to known functions. We keep around the original parsed lambda-list and related information so that people can figure out how to call the main entry.

4.3 ICR representation of non-local exits

All exits are initially represented by EXIT nodes: How about an Exit node:

```
(defstruct (exit (:include node))
  value)
```

The Exit node uses the continuation that is to receive the thrown Value. During optimization, if we discover that the Cont's home-lambda is the same as the exit node's, then we can delete the Exit node, substituting the Cont for all of the Value's uses.

The successor block of an EXIT is the entry block in the entered environment. So we use the Exit node to mark the place where exit code is inserted. During environment analysis, we need only insert a single block containing the entry point stub.

We ensure that all Exits that aren't for a NLX don't have any Value, so that local exits never require any value massaging.

The Entry node marks the beginning of a block or tagbody:

```
 (defstruct (entry (:include node))
   (continuations nil :type list))
```

It contains a list of all the continuations that the body could exit to. The Entry node is used as a marker for the place to snapshot state, including the control stack pointer. Each lambda has a list of its Entries so that environment analysis can figure out which continuations are really being closed over. There is no reason for optimization to delete Entry nodes, since they are harmless in the degenerate case: we just emit no code (like a no-var let).

We represent CATCH using the lexical exit mechanism. We do a transformation like this:

```
(catch 'foo xxx)  ==>
(block #:foo
  (%catch #'(lambda () (return-from #:foo (%unknown-values))) 'foo)
  (%within-cleanup :catch
    xxx))
```

%CATCH just sets up the catch frame which points to the exit function. %Catch is an ordinary function as far as ICR is concerned. The fact that the catcher needs to be cleaned up is expressed by the Cleanup slots in the continuations in the body. %UNKNOWN-VALUES is a dummy function call which represents the fact that we don't know what values will be thrown.

%WITHIN-CLEANUP is a special special form that instantiates its first argument as the current cleanup when converting the body. In reality, the lambda is also created by the special special form %ESCAPE-FUNCTION, which gives the lambda a special :ESCAPE kind so that the back end knows not to generate any code for it.

We use a similar hack in Unwind-Protect to represent the fact that the cleanup forms can be invoked at arbitrarily random times.

```
     (unwind-protect p c)  ==>
     (flet ((#:cleanup () c))
       (block #:return
(multiple-value-bind
     (#:next #:start #:count)
     (block #:unwind
               (%unwind-protect #'(lambda (x) (return-from #:unwind x)))
               (%within-cleanup :unwind-protect
(return-from #:return p)))
  (#:cleanup)
          (%continue-unwind #:next #:start #:count))))
```

We use the block #:unwind to represent the entry to cleanup code in the case where we are non-locally unwound. Calling of the cleanup function in the drop-through case (or any local exit) is handled by cleanup generation. We make the cleanup a function so that cleanup generation can add calls at local exits from the protected form. #:next, #:start and #:count are state used in the case where we are unwound. They indicate where to go after doing the cleanup and what values are being thrown. The cleanup encloses only the protected form. As in CATCH, the escape function is specially tagged as :ESCAPE. The cleanup function is tagged as :CLEANUP to inhibit let conversion (since references are added in environment analysis.)

Notice that implementing these forms using closures over continuations eliminates any need to special-case ICR flow analysis. Obviously we don't really want to make heap-closures here. In reality these functions are special-cased by the back-end according to their KIND.

4.4 Block compilation

One of the properties of ICR is that it supports "block compilation" by allowing arbitrarily large amounts of code to be converted at once, with actual compilation of the code being done at will.

In order to preserve the normal semantics we must recognize that proclamations (possibly implicit) are scoped. A proclamation is in effect only from the time of appearance of the proclamation to the time it is contradicted. The current global environment at the end of a block is not necessarily the correct global environment for compilation of all the code within the block. We solve this problem by closing over the relevant information in the ICR at the time it is converted. For example, each functional variable reference is marked as inline, notinline or don't care. Similarly, each node contains a structure known as a Cookie which contains the appropriate settings of the compiler policy switches.

We actually convert each form in the file separately, creating a separate "initial component" for each one. Later on, these components are merged as needed. The main reason for doing this is to cause EVAL-WHEN processing to be interleaved with reading.

4.5 Entry points

#—

Since we need to evaluate potentially arbitrary code in the XEP argument forms (for type checking), we can't leave the arguments in the wired passing locations. Instead, it seems better to give the XEP max-args fixed arguments, with the passing locations being the true passing locations. Instead of using %XEP-ARG, we reference the appropriate variable.

Also, it might be a good idea to do argument count checking and dispatching with explicit conditional code in the XEP. This would simplify both the code that creates the XEP and the VMR conversion of XEPs. Also, argument count dispatching would automatically benefit from any cleverness in compilation of case-like forms (jump tables, etc). On the downside, this would push some assumptions about how arg dispatching is done into ICR. But then we are currently violating abstraction at least as badly in VMR conversion, which is also supposed to be implementation independent. —#

As a side-effect of finding which references to known functions can be converted to local calls, we find any references that cannot be converted. References that cannot be converted to a local call must evaluate to a "function object" (or function-entry) that can be called using the full call convention. A function that can be called from outside the component is called an "entry-point".

Lots of stuff that happens at compile-time with local function calls must be done at run-time when an entry-point is called.

It is desirable for optimization and other purposes if all the calls to every function were directly present in ICR as local calls. We cannot directly do this with entry-point functions, since we don't know where and how the entry-point will be called until run-time.

What we do is represent all the calls possible from outside the component by local calls within the component. For each entry-point function, we create a corresponding lambda called the external entry point or XEP. This is a function which takes the number of arguments passed as the first argument, followed by arguments corresponding to each required or optional argument.

If an optional argument is unsupplied, the value passed into the XEP is undefined. The XEP is responsible for doing argument count checking and dispatching.

In the case of a fixed-arg lambda, we emit a call to the %VERIFY-ARGUMENT-COUNT funny function (conditional on policy), then call the real function on the passed arguments. Even in this simple case, we benefit several ways from having a separate XEP:

- The argument count checking is factored out, and only needs to be done in full calls.
- Argument type checking happens automatically as a consequence of passing the XEP arguments in a local call to the real function. This type checking is also only done in full calls.
- The real function may use a non-standard calling convention for the benefit of recursive or block-compiled calls. The XEP converts arguments/return values to/from the standard convention. This also requires little special-casing of XEPs.

If the function has variable argument count (represented by an OPTIONAL-DISPATCH), then the XEP contains a COND which dispatches off of the argument count, calling the appropriate entry-point function (which then does defaulting). If there is a more entry (for keyword or rest args), then the XEP obtains the more arg context and count by calling the %MORE-ARG-CONTEXT funny function.

All non-local-call references to functions are replaced with references to the corresponding XEP. ICR optimization may discover a local call that was previously a non-local reference. When we delete the reference to the XEP, we may find that it has no references. In this case, we can delete the XEP, causing the function to no longer be an entry-point.

Chapter 5

ICR conversion

5.1 Canonical forms

#—

Would be useful to have a Freeze-Type proclamation. Its primary use would be to say that the indicated type won't acquire any new subtypes in the future. This allows better open-coding of structure type predicates, since the possible types that would satisfy the predicate will be constant at compile time, and thus can be compiled as a skip-chain of EQ tests.

Of course, this is only a big win when the subtypes are few: the most important case is when there are none. If the closure of the subtypes is much larger than the average number of supertypes of an inferior, then it is better to grab the list of superiors out of the object's type, and test for membership in that list.

Should type-specific numeric equality be done by EQL rather than =? i.e. should = on two fixnums become EQL and then convert to EQL/FIXNUM? Currently we transform EQL into =, which is complicated, since we have to prove the operands are the class of numeric type before we do it. Also, when EQL sees one operand is a FIXNUM, it transforms to EQ, but the generator for EQ isn't expecting numbers, so it doesn't use an immediate compare.

5.1.1 Array hackery

Array type tests are transformed to `%array-typep`, separation of the implementation-dependent array-type handling. This way we can transform STRINGP to:

```
     (or (simple-string-p x)
 (and (complex-array-p x)
      (= (array-rank x) 1)
      (simple-string-p (%array-data x))))
```

In addition to the similar bit-vector-p, we also handle vectorp and any type tests on which the a dimension isn't wild. [Note that we will want to expand into frobs compatible with those that array references expand into so that the same optimizations will work on both.]

These changes combine to convert hairy type checks into hairy typep's, and then convert hairyp typeps into simple typeps.

Do we really need non-VOP templates? It seems that we could get the desired effect through implementation-dependent ICR transforms. The main risk would be of obscuring the type semantics of the code. We could fairly easily retain all the type information present at the time the tranform is run, but if we discover new type information, then it won't be propagated unless the VM also supplies type inference methods for its internal frobs (precluding the use of `%PRIMITIVE`, since primitives don't have derive-type methods.)

I guess one possibility would be to have the call still considered "known" even though it has been transformed. But this doesn't work, since we start doing LET optimizations that trash the arglist once the call has been transformed (and indeed we want to.)

Actually, I guess the overhead for providing type inference methods for the internal frobs isn't that great, since we can usually borrow the inference method for a Common Lisp function. For example, in our AREF case:

```
    (aref x y)
==>
    (let ((#:len (array-dimension x 0)))
      (%unchecked-aref x (%check-in-bounds y #:len)))
```

Now in this case, if we made `%UNCHECKED-AREF` have the same derive-type method as AREF, then if we discovered something new about X's element type, we could derive a new type for the entire expression.

Actually, it seems that baring this detail at the ICR level is beneficial, since it admits the possibility of optimizing away the bounds check using type information. If we discover X's dimensions, then `#:LEN` becomes a constant that can be substituted. Then `%CHECK-IN-BOUNDS` can notice that the bound is constant and check it against the type for Y. If Y is known to be in range, then we can optimize away the bounds check.

Actually in this particular case, the best thing to do would be if we discovered the bound is constant, then replace the bounds check with an implicit type check. This way all the type check optimization mechanisms would be brought into the act.

So we actually want to do the bounds-check expansion as soon as possible, rather than later than possible: it should be a source-transform, enabled by the fast-safe policy.

With multi-dimensional arrays we probably want to explicitly do the index computation: this way portions of the index computation can become loop invariants. In a scan in row-major order, the inner loop wouldn't have to do any multiplication: it would only do an addition. We would use normal fixnum arithmetic, counting on * to cleverly handle multiplication by a constant, and appropriate inline expansion.

Note that in a source transform, we can't make any assumptions the type of the array. If it turns out to be a complex array without declared dimensions, then the calls to ARRAY-DIMENSION will have to turn into a VOP that can be affected. But if it is simple, then the VOP is unaffected, and if we know the bounds, it is constant. Similarly, we would have operations. is simple. [This is somewhat inefficient when the array isn't eventually discovered to be simple, since finding the data and finding the displacement duplicate each other. We could make optimize to (VALUES (optimization of trivial VALUES uses.]

Also need (THE (ARRAY * * * ...) x) to assert correct rank.

—#

A bunch of functions have source transforms that convert them into the canonical form that later parts of the compiler want to see. It is not legal to rely on the canonical form since source transforms can be inhibited by a Notinline declaration. This shouldn't be a problem, since everyone should keep their hands off of Notinline calls.

Some transformations:

```
Endp  ==>  (NULL (THE LIST ...))
(NOT xxx) or (NULL xxx) => (IF xxx NIL T)

(typep x '<simple type>) => (<simple predicate> x)
(typep x '<complex type>) => ...composition of simpler operations...
```

TYPEP of AND, OR and NOT types turned into conditionals over multiple TYPEP calls. This makes hairy TYPEP calls more digestible to type constraint propagation, and also means that the TYPEP code generators don't have to deal with these cases. [### In the case of union types we may want to do something to preserve information for type constraint propagation.]

```
    (apply \#'foo a b c)
==>
    (multiple-value-call \#'foo (values a) (values b) (values-list c))
```

This way only MV-CALL needs to know how to do calls with unknown numbers of arguments. It should be nearly as efficient as a special-case VMR-Convert method could be.

```
Make-String => Make-Array
N-arg predicates associated into two-arg versions.
Associate N-arg arithmetic ops.
Expand CxxxR and FIRST...nTH
Zerop, Plusp, Minusp, 1+, 1-, Min, Max, Rem, Mod
```

```
(Values x), (Identity x) => (Prog1 x)

All specialized aref functions => (aref (the xxx) ...)
```

Convert (ldb (byte ...) ...) into internal frob that takes size and position as separate args. Other byte functions also...

Change for-value primitive predicates into `(if <pred> t nil)`. This isn't particularly useful during ICR phases, but makes life easy for VMR conversion.

This last can't be a source transformation, since a source transform can't tell where the form appears. Instead, ICR conversion special-cases calls to known functions with the Predicate attribute by doing the conversion when the destination of the result isn't an IF. It isn't critical that this never be done for predicates that we ultimately discover to deliver their value to an IF, since IF optimizations will flush unnecessary IFs in a predicate.

5.2 Inline functions

[### Inline expansion is especially powerful in the presence of good lisp-level optimization ("partial evaluation"). Many "optimizations" usually done in Lisp compilers by special-case source-to-source transforms can be had simply by making the source of the general case function available for inline expansion. This is especially helpful in Common Lisp, which has many commonly used functions with simple special cases but bad general cases (list and sequence functions, for example.)

Inline expansion of recursive functions is allowed, and is not as silly as it sounds. When expanded in a specific context, much of the overhead of the recursive calls may be eliminated (especially if there are many keyword arguments, etc.)

[Also have MAYBE-INLINE]]

We only record a function's inline expansion in the global environment when the function is in the null lexical environment, since the expansion must be represented as source.

We do inline expansion of functions locally defined by FLET or LABELS even when the environment is not null. Since the appearances of the local function must be nested within the desired environment, it is possible to expand local functions inline even when they use the environment. We just stash the source form and environments in the Functional for the local function. When we convert a call to it, we just reconvert the source in the saved environment.

An interesting alternative to the inline/full-call dichotomy is "semi-inline" coding. Whenever we have an inline expansion for a function, we can expand it only once per block compilation, and then use local call to call this copied version. This should get most of the speed advantage of real inline coding with much less code bloat. This is especially attractive for simple system functions such as Read-Char.

The main place where true inline expansion would still be worth doing is where large amounts of the function could be optimized away by constant folding or other optimizations that depend on the exact arguments to the call.

5.3 Compilation policy

We want more sophisticated control of compilation safety than is offered in CL, so that we can emit only those type checks that are likely to discover something (i.e. external interfaces.)

5.4 Notes

Generalized back-end notion provides dynamic retargeting? (for byte code)

The current node type annotations seem to be somewhat unsatisfactory, since we lose information when we do a THE on a continuation that already has uses, or when we convert a let where the actual result continuation has other uses.

But the case with THE isn't really all that bad, since the test of whether there are any uses happens before conversion of the argument, thus THE loses information only when there are uses outside of the declared form. The LET case may not be a big deal either.

Note also that losing user assertions isn't really all that bad, since it won't damage system integrity. At worst, it will cause a bug to go undetected. More likely, it will just cause the error to be signaled in a different place (and possibly in a less informative way). Of course, there is an efficiency hit for losing type information, but if it only happens in strange cases, then this isn't a big deal.

Chapter 6

Local call analysis

All calls to local functions (known named functions and LETs) are resolved to the exact LAMBDA node which is to be called. If the call is syntactically illegal, then we emit a warning and mark the reference as :notinline, forcing the call to be a full call. We don't even think about converting APPLY calls; APPLY is not special-cased at all in ICR. We also take care not to convert calls in the top-level component, which would join it to normal code. Calls to functions with rest args and calls with non-constant keywords are also not converted.

We also convert MV-Calls that look like MULTIPLE-VALUE-BIND to local calls, since we know that they can be open-coded. We replace the optional dispatch with a call to the last optional entry point, letting MV-Call magically default the unsupplied values to NIL.

When ICR optimizations discover a possible new local call, they explicitly invoke local call analysis on the code that needs to be reanalyzed.

[### Let conversion. What it means to be a let. Argument type checking done by caller. Significance of local call is that all callers are known, so special call conventions may be used.] A lambda called in only one place is called a "let" call, since a Let would turn into one.

In addition to enabling various ICR optimizations, the let/non-let distinction has important environment significance. We treat the code in function and all of the lets called by that function as being in the same environment. This allows exits from lets to be treated as local exits, and makes life easy for environment analysis.

Since we will let-convert any function with only one call, we must be careful about cleanups. It is possible that a lexical exit from the let function may have to clean up dynamic bindings not lexically apparent at the exit point. We handle this by annotating lets with any cleanup in effect at the call site. The cleanup for continuations with no immediately enclosing cleanup is the lambda that the continuation is in. In this case, we look at the lambda to see if any cleanups need to be done.

Let conversion is disabled for entry-point functions, since otherwise we might convert the call from the XEP to the entry point into a let. Then later on, we might want to convert a non-local reference into a local call, and not be able to, since once a function has been converted to a let, we can't convert it back.

A function's return node may also be deleted if it is unreachable, which can happen if the function never returns normally. Such functions are not lets.

Chapter 7

Find components

This is a post-pass to ICR conversion that massages the flow graph into the shape subsequent phases expect. Things done: Compute the depth-first ordering for the flow graph. Find the components (disconnected parts) of the flow graph.

This pass need only be redone when newly converted code has been added to the flow graph. The reanalyze flag in the component structure should be set by people who mess things up.

We create the initial DFO using a variant of the basic algorithm. The initial DFO computation breaks the ICR up into components, which are parts that can be compiled independently. This is done to increase the efficiency of large block compilations. In addition to improving locality of reference and reducing the size of flow analysis problems, this allows back-end data structures to be reclaimed after the compilation of each component.

ICR optimization can change the connectivity of the flow graph by discovering new calls or eliminating dead code. Initial DFO determination splits up the flow graph into separate components, but does so conservatively, ensuring that parts that might become joined (due to local call conversion) are joined from the start. Initial DFO computation also guarantees that all code which shares a lexical environment is in the same component so that environment analysis needs to operate only on a single component at a time.

[This can get a bit hairy, since code seemingly reachable from the environment entry may be reachable from a NLX into that environment. Also, function references must be considered as links joining components even though the flow graph doesn't represent these.]

After initial DFO determination, components are neither split nor joined. The standard DFO computation doesn't attempt to split components that have been disconnected.

Chapter 8

ICR optimize

Somewhere describe basic ICR utilities: continuation-type, constant-continuation-p, etc. Perhaps group by type in ICR description?

We are conservative about doing variable-for-variable substitution in ICR optimization, since if we substitute a variable with a less restrictive type, then we may prevent use of a "good" representation within the scope of the inner binding.

Note that variable-variable substitutions aren't really crucial in ICR, since they don't create opportunities for new optimizations (unlike substitution of constants and functions). A spurious variable-variable binding will show up as a Move operation in VMR. This can be optimized away by reaching-definitions and also by targeting. [### But actually, some optimizers do see if operands are the same variable.]

#—

The IF-IF optimization can be modeled as a value driven optimization, since adding a use definitely is cause for marking the continuation for reoptimization. [When do we add uses? Let conversion is the only obvious time.] I guess IF-IF conversion could also be triggered by a non-immediate use of the test continuation becoming immediate, but to allow this to happen would require Delete-Block (or somebody) to mark block-starts as needing to be reoptimized when a predecessor changes. It's not clear how important it is that IF-IF conversion happen under all possible circumstances, as long as it happens to the obvious cases.

[### It isn't totally true that code flushing never enables other worthwhile optimizations. Deleting a functional reference can cause a function to cease being an XEP, or even trigger let conversion. It seems we still want to flush code during ICR optimize, but maybe we want to interleave it more intimately with the optimization pass.

Ref-flushing works just as well forward as backward, so it could be done in the forward pass. Call flushing doesn't work so well, but we could scan the block backward looking for any new flushable stuff if we flushed a call on the forward pass.

When we delete a variable due to lack of references, we leave the variable in the lambda-list so that positional references still work. The initial value continuation is flushed, though (replaced with NIL) allowing the initial value for to be deleted (modulo side-effects.)

Note that we can delete vars with no refs even when they have sets. I guess when there are no refs, we should also flush all sets, allowing the value expressions to be flushed as well.

Squeeze out single-reference unset let variables by changing the dest of the initial value continuation to be the node that receives the ref. This can be done regardless of what the initial value form is, since we aren't actually moving the evaluation. Instead, we are in effect using the continuation's locations in place of the temporary variable.

Doing this is of course, a wild violation of stack discipline, since the ref might be inside a loop, etc. But with the VMR back-end, we only need to preserve stack discipline for unknown-value continuations; this ICR transformation must be already inhibited when the DEST of the REF is a multiple-values receiver (EXIT, RETURN or MV-COMBINATION), since we must preserve the single-value semantics of the let-binding in this case.

The REF and variable must be deleted as part of this operation, since the ICR would otherwise be left in an inconsistent state; we can't wait for the REF to be deleted due to being unused, since we have grabbed the arg continuation and substituted it into the old DEST.

The big reason for doing this transformation is that in macros such as INCF and PSETQ, temporaries are squeezed out, and the new value expression is evaluated directly to the setter, allowing any result type assertion to be applied to the expression evaluation. Unlike in the case of substitution, there is no point in inhibiting this transformation when the initial value type is weaker than the variable type. Instead, we intersect the asserted type for the old REF's CONT with the type assertion on the initial value continuation. Note that the variable's type has already been asserted on the initial-value continuation.

Of course, this transformation also simplifies the ICR even when it doesn't discover interesting type assertions, so it makes sense to do it whenever possible. This reduces the demands placed on register allocation, etc.

There are three dead-code flushing rules:

1. Refs with no DEST may be flushed.

2. Known calls with no dest that are flushable may be flushed. We null the DEST in all the args.

3. If a lambda-var has no refs, then it may be deleted. The flushed argument continuations have their DEST nulled.

These optimizations all enable one another. We scan blocks backward, looking for nodes whose CONT has no DEST, then type-dispatching off of the node. If we delete a ref, then we check to see if it is a lambda-var with no refs. When we flush an argument, we mark the blocks for all uses of the CONT as needing to be reoptimized.

8.1 Goals for ICR optimizations

#—

When an optimization is disabled, code should still be correct and not ridiculously inefficient. Phases shouldn't be made mandatory when they have lots of non-required stuff jammed into them.

—#

This pass is optional, but is desirable if anything is more important than compilation speed.

This phase is a grab-bag of optimizations that concern themselves with the flow of values through the code representation. The main things done are type inference, constant folding and dead expression elimination. This phase can be understood as a walk of the expression tree that propagates assertions down the tree and propagates derived information up the tree. The main complication is that there isn't any expression tree, since ICR is flow-graph based.

We repeat this pass until we don't discover anything new. This is a bit of feat, since we dispatch to arbitrary functions which may do arbitrary things, making it hard to tell if anything really happened. Even if we solve this problem by requiring people to flag when they changed or by checking to see if they changed something, there are serious efficiency problems due to massive redundant computation, since in many cases the only way to tell if anything changed is to recompute the value and see if it is different from the old one.

We solve this problem by requiring that optimizations for a node only depend on the properties of the CONT and the continuations that have the node as their DEST. If the continuations haven't changed since the last pass, then we don't attempt to re-optimize the node, since we know nothing interesting will happen.

We keep track of which continuations have changed by a REOPTIMIZE flag that is set whenever something about the continuation's value changes.

When doing the bottom up pass, we dispatch to type specific code that knows how to tell when a node needs to be reoptimized and does the optimization. These node types are special-cased: COMBINATION, IF, RETURN, EXIT, SET.

The REOPTIMIZE flag in the COMBINATION-FUN is used to detect when the function information might have changed, so that we know when there are new assertions that could be propagated from the function type to the arguments.

When we discover something about a leaf, or substitute for leaf, we reoptimize the CONT for all the REF and SET nodes.

We have flags in each block that indicate when any nodes or continuations in the block need to be re-optimized, so we don't have to scan blocks where there is no chance of anything happening.

It is important for efficiency purposes that optimizers never say that they did something when they didn't, but this by itself doesn't guarantee timely termination. I believe that with the type system implemented, type inference will converge in finite time, but as a practical matter, it can take far too long to discover not much. For this reason, ICR optimization is terminated after three consecutive passes that don't add or delete code. This premature termination only happens 2% of the time.

8.2 Flow graph simplification

Things done:

- Delete blocks with no predecessors.
- Merge blocks that can be merged.
- Convert local calls to Let calls.
- Eliminate degenerate IFs.

We take care not to merge blocks that are in different functions or have different cleanups. This guarantees that non-local exits are always at block ends and that cleanup code never needs to be inserted within a block.

We eliminate IFs with identical consequent and alternative. This would most likely happen if both the consequent and alternative were optimized away.

[Could also be done if the consequent and alternative were different blocks, but computed the same value. This could be done by a sort of cross-jumping optimization that looked at the predecessors for a block and merged code shared between predecessors. IFs with identical branches would eventually be left with nothing in their branches.]

We eliminate IF-IF constructs:

```
(IF (IF A B C) D E) ==>
(IF A (IF B D E) (IF C D E))
```

In reality, what we do is replicate blocks containing only an IF node where the predicate continuation is the block start. We make one copy of the IF node for each use, leaving the consequent and alternative the same. If you look at the flow graph representation, you will see that this is really the same thing as the above source to source transformation.

8.3 Forward ICR optimizations

In the forward pass, we scan the code in forward depth-first order. We examine each call to a known function, and:

- Eliminate any bindings for unused variables.
- Do top-down type assertion propagation. In local calls, we propagate asserted and derived types between the call and the called lambda.
- Replace calls of foldable functions with constant arguments with the result. We don't have to actually delete the call node, since Top-Down optimize will delete it now that its value is unused.
- Run any Optimizer for the current function. The optimizer does arbitrary transformations by hacking directly on the IR. This is useful primarily for arithmetic simplification and similar things that may need to examine and modify calls other than the current call. The optimizer is responsible for recording any changes that it makes. An optimizer can inhibit further optimization of the node during the current pass by returning true. This is useful when deleting the node.
- Do ICR transformations, replacing a global function call with equivalent inline lisp code.
- Do bottom-up type propagation/inferencing. For some functions such as Coerce we will dispatch to a function to find the result type. The Derive-Type function just returns a type structure, and we check if it is different from the old type in order to see if there was a change.
- Eliminate IFs with predicates known to be true or false.

- Substitute the value for unset let variables that are bound to constants, unset lambda variables or functionals.
- Propagate types from local call args to var refs.

We use type info from the function continuation to find result types for functions that don't have a derive-type method.

8.3.1 ICR transformation

ICR transformation does "source to source" transformations on known global functions, taking advantage of semantic information such as argument types and constant arguments. Transformation is optional, but should be done if speed or space is more important than compilation speed. Transformations which increase space should pass when space is more important than speed.

A transform is actually an inline function call where the function is computed at compile time. The transform gets to peek at the continuations for the arguments, and computes a function using the information gained. Transforms should be cautious about directly using the values of constant continuations, since the compiler must preserve eqlness of named constants, and it will have a hard time if transforms go around randomly copying constants.

The lambda that the transform computes replaces the original function variable reference as the function for the call. This lets the compiler worry about evaluating each argument once in the right order. We want to be careful to preserve type information when we do a transform, since it may be less than obvious what the transformed code does.

There can be any number of transforms for a function. Each transform is associated with a function type that the call must be compatible with. A transform is only invoked if the call has the right type. This provides a way to deal with the common case of a transform that only applies when the arguments are of certain types and some arguments are not specified. We always use the derived type when determining whether a transform is applicable. Type check is responsible for setting the derived type to the intersection of the asserted and derived types.

If the code in the expansion has insufficient explicit or implicit argument type checking, then it should cause checks to be generated by making declarations.

A transformation may decide to pass if it doesn't like what it sees when it looks at the args. The Give-Up function unwinds out of the transform and deals with complaining about inefficiency if speed is more important than brevity. The format args for the message are arguments to Give-Up. If a transform can't be done, we just record the message where ICR finalize can find it. note. We can't complain immediately, since it might get transformed later on.

8.4 Backward ICR optimizations

In the backward pass, we scan each block in reverse order, and eliminate any effectless nodes with unused values. In ICR this is the only way that code is deleted other than the elimination of unreachable blocks.

Chapter 9

Type checking

We need to do a pretty good job of guessing when a type check will ultimately need to be done. Generic arithmetic, for example: In the absence of declarations, we will use the safe variant, but if we don't know this, we will generate a check for NUMBER anyway. We need to look at the fast-safe templates and guess if any of them could apply.

We compute a function type from the VOP arguments and assertions on those arguments. This can be used with Valid-Function-Use to see which templates do or might apply to a particular call. If we guess that a safe implementation will be used, then we mark the continuation so as to force a safe implementation to be chosen. [This will happen if ICR optimize doesn't run to completion, so the ICR optimization after type check generation can discover new type information. Since we won't redo type check at that point, there could be a call that has applicable unsafe templates, but isn't type checkable.]

[### A better and more general optimization of structure type checks: in type check conversion, we look at the *original derived* type of the continuation: if the difference between the proven type and the asserted type is a simple type check, then check for the negation of the difference. e.g. if we want a FOO and we know we've got (OR FOO NULL), then test for (NOT NULL). This is a very important optimization for linked lists of structures, but can also apply in other situations.]

If after ICR phases, we have a continuation with check-type set in a context where it seems likely a check will be emitted, and the type is too hairy to be easily checked (i.e. no CHECK-xxx VOP), then we do a transformation on the ICR equivalent to:

```
  (... (the hair <foo>) ...)
==>
  (... (funcall \#'(lambda (\#:val)
    (if (typep \#:val 'hair)
\#:val
(%type-check-error \#:val 'hair)))
<foo>)
        ...)
```

This way, we guarantee that VMR conversion never has to emit type checks for hairy types.

[Actually, we need to do a MV-bind and several type checks when there is a MV continuation. And some values types are just too hairy to check. We really can't check any assertion for a non-fixed number of values, since there isn't any efficient way to bind arbitrary numbers of values. (could be done with MV-call of a more-arg function, I guess...)]

[Perhaps only use CHECK-xxx VOPs for types equivalent to a ptype? Exceptions for CONS and SYMBOL? Anyway, no point in going to trouble to implement and emit rarely used CHECK-xxx vops.]

One potential lose in converting a type check to explicit conditionals rather than to a CHECK-xxx VOP is that VMR code motion optimizations won't be able to do anything. This shouldn't be much of an issue, though, since type constraint propagation has already done global optimization of type checks.

This phase is optional, but should be done if anything is more important than compile speed.

Type check is responsible for reconciling the continuation asserted and derived types, emitting type checks if appropriate. If the derived type is a subtype of the asserted type, then we don't need to do anything.

If there is no intersection between the asserted and derived types, then there is a manifest type error. We print a warning message, indicating that something is almost surely wrong. This will inhibit any transforms or generators that care about their argument types, yet also inhibits further error messages, since NIL is a subtype of every type.

If the intersection is not null, then we set the derived type to the intersection of the asserted and derived types and set the Type-Check flag in the continuation. We always set the flag when we can't prove that the type assertion is satisfied, regardless of whether we will ultimately actually emit a type check or not. This is so other phases such as type constraint propagation can use the Type-Check flag to detect an interesting type assertion, instead of having to duplicate much of the work in this phase. [### 7 extremely random values for CONTINUATION-TYPE-CHECK.]

Type checks are generated on the fly during VMR conversion. When VMR conversion generates the check, it prints an efficiency note if speed is important. We don't flame now since type constraint progpagation may decide that the check is unnecessary. [### Not done now, maybe never.]

In local function call, it is the caller that is in effect responsible for checking argument types. This happens in the same way as any other type check, since ICR optimize propagates the declared argument types to the type assertions for the argument continuations in all the calls.

Since the types of arguments to entry points are unknown at compile time, we want to do runtime checks to ensure that the incoming arguments are of the correct type. This happens without any special effort on the part of type check, since the XEP is represented as a local call with unknown type arguments. These arguments will be marked as needing to be checked.

Chapter 10

Constraint propagation

New lambda-var-slot:

constraints: a list of all the constraints on this var for either X or Y.

How to maintain consistency? Does it really matter if there are constraints with deleted vars lying around? Note that whatever mechanism we use for getting the constraints in the first place should tend to keep them up to date. Probably we would define optimizers for the interesting relations that look at their CONT's dest and annotate it if it is an IF.

But maybe it is more trouble then it is worth trying to build up the set of constraints during ICR optimize (maintaining consistency in the process). Since ICR optimize iterates a bunch of times before it converges, we would be wasting time recomputing the constraints, when nobody uses them till constraint propagation runs.

It seems that the only possible win is if we re-ran constraint propagation (which we might want to do.) In that case, we wouldn't have to recompute all the constraints from scratch. But it seems that we could do this just as well by having ICR optimize invalidate the affected parts of the constraint annotation, rather than trying to keep them up to date. This also fits better with the optional nature of constraint propagation, since we don't want ICR optimize to commit to doing a lot of the work of constraint propagation.

For example, we might have a per-block flag indicating that something happened in that block since the last time constraint propagation ran. We might have different flags to represent the distinction between discovering a new type assertion inside the block and discovering something new about an if predicate, since the latter would be cheaper to update and probably is more common.

It's fairly easy to see how we can build these sets of restrictions and propagate them using flow analysis, but actually using this information seems a bit more ad-hoc.

Probably the biggest thing we do is look at all the refs. If we have proven that the value is EQ (EQL for a number) to some other leaf (constant or lambda-var), then we can substitute for that reference. In some cases, we will want to do special stuff depending on the DEST. If the dest is an IF and we proved (not null), then we can substitute T. And if the dest is some relation on the same two lambda-vars, then we want to see if we can show that relation is definitely true or false.

Otherwise, we can do our best to invert the set of restrictions into a type. Since types hold only constant info, we have to ignore any constraints between two vars. We can make some use of negated type restrictions by using TYPE-DIFFERENCE to remove the type from the ref types. If our inferred type is as good as the type assertion, then the continuation's type-check flag will be cleared.

It really isn't much of a problem that we don't infer union types on joins, since union types are relatively easy to derive without using flow information. The normal bottom-up type inference done by ICR optimize does this for us: it annotates everything with the union of all of the things it might possibly be. Then constraint propagation subtracts out those types that can't be in effect because of predicates or checks.

This phase is optional, but is desirable if anything is more important than compilation speed. We use an algorithm similar to available expressions to propagate variable type information that has been discovered by implicit or explicit type tests, or by type inference.

We must do a pre-pass which locates set closure variables, since we cannot do flow analysis on such variables. We set a flag in each set closure variable so that we can quickly tell that it is losing when we see it again. Although this may seem to be wastefully redundant with environment analysis, the overlap isn't really that great, and the cost should be small compared to that of the flow analysis that we are preparing to do. [Or we could punt on set variables...]

A type constraint is a structure that includes sset-element and has the type and variable. [Also a not-p flag indicating whether the sense is negated.]

Each variable has a list of its type constraints. We create a type constraint when we see a type test or check. If there is already a constraint for the same variable and type, then we just re-use it. If there is already a weaker constraint, then we generate both the weak constraints and the strong constraint so that the weak constraints won't be lost even if the strong one is unavailable.

We find all the distinct type constraints for each variable during the pre-pass over the lambda nesting. Each constraint has a list of the weaker constraints so that we can easily generate them.

Every block generates all the type constraints in it, but a constraint is available in a successor only if it is available in all predecessors. We determine the actual type constraint for a variable at a block by intersecting all the available type constraints for that variable.

This isn't maximally tense when there are constraints that are not hierarchically related, e.g. (or a b) (or b c). If these constraints were available from two predecessors, then we could infer that we have an (or a b c) constraint, but the above algorithm would come up with none. This probably isn't a big problem.

[### Do we want to deal with `(if (eq <var> '<foo>) ...)` indicating singleton member type?]

We detect explicit type tests by looking at type test annotation in the IF node. If there is a type check, the OUT sets are stored in the node, with different sets for the consequent and alternative. Implicit type checks are located by finding Ref nodes whose Cont has the Type-Check flag set. We don't actually represent the GEN sets, we just initialize OUT to it, and then form the union in place.

When we do the post-pass, we clear the Type-Check flags in the continuations for Refs when we discover that the available constraints satisfy the asserted type. Any explicit uses of typep should be cleaned up by the ICR optimizer for typep. We can also set the derived type for Refs to the intersection of the available type assertions. If we discover anything, we should consider redoing ICR optimization, since better type information might enable more optimizations.

Chapter 11

ICR finalize

This pass looks for interesting things in the ICR so that we can forget about them. Used and not defined things are flamed about.

We postpone these checks until now because the ICR optimizations may discover errors that are not initially obvious. We also emit efficiency notes about optimizations that we were unable to do. We can't emit the notes immediately, since we don't know for sure whether a repeated attempt at optimization will succeed.

We examine all references to unknown global function variables and update the approximate type accordingly. We also record the names of the unknown functions so that they can be flamed about if they are never defined. Unknown normal variables are flamed about on the fly during ICR conversion, so we ignore them here.

We check each newly defined global function for compatibility with previously recorded type information. If there is no :defined or :declared type, then we check for compatibility with any approximate function type inferred from previous uses.

Chapter 12

Environment analysis

A related change would be to annotate ICR with information about tail-recursion relations. What we would do is add a slot to the node structure that points to the corresponding Tail-Info when a node is in a TR position. This annotation would be made in a final ICR pass that runs after cleanup code is generated (part of environment analysis). When true, the node is in a true TR position (modulo return-convention incompatibility). When we determine return conventions, we null out the tail-p slots in XEP calls or known calls where we decided not to preserve tail-recursion.

In this phase, we also check for changes in the dynamic binding environment that require cleanup code to be generated. We just check for changes in the Continuation-Cleanup on local control transfers. If it changes from an inner dynamic context to an outer one that is in the same environment, then we emit code to clean up the dynamic bindings between the old and new continuation. We represent the result of cleanup detection to the back end by interposing a new block containing a call to a funny function. Local exits from CATCH or UNWIND-PROTECT are detected in the same way.

—#

The primary activity in environment analysis is the annotation of ICR with environment structures describing where variables are allocated and what values the environment closes over.

Each lambda points to the environment where its variables are allocated, and the environments point back. We always allocate the environment at the Bind node for the sole non-let lambda in the environment, so there is a close relationship between environments and functions. Each "real function" (i.e. not a LET) has a corresponding environment.

We attempt to share the same environment among as many lambdas as possible so that unnecessary environment manipulation is not done. During environment analysis the only optimization of this sort is realizing that a Let (a lambda with no Return node) cannot need its own environment, since there is no way that it can return and discover that its old values have been clobbered.

When the function is called, values from other environments may need to be made available in the function's environment. These values are said to be "closed over".

Even if a value is not referenced in a given environment, it may need to be closed over in that environment so that it can be passed to a called function that does reference the value. When we discover that a value must be closed over by a function, we must close over the value in all the environments where that function is referenced. This applies to all references, not just local calls, since at other references we must have the values on hand so that we can build a closure. This propagation must be applied recursively, since the value must also be available in *those* functions' callers.

If a closure reference is known to be "safe" (not an upward funarg), then the closure structure may be allocated on the stack.

Closure analysis deals only with closures over values, while Common Lisp requires closures over variables. The difference only becomes significant when variables are set. If a variable is not set, then we can freely make copies of it without keeping track of where they are. When a variable is set, we must maintain a single value cell, or at least the illusion thereof. We achieve this by creating a heap-allocated "value cell" structure for each set variable that is closed over. The pointer to this value cell is passed around as the "value" corresponding to that variable. References to the variable must explicitly indirect through the value cell.

When we are scanning over the lambdas in the component, we also check for bound but not referenced variables.

Environment analysis emits cleanup code for local exits and markers for non-local exits.

A non-local exit is a control transfer from one environment to another. In a non-local exit, we must close over the continuation that we transfer to so that the exiting function can find its way back. We indicate the need to close a continuation by placing the continuation structure in the closure and also pushing it on a list in the environment structure for the target of the exit. [### To be safe, we would treat the continuation as a set closure variable so that we could invalidate it when we leave the dynamic extent of the exit point. Transferring control to a meaningless stack pointer would be apt to cause horrible death.]

Each local control transfer may require dynamic state such as special bindings to be undone. We represent cleanup actions by funny function calls in a new block linked in as an implicit MV-PROG1.

Chapter 13

Virtual Machine Representation Introduction

Chapter 14

Global TN assignment

The basic mechanism for closing over values is to pass the values as additional implicit arguments in the function call. This technique is only applicable when:

- the calling function knows which values the called function wants to close over, and
- the values to be closed over are available in the calling environment.

The first condition is always true of local function calls. Environment analysis can guarantee that the second condition holds by closing over any needed values in the calling environment.

If the function that closes over values may be called in an environment where the closed over values are not available, then we must store the values in a "closure" so that they are always accessible. Closures are called using the "full call" convention. When a closure is called, control is transferred to the "external entry point", which fetches the values out of the closure and then does a local call to the real function, passing the closure values as implicit arguments.

In this scheme there is no such thing as a "heap closure variable" in code, since the closure values are moved into TNs by the external entry point. There is some potential for pessimization here, since we may end up moving the values from the closure into a stack memory location, but the advantages are also substantial. Simplicity is gained by always representing closure values the same way, and functions with closure references may still be called locally without allocating a closure. All the TN based VMR optimizations will apply to closure variables, since closure variables are represented in the same way as all other variables in VMR. Closure values will be allocated in registers where appropriate.

Closures are created at the point where the function is referenced, eliminating the need to be able to close over closures. This lazy creation of closures has the additional advantage that when a closure reference is conditionally not done, then the closure consing will never be done at all. The corresponding disadvantage is that a closure over the same values may be created multiple times if there are multiple references. Note however, that VMR loop and common subexpression optimizations can eliminate redundant closure consing. In any case, multiple closures over the same variables doesn't seem to be that common.

#— Having the Tail-Info would also make return convention determination trivial. We could just look at the type, checking to see if it represents a fixed number of values. To determine if the standard return convention is necessary to preserve tail-recursion, we just iterate over the equivalent functions, looking for XEPs and uses in full calls. —#

The Global TN Assignment pass (GTN) can be considered a post-pass to environment analysis. This phase assigns the TNs used to hold local lexical variables and pass arguments and return values and determines the value-passing strategy used in local calls.

To assign return locations, we look at the function's tail-set.

If the result continuation for an entry point is used as the continuation for a full call, then we may need to constrain the continuation's values passing convention to the standard one. This is not necessary when the call is known not to be part of a tail-recursive loop (due to being a known function).

Once we have figured out where we must use the standard value passing strategy, we can use a more flexible strategy to determine the return locations for local functions. We determine the possible numbers of return values from each function by examining the uses of all the result continuations in the equivalence class of the result continuation.

If the tail-set type is for a fixed number of values, then we return that fixed number of values from all the functions whose result continuations are equated. If the number of values is not fixed, then we must use the unknown-values convention, although we are not forced to use the standard locations. We assign the result TNs at this time.

We also use the tail-sets to see what convention we want to use. What we do is use the full convention for any function that has a XEP its tail-set, even if we aren't required to do so by a tail-recursive full call, as long as there are no non-tail-recursive local calls in the set. This prevents us from gratuitously using a non-standard convention when there is no reason to.

Chapter 15

Local TN assignment

[Want a different name for this so as not to be confused with the different local/global TN representations. The really interesting stuff in this phase is operation selection, values representation selection, return strategy, etc. Maybe this phase should be conceptually lumped with GTN as "implementation selection", since GTN determines call strategies and locations.]

#—

[### I guess I believe that it is OK for VMR conversion to dick the ICR flow graph. An alternative would be to give VMR its very own flow graph, but that seems like overkill.

In particular, it would be very nice if a TR local call looked exactly like a jump in VMR. This would allow loop optimizations to be done on loops written as recursions. In addition to making the call block transfer to the head of the function rather than to the return, we would also have to do something about skipping the part of the function prolog that moves arguments from the passing locations, since in a TR call they are already in the right frame.

In addition to directly indicating whether a call should be coded with a TR variant, the Tail-P annotation flags non-call nodes that can directly return the value (an "advanced return"), rather than moving the value to the result continuation and jumping to the return code. Then (according to policy), we can decide to advance all possible returns. If all uses of the result are Tail-P, then LTN can annotate the result continuation as :Unused, inhibiting emission of the default return code.

[### But not really. Now there is a single list of templates, and a given template has only one policy.]

In LTN, we use the :Safe template as a last resort even when the policy is unsafe. Note that we don't try :Fast-Safe; if this is also a good unsafe template, then it should have the unsafe policies explicitly specified.

With a :Fast-Safe template, the result type must be proven to satisfy the output type assertion. This means that a fast-safe template with a fixnum output type doesn't need to do fixnum overflow checking. [### Not right to just check against the Node-Derived-Type, since type-check intersects with this.]

It seems that it would be useful to have a kind of template where the args must be checked to be fixnum, but the template checks for overflow and signals an error. In the case where an output assertion is present, this would generate better code than conditionally branching off to make a bignum, and then doing a type check on the result.

How do we deal with deciding whether to do a fixnum overflow check? This is perhaps a more general problem with the interpretation of result type restrictions in templates. It would be useful to be able to discriminate between the case where the result has been proven to be a fixnum and where it has simply been asserted to be so.

The semantics of result type restriction is that the result must be proven to be of that type *except* for safe generators, which are assumed to verify the assertion. That way "is-fixnum" case can be a fast-safe generator and the "should-be-fixnum" case is a safe generator. We could choose not to have a safe "should-be-fixnum" generator, and let the unrestricted safe generator handle it. We would then have to do an explicit type check on the result.

In other words, for all template except Safe, a type restriction on either an argument or result means "this must be true; if it is not the system may break." In contrast, in a Safe template, the restriction means "If this is not true, I will signal an error."

Since the node-derived-type only takes into consideration stuff that can be proved from the arguments, we can use the node-derived-type to select fast-safe templates. With unsafe policies, we don't care, since the code is supposed to be unsafe.

—#

Local TN assignment (LTN) assigns all the TNs needed to represent the values of continuations. This pass scans over the code for the component, examining each continuation and its destination. A number of somewhat unrelated things are also done at the same time so that multiple passes aren't necessary. – Determine the Primitive-Type for each continuation value and assigns TNs to hold the values. – Use policy information to determine the implementation strategy for each call to a known function. – Clear the type-check flags in continuations whose destinations have safe implementations. – Determine the value-passing strategy for each continuation: known or unknown. – Note usage of unknown-values continuations so that stack analysis can tell when stack values must be discarded.

If safety is more important than speed and space, then we consider generating type checks on the values of nodes whose CONT has the Type-Check flag set. If the destination for the continuation value is safe, then we don't need to do a check. We assume that all full calls are safe, and use the template information to determine whether inline operations are safe.

This phase is where compiler policy switches have most of their effect. The speed/space/safety tradeoff can determine which of a number of coding strategies are used. It is important to make the policy choice in VMR conversion rather than in code generation because the cost and storage requirement information which drives TNBIND will depend strongly on what actual VOP is chosen. In the case of +/FIXNUM, there might be three or more implementations, some optimized for speed, some for space, etc. Some of these VOPS might be open-coded and some not.

We represent the implementation strategy for a call by either marking it as a full call or annotating it with a "template" representing the open-coding strategy. Templates are selected using a two-way dispatch off of operand primitive-types and policy. The general case of LTN is handled by the LTN-Annotate function in the function-info, but most functions are handled by a table-driven mechanism. There are four different translation policies that a template may have:

Safe The safest implementation; must do argument type checking.

Small The (unsafe) smallest implementation.

Fast The (unsafe) fastest implementation.

Fast-Safe An implementation optimized for speed, but which does any necessary checks exclusive of argument type checking. Examples are array bounds checks and fixnum overflow checks.

Usually a function will have only one or two distinct templates. Either or both of the safe and fast-safe templates may be omitted; if both are specified, then they should be distinct. If there is no safe template and our policy is safe, then we do a full call.

We use four different coding strategies, depending on the policy:

Safe: safety > space > speed, or we want to use the fast-safe template, but there isn't one.

Small: space > (max speed safety)

Fast: speed > (max space safety)

Fast-Safe (and type check): safety > speed > space, or we want to use the safe template, but there isn't one.

"Space" above is actually the maximum of space and cspeed, under the theory that less code will take less time to generate and assemble. [### This could lose if the smallest case is out-of-line, and must allocate many linkage registers.]

Chapter 16

Control optimization

In this phase we annotate blocks with drop-throughs. This controls how code generation linearizes code so that drop-throughs are used most effectively. We totally linearize the code here, allowing code generation to scan the blocks in the emit order.

There are basically two aspects to this optimization:

1. Dynamically reducing the number of branches taken v.s. branches not taken under the assumption that branches not taken are cheaper.

2. Statically minimizing the number of unconditional branches, saving space and presumably time.

These two goals can conflict, but if they do it seems pretty clear that the dynamic optimization should get preference. The main dynamic optimization is changing the sense of a conditional test so that the more commonly taken branch is the fall-through case. The problem is determining which branch is more commonly taken.

The most clear-cut case is where one branch leads out of a loop and the other is within. In this case, clearly the branch within the loop should be preferred. The only added complication is that at some point in the loop there has to be a backward branch, and it is preferable for this branch to be conditional, since an unconditional branch is just a waste of time.

In the absence of such good information, we can attempt to guess which branch is more popular on the basis of difference in the cost between the two cases. Min-max strategy suggests that we should choose the cheaper alternative, since the percentagewise improvement is greater when the branch overhead is significant with respect to the cost of the code branched to. A tractable approximation of this is to compare only the costs of the two blocks immediately branched to, since this would avoid having to do any hairy graph walking to find all the code for the consequent and the alternative. It might be worthwhile discriminating against ultra-expensive functions such as ERROR.

For this to work, we have to detect when one of the options is empty. In this case, the next for one branch is a successor of the other branch, making the comparison meaningless. We use dominator information to detect this situation. When a branch is empty, one of the predecessors of the first block in the empty branch will be dominated by the first block in the other branch. In such a case we favor the empty branch, since that's about as cheap as you can get.

Statically minimizing branches is really a much more tractable problem, but what literature there is makes it look hard. Clearly the thing to do is to use a non-optimal heuristic algorithm.

A good possibility is to use an algorithm based on the depth first ordering. We can modify the basic DFO algorithm so that it chooses an ordering which favors any drop-thrus that we may choose for dynamic reasons. When we are walking the graph, we walk the desired drop-thru arc last, which will place it immediately after us in the DFO unless the arc is a retreating arc.

We scan through the DFO and whenever we find a block that hasn't been done yet, we build a straight-line segment by setting the drop-thru to the unreached successor block which has the lowest DFN greater than that for the block. We move to the drop-thru block and repeat the process until there is no such block. We then go back to our original scan through the DFO, looking for the head of another straight-line segment.

This process will automagically implement all of the dynamic optimizations described above as long as we favor the appropriate IF branch when creating the DFO. Using the DFO will prevent us from making the back branch in a loop the drop-thru, but we need to be clever about favoring IF branches within loops while computing the DFO. The IF join will be favored without any special effort, since we follow through the most favored path until we reach the end.

This needs some knowledge about the target machine, since on most machines non-tail-recursive calls will use some sort of call instruction. In this case, the call actually wants to drop through to the return point, rather than dropping through to the beginning of the called function.

Chapter 17

VMR conversion

#— Single-use let var continuation substitution not really correct, since it can cause a spurious type error. Maybe we do want stuff to prove that an NLX can't happen after all. Or go back to the idea of moving a combination arg to the ref location, and having that use the ref cont (with its output assertion.) This lossage doesn't seem very likely to actually happen, though. [### must-reach stuff wouldn't work quite as well as combination substitute in psetq, etc., since it would fail when one of the new values is random code (might unwind.)]

Is this really a general problem with eager type checking? It seems you could argue that there was no type error in this code:

```
(+ :foo (throw 'up nil))
```

But we would signal an error.

Emit explicit you-lose operation when we do a move between two non-T ptypes, even when type checking isn't on. Can this really happen? Seems we should treat continuations like this as though type-check was true. Maybe LTN should leave type-check true in this case, even when the policy is unsafe. (Do a type check against NIL?)

At continuation use time, we may in general have to do both a coerce-to-t and a type check, allocating two temporary TNs to hold the intermediate results.

17.1 VMR Control representation

We represent all control transfer explicitly. In particular, :Conditional VOPs take a single Target continuation and a Not-P flag indicating whether the sense of the test is negated. Then an unconditional Branch VOP will be emitted afterward if the other path isn't a drop-through.

So we linearize the code before VMR-conversion. This isn't a problem, since there isn't much change in control flow after VMR conversion (none until loop optimization requires introduction of header blocks.) It does make cost-based branch prediction a bit ucky, though, since we don't have any cost information in ICR. Actually, I guess we do have pretty good cost information after LTN even before VMR conversion, since the most important thing to know is which functions are open-coded.

—#

VMR preserves the block structure of ICR, but replaces the nodes with a target dependent virtual machine (VM) representation. Different implementations may use different VMs without making major changes in the back end. The two main components of VMR are Temporary Names (TNs) and Virtual OPerations (VOPs). TNs represent the locations that hold values, and VOPs represent the operations performed on the values.

A "primitive type" is a type meaningful at the VM level. Examples are Fixnum, String-Char, Short-Float. During VMR conversion we use the primitive type of an expression to determine both where we can store the result of the expression and which type-specific implementations of an operation can be applied to the value. [Ptype is a set of SCs == representation choices and representation specific operations]

The VM specific definitions provide functions that do stuff like find the primitive type corresponding to a type and test for primitive type subtypep. Usually primitive types will be disjoint except for T, which represents all types.

The primitive type T is special-cased. Not only does it overlap with all the other types, but it implies a descriptor ("boxed" or "pointer") representation. For efficiency reasons, we sometimes want to use alternate representations for some objects such as numbers. The majority of operations cannot exploit alternate representations, and would only be complicated if they had to be able to convert alternate representations into descriptors. A template can require an operand to be a descriptor by constraining the operand to be of type T.

A TN can only represent a single value, so we bare the implementation of MVs at this point. When we know the number of multiple values being handled, we use multiple TNs to hold them. When the number of values is actually unknown, we use a convention that is compatible with full function call.

Everything that is done is done by a VOP in VMR. Calls to simple primitive functions such as + and CAR are translated to VOP equivalents by a table-driven mechanism. This translation is specified by the particular VM definition; VMR conversion makes no assumptions about which operations are primitive or what operand types are worth special-casing. The default calling mechanisms and other miscellaneous builtin features are implemented using standard VOPs that must be implemented by each VM.

Type information can be forgotten after VMR conversion, since all type-specific operation selections have been made.

Simple type checking is explicitly done using CHECK-xxx VOPs. They act like innocuous effectless/unaffected VOPs which return the checked thing as a result. This allows loop-invariant optimization and common subexpression elimination to remove redundant checks. All type checking is done at the time the continuation is used.

Note that we need only check asserted types, since if type inference works, the derived types will also be satisfied. We can check whichever is more convenient, since both should be true.

Constants are turned into special Constant TNs, which are wired down in a SC that is determined by their type. The VM definition provides a function that returns a constant TN to represent a Constant Leaf.

Each component has a constant pool. There is a register dedicated to holding the constant pool for the current component. The back end allocates non-immediate constants in the constant pool when it discovers them during translation from ICR.

[### Check that we are describing what is actually implemented. But this really isn't very good in the presence of interesting unboxed representations...] Since LTN only deals with values from the viewpoint of the receiver, we must be prepared during the translation pass to do stuff to the continuation at the time it is used. – If a VOP yields more values than are desired, then we must create TNs to hold the discarded results. An important special-case is continuations whose value is discarded. These continuations won't be annotated at all. In the case of a Ref, we can simply skip evaluation of the reference when the continuation hasn't been annotated. Although this will eliminate bogus references that for some reason weren't optimized away, the real purpose is to handle deferred references. – If a VOP yields fewer values than desired, then we must default the extra values to NIL. – If a continuation has its type-check flag set, then we must check the type of the value before moving it into the result location. In general, this requires computing the result in a temporary, and having the type-check operation deliver it in the actual result location. – If the template's result type is T, then we must generate a boxed temporary to compute the result in when the continuation's type isn't T.

We may also need to do stuff to the arguments when we generate code for a template. If an argument continuation isn't annotated, then it must be a deferred reference. We use the leaf's TN instead. We may have to do any of the above use-time actions also. Alternatively, we could avoid hair by not deferring references that must be type-checked or may need to be boxed.

17.2 Stack analysis

Think of this as a lifetime problem: a values generator is a write and a values receiver is a read. We want to annotate each VMR-Block with the unknown-values continuations that are live at that point. If we do a control transfer to a place where fewer continuations are live, then we must deallocate the newly dead continuations.

We want to convince ourselves that values deallocation based on lifetime analysis actually works. In particular, we need to be sure that it doesn't violate the required stack discipline. It is clear that it is impossible to deallocate the values before they become dead, since later code may decide to use them. So the only thing we need to ensure is that the "right" time isn't later than the time that the continuation becomes dead.

The only reason why we couldn't deallocate continuation A as soon as it becomes dead would be that there is another continuation B on top of it that isn't dead (since we can only deallocate the topmost continuation).

The key to understanding why this can't happen is that each continuation has only one read (receiver). If B is on top of A, then it must be the case that A is live at the receiver for B. This means that it is impossible for B to be live without A being live.

The reason that we don't solve this problem using a normal iterative flow analysis is that we also need to know the ordering of the continuations on the stack so that we can do deallocation. When it comes time to discard values, we want to know which discarded continuation is on the bottom so that we can reset SP to its start.

[I suppose we could also decrement SP by the aggregate size of the discarded continuations.] Another advantage of knowing the order in which we expect continuations to be on the stack is that it allows us to do some consistency checking. Also doing a localized graph walk around the values-receiver is likely to be much more efficient than doing an iterative flow analysis problem over all the code in the component (not that big a consideration.)

#— Actually, what we do is a backward graph walk from each unknown-values receiver. As we go, we mark each walked block with the ordered list of continuations we believe are on the stack. Starting with an empty stack, we: – When we encounter another unknown-values receiver, we push that continuation on our simulated stack. – When we encounter a receiver (which had better be for the topmost continuation), we pop that continuation. – When we pop all continuations, we terminate our walk.

[### not quite right... It seems we may run into "dead values" during the graph walk too. It seems that we have to check if the pushed continuation is on stack top, and if not, add it to the ending stack so that the post-pass will discard it.]

[### Also, we can't terminate our walk just because we hit a block previously walked. We have to compare the End-Stack with the values received along the current path: if we have more values on our current walk than on the walk that last touched the block, then we need to re-walk the subgraph reachable from that block, using our larger set of continuations. It seems that our actual termination condition is reaching a block whose End-Stack is already EQ to our current stack.]

If at the start, the block containing the values receiver has already been walked, we skip the walk for that continuation, since it has already been handled by an enclosing values receiver. Once a walk has started, we ignore any signs of a previous walk, clobbering the old result with our own, since we enclose that continuation, and the previous walk doesn't take into consideration the fact that our values block underlies its own.

When we are done, we have annotated each block with the stack current both at the beginning and at the end of that block. Blocks that aren't walked don't have anything on the stack either place (although they may hack MVs internally).

We then scan all the blocks in the component, looking for blocks that have predecessors with a different ending stack than that block's starting stack. (The starting stack had better be a tail of the predecessor's ending stack.) We insert a block intervening between all of these predecessors that sets SP to the end of the values for the continuation that should be on stack top. Of course, this pass needn't be done if there aren't any global unknown MVs.

Also, if we find any block that wasn't reached during the walk, but that USEs an outside unknown-values continuation, then we know that the DEST can't be reached from this point, so the values are unused. We either insert code to pop the values, or somehow mark the code to prevent the values from ever being pushed. (We could cause the popping to be done by the normal pass if we iterated over the pushes beforehand, assigning a correct END-STACK.)

[### But I think that we have to be a bit clever within blocks, given the possibility of blocks being joined. We could collect some unknown MVs in a block, then do a control transfer out of the receiver, and this control transfer could be squeezed out by merging blocks. How about:

```
    (tagbody
      (return
       (multiple-value-prog1 (foo)
(when bar
  (go UNWIND))))

     UNWIND
      (return
       (multiple-value-prog1 (baz)
bletch)))
```

But the problem doesn't happen here (can't happen in general?) since a node buried within a block can't use a continuation outside of the block. In fact, no block can have more then one PUSH continuation, and this must always be the last continuation. So it is trivially (structurally) true that all pops come before any push.

[### But not really: the DEST of an embedded continuation may be outside the block. There can be multiple pushes, and we must find them by iterating over the uses of MV receivers in LTN. But it would be hard to get the order right this way. We could easily get the order right if we added the generators as we saw the uses, except that we can't guarantee that the continuations will be annotated at that point. (Actually, I think we only need the order for consistency checks, but that is probably worthwhile). I guess the thing to do is when we process the receiver, add the generator blocks to the Values-Generators, then do a post-pass that re-scans the blocks adding the pushes.]

I believe that above concern with a dead use getting mashed inside a block can't happen, since the use inside the block must be the only use, and if the use isn't reachable from the push, then the use is totally unreachable, and should have been deleted, which would prevent it from ever being annotated.]] —#

We find the partial ordering of the values globs for unknown values continuations in each environment. We don't have to scan the code looking for unknown values continuations since LTN annotates each block with the continuations that were popped and not pushed or pushed and not popped. This is all we need to do the inter-block analysis.

After we have found out what stuff is on the stack at each block boundary, we look for blocks with predecessors that have junk on the stack. For each such block, we introduce a new block containing code to restore the stack pointer. Since unknown-values continuations are represented as `<start, count>`, we can easily pop a continuation using the Start TN.

Note that there is only doubt about how much stuff is on the control stack, since only it is used for unknown values. Any special stacks such as number stacks will always have a fixed allocation.

17.3 Non-local exit

If the starting and ending continuations are not in the same environment, then the control transfer is a non-local exit. In this case just call Unwind with the appropriate stack pointer, and let the code at the re-entry point worry about fixing things up.

It seems like maybe a good way to organize VMR conversion of NLX would be to have environment analysis insert funny functions in new interposed cleanup blocks. The thing is that we need some way for VMR conversion to: 1] Get its hands on the returned values. 2] Do weird control shit. 3] Deliver the values to the original continuation destination. I.e. we need some way to interpose arbitrary code in the path of value delivery.

What we do is replace the NLX uses of the continuation with another continuation that is received by a MV-Call to %NLX-VALUES in a cleanup block that is interposed between the NLX uses and the old continuation's block. The MV-Call uses the original continuation to deliver its values to.

[Actually, it's not really important that this be an MV-Call, since it has to be special-cased by LTN anyway. Or maybe we would want it to be an MV call. If we did normal LTN analysis of an MV call, it would force the returned values into the unknown values convention, which is probably pretty convenient for use in NLX.

Then the entry code would have to use some special VOPs to receive the unknown values. But we probably need special VOPs for NLX entry anyway, and the code can share with the call VOPs. Also we probably need the technology anyway, since THROW will use truly unknown values.]

On entry to a dynamic extent that has non-local-exists into it (always at an ENTRY node), we take a complete snapshot of the dynamic state:

- the top pointers for all stacks
- current Catch and Unwind-Protect
- current special binding (binding stack pointer in shallow binding)

We insert code at the re-entry point which restores the saved dynamic state. All TNs live at an NLX EP are forced onto the stack, so we don't have to restore them, and we don't have to worry about getting them saved.

Chapter 18

Copy propagation

File: `copyprop`

This phase is optional, but should be done whenever speed or space is more important than compile speed. We use global flow analysis to find the reaching definitions for each TN. This information is used here to eliminate unnecessary TNs, and is also used later on by loop invariant optimization.

In some cases, VMR conversion will unnecessarily copy the value of a TN into another TN, since it may not be able to tell that the initial TN has the same value at the time the second TN is referenced. This can happen when ICR optimize is unable to eliminate a trivial variable binding, or when the user does a setq, or may also result from creation of expression evaluation temporaries during VMR conversion. Whatever the cause, we would like to avoid the unnecessary creation and assignment of these TNs.

What we do is replace TN references whose only reaching definition is a Move VOP with a reference to the TN moved from, and then delete the Move VOP if the copy TN has no remaining references. There are several restrictions on copy propagation:

- The TNs must be "ordinary" TNs, not restricted or otherwise unusual. Extending the life of restricted (or wired) TNs can make register allocation impossible. Some other TN kinds have hidden references.
- We don't want to defeat source-level debugging by replacing named variables with anonymous temporaries.
- We can't delete moves that representation selected might want to change into a representation conversion, since we need the primitive types of both TNs to select a conversion.

Some cleverness reduces the cost of flow analysis. As for lifetime analysis, we only need to do flow analysis on global packed TNs. We can't do the real local TN assignment pass before this, since we allocate TNs afterward, so we do a pre-pass that marks the TNs that are local for our purposes. We don't care if block splitting eventually causes some of them to be considered global.

Note also that we are really only interested in knowing if there is a unique reaching definition, which we can mash into our flow analysis rules by doing an intersection. Then a definition only appears in the set when it is unique. We then propagate only definitions of TNs with only one write, which allows the TN to stand for the definition.

Chapter 19

Representation selection

File: `represent`

Some types of object (such as `single-float`) have multiple possible representations. Multiple representations are useful mainly when there is a particularly efficient non-descriptor representation. In this case, there is the normal descriptor representation, and an alternate non-descriptor representation.

This possibility brings up two major issues:

- The compiler must decide which representation will be most efficient for any given value, and
- Representation conversion code must be inserted where the representation of a value is changed.

First, the representations for TNs are selected by examining all the TN references and attempting to minimize reference costs. Then representation conversion code is introduced.

This phase is in effect a pre-pass to register allocation. The main reason for its existence is that representation conversions may be farily complex (e.g. involving memory allocation), and thus must be discovered before register allocation.

VMR conversion leaves stubs for representation specific move operations. Representation selection recognizes `move` by name. Argument and return value passing for call VOPs is controlled by the `:move-arguments` option to `define-vop`.

Representation selection is also responsible for determining what functions use the number stack. If any representation is chosen which could involve packing into the `non-descriptor-stack` SB, then we allocate the NFP register throughout the component. As an optimization, permit the decision of whether a number stack frame needs to be allocated to be made on a per-function basis. If a function doesn't use the number stack, and isn't in the same tail-set as any function that uses the number stack, then it doesn't need a number stack frame, even if other functions in the component do.

Chapter 20

Lifetime analysis

File: `life`

This phase is a preliminary to Pack. It involves three passes: – A pre-pass that computes the DEF and USE sets for live TN analysis, while also assigning local TN numbers, splitting blocks if necessary. ### But not really... – A flow analysis pass that does backward flow analysis on the component to find the live TNs at each block boundary. – A post-pass that finds the conflict set for each TN.

#— Exploit the fact that a single VOP can only exhaust LTN numbers when there are large more operands. Since more operand reference cannot be interleaved with temporary reference, the references all effectively occur at the same time. This means that we can assign all the more args and all the more results the same LTN number and the same lifetime info. —#

20.1 Flow analysis

It seems we could use the global-conflicts structures during compute the inter-block lifetime information. The pre-pass creates all the global-conflicts for blocks that global TNs are referenced in. The flow analysis pass just adds always-live global-conflicts for the other blocks the TNs are live in. In addition to possibly being more efficient than SSets, this would directly result in the desired global-conflicts information, rather than having to create it from another representation.

The DFO sorted per-TN global-conflicts thread suggests some kind of algorithm based on the manipulation of the sets of blocks each TN is live in (which is what we really want), rather than the set of TNs live in each block.

If we sorted the per-TN global-conflicts in reverse DFO (which is just as good for determining conflicts between TNs), then it seems we could scan though the conflicts simultaneously with our flow-analysis scan through the blocks.

The flow analysis step is the following: If a TN is always-live or read-before-written in a successor block, then we make it always-live in the current block unless there are already global-conflicts recorded for that TN in this block.

The iteration terminates when we don't add any new global-conflicts during a pass.

We may also want to promote TNs only read within a block to always-live when the TN is live in a successor. This should be easy enough as long as the global-conflicts structure contains this kind of info.

The critical operation here is determining whether a given global TN has global conflicts in a given block. Note that since we scan the blocks in DFO, and the global-conflicts are sorted in DFO, if we give each global TN a pointer to the global-conflicts for the last block we checked the TN was in, then we can guarantee that the global-conflicts we are looking for are always at or after that pointer. If we need to insert a new structure, then the pointer will help us rapidly find the place to do the insertion.]

20.2 Conflict detection

[### Environment, :more TNs.]

This phase makes use of the results of lifetime analysis to find the set of TNs that have lifetimes overlapping with those of each TN. We also annotate call VOPs with information about the live TNs so that code generation knows which registers need to be saved.

The basic action is a backward scan of each block, looking at each TN-Ref and maintaining a set of the currently live TNs. When we see a read, we check if the TN is in the live set. If not, we: – Add the TN to the conflict set for every currently live TN, – Union the set of currently live TNs with the conflict set for the TN, and – Add the TN to the set of live TNs.

When we see a write for a live TN, we just remove it from the live set. If we see a write to a dead TN, then we update the conflicts sets as for a read, but don't add the TN to the live set. We have to do this so that the bogus write doesn't clobber anything.

[We don't consider always-live TNs at all in this process, since the conflict of always-live TNs with other TNs in the block is implicit in the global-conflicts structures.

Before we do the scan on a block, we go through the global-conflicts structures of TNs that change liveness in the block, assigning the recorded LTN number to the TN's LTN number for the duration of processing of that block.]

Efficiently computing and representing this information calls for some cleverness. It would be prohibitively expensive to represent the full conflict set for every TN with sparse sets, as is done at the block-level. Although it wouldn't cause non-linear behavior, it would require a complex linked structure containing tens of elements to be created for every TN. Fortunately we can improve on this if we take into account the fact that most TNs are "local" TNs: TNs which have all their uses in one block.

First, many global TNs will be either live or dead for the entire duration of a given block. We can represent the conflict between global TNs live throughout the block and TNs local to the block by storing the set of always-live global TNs in the block. This reduces the number of global TNs that must be represented in the conflicts for local TNs.

Second, we can represent conflicts within a block using bit-vectors. Each TN that changes liveness within a block is assigned a local TN number. Local conflicts are represented using a fixed-size bit-vector of 64 elements or so which has a 1 for the local TN number of every TN live at that time. The block has a simple-vector which maps from local TN numbers to TNs. Fixed-size vectors reduce the hassle of doing allocations and allow operations to be open-coded in a maximally tense fashion.

We can represent the conflicts for a local TN by a single bit-vector indexed by the local TN numbers for that block, but in the global TN case, we need to be able to represent conflicts with arbitrary TNs. We could use a list-like sparse set representation, but then we would have to either special-case global TNs by using the sparse representation within the block, or convert the local conflicts bit-vector to the sparse representation at the block end. Instead, we give each global TN a list of the local conflicts bit-vectors for each block that the TN is live in. If the TN is always-live in a block, then we record that fact instead. This gives us a major reduction in the amount of work we have to do in lifetime analysis at the cost of some increase in the time to iterate over the set during Pack.

Since we build the lists of local conflict vectors a block at a time, the blocks in the lists for each TN will be sorted by the block number. The structure also contains the local TN number for the TN in that block. These features allow pack to efficiently determine whether two arbitrary TNs conflict. You just scan the lists in order, skipping blocks that are in only one list by using the block numbers. When we find a block that both TNs are live in, we just check the local TN number of one TN in the local conflicts vector of the other.

In order to do these optimizations, we must do a pre-pass that finds the always-live TNs and breaks blocks up into small enough pieces so that we don't run out of local TN numbers. If we can make a block arbitrarily small, then we can guarantee that an arbitrarily small number of TNs change liveness within the block. We must be prepared to make the arguments to unbounded arg count VOPs (such as function call) always-live even when they really aren't. This is enabled by a panic mode in the block splitter: if we discover that the block only contains one VOP and there are still too many TNs that aren't always-live, then we promote the arguments (which we'd better be able to do...).

This is done during the pre-scan in lifetime analysis. We can do this because all TNs that change liveness within a block can be found by examining that block: the flow analysis only adds always-live TNs.

When we are doing the conflict detection pass, we set the LTN number of global TNs. We can easily detect global TNs that have not been locally mapped because this slot is initially null for global TNs and we null it out after processing each block. We assign all Always-Live TNs to the same local number so that we don't need to treat references to them specially when making the scan.

We also annotate call VOPs that do register saving with the TNs that are live during the call, and thus would need to be saved if they are packed in registers.

We adjust the costs for TNs that need to be saved so that TNs costing more to save and restore than to reference get packed on the stack. We would also like more often saved TNs to get higher costs so that they are packed in more savable locations.

Chapter 21

Packing

File: `pack`

#—

Add lifetime/pack support for pre-packed save TNs.

Fix GTN/VMR conversion to use pre-packed save TNs for old-cont and return-PC. (Will prevent preference from passing location to save location from ever being honored?)

We will need to make packing of passing locations smarter before we will be able to target the passing location on the stack in a tail call (when that is where the callee wants it.) Currently, we will almost always pack the passing location in a register without considering whether that is really a good idea. Maybe we should consider schemes that explicitly understand the parallel assignment semantics, and try to do the assignment with a minimum number of temporaries. We only need assignment temps for TNs that appear both as an actual argument value and as a formal parameter of the called function. This only happens in self-recursive functions.

Could be a problem with lifetime analysis, though. The write by a move-arg VOP would look like a write in the current env, when it really isn't. If this is a problem, then we might want to make the result TN be an info arg rather than a real operand. But this would only be a problem in recursive calls, anyway. [This would prevent targeting, but targeting across passing locations rarely seems to work anyway.] [### But the :ENVIRONMENT TN mechanism would get confused. Maybe put env explicitly in TN, and have it only always-live in that env, and normal in other envs (or blocks it is written in.) This would allow targeting into environment TNs.

I guess we would also want the env/PC save TNs normal in the return block so that we can target them. We could do this by considering env TNs normal in read blocks with no successors.

ENV TNs would be treated totally normally in non-env blocks, so we don't have to worry about lifetime analysis getting confused by variable initializations. Do some kind of TN costing to determine when it is more trouble than it is worth to allocate TNs in registers.

Change pack ordering to be less pessimal. Pack TNs as they are seen in the LTN map in DFO, which at least in non-block compilations has an effect something like packing main trace TNs first, since control analysis tries to put the good code first. This could also reduce spilling, since it makes it less likely we will clog all registers with global TNs.

If we pack a TN with a specified save location on the stack, pack in the specified location.

Allow old-cont and return-pc to be kept in registers by adding a new "keep around" kind of TN. These are kind of like environment live, but are only always-live in blocks that they weren't referenced in. Lifetime analysis does a post-pass adding always-live conflicts for each "keep around" TN to those blocks with no conflict for that TN. The distinction between always-live and keep-around allows us to successfully target old-cont and return-pc to passing locations. MAKE-KEEP-AROUND-TN (ptype), PRE-PACK-SAVE-TN (tn scn offset). Environment needs a KEEP-AROUND-TNS slot so that conflict analysis can find them (no special casing is needed after then, they can be made with :NORMAL kind). VMR-component needs PRE-PACKED-SAVE-TNS so that conflict analysis or somebody can copy conflict info from the saved TN.

Note that having block granularity in the conflict information doesn't mean that a localized packing scheme would have to do all moves at block boundaries (which would clash with the desire to have saving done as part of this mechanism.) All that it means is that if we want to do a move within the block, we would need to allocate both locations throughout that block (or something).

Load TN pack:

A location is out for load TN packing if:

The location has TN live in it after the VOP for a result, or before the VOP for an argument, or

The location is used earlier in the TN-ref list (after) the saved results ref or later in the TN-Ref list (before) the loaded argument's ref.

To pack load TNs, we advance the live-tns to the interesting VOP, then repeatedly scan the vop-refs to find vop-local conflicts for each needed load TN. We insert move VOPs and change over the TN-Ref-TNs as we go so the TN-Refs will reflect conflicts with already packed load-TNs.

If we fail to pack a load-TN in the desired SC, then we scan the Live-TNs for the SB, looking for a TN that can be packed in an unbounded SB. This TN must then be repacked in the unbounded SB. It is important the load-TNs are never packed in unbounded SBs, since that would invalidate the conflicts info, preventing us from repacking TNs in unbounded SBs. We can't repack in a finite SB, since there might have been load TNs packed in that SB which aren't represented in the original conflict structures.

Is it permissible to "restrict" an operand to an unbounded SC? Not impossible to satisfy as long as a finite SC is also allowed. But in practice, no restriction would probably be as good.

We assume all locations can be used when an sc is based on an unbounded sb.

]

TN-Refs are convenient structures to build the target graph out of. If we allocated space in every TN-Ref, then there would certainly be enough to represent arbitrary target graphs. Would it be enough to allocate a single Target slot? If there is a target path through a given VOP, then the Target of the write ref would be the read, and vice-versa. To find all the TNs that target us, we look at the TN for the target of all our write refs.

We separately chain together the read refs and the write refs for a TN, allowing easy determination of things such as whether a TN has only a single definition or has no reads. It would also allow easier traversal of the target graph.

Represent per-location conflicts as vectors indexed by block number of per-block conflict info. To test whether a TN conflicts on a location, we would then have to iterate over the TNs global-conflicts, using the block number and LTN number to check for a conflict in that block. But since most TNs are local, this test actually isn't much more expensive than indexing into a bit-vector by GTN numbers.

The big win of this scheme is that it is much cheaper to add conflicts into the conflict set for a location, since we never need to actually compute the conflict set in a list-like representation (which requires iterating over the LTN conflicts vectors and unioning in the always-live TNs). Instead, we just iterate over the global-conflicts for the TN, using BIT-IOR to combine the conflict set with the bit-vector for that block in that location, or marking that block/location combination as being always-live if the conflict is always-live.

Generating the conflict set is inherently more costly, since although we believe the conflict set size to be roughly constant, it can easily contain tens of elements. We would have to generate these moderately large lists for all TNs, including local TNs. In contrast, the proposed scheme does work proportional to the number of blocks the TN is live in, which is small on average (1 for local TNs). This win exists independently from the win of not having to iterate over LTN conflict vectors.

[### Note that since we never do bitwise iteration over the LTN conflict vectors, part of the motivation for keeping these a small fixed size has been removed. But it would still be useful to keep the size fixed so that we can easily recycle the bit-vectors, and so that we could potentially have maximally tense special primitives for doing clear and bit-ior on these vectors.]

This scheme is somewhat more space-intensive than having a per-location bit-vector. Each vector entry would be something like 150 bits rather than one bit, but this is mitigated by the number of blocks being 5-10x smaller than the number of TNs. This seems like an acceptable overhead, a small fraction of the total VMR representation.

The space overhead could also be reduced by using something equivalent to a two-dimensional bit array, indexed first by LTN numbers, and then block numbers (instead of using a simple-vector of separate bit-vectors.) This would eliminate space wastage due to bit-vector overheads, which might be 50more, and would also make efficient zeroing of the vectors more straightforward. We would then want efficient operations for OR'ing LTN conflict vectors with rows in the array.

This representation also opens a whole new range of allocation algorithms: ones that store allocate TNs in different locations within different portions of the program. This is because we can now represent a location being used to hold a certain TN within an arbitrary subset of the blocks the TN is referenced in.

Pack goals:

Pack should:

Subject to resource constraints: – Minimize use costs – "Register allocation" Allocate as many values as possible in scarce "good" locations, attempting to minimize the aggregate use cost for the entire program. – "Save optimization" Don't allocate values in registers when the save/restore costs exceed the expected gain for keeping the value in a register. (Similar to "opening costs" in RAOC.) [Really just a case of representation selection.]

– Minimize preference costs Eliminate as many moves as possible.

"Register allocation" is basically an attempt to eliminate moves between registers and memory. "Save optimization" counterbalances "register allocation" to prevent it from becoming a pessimization, since saves can introduce register/memory moves.

Preference optimization reduces the number of moves within an SC. Doing a good job of honoring preferences is important to the success of the compiler, since we have assumed in many places that moves will usually be optimized away.

The scarcity-oriented aspect of "register allocation" is handled by a greedy algorithm in pack. We try to pack the "most important" TNs first, under the theory that earlier packing is more likely to succeed due to fewer constraints.

The drawback of greedy algorithms is their inability to look ahead. Packing a TN may mess up later "register allocation" by precluding packing of TNs that are individually "less important," but more important in aggregate. Packing a TN may also prevent preferences from being honored.

Initial packing:

Pack all TNs restricted to a finite SC first, before packing any other TNs.

One might suppose that Pack would have to treat TNs in different environments differently, but this is not the case. Pack simply assigns TNs to locations so that no two conflicting TNs are in the same location. In the process of implementing call semantics in conflict analysis, we cause TNs in different environments not to conflict. In the case of passing TNs, cross environment conflicts do exist, but this reflects reality, since the passing TNs are live in both the caller and the callee. Environment semantics has already been implemented at this point.

This means that Pack can pack all TNs simultaneously, using one data structure to represent the conflicts for each location. So we have only one conflict set per SB location, rather than separating this information by environment.

Load TN packing:

We create load TNs as needed in a post-pass to the initial packing. After TNs are packed, it may be that some references to a TN will require it to be in a SC other than the one it was packed in. We create load-TNs and pack them on the fly during this post-pass.

What we do is have an optional SC restriction associated with TN-refs. If we pack the TN in an SC which is different from the required SC for the reference, then we create a TN for each such reference, and pack it into the required SC.

In many cases we will be able to pack the load TN with no hassle, but in general we may need to spill a TN that has already been packed. We choose a TN that isn't in use by the offending VOP, and then spill that TN onto the stack for the duration of that VOP. If the VOP is a conditional, then we must insert a new block interposed before the branch target so that the TN value is restored regardless of which branch is taken.

Instead of remembering lifetime information from conflict analysis, we rederive it. We scan each block backward while keeping track of which locations have live TNs in them. When we find a reference that needs a load TN packed, we try to pack it in an unused location. If we can't, we unpack the currently live TN with the lowest cost and force it into an unbounded SC.

The per-location and per-TN conflict information used by pack doesn't need to be updated when we pack a load TN, since we are done using those data structures.

We also don't need to create any TN-Refs for load TNs. [??? How do we keep track of load-tn lifetimes? It isn't really that hard, I guess. We just remember which load TNs we created at each VOP, killing them when we pass the loading (or saving) step. This suggests we could flush the Refs thread if we were willing to sacrifice some flexibility in explicit temporary lifetimes. Flushing the Refs would make creating the VMR representation easier.]

The lifetime analysis done during load-TN packing doubles as a consistency check. If we see a read of a TN packed in a location which has a different TN currently live, then there is a packing bug. If any of the TNs recorded as being live at the block beginning are packed in a scarce SB, but aren't current in that location, then we also have a problem.

The conflict structure for load TNs is fairly simple, the load TNs for arguments and results all conflict with each other, and don't conflict with much else. We just try packing in targeted locations before trying at random.

Chapter 22

Code generation

This is fairly straightforward. We translate VOPs into instruction sequences on a per-block basis.

After code generation, the VMR representation is gone. Everything is represented by the assembler data structures.

Chapter 23

Assembly

In effect, we do much of the work of assembly when the compiler is compiled.

The assembler makes one pass fixing up branch offsets, then squeezes out the space left by branch shortening and dumps out the code along with the load-time fixup information. The assembler also deals with dumping unboxed non-immediate constants and symbols. Boxed constants are created by explicit constructor code in the top-level form, while immediate constants are generated using inline code.

[### The basic output of the assembler is: A code vector A representation of the fixups along with indices into the code vector for the fixup locations A PC map translating PCs into source paths

This information can then be used to build an output file or an in-core function object.]

The assembler is table-driven and supports arbitrary instruction formats. As far as the assembler is concerned, an instruction is a bit sequence that is broken down into subsequences. Some of the subsequences are constant in value, while others can be determined at assemble or load time.

```
Assemble Node Form*
    Allow instructions to be emitted during the evaluation of the Forms by
    defining Inst as a local macro.  This macro caches various global
    information in local variables.  Node tells the assembler what node
    ultimately caused this code to be generated.  This is used to create the
    pc=>source map for the debugger.

Assemble-Elsewhere Node Form*
    Similar to Assemble, but the current assembler location is changed to
    somewhere else.  This is useful for generating error code and similar
    things.  Assemble-Elsewhere may not be nested.

Inst Name Arg*
    Emit the instruction Name with the specified arguments.

Gen-Label
Emit-Label (Label)
    Gen-Label returns a Label object, which describes a place in the code.
    Emit-Label marks the current position as being the location of Label.
```

Chapter 24

Dumping

So far as input to the dumper/loader, how about having a list of Entry-Info structures in the VMR-Component? These structures contain all information needed to dump the associated function objects, and are only implicitly associated with the functional/XEP data structures. Load-time constants that reference these function objects should specify the Entry-Info, rather than the functional (or something). We would then need to maintain some sort of association so VMR conversion can find the appropriate Entry-Info. Alternatively, we could initially reference the functional, and then later clobber the reference to the Entry-Info.

We have some kind of post-pass that runs after assembly, going through the functions and constants, annotating the VMR-Component for the benefit of the dumper: Resolve :Label load-time constants. Make the debug info. Make the entry-info structures.

Fasl dumper and in-core loader are implementation (but not instruction set) dependent, so we want to give them a clear interface.

```
open-fasl-file name => fasl-file
    Returns a ``fasl-file'' object representing all state needed by the dumper.
    We objectify the state, since the fasdumper should be reentrant.  (but
    could fail to be at first.)

close-fasl-file fasl-file abort-p
    Close the specified fasl-file.

fasl-dump-component component code-vector length fixups fasl-file
    Dump the code, constants, etc. for component.  Code-Vector is a vector
    holding the assembled code.  Length is the number of elements of Vector
    that are actually in use.  Fixups is a list of conses (offset . fixup)
    describing the locations and things that need to be fixed up at load time.
    If the component is a top-level component, then the top-level lambda will
    be called after the component is loaded.

load-component component code-vector length fixups
    Like Fasl-Dump-Component, but directly installs the code in core, running
    any top-level code immediately.  (???) but we need some way to glue
    together the componenents, since we don't have a fasl table.
```

Dumping:

Dump code for each component after compiling that component, but defer dumping of other stuff. We do the fixups on the code vectors, and accumulate them in the table.

We have to grovel the constants for each component after compiling that component so that we can fix up load-time constants. Load-time constants are values needed by the code that are computed after code generation/assembly time. Since the code is fixed at this point, load-time constants are always represented as non-immediate constants in the constant pool. A load-time constant is distinguished by being a cons (Kind . What), instead of a Constant leaf. Kind is a keyword indicating how the constant is computed, and What is some context.

Some interesting load-time constants:

```
  (:label . <label>)
      Is replaced with the byte offset of the label within the code-vector.

  (:code-vector . <component>)
      Is replaced by the component's code-vector.

  (:entry . <function>)
  (:closure-entry . <function>)
Is replaced by the function-entry structure for the specified function.
:Entry is how the top-level component gets a handle on the function
definitions so that it can set them up.
```

We also need to remember the starting offset for each entry, although these don't in general appear as explicit constants.

We then dump out all the :Entry and :Closure-Entry objects, leaving any constant-pool pointers uninitialized. After dumping each :Entry, we dump some stuff to let genesis know that this is a function definition. Then we dump all the constant pools, fixing up any constant-pool pointers in the already-dumped function entry structures.

The debug-info *is* a constant: the first constant in every constant pool. But the creation of this constant must be deferred until after the component is compiled, so we leave a (:debug-info) placeholder. [Or maybe this is implicitly added in by the dumper, being supplied in a VMR-component slot.]

Work out details of the interface between the back-end and the assembler/dumper.

Support for multiple assemblers concurrently loaded? (for byte code)

We need various mechanisms for getting information out of the assembler.

We can get entry PCs and similar things into function objects by making a Constant leaf, specifying that it goes in the closure, and then setting the value after assembly.

We have an operation Label-Value which can be used to get the value of a label after assembly and before the assembler data structures are deallocated.

The function map can be constructed without any special help from the assembler. Codegen just has to note the current label when the function changes from one block to the next, and then use the final value of these labels to make the function map.

Probably we want to do the source map this way too. Although this will make zillions of spurious labels, we would have to effectively do that anyway.

With both the function map and the source map, getting the locations right for uses of Elsewhere will be a bit tricky. Users of Elsewhere will need to know about how these maps are being built, since they must record the labels and corresponding information for the elsewhere range. It would be nice to have some cooperation from Elsewhere so that this isn't necessary, otherwise some VOP writer will break the rules, resulting in code that is nowhere.

The Debug-Info and related structures are dumped by consing up the structure and making it be the value of a constant.

Getting the code vector and fixups dumped may be a bit more interesting. I guess we want a Dump-Code-Vector function which dumps the code and fixups accumulated by the current assembly, returning a magic object that will become the code vector when it is dumped as a constant.]

Chapter 25

User Interface of the Compiler

25.1 Error Message Utilities

25.2 Source Paths

Part III

Compiler Retargeting

Chapter 26

Retargeting the Compiler

[###

In general, it is a danger sign if a generator references a TN that isn't an operand or temporary, since lifetime analysis hasn't been done for that use. We are doing weird stuff for the old-cont and return-pc passing locations, hoping that the conflicts at the called function have the desired effect. Other stuff? When a function returns unknown values, we don't reference the values locations when a single-value return is done. But nothing is live at a return point anyway.

Have a way for template conversion to special-case constant arguments? How about: If an arg restriction is (:satisfies [<predicate function>]), and the corresponding argument is constant, with the constant value satisfying the predicate, then (if any other restrictions are satisfied), the template will be emitted with the literal value passed as an info argument. If the predicate is omitted, then any constant will do.

We could sugar this up a bit by allowing (:member <object>*) for (:satisfies (lambda (x) (member x '(<object>*))))

We could allow this to be translated into a Lisp type by adding a new Constant type specifier. This could only appear as an argument to a function type. To satisfy (Constant <type>), the argument must be a compile-time constant of the specified type. Just Constant means any constant (i.e. (Constant *)). This would be useful for the type constraints on ICR transforms.

Constant TNs: we count on being able to indirect to the leaf, and don't try to wedge the information into the offset. We set the FSC to an appropriate immediate SC.

Allow "more operands" to VOPs in define-vop. You can't do much with the more operands: define-vop just fills in the cost information according to the loading costs for a SC you specify. You can't restrict more operands, and you can't make local preferences. In the generator, the named variable is bound to the TN-ref for the first extra operand. This should be good enough to handle all the variable arg VOPs (primarily function call and return). Usually more operands are used just to get TN lifetimes to work out; the generator actually ignores them.

Variable-arg VOPs can't be used with the VOP macro. You must use VOP*. VOP* doesn't do anything with these extra operand except stick them on the ends of the operand lists passed into the template. VOP* is often useful within the convert functions for non-VOP templates, since it can emit a VOP using an already prepared TN-Ref list.

It is pretty basic to the whole primitive-type idea that there is only one primitive-type for a given lisp type. This is really the same as saying primitive types are disjoint. A primitive type serves two somewhat unrelated purposes: – It is an abstraction of a Lisp type used to select type specific operations. Originally kind of an efficiency hack, but it lets a template's type signature be used both for selection and operand representation determination. – It represents a set of possible representations for a value (SCs). The primitive type is used to determine the legal SCs for a TN, and is also used to determine which type-coercion/move VOP to use.

]

There are basically three levels of target dependence:

– Code in the "front end" (before VMR conversion) deals only with Lisp semantics, and is totally target independent.

– Code after VMR conversion and before code generation depends on the VM, but should work with little modification across a wide range of "conventional" architectures.

– Code generation depends on the machine's instruction set and other implementation details, so it will have to be redone for each implementation. Most of the work here is in defining the translation into assembly code of all the supported VOPs.

Chapter 27

Storage bases and classes

New interface: instead of CURRENT-FRAME-SIZE, have CURRENT-SB-SIZE `<name>` which returns the current element size of the named SB.

How can we have primitive types that overlap, i.e. (UNSIGNED-BYTE 32), (SIGNED-BYTE 32), FIXNUM? Primitive types are used for two things: Representation selection: which SCs can be used to represent this value? For this purpose, it isn't necessary that primitive types be disjoint, since any primitive type can choose an arbitrary set of representations. For moves between the overlapping representations, the move/load operations can just be noops when the locations are the same (vanilla MOVE), since any bad moves should be caught out by type checking. VOP selection: Is this operand legal for this VOP? When ptypes overlap in interesting ways, there is a problem with allowing just a simple ptype restriction, since we might want to allow multiple ptypes. This could be handled by allowing "union primitive types", or by allowing multiple primitive types to be specified (only in the operand restriction.) The latter would be along the lines of other more flexible VOP operand restriction mechanisms, (constant, etc.)

Ensure that load/save-operand never need to do representation conversion.

The PRIMITIVE-TYPE more/coerce info would be moved into the SC. This could perhaps go along with flushing the TN-COSTS. We would annotate the TN with best SC, which implies the representation (boxed or unboxed). We would still need to represent the legal SCs for restricted TNs somehow, and also would have to come up with some other way for pack to keep track of which SCs we have already tried.

An SC would have a list of "alternate" SCs and a boolean SAVE-P value that indicates it needs to be saved across calls in some non-SAVE-P SC. A TN is initially given its "best" SC. The SC is annotated with VOPs that are used for moving between the SC and its alternate SCs (load/save operand, save/restore register). It is also annotated with the "move" VOPs used for moving between this SC and all other SCs it is possible to move between. We flush the idea that there is only c-to-t and c-from-t.

But how does this mesh with the idea of putting operand load/save back into the generator? Maybe we should instead specify a load/save function? The load/save functions would also differ from the move VOPs in that they would only be called when the TN is in fact in that particular alternate SC, whereas the move VOPs will be associated with the primary SC, and will be emitted before it is known whether the TN will be packed in the primary SC or an alternate.

I guess a packed SC could also have immediate SCs as alternate SCs, and constant loading functions could be associated with SCs using this mechanism.

So given a TN packed in SC X and an SC restriction for Y and Z, how do we know which load function to call? There would be ambiguity if X was an alternate for both Y and Z and they specified different load functions. This seems unlikely to arise in practice, though, so we could just detect the ambiguity and give an error at define-vop time. If they are doing something totally weird, they can always inhibit loading and roll their own.

Note that loading costs can be specified at the same time (same syntax) as association of loading functions with SCs. It seems that maybe we will be rolling DEFINE-SAVE-SCS and DEFINE-MOVE-COSTS into DEFINE-STORAGE-CLASS.

Fortunately, these changes will affect most VOP definitions very little.

A Storage Base represents a physical storage resource such as a register set or stack frame. Storage bases for non-global resources such as the stack are relativized by the environment that the TN is allocated in. Packing conflict information is kept in the storage base, but non-packed storage resources such as closure environments also have storage bases. Some storage bases:

```
    General purpose registers
    Floating point registers
    Boxed (control) stack environment
    Unboxed (number) stack environment
    Closure environment
```

A storage class is a potentially arbitrary set of the elements in a storage base. Although conceptually there may be a hierarchy of storage classes such as “all registers”, “boxed registers”, “boxed scratch registers”, this doesn’t exist at the implementation level. Such things can be done by specifying storage classes whose locations overlap. A TN shouldn’t have lots of overlapping SC’s as legal SC’s, since time would be wasted repeatedly attempting to pack in the same locations.

There will be some SC’s whose locations overlap a great deal, since we get Pack to do our representation analysis by having lots of SC’s. An SC is basically a way of looking at a storage resource. Although we could keep a fixnum and an unboxed representation of the same number in the same register, they correspond to different SC’s since they are different representation choices.

TNs are annotated with the primitive type of the object that they hold: T: random boxed object with only one representation. Fixnum, Integer, XXX-Float: Object is always of the specified numeric type. String-Char: Object is always a string-char.

When a TN is packed, it is annotated with the SC it was packed into. The code generator for a VOP must be able to uniquely determine the representation of its operands from the SC. (debugger also...)

Some SCs: Reg: any register (immediate objects) Save-Reg: a boxed register near r15 (registers easily saved in a call) Boxed-Reg: any boxed register (any boxed object) Unboxed-Reg: any unboxed register (any unboxed object) Float-Reg, Double-Float-Reg: float in FP register. Stack: boxed object on the stack (on cstack) Word: any 32bit unboxed object on nstack. Double: any 64bit unboxed object on nstack.

We have a number of non-packed storage classes which serve to represent access costs associated with values that are not allocated using conflicts information. Non-packed TNs appear to already be packed in the appropriate storage base so that Pack doesn’t get confused. Costs for relevant non-packed SC’s appear in the TN-Ref cost information, but need not ever be summed into the TN cost vectors, since TNs cannot be packed into them.

There are SCs for non-immediate constants and for each significant kind of immediate operand in the architecture. On the RT, 4, 8 and 20 bit integer SCs are probably worth having.

```
Non-packed SCs:
    Constant
    Immediate constant SCs:
        Signed-Byte-<N>, Unsigned-Byte-<N>, for various architecture dependent
    values of <N>
String-Char
XXX-Float
Magic values: T, NIL, 0.
```

Chapter 28

Type system parameterization

The main aspect of the VM that is likely to vary for good reason is the type system:

– Different systems will have different ways of representing dynamic type information. The primary effect this has on the compiler is causing VMR conversion of type tests and checks to be implementation dependent. Rewriting this code for each implementation shouldn't be a big problem, since the portable semantics of types has already been dealt with.

– Different systems will have different specialized number and array types, and different VOPs specialized for these types. It is easy to add this kind of knowledge without affecting the rest of the compiler. All you have to do is define the VOPs and translations.

– Different systems will offer different specialized storage resources such as floating-point registers, and will have additional kinds of primitive-types. The storage class mechanism handles a large part of this, but there may be some problem in getting VMR conversion to realize the possibly large hidden costs in implicit moves to and from these specialized storage resources. Probably the answer is to have some sort of general mechanism for determining the primitive-type for a TN given the Lisp type, and then to have some sort of mechanism for automatically using specialized Move VOPs when the source or destination has some particular primitive-type.

#— How to deal with list/null(symbol)/cons in primitive-type structure? Since cons and symbol aren't used for type-specific template selection, it isn't really all that critical. Probably Primitive-Type should return the List primitive type for all of Cons, List and Null (indicating when it is exact). This would allow type-dispatch for simple sequence functions (such as length) to be done using the standard template-selection mechanism. [Not a wired assumption] —#

Chapter 29

VOP Definition

Before the operand TN-refs are passed to the emit function, the following stuff is done: – The refs in the operand and result lists are linked together in order using the Across slot. This list is properly NIL terminated. – The TN slot in each ref is set, and the ref is linked into that TN's refs using the Next slot. – The Write-P slot is set depending on whether the ref is an argument or result. – The other slots have the default values.

The template emit function fills in the Vop, Costs, Cost-Function, SC-Restriction and Preference slots, and links together the Next-Ref chain as appropriate.

29.1 Lifetime model

#— Note in doc that the same TN may not be used as both a more operand and as any other operand to the same VOP, to simplify more operand LTN number coalescing. —#

It seems we need a fairly elaborate model for intra-VOP conflicts in order to allocate temporaries without introducing spurious conflicts. Consider the important case of a VOP such as a miscop that must have operands in certain registers. We allocate a wired temporary, create a local preference for the corresponding operand, and move to (or from) the temporary. If all temporaries conflict with all arguments, the result will be correct, but arguments could never be packed in the actual passing register. If temporaries didn't conflict with any arguments, then the temporary for an earlier argument might get packed in the same location as the operand for a later argument; loading would then destroy an argument before it was read.

A temporary's intra-VOP lifetime is represented by the times at which its life starts and ends. There are various instants during the evaluation that start and end VOP lifetimes. Two TNs conflict if the live intervals overlap. Lifetimes are open intervals: if one TN's lifetime begins at a point where another's ends, then the TNs don't conflict.

The times within a VOP are the following:

:Load This is the beginning of the argument's lives, as far as intra-vop conflicts are concerned. If load-TNs are allocated, then this is the beginning of their lives.

(:Argument <n>) The point at which the N'th argument is read for the last time (by this VOP). If the argument is dead after this VOP, then the argument becomes dead at this time, and may be reused as a temporary or result load-TN.

(:Eval <n>) The N'th evaluation step. There may be any number of evaluation steps, but it is unlikely that more than two are needed.

(:Result <n>) The point at which the N'th result is first written into. This is the point at which that result becomes live.

:Save Similar to :Load, but marks the end of time. This is the point at which result load-TNs are stored back to the actual location.

In any of the list-style time specifications, the keyword by itself stands for the first such time, i.e.

```
  :argument  <==>  (:argument 0)
```

Note that argument/result read/write times don't actually have to be in the order specified, but they must *appear* to happen in that order as far as conflict analysis is concerned. For example, the arguments can be read in any order as long as no TN is written that has a life beginning at or after (:Argument <n>), where N is the number of an argument whose reading was postponed.

[### (???)

We probably also want some syntactic sugar in Define-VOP for automatically moving operands to/from explicitly allocated temporaries so that this kind of thing is somewhat easy. There isn't really any reason to consider the temporary to be a load-TN, but we want to compute costs as though it was and want to use the same operand loading routines.

We also might consider allowing the lifetime of an argument/result to be extended forward/backward. This would in many cases eliminate the need for temporaries when operands are read/written out of order.]

29.2 VOP Cost model

Note that in this model, if an operand has no restrictions, it has no cost. This makes sense, since the purpose of the cost is to indicate the relative value of packing in different SCs. If the operand isn't required to be in a good SC (i.e. a register), then we might as well leave it in memory. The SC restriction mechanism can be used even when doing a move into the SC is too complex to be generated automatically (perhaps requiring temporary registers), since Define-VOP allows operand loading to be done explicitly.

29.3 Efficiency notes

In addition to being used to tell whether a particular unsafe template might get emitted, we can also use it to give better efficiency notes: – We can say what is wrong with the call types, rather than just saying we failed to open-code. – We can tell whether any of the "better" templates could possibly apply, i.e. is the inapplicability of a template because of inadequate type information or because the type is just plain wrong. We don't want to flame people when a template that couldn't possibly match doesn't match, e.g. complaining that we can't use fixnum+ when the arguments are known to be floats.

This is how we give better efficiency notes:

The Template-Note is a short noun-like string without capitalization or punctuation that describes what the template "does", i.e. we say "Unable to do A, doing A instead."

The Cost is moved from the Vop-Info to the Template structure, and is used to determine the "goodness" of possibly applicable templates. [Could flush Template/Vop-Info distinction] The cost is used to choose the best applicable template to emit, and also to determine what better templates we might have been able to use.

A template is possibly applicable if there is an intersection between all of the arg/result types and the corresponding arg/result restrictions, i.e. the template is not clearly impossible: more declarations might allow it to be emitted.

Chapter 30

Assembler Retargeting

Chapter 31

Writing Assembly Code

VOP writers expect:

MOVE You write when you port the assembler.)

EMIT-LABEL
: Assembler interface like INST. Takes a label you made and says "stick it here."

GEN-LABEL
: Returns a new label suitable for use with EMIT-LABEL exactly once and for referencing as often as necessary.

INST Recognizes and dispatches to instructions you defined for assembler.

ALIGN This takes the number of zero bits you want in the low end of the address of the next instruction.

ASSEMBLE

ASSEMBLE-ELSEWHERE
: Get ready for assembling stuff. Takes a VOP and arbitrary PROGN-style body. Wrap these around instruction emission code announcing the first pass of our assembler.

CURRENT-NFP-TN
: This returns a TN for the NFP if the caller uses the number stack, or nil.

SB-ALLOCATED-SIZE
: This returns the size of some storage base used by the currently compiling component.

...

;;; ;;; VOP idioms ;;;

STORE-STACK-TN

LOAD-STACK-TN
: These move a value from a register to the control stack, or from the control stack to a register. They take care of checking the TN types, modifying offsets according to the address units per word, etc.

Chapter 32

Required VOPS

Note: the move VOP cannot have any wired temps. (Move-Argument also?) This is so we can move stuff into wired TNs without stepping on our toes.

We create set closure variables using the Value-Cell VOP, which takes a value and returns a value cell containing the value. We can basically use this instead of a Move VOP when initializing the variable. Value-Cell-Set and Value-Cell-Ref are used to access the value cell. We can have a special effect for value cells so that value cells references can be discovered to be common subexpressions or loop invariants.

Represent unknown-values continuations as (start, count). Unknown values continuations are always outside of the current frame (on stack top). Within a function, we always set up and receive values in the standard passing locations. If we receive stack values, then we must BLT them down to the start of our frame, filling in any unsupplied values. If we generate unknown values (i.e. PUSH-VALUES), then we set the values up in the standard locations, then BLT them to stack top. When doing a tail-return of MVs, we just set them up in the standard locations and decrement SP: no BLT is necessary.

Unknown argument call (MV-CALL) takes its arguments on stack top (is given a base pointer). If not a tail call, then we just set the arg pointer to the base pointer and call. If a tail call, we must BLT the arguments down to the beginning of the current frame.

Implement more args by BLT'ing the more args *on top* of the current frame. This solves two problems:

- Any register more arguments can be made uniformly accessibly by copying them into memory. [We can't store the registers in place, since the beginning of the frame gets double use for storing the old-cont, return-pc and env.]
- It solves the deallocation problem: the arguments will be deallocated when the frame is returned from or a tail full call is done out of it. So keyword args will be properly tail-recursive without any special mechanism for squeezing out the more arg once the parsing is done. Note that a tail local call won't blast the more arg, since in local call the callee just takes the frame it is given (in this case containing the more arg).

More args in local call??? Perhaps we should not attempt local call conversion in this case. We already special-case keyword args in local call. It seems that the main importance of more args is primarily related to full call: it is used for defining various kinds of frobs that need to take arbitrary arguments:

- Keyword arguments
- Interpreter stubs
- "Pass through" applications such as dispatch functions

Given the marginal importance of more args in local call, it seems unworth going to any implementation difficulty. In fact, it seems that it would cause complications both at the VMR level and also in the VM definition. This being the case, we should flush it.

32.1 Function Call

32.1.1 Registers and frame format

These registers are used in function call and return:

A0..An In full call, the first three arguments. In unknown values return, the first three return values.

CFP The current frame pointer. In full call, this initially points to a partial frame large enough to hold the passed stack arguments (zero-length if none).

CSP The current control stack top pointer.

OCFP In full call, the passing location for the frame to return to.

In unknown-values return of other than one value, the pointer to returned stack values. In such a return, OCFP is always initialized to point to the frame returned from, even when no stack values are returned. This allows OCFP to be used to restore CSP.

LRA In full call, the passing location for the return PC.

NARGS In full call, the number of arguments passed. In unknown-values return of other than one value, the number of values returned.

32.1.2 Full call

What is our usage of CFP, OCFP and CSP?

It is an invariant that CSP always points after any useful information so that at any time an interrupt can come and allocate stuff in the stack.

TR call is also a constraint: we can't deallocate the caller's frame before the call, since it holds the stack arguments for the call.

What we do is have the caller set up CFP, and have the callee set CSP to CFP plus the frame size. The caller leaves CSP alone: the callee is the one who does any necessary stack deallocation.

In a TR call, we don't do anything: CFP is left as CFP, and CSP points to the end of the frame, keeping the stack arguments from being trashed.

In a normal call, CFP is set to CSP, causing the callee's frame to be allocated after the current frame.

32.1.3 Unknown values return

The unknown values return convention is always used in full call, and is used in local call when the compiler either can't prove that a fixed number of values are returned, or decides not to use the fixed values convention to allow tail-recursive XEP calls.

The unknown-values return convention has variants: single value and variable values. We make this distinction to optimize the important case of a returner who knows exactly one value is being returned. Note that it is possible to return a single value using the variable-values convention, but it is less efficient.

We indicate single-value return by returning at the return-pc+4; variable value return is indicated by returning at the return PC.

Single-value return makes only the following guarantees: A0 holds the value returned. CSP has been reset: there is no garbage on the stack.

In variable value return, more information is passed back: A0..A2 hold the first three return values. If fewer than three values are returned, then the unused registers are initialized to NIL.

OCFP points to the frame returned from. Note that because of our tail-recursive implementation of call, the frame receiving the values is always immediately under the frame returning the values. This means that we can use OCFP to index the values when we access them, and to restore CSP when we want to discard them.

NARGS holds the number of values returned.

CSP is always (+ OCFP (* NARGS 4)), i.e. there is room on the stack allocated for all returned values, even if they are all actually passed in registers.

32.1.4 External Entry Points

Things that need to be done on XEP entry: 1] Allocate frame 2] Move more arg above the frame, saving context 3] Set up env, saving closure pointer if closure 4] Move arguments from closure to local home Move old-cont and return-pc to the save locations 5] Argument count checking and dispatching

XEP VOPs:

```
Allocate-Frame
Copy-More-Arg <nargs-tn> 'fixed {in a3} => <context>, <count>
Setup-Environment
Setup-Closure-Environment => <closure>
Verify-Argument-Count <nargs-tn> 'count {for fixed-arg lambdas}
Argument-Count-Error <nargs-tn> {Drop-thru on hairy arg dispatching}
Use fast-if-=/fixnum and fast-if-</fixnum for dispatching.
```

Closure vops:

```
make-closure <fun entry> <slot count> => <closure>
closure-init <closure> <values> 'slot
```

Things that need to be done on all function entry:

- Move arguments to the variable home (consing value cells as necessary)
- Move environment values to the local home
- Move old-cont and return-pc to the save locations

32.2 Calls

Calling VOP's are a cross product of the following sets (with some members missing): Return values multiple (all values) fixed (calling with unknown values conventions, wanting a certain number.) known (only in local call where caller/callee agree on number of values.) tail (doesn't return but does tail call) What function local named (going through symbol, like full but stash fun name for error sys) full (have a function) Args fixed (number of args are known at compile-time) variable (MULTIPLE-VALUE-CALL and APPLY)

Note on all jumps for calls and returns that we want to put some instruction in the jump's delay slot(s).

Register usage at the time of the call:

LEXENV This holds the lexical environment to use during the call if it's a closure, and it is undefined otherwise.

CNAME This holds the symbol for a named call and garbage otherwise.

OCFP This holds the frame pointer, which the system restores upon return. The callee saves this if necessary; this is passed as a pseudo-argument.

A0 ... An These holds the first n+1 arguments.

NARGS This holds the number of arguments, as a fixnum.

LRA This holds the lisp-return-address object which indicates where to return. For a tail call, this retains its current value. The callee saves this if necessary; this is passed as a pseudo-argument.

CODE This holds the function object being called.

CSP The caller ignores this. The callee sets it as necessary based on CFP.

CFP This holds the callee's frame pointer. Caller sets this to the new frame pointer, which it remembered when it started computing arguments; this is CSP if there were no stack arguments. For a tail call CFP retains its current value.

NSP The system uses this within a single function. A function using NSP must allocate and deallocate before returning or making a tail call.

Register usage at the time of the return for single value return, which goes with the unknown-values convention the caller used.

A0 This holds the value.

CODE This holds the lisp-return-address at which the system continues executing.

CSP This holds the CFP. That is, the stack is guaranteed to be clean, and there is no code at the return site to adjust the CSP.

CFP This holds the OCFP.

Additional register usage for multiple value return:

NARGS This holds the number of values returned.

A0 ... An These holds the first n+1 values, or NIL if there are less than n+1 values.

CSP Returner stores CSP to hold its CFP + NARGS * `<address units per word>`

OCFP Returner stores this as its CFP, so the returnee has a handle on either the start of the returned values on the stack.

ALLOCATE FULL CALL FRAME.

If the number of call arguments (passed to the VOP as an info argument) indicates that there are stack arguments, then it makes some callee frame for arguments:

```
   VOP-result <- CSP
   CSP <- CSP + value of VOP info arg times address units per word.
```

In a call sequence, move some arguments to the right places.

There's a variety of MOVE-ARGUMENT VOP's.

FULL CALL VOP'S (variations determined by whether it's named, it's a tail call, there is a variable arg count, etc.)

```
  if variable number of arguments
    NARGS <- (CSP - value of VOP argument) shift right by address units per word.
    A0...An <- values off of VOP argument (just fill them all)
  else
    NARGS <- value of VOP info argument (always a constant)

  if tail call
    OCFP <- value from VOP argument
    LRA <- value from VOP argument
    CFP stays the same since we reuse the frame
    NSP <- NFP
  else
    OCFP <- CFP
    LRA <- compute LRA by adding an assemble-time determined constant to
        CODE.
    CFP <- new frame pointer (remembered when starting to compute args)
           This is CSP if no stack args.
    when (current-nfp-tn VOP-self-pointer)
      stack-temp <- NFP

  if named
    CNAME <- function symbol name
    the-fun <- function object out of symbol

  LEXENV <- the-fun (from previous line or VOP argument)
  CODE <- function-entry (the first word after the-fun)
  LIP <- calc first instruction addr (CODE + constant-offset)
  jump and run off temp

  <emit Lisp return address data-block>
  <default and move return values OR receive return values>
  when (current-nfp-tn VOP-self-pointer)
    NFP <- stack-temp
```

Callee:

```
XEP-ALLOCATE-FRAME
  emit function header (maybe initializes offset back to component start,
   but other pointers are set up at load-time.  Pads
to dual-word boundary.)
  CSP <- CFP + compile-time determined constant (frame size)
  if the function uses the number stack
    NFP <- NSP
    NSP <- NSP + compile-time determined constant (number stack frame size)

SETUP-ENVIRONMENT
(either use this or the next one)

CODE <- CODE - assembler-time determined offset from function-entry back to
       the code data-block address.

SETUP-CLOSURE-ENVIRONMENT
(either use this or the previous one)
After this the CLOSURE-REF VOP can reference closure variables.

VOP-result <- LEXENV
CODE <- CODE - assembler-time determined offset from function-entry back to
       the code data-block address.
```

Return VOP's RETURN and RETURN-MULTIPLE are for the unknown-values return convention. For some previous caller this is either it wants n values (and it doesn't know how many are coming), or it wants all the values returned (and it doesn't know how many are coming).

RETURN (known fixed number of values, used with the unknown-values convention in the caller.) When compiler invokes VOP, all values are already where they should be; just get back to caller.

```
when (current-nfp-tn VOP-self-pointer)
  ;; The number stack grows down in memory.
  NSP <- NFP + number stack frame size for calls within the currently
                  compiling component
       times address units per word
CODE <- value of VOP argument with LRA
if VOP info arg is 1 (number of values we know we're returning)
  CSP <- CFP
  LIP <- calc target addr
          (CODE + skip over LRA header word + skip over address units per branch)
  (The branch is in the caller to skip down to the MV code.)
else
  NARGS <- value of VOP info arg
  nil out unused arg regs
  OCFP <- CFP  (This indicates the start of return values on the stack,
   but you leave space for those in registers for convenience.)
  CSP <- CFP + NARGS * address-units-per-word
  LIP <- calc target addr (CODE + skip over LRA header word)
CFP <- value of VOP argument with OCFP
jump and run off LIP
```

RETURN-MULTIPLE (unknown number of values, used with the unknown-values convention in the caller.) When compiler invokes VOP, it gets TN's representing a pointer to the values on the stack and how many values were computed.

```
when (current-nfp-tn VOP-self-pointer)
  ;; The number stack grows down in memory.
  NSP <- NFP + number stack frame size for calls within the currently
                  compiling component
       times address units per word
NARGS <- value of VOP argument
copy the args to the beginning of the current (returner's) frame.
   Actually some go into the argument registers.  When putting the rest at
   the beginning of the frame, leave room for those in the argument registers.
CSP <- CFP + NARGS * address-units-per-word
nil out unused arg regs
OCFP <- CFP  (This indicates the start of return values on the stack,
      but you leave space for those in registers for convenience.)
CFP <- value of VOP argument with OCFP
CODE <- value of VOP argument with LRA
LIP <- calc target addr (CODE + skip over LRA header word)
jump and run off LIP
```

Returnee The call VOP's call DEFAULT-UNKNOWN-VALUES or RECEIVE-UNKNOWN-VALUES after spitting out transfer control to get stuff from the returner.

DEFAULT-UNKNOWN-VALUES (We know what we want and we got something.) If returnee wants one value, it never does anything to deal with a shortage of return values. However, if start at PC, then it has to adjust the stack pointer to dump extra values (move OCFP into CSP). If it starts at PC+N, then it just goes along with the "want one value, got it" case. If the returnee wants multiple values, and there's a shortage of return values, there are two cases to handle. One, if the returnee wants fewer values than there are return registers, and we start at PC+N, then it fills in return registers `A1..A<desired values necessary>`; if we start at PC, then the returnee is fine since the returning conventions have filled in the unused return registers with nil, but the returnee must adjust the stack pointer to dump possible stack return values (move OCFP to CSP). Two, if the returnee wants more values than the number of return registers, and it starts at PC+N (got one value), then it sets up returnee state as if an unknown number of values came back:

```
   A0 has the one value
   A1..An get nil
   NARGS gets 1
   OCFP gets CSP, so general code described below can move OCFP into CSP
If we start at PC, then branch down to the general ``got k values, wanted n''
code which takes care of the following issues:
   If k < n, fill in stack return values of nil for shortage of return
      values and move OCFP into CSP
   If k >= n, move OCFP into CSP
This also restores CODE from LRA by subtracting an assemble-time constant.
```

RECEIVE-UKNOWN-VALUES (I want whatever I get.) We want these at the end of our frame. When the returnee starts at PC, it moves the return value registers to OCFP..OCFP[An] ignoring where the end of the stack is and whether all the return value registers had values. The returner left room on the stack before the stack return values for the register return values. When the returnee starts at PC+N, bump CSP by 1 and copy A0 there. This also restores CODE from LRA by subtracting an assemble-time constant.

Local call

There are three flavors: 1] KNOWN-CALL-LOCAL Uses known call convention where caller and callee agree where all the values are, and there's a fixed number of return values. 2] CALL-LOCAL Uses the unknown-values convention, but we expect a particular number of values in return. 3] MULTIPLE-CALL-LOCAL Uses the unknown-values convention, but we want all values returned.

ALLOCATE-FRAME

If the number of call arguments (passed to the VOP as an info argument) indicates that there are stack arguments, then it makes some callee frame for arguments:

```
   VOP-result1 <- CSP
   CSP <- CSP + control stack frame size for calls within the currently
       compiling component
    times address units per word.
   when (callee-nfp-tn <VOP info arg holding callee>)
     ;; The number stack grows down.
     ;; May have to round to dual-word boundary if machines C calling
     ;;    conventions demand this.
     NSP <- NSP - number stack frame size for calls within the currently
```

```
          compiling component
  times address units per word
     VOP-result2 <- NSP
```

KNOWN-CALL-LOCAL, CALL-LOCAL, MULTIPLE-CALL-LOCAL KNOWN-CALL-LOCAL has no need to affect CODE since CODE is the same for the caller/returnee and the returner. This uses KNOWN-RETURN. With CALL-LOCAL and MULTIPLE-CALL-LOCAL, the caller/returnee must fixup CODE since the callee may do a tail full call. This happens in the code emitted by DEFAULT-UNKNOWN-VALUES and RECEIVE-UNKNOWN-VALUES. We use these return conventions since we don't know what kind of values the returner will give us. This could happen due to a tail full call to an unknown function, or because the callee had different return points that returned various numbers of values.

```
when (current-nfp-tn VOP-self-pointer)   ;Get VOP self-pointer with
 ;DEFINE-VOP switch :vop-var.
  stack-temp <- NFP
CFP <- value of VOP arg
when (callee-nfp-tn <VOP info arg holding callee>)
  <where-callee-wants-NFP-tn>  <-  value of VOP arg
<where-callee-wants-LRA-tn>  <-  compute LRA by adding an assemble-time
 determined constant to CODE.
jump and run off VOP info arg holding start instruction for callee

<emit Lisp return address data-block>
<case call convention
  known: do nothing
  call: default and move return values
  multiple: receive return values
>
when (current-nfp-tn VOP-self-pointer)
  NFP <- stack-temp
```

KNOWN-RETURN

```
CSP <- CFP
when (current-nfp-tn VOP-self-pointer)
  ;; number stack grows down in memory.
  NSP <- NFP + number stack frame size for calls within the currently
                compiling component
       times address units per word
LIP <- calc target addr (value of VOP arg + skip over LRA header word)
CFP <- value of VOP arg
jump and run off LIP
```

Chapter 33

Standard Primitives

Chapter 34

Customizing VMR Conversion

Another way in which different implementations differ is in the relative cost of operations. On machines without an integer multiply instruction, it may be desirable to convert multiplication by a constant into shifts and adds, while this is surely a bad idea on machines with hardware support for multiplication. Part of the tuning process for an implementation will be adding implementation dependent transforms and disabling undesirable standard transforms.

When practical, ICR transforms should be used instead of VMR generators, since transforms are more portable and less error-prone. Note that the Lisp code need not be implementation independent: it may contain all sorts of sub-primitives and similar stuff. Generally a function should be implemented using a transform instead of a VMR translator unless it cannot be implemented as a transform due to being totally evil or it is just as easy to implement as a translator because it is so simple.

34.1 Constant Operands

If the code emitted for a VOP when an argument is constant is very different than the non-constant case, then it may be desirable to special-case the operation in VMR conversion by emitting different VOPs. An example would be if SVREF is only open-coded when the index is a constant, and turns into a miscop call otherwise. We wouldn't want constant references to spuriously allocate all the miscop linkage registers on the off chance that the offset might not be constant. See the :constant feature of VOP primitive type restrictions.

34.2 Supporting Multiple Hardware Configurations

A winning way to change emitted code depending on the hardware configuration, i.e. what FPA is present is to do this using primitive types. Note that the Primitive-Type function is VM supplied, and can look at any appropriate hardware configuration switches. Short-Float can become 6881-Short-Float, AFPA-Short-Float, etc. There would be separate SBs and SCs for the registers of each kind of FP hardware, with each hardware-specific primitive type using the appropriate float register SC. Then the hardware specific templates would provide AFPA-Short-Float as the argument type restriction.

Primitive type changes:

The primitive-type structure is given a new %Type slot, which is the CType structure that is equivalent to this type. There is also a Guard slot, which, if true is a function that control whether this primitive type is allowed (due to hardware configuration, etc.)

We add new :Type and :Guard keywords to Def-Primitive-Type. Type is the type specifier that is equivalent (default to the primitive-type name), and Guard is an expression evaluated in the null environment that controls whether this type applies (default to none, i.e. constant T).

The Primitive-Type-Type function returns the Lisp CType corresponding to a primitive type. This is the %Type unless there is a guard that returns false, in which case it is the empty type (i.e. NIL).

[But this doesn't do what we want it to do, since we will compute the function type for a template at load-time, so they will correspond to whatever configuration was in effect then. Maybe we don't want to dick with guards here (if at all). I guess we can defer this issue until we actually support different FP configurations. But it would seem pretty losing to separately flame about all the different FP configurations that could be used to open-code + whenever we are forced to closed-code +.

If we separately report each better possibly applicable template that we couldn't use, then it would be reasonable to report any conditional template allowed by the configuration.

But it would probably also be good to give some sort of hint that perhaps it would be a good time to make sure you understand how to tell the compiler to compile for a particular configuration. Perhaps if there is a template that applies *but for the guard*, then we could give a note. This way, if someone thinks they are being efficient by throwing in lots of declarations, we can let them know that they may have to do more.

I guess the guard should be associated with the template rather than the primitive type. This would allow LTN and friends to easily tell whether a template applies in this configuration. It is also probably more natural for some sorts of things: with some hardware variants, it may be that the SBs and representations (SCs) are really the same, but there are some different allowed operations. In this case, we could easily conditionalize VOPs without the increased complexity due to bogus SCs. If there are different storage resources, then we would conditionalize Primitive-Type as well.

34.3 Special-case VMR convert methods

(defun continuation-tn (cont &optional (check-p t)) ...) Return the TN which holds Continuation's first result value. In general this may emit code to load the value into a TN. If Check-P is true, then when policy indicates, code should be emitted to check that the value satisfies the continuation asserted type.

(defun result-tn (cont) ...) Return the TN that Continuation's first value is delivered in. In general, may emit code to default any additional values to NIL.

(defun result-tns (cont n) ...) Similar to Result-TN, except that it returns a list of N result TNs, one for each of the first N values.

Nearly all open-coded functions should be handled using standard template selection. Some (all?) exceptions:

- List, List* and Vector take arbitrary numbers of arguments. Could implement Vector as a source transform. Could even do List in a transform if we explicitly represent the stack args using %More-Args or something.
- %Typep varies a lot depending on the type specifier. We don't want to transform it, since we want %Typep as a canonical form so that we can do type optimizations.
- Apply is weird.
- Funny functions emitted by the compiler: %Listify-Rest-Args, Arg, %More-Args, %Special-Bind, %Catch, %Unknown-Values (?), %Unwind-Protect, %Unwind, %%Primitive.

Part IV

Run-Time System

Chapter 35

The Type System

Chapter 36

The Info Database

The info database provides a functional interface to global information about named things in CMUCL. Information is considered to be global if it must persist between invocations of the compiler. The use of a functional interface eliminates the need for the compiler to worry about the details of the representation. The info database also handles the need to multiple "global" environments, which makes it possible to change something in the compiler without trashing the running Lisp environment.

The info database contains arbitrary lisp values, addressed by a combination of name, class and type. The Name is an EQUAL-thing which is the name of the thing that we are recording information about. Class is the kind of object involved: typical classes are Function, Variable, Type. A type names a particular piece of information within a given class. Class and Type are symbols, but are compared with STRING=.

Chapter 37

The IR1 Interpreter

May be worth having a byte-code representation for interpreted code. This way, an entire system could be compiled into byte-code for debugging (the "check-out" compiler?).

Given our current inclination for using a stack machine to interpret IR1, it would be straightforward to layer a byte-code interpreter on top of this.

Instead of having no interpreter, or a more-or-less conventional interpreter, or byte-code interpreter, how about directly executing IR1?

We run through the IR1 passes, possibly skipping optional ones, until we get through environment analysis. Then we run a post-pass that annotates IR1 with information about where values are kept, i.e. the stack slot.

We can lazily convert functions by having FUNCTION make an interpreted function object that holds the code (really a closure over the interpreter). The first time that we try to call the function, we do the conversion and processing. Also, we can easily keep track of which interpreted functions we have expanded macros in, so that macro redefinition automatically invalidates the old expansion, causing lazy reconversion.

Probably the interpreter will want to represent MVs by a recognizable structure that is always heap-allocated. This way, we can punt the stack issues involved in trying to spread MVs. So a continuation value can always be kept in a single cell.

The compiler can have some special frobs for making the interpreter efficient, such as a call operation that extracts arguments from the stack slots designated by a continuation list. Perhaps

```
    (values-mapcar fun . lists)
<==>
    (values-list (mapcar fun . lists))
```

This would be used with MV-CALL.

This scheme seems to provide nearly all of the advantages of both the compiler and conventional interpretation. The only significant disadvantage with respect to a conventional interpreter is that there is the one-time overhead of conversion, but doing this lazily should make this quite acceptable.

With respect to a conventional interpreter, we have major advantages: + Full syntax checking: safety comparable to compiled code. + Semantics similar to compiled code due to code sharing. Similar diagnostic messages, etc. Reduction of error-prone code duplication. + Potential for full type checking according to declarations (would require running IR1 optimize?) + Simplifies debugger interface, since interpreted code can look more like compiled code: source paths, edit definition, etc.

For all non-run-time symbol annotations (anything other than SYMBOL-FUNCTION and SYMBOL-VALUE), we use the compiler's global database. MACRO-FUNCTION will use INFO, rather than vice-versa.

When doing the IR1 phases for the interpreter, we probably want to suppress optimizations that change user-visible function calls: – Don't do local call conversion of any named functions (even lexical ones). This is so that a call will appear on the stack that looks like the call in the original source. The keyword and optional argument transformations done by local call mangle things quite a bit. Also, note local-call converting prevents unreferenced arguments from being deleted, which is another non-obvious transformation. – Don't run source-transforms, IR1 transforms and IR1 optimizers. This way, TRACE and BACKTRACE will show calls with the original arguments, rather than the "optimized" form, etc. Also, for the interpreter it will actually be faster to call the original function (which is compiled) than to "inline expand" it. Also, this allows implementation-dependent transforms to expand into

There are some problems with stepping, due to our non-syntactic IR1 representation. The source path information is the key that makes this conceivable. We can skip over the stepping of a subform by quietly evaluating nodes whose source path lies within the form being skipped.

One problem with determining what value has been returned by a form. With a function call, it is theoretically possible to precisely determine this, since if we complete evaluation of the arguments, then we arrive at the Combination node whose value is synonymous with the value of the form. We can even detect this case, since the Node-Source will be EQ to the form. And we can also detect when we unwind out of the evaluation, since we will leave the form without having ever reached this node.

But with macros and special-forms, there is no node whose value is the value of the form, and no node whose source is the macro call or special form. We can still detect when we leave the form, but we can't be sure whether this was a normal evaluation result or an explicit RETURN-FROM.

But does this really matter? It seems that we can print the value returned (if any), then just print the next form to step. In the rare case where we did unwind, the user should be able to figure it out.

[We can look at this as a side-effect of CPS: there isn't any difference between a "normal" return and a non-local one.]

[Note that in any control transfer (normal or otherwise), the stepper may need to unwind out of an arbitrary number of levels of stepping. This is because a form in a TR position may yield its to a node arbitrarily far out.]

Another problem is with deciding what form is being stepped. When we start evaluating a node, we dive into code that is nested somewhere down inside that form. So we actually have to do a loop of asking questions before we do any evaluation. But what do we ask about?

If we ask about the outermost enclosing form that is a subform of the last form that the user said to execute, then we might offer a form that isn't really evaluated, such as a LET binding list.

But once again, is this really a problem? It is certainly different from a conventional stepper, but a pretty good argument could be made that it is superior. Haven't you ever wanted to skip the evaluation of all the LET bindings, but not the body? Wouldn't it be useful to be able to skip the DO step forms?

All of this assumes that nobody ever wants to step through the guts of a macroexpansion. This seems reasonable, since steppers are for weenies, and weenies don't define macros (hence don't debug them). But there are probably some weenies who don't know that they shouldn't be writing macros.

We could handle this by finding the "source paths" in the expansion of each macro by sticking some special frob in the source path marking the place where the expansion happened. When we hit code again that is in the source, then we revert to the normal source path. Something along these lines might be a good idea anyway (for compiler error messages, for example).

The source path hack isn't guaranteed to work quite so well in generated code, though, since macros return stuff that isn't freshly consed. But we could probably arrange to win as long as any given expansion doesn't return two EQ forms.

It might be nice to have a command that skipped stepping of the form, but printed the results of each outermost enclosed evaluated subform, i.e. if you used this on the DO step-list, it would print the result of each new-value form. I think this is implementable. I guess what you would do is print each value delivered to a DEST whose source form is the current or an enclosing form. Along with the value, you would print the source form for the node that is computing the value.

The stepper can also have a "back" command that "unskips" or "unsteps". This would allow the evaluation of forms that are pure (modulo lexical variable setting) to be undone. This is useful, since in stepping it is common that you skip a form that you shouldn't have, or get confused and want to restart at some earlier point.

What we would do is remember the current node and the values of all local variables. heap before doing each step or skip action. We can then back up the state of all lexical variables and the "program counter". To make this work right with set closure variables, we would copy the cell's value, rather than the value cell itself.

[To be fair, note that this could easily be done with our current interpreter: the stepper could copy the environment alists.]

We can't back up the "program counter" when a control transfer leaves the current function, since this state is implicitly represented in the interpreter's state, and is discarded when we exit. We probably want to ask for confirmation before leaving the function to give users a chance to "unskip" the forms in a TR position.

Another question is whether the conventional stepper is really a good thing to imitate... How about an editor-based mouse-driven interface? Instead of "skipping" and "stepping", you would just designate the next form that you wanted to stop at. Instead of displaying return values, you replace the source text with the printed representation of the value.

It would show the “program counter” by highlighting the *innermost* form that we are about to evaluate, i.e. the source form for the node that we are stopped at. It would probably also be useful to display the start of the form that was used to designate the next stopping point, although I guess this could be implied by the mouse position.

Such an interface would be a little harder to implement than a dumb stepper, but it would be much easier to use. [It would be impossible for an evalhook stepper to do this.]

37.1 Use of %PRIMITIVE

Note: `%PRIMITIVE` can only be used in compiled code. It is a trapdoor into the compiler, not a general syntax for accessing “sub-primitives”. It’s main use is in implementation-dependent compiler transforms. It saves us the effort of defining a “phony function” (that is not really defined), and also allows direct communication with the code generator through codegen-info arguments.

Some primitives may be exported from the VM so that `%PRIMITIVE` can be used to make it explicit that an escape routine or interpreter stub is assuming an operation is implemented by the compiler.

Chapter 38

Debugger

Two classes of errors are handled by the Lisp debugger. These are synchronous errors caused by something erring in program code and asynchronous errors caused by some external context of execution (clock interrupts, control-c interrupts). Asynchronous errors can often be postponed if they are delivered at an inconvenient time.

Synchronous errors are frequently handled by directly invoking the debugger. However, there are several places where the strategy of jumping into the debugger is not used. In those situations the compiler emits a stylized breakpoint; a breakpoint instruction (usually an INT3) followed by several bytes of argument data. This will cause a trip through the operating system and ultimately the invocation of the C-level SIGTRAP handler which, in turn, interprets the argument bytes following the breakpoint and dispatches to the correct handler. There is a switch statement in "sigtrap_handler" which gives the whole story on what types of errors rely on this mechanism. The most commonly invoked handler is probably "interrupt_internal_error" as it fields such common exceptions as the use of unbound symbols. To familiarize with the context these traps are created in, one can disassemble just about any function and look at the bottom of the disassembly for blocks of error handling code. There will often be "BREAK 10" opcodes followed by several "BYTE" opcodes with the meaning of the arguments in neatly decoded form off in the right-hand column.

The other types of synchronous errors are those errors delivered by the operating system such as FPU traps and SIGSEGVs. The invocation of those signals should be funneled through a C-level trampoline which makes a callback into Lisp passing all of the signal handler arguments. That code is pretty straight forward and the "interrupt_handle_now" function is pretty much where all of the runtime logic is localized.

Handling asynchronous errors and deferred asynchronous errors is a bit more involved...

38.1 Tracing and Breakpoints

Here are a few notes on how tracing of compiled code works.

When a function is traced, a breakpoint instruction is placed at the start of the function, replacing the instruction that was there. (This is a `:function-start` breakpoint.) (This appears to be one instruction after the no-arg parsing entry point.) The breakpoint instruction is, of course, architecture-specific, but it must signal a `trap_Breakpoint` trap.

When the code is run, the breakpoint instruction is executed causing a trap. The trap handler runs `HANDLE-BREAKPOINT` to process it. After doing the appropriate processing, we now need to continue. Of course, since the real instruction has been replaced, we to run the original instruction. This is done by now inserting a *new* breakpoint after the original breakpoint. This breakpoint must be of the type `trap_AfterBreakpoint`. The original instruction is restored and execution continues from there. Then the `trap_AfterBreakpoint` instruction gets executed. The handler for this puts back the original breakpoint, thereby preserving the breakpoint. Then we replace the AfterBreakpoint with the original instruction and continue from there.

That's all pretty straightforward in concept.

When tracing, additional information is needed. Breakpoints have the ability to run arbitrary lisp code to process the breakpoint. Tracing uses this feature.

When this breakpoint is reached, `HANDLE-BREAKPOINT` runs the breakpoint hook function. This function figures out where this function would return to and creates a new return area and replaces the original return address with this new address. Thus, when the function returns, it returns to this new location instead of the original.

This new return address is a specially created bogus LRA object. It is a code-component whose body consists of a code template copied from an assembly routine into the body. The assembly routine is the code in `function_end_breakpoint_guts`. This bogus LRA object stores the real LRA for the function, and also an indication if the known-return convention is used for this function.

The bogus LRA object contains a function-end breakpoint (`trap_FunctionEndBreakpoint`). When it's executed the trap handler handles this breakpoint. It figures out where this trap come from and calls `HANDLE-BREAKPOINT` to handle it. `HANDLE-BREAKPOINT` returns and the trap handler arranges it so that this bogus LRA returns to the real LRA.

Thus, we can do something when a Lisp function returns, like printing out the return value for the function for tracing.

There are lots of internal details left out here, but gives a short overview of how this works. For more info, look at `code/debug-int.lisp` and `lisp/breakpoint.c`, and, of course, the various `<foo>-arch.c` files.

Chapter 39

Debugger Information

Although the compiler's great freedom in choice of function call conventions and variable representations has major efficiency advantages, it also has unfortunate consequences for the debugger. The debug information that we need is even more elaborate than for conventional "compiled" languages, since we cannot even do a simple backtrace without some debug information. However, once having gone this far, it is not that difficult to go the extra distance, and provide full source level debugging of compiled code.

Full debug information has a substantial space penalty, so we allow different levels of debug information to be specified. In the extreme case, we can totally omit debug information.

39.1 The Debug-Info Structure

The Debug-Info structure directly represents information about the source code, and points to other structures that describe the layout of run-time data structures.

Make some sort of minimal debug-info format that would support at least the common cases of level 1 (since that is what we would release), and perhaps level 0. Actually, it seems it wouldn't be hard to crunch nearly all of the debug-function structure and debug-info function map into a single byte-vector. We could have an uncrunch function that restored the current format. This would be used by the debugger, and also could be used by purify to delete parts of the debug-info even when the compiler dumps it in crunched form. [Note that this isn't terribly important if purify is smart about debug-info...]

Compiled source map representation:

[### store in debug-function PC at which env is properly initialized, i.e. args (and return-pc, etc.) in internal locations. This is where a :function-start breakpoint would break.]

[### Note that that we can easily cache the form-number =¿ source-path or form-number =¿ form translation using a vector indexed by form numbers that we build during a walk.]

Instead of using source paths in the debug-info, use "form numbers". The form number of a form is the number of forms that we walk to reach that form when doing a pre-order walk of the source form. [Might want to use a post-order walk, as that would more closely approximate evaluation order.]

We probably want to continue using source-paths in the compiler, since they are quick to compute and to get you to a particular form. [### But actually, I guess we don't have to precompute the source paths and annotate nodes with them: instead we could annotate the nodes with the actual original source form. Then if we wanted to find the location of that form, we could walk the root source form, looking that original form. But we might still need to enter all the forms in a hashtable so that we can tell during IR1 conversion that a given form appeared in the original source.]

Note that form numbers have an interesting property: it is quite efficient to determine whether an arbitrary form is a subform of some other form, since the form number of B will be > than A's number and < A's next sibling's number iff B is a subform of A.

This should be quite useful for doing the `source=>pc` mapping in the debugger, since that problem reduces to finding the subset of the known locations that are for subforms of the specified form.

Assume a byte vector with a standard variable-length integer format, something like this:

```
0..253 => the integer
```

```
    254 => read next two bytes for integer
    255 => read next four bytes for integer
```

Then a compiled debug block is just a sequence of variable-length integers in a particular order, something like this:

```
    number of successors
    ...offsets of each successor in the function's blocks vector...
    first PC
    [offset of first top-level form (in forms) (only if not component default)]
    form number of first source form
    first live mask (length in bytes determined by number of VARIABLES)
    ...more <PC, top-level form offset, form-number, live-set> tuples...
```

We determine the number of locations recorded in a block by finding the start of the next compiled debug block in the blocks vector.

[### Actually, only need 2 bits for number of successors 0,1,2. We might want to use other bits in the first byte to indicate the kind of location.] [### We could support local packing by having a general concept of "alternate locations" instead of just regular and save locations. The location would have a bit indicating that there are alternate locations, in which case we read the number of alternate locations and then that many more SC-OFFSETs. In the debug-block, we would have a second bit mask with bits set for TNs that are in an alternate location. We then read a number for each such TN, with the value being interpreted as an index into the Location's alternate locations.]

It looks like using structures for the compiled-location-info is too bulky. Instead we need some packed binary representation.

First, let's represent an SC/offset pair with an "SC-Offset", which is an integer with the SC in the low 5 bits and the offset in the remaining bits:

```
    ----------------------------------------------------
    | Offset (as many bits as necessary) | SC (5 bits) |
    ----------------------------------------------------
```

Probably the result should be constrained to fit in a fixnum, since it will be more efficient and gives more than enough possible offsets.

We can then represent a compiled location like this:

```
    single byte of boolean flags:
uninterned name
packaged name
environment-live
has distinct save location
        has ID (name not unique in this fun)
    name length in bytes (as var-length integer)
    ...name bytes...
    [if packaged, var-length integer that is package name length]
     ...package name bytes...]
    [If has ID, ID as var-length integer]
    SC-Offset of primary location (as var-length integer)
    [If has save SC, SC-Offset of save location (as var-length integer)]
```

But for a whizzy breakpoint facility, we would need a good `source=>code` map. Dumping a complete `code=>source map` might be as good a way as any to represent this, due to the one-to-many relationship between source and code locations.

We might be able to get away with just storing the source locations for the beginnings of blocks and maintaining a mapping from code ranges to blocks. This would be fine both for the profiler and for the "where am I running now" indication. Users might also be convinced that it was most interesting to break at block starts, but I don't really know how easily people could develop an understanding of basic blocks.

It could also be a bit tricky to map an arbitrary user-designated source location to some "closest" source location actually in the debug info. This problem probably exists to some degree even with a full source map, since some forms will never appear as the source of any node. It seems you might have to negotiate with the user. He would mouse something, and then you would highlight some source form that has a common prefix (i.e. is a prefix of the user path, or vice-versa.) If they aren't happy with the result, they could try something else. In some cases, the designated path might be a prefix of several paths. This ambiguity might be resolved by picking the shortest path or letting the user choose.

At the primitive level, I guess what this means is that the structure of source locations (i.e. source paths) must be known, and the `source=>code` operation should return a list of `<source,code>` pairs, rather than just a list of code locations. This allows the debugger to resolve the ambiguity however it wants.

I guess the formal definition of which source paths we would return is:

> All source paths in the debug info that have a maximal common prefix with the specified path. i.e. if several paths have the complete specified path as a prefix, we return them all. Otherwise, all paths with an equally large common prefix are returned: if the path with the most in common matches only the first three elements, then we return all paths that match in the first three elements. As a degenerate case (which probably shouldn't happen), if there is no path with anything in common, then we return *all* of the paths.

In the DEBUG-SOURCE structure we may ultimately want a vector of the start positions of each source form, since that would make it easier for the debugger to locate the source. It could just open the file, FILE-POSITION to the form, do a READ, then loop down the source path. Of course, it could read each form starting from the beginning, but that might be too slow.

Do XEPs really need Debug-Functions? The only time that we will commonly end up in the debugger on an XEP is when an argument type check fails. But I suppose it would be nice to be able to print the arguments passed...

Note that assembler-level code motion such as pipeline reorganization can cause problems with our PC maps. The assembler needs to know that debug info markers are different from real labels anyway, so I suppose it could inhibit motion across debug markers conditional on policy. It seems unworthwhile to remember the node for each individual instruction.

For tracing block-compiled calls:

```
Info about return value passing locations?
Info about where all the returns are?
```

We definitely need the return-value passing locations for debug-return. The question is what the interface should be. We don't really want to have a visible debug-function-return-locations operation, since there are various value passing conventions, and we want to paper over the differences.

Probably should be a compiler option to initialize stack frame to a special uninitialized object (some random immediate type). This would aid debugging, and would also help GC problems. For the latter reason especially, this should be locally-turn-onable (off of policy? the new debug-info quality?).

What about the interface between the evaluator and the debugger? (i.e. what happens on an error, etc.) Compiler error handling should be integrated with run-time error handling. Ideally the error messages should look the same. Practically, in some cases the run-time errors will have less information. But the error should look the same to the debugger (or at least similar).

39.1.1 Debugger Interface

How does the debugger interface to the "evaluator" (where the evaluator means all of native code, byte-code and interpreted IR1)? It seems that it would be much more straightforward to have a consistent user interface to debugging all code representations if there was a uniform debugger interface to the underlying stuff, and vice-versa.

Of course, some operations might not be supported by some representations, etc. For example, fine-control stepping might not be available in native code. In other cases, we might reduce an operation to the lowest common denominator, for example fetching lexical variables by string and admitting the possibility of ambiguous matches. [Actually, it would probably be a good idea to store the package if we are going to allow variables to be closed over.]

Some objects we would need:

```
Location:
The constant information about the place where a value is stored,
        everything but which particular frame it is in.  Operations:
        location name, type, etc.
        location-value frame location (setf'able)
monitor-location location function
            Function is called whenever location is set with the location,
            frame and old value.  If active values aren't supported, then we
            dummy the effect using breakpoints, in which case the change won't
            be noticed until the end of the block (and intermediate changes
            will be lost.)
debug info:
        All the debug information for a component.
Frame:
frame-changed-locations frame => location*
            Return a list of the locations in frame that were changed since the
            last time this function was called.  Or something.  This is for
            displaying interesting state changes at breakpoints.
save-frame-state frame => frame-state
restore-frame-state frame frame-state
    These operations allow the debugger to back up evaluation, modulo
    side-effects and non-local control transfers.  This copies and
    restores all variables, temporaries, etc, local to the frame, and
    also the current PC and dynamic environment (current catch, etc.)

    At the time of the save, the frame must be for the running function
    (not waiting for a call to return.)  When we restore, the frame
    becomes current again, effectively exiting from any frames on top.
    (Of course, frame must not already be exited.)

Thread:
        Representation of which stack to use, etc.
Block:
        What successors the block has, what calls there are in the block.
        (Don't need to know where calls are as long as we know called function,
        since can breakpoint at the function.)  Whether code in this block is
        wildly out of order due to being the result of loop-invariant
        optimization, etc.  Operations:
        block-successors block => code-location*
        block-forms block => (source-location code-location)*
            Return the corresponding source locations and code locations for
            all forms (and form fragments) in the block.
```

39.1.2 Variable maps

There are about five things that the debugger might want to know about a variable:

Name Although a lexical variable's name is "really" a symbol (package and all), in practice it doesn't seem worthwhile to require all the symbols for local variable names to be retained. There is much less VM and GC overhead for a constant string than for a symbol. (Also it is useful to be able to access gensyms in the debugger, even though they are theoretically ineffable).

ID Which variable with the specified name is this? It is possible to have multiple variables with the same name in a given function. The ID is something that makes Name unique, probably a small integer. When variables aren't unique, we could make this be part of the name, e.g. "FOO#1", "FOO#2". But there are advantages to keeping this separate, since in many cases lifetime information can be used to disambiguate, making qualification unnecessary.

SC When unboxed representations are in use, we must have type information to properly read and write a location. We only need to know the SC for this, which would be amenable to a space-saving numeric encoding.

Location Simple: the offset in SC. [Actually, we need the save location too.]

Lifetime In what parts of the program does this variable hold a meaningful value? It seems prohibitive to record precise lifetime information, both in space and compiler effort, so we will have to settle for some sort of approximation.

The finest granularity at which it is easy to determine liveness is the block: we can regard the variable lifetime as the set of blocks that the variable is live in. Of course, the variable may be dead (and thus contain meaningless garbage) during arbitrarily large portions of the block.

Note that this subsumes the notion of which function a variable belongs to. A given block is only in one function, so the function is implicit.

The variable map should represent this information space-efficiently and with adequate computational efficiency.

The SC and ID can be represented as small integers. Although the ID can in principle be arbitrarily large, it should be <100 in practice. The location can be represented by just the offset (a moderately small integer), since the SB is implicit in the SC.

The lifetime info can be represented either as a bit-vector indexed by block numbers, or by a list of block numbers. Which is more compact depends both on the size of the component and on the number of blocks the variable is live in. In the limit of large component size, the sparse representation will be more compact, but it isn't clear where this crossover occurs. Of course, it would be possible to use both representations, choosing the more compact one on a per-variable basis. Another interesting special case is when the variable is live in only one block: this may be common enough to be worth picking off, although it is probably rarer for named variables than for TNs in general.

If we dump the type, then a normal list-style type descriptor is fine: the space overhead is small, since the shareability is high.

We could probably save some space by cleverly representing the var-info as parallel vectors of different types, but this would be more painful in use. It seems better to just use a structure, encoding the unboxed fields in a fixnum. This way, we can pass around the structure in the debugger, perhaps even exporting it from the low-level debugger interface.

[### We need the save location too. This probably means that we need two slots of bits, since we need the save offset and save SC. Actually, we could let the save SC be implied by the normal SC, since at least currently, we always choose the same save SC for a given SC. But even so, we probably can't fit all that stuff in one fixnum without squeezing a lot, so we might as well split and record both SCs.

In a localized packing scheme, we would have to dump a different var-info whenever either the main location or the save location changes. As a practical matter, the save location is less likely to change than the main location, and should never change without the main location changing.

One can conceive of localized packing schemes that do saving as a special case of localized packing. If we did this, then the concept of a save location might be eliminated, but this would require major changes in the IR2 representation for call and/or lifetime info. Probably we will want saving to continue to be somewhat magical.]

How about:

```
(defstruct var-info
  ;;
  ;; This variable's name. (symbol-name of the symbol)
  (name nil :type simple-string)
  ;;
  ;; The SC, ID and offset, encoded as bit-fields.
  (bits nil :type fixnum)
  ;;
  ;; The set of blocks this variable is live in.  If a bit-vector, then it has
  ;; a 1 when indexed by the number of a block that it is live in.  If an
  ;; I-vector, then it lists the live block numbers.  If a fixnum, then that is
  ;; the number of the sole live block.
  (lifetime nil :type (or vector fixnum))
  ;;
  ;; The variable's type, represented as list-style type descriptor.
  type)
```

Then the debug-info holds a simple-vector of all the var-info structures for that component. We might as well make it sorted alphabetically by name, so that we can binary-search to find the variable corresponding to a particular name.

We need to be able to translate PCs to block numbers. This can be done by an I-Vector in the component that contains the start location of each block. The block number is the index at which we find the correct PC range. This requires that we use an emit-order block numbering distinct from the IR2-Block-Number, but that isn't any big deal. This seems space-expensive, but it isn't too bad, since it would only be a fraction of the code size if the average block length is a few words or more.

An advantage of our per-block lifetime representation is that it directly supports keeping a variable in different locations when in different blocks, i.e. multi-location packing. We use a different var-info for each different packing, since the SC and offset are potentially different. The Name and ID are the same, representing the fact that it is the same variable. It is here that the ID is most significant, since the debugger could otherwise make same-name variables unique all by itself.

39.1.3 Stack parsing

[### Probably not worth trying to make the stack parseable from the bottom up. There are too many complications when we start having variable sized stuff on the stack. It seems more profitable to work on making top-down parsing robust. Since we are now planning to wire the bottom-up linkage info, scanning from the bottom to find the top frame shouldn't be too inefficient, even when there was a runaway recursion. If we somehow jump into hyperspace, then the debugger may get confused, but we can debug this sort of low-level system lossage using ADB.]

There are currently three relevant context pointers:

- The PC. The current PC is wired (implicit in the machine). A saved PC (RETURN-PC) may be anywhere in the current frame.
- The current stack context (CONT). The current CONT is wired. A saved CONT (OLD-CONT) may be anywhere in the current frame.
- The current code object (ENV). The current ENV is wired. When saved, this is extra-difficult to locate, since it is saved by the caller, and is thus at an unknown offset in OLD-CONT, rather than anywhere in the current frame.

We must have all of these to parse the stack.

With the proposed Debug-Function, we parse the stack (starting at the top) like this:

1. Use ENV to locate the current Debug-Info
2. Use the Debug-Info and PC to determine the current Debug-Function.
3. Use the Debug-Function to find the OLD-CONT and RETURN-PC.
4. Find the old ENV by searching up the stack for a saved code object containing the RETURN-PC.
5. Assign old ENV to ENV, OLD-CONT to CONT, RETURN-PC to PC and goto 1.

If we changed the function representation so that the code and environment were a single object, then the location of the old ENV would be simplified. But we still need to represent ENV as separate from PC, since interrupts and errors can happen when the current PC isn't positioned at a valid return PC.

It seems like it might be a good idea to save OLD-CONT, RETURN-PC and ENV at the beginning of the frame (before any stack arguments). Then we wouldn't have to search to locate ENV, and we also have a hope of parsing the stack even if it is damaged. As long as we can locate the start of some frame, we can trace the stack above that frame. We can recognize a probable frame start by scanning the stack for a code object (presumably a saved ENV).

Probably we want some fairly general mechanism for specifying that a TN should be considered to be live for the duration of a specified environment. It would be somewhat easier to specify that the TN is live for all time, but this would become very space-inefficient in large block compilations.

This mechanism could be quite useful for other debugger-related things. For example, when debuggability is important, we could make the TNs holding arguments live for the entire environment. This would guarantee that a backtrace would always get the right value (modulo setqs).

Note that in this context, "environment" means the Environment structure (one per non-let function). At least according to current plans, even when we do inter-routine register allocation, the different functions will have different environments: we just "equate" the environments. So the number of live per-environment TNs is bounded by the size of a "function", and doesn't blow up in block compilation.

The implementation is simple: per-environment TNs are flagged by the :Environment kind. :Environment TNs are treated the same as :Normal TNs by everyone except for lifetime/conflict analysis. An environment's TNs are also stashed in a list in the IR2-Environment structure. During the conflict analysis post-pass, we look at each block's environment, and make all the environment's TNs always-live in that block.

We can implement the "fixed save location" concept needed for lazy frame creation by allocating the save TNs as wired TNs at IR2 conversion time. We would use the new "environment lifetime" concept to specify the lifetimes of the save locations. There isn't any run-time overhead if we never get around to using the save TNs. [Pack would also have to notice TNs with pre-allocated save TNs, packing the original TN in the stack location if its FSC is the stack.]

We want a standard (recognizable) format for an "escape" frame. We must make an escape frame whenever we start running another function without the current function getting a chance to save its registers. This may be due either to a truly asynchronous event such as a software interrupt, or due to an "escape" from a miscop. An escape frame marks a brief conversion to a callee-saves convention.

Whenever a miscop saves registers, it should make an escape frame. This ensures that the "current" register contents can always be located by the debugger. In this case, it may be desirable to be able to indicate that only partial saving has been done. For example, we don't want to have to save all the FP registers just so that we can use a couple extra general registers.

When the debugger see an escape frame, it knows that register values are located in the escape frame's "register save" area, rather than in the normal save locations.

It would be nice if there was a better solution to this internal error concept. One problem is that it seems there is a substantial space penalty for emitting all that error code, especially now that we don't share error code between errors because we want to preserve the source context in the PC. But this probably isn't really all that bad when considered as a fraction of the code. For example, the check part of a type check is 12 bytes, whereas the error part is usually only 6. In this case, we could never reduce the space overhead for type checks by more than 1/3, thus the total code size reduction would be small. This will be made even less important when we do type check optimizations to reduce the number of type checks.

Probably we should stick to the same general internal error mechanism, but make it interact with the debugger better by allocating linkage registers and allowing proceedable errors. We could support shared error calls and non-proceedable errors when space is more important than debuggability, but this is probably more complexity than is worthwhile.

We jump or trap to a routine that saves the context (allocating at most the return PC register). We then encode the error and context in the code immediately following the jump/trap. (On the MIPS, the error code can be encoded in the trap itself.) The error arguments would be encoded as SC-offsets relative to the saved context. This could solve both the arg-trashing problem and save space, since we could encode the SC-offsets more tersely than the corresponding move instructions.

Chapter 40

Object Format

40.1 Tagging

The following is a key of the three bit low-tagging scheme:

000 even fixnum

001 function pointer

010 even other-immediate (header-words, characters, symbol-value trap value, etc.)

011 list pointer

100 odd fixnum

101 structure pointer

110 odd other immediate

111 other-pointer to data-blocks (other than conses, structures, and functions)

This tagging scheme forces a dual-word alignment of data-blocks on the heap, but this can be pretty negligible:

- RATIOS and COMPLEX must have a header-word anyway since they are not a major type. This wastes one word for these infrequent data-blocks since they require two words for the data.
- BIGNUMS must have a header-word and probably contain only one other word anyway, so we probably don't waste any words here. Most bignums just barely overflow fixnums, that is by a bit or two.
- Single and double FLOATS? no waste, or one word wasted
- SYMBOLS have a pad slot (current called the setf function, but unused.)

Everything else is vector-like including code, so these probably take up so many words that one extra one doesn't matter.

40.2 GC Comments

Data-Blocks comprise only descriptors, or they contain immediate data and raw bits interpreted by the system. GC must skip the latter when scanning the heap, so it does not look at a word of raw bits and interpret it as a pointer descriptor. These data-blocks require headers for GC as well as for operations that need to know how to interpret the raw bits. When GC is scanning, and it sees a header-word, then it can determine how to skip that data-block if necessary. Header-Words are tagged as other-immediates. See "Other-Immediates", section 40.5 and "Data-Blocks and Header-Words", section 40.6 for comments on distinguishing header-words from other-immediate data. This distinction is necessary since we scan through data-blocks containing only descriptors just as we scan through the heap looking for header-words introducing data-blocks.

Data-Blocks containing only descriptors do not require header-words for GC since the entire data-block can be scanned by GC a word at a time, taking whatever action is necessary or appropriate for the data in that slot. For example, a cons is referenced by a descriptor with a specific tag, and the system always knows the size of this data-block. When GC encounters a pointer to a cons, it can transport it into the new space, and when scanning, it can simply scan the two words manifesting the cons interpreting each word as a descriptor. Actually there is no cons tag, but a list tag, so we make sure the cons is not nil when appropriate. A header may still be desired if the pointer to the data-block does not contain enough information to adequately maintain the data-block. An example of this is a simple-vector containing only descriptor slots, and we attach a header-word because the descriptor pointing to the vector lacks necessary information – the type of the vector's elements, its length, etc.

There is no need for a major tag for GC forwarding pointers. Since the tag bits are in the low end of the word, a range check on the start and end of old space tells you if you need to move the thing. This is all GC overhead.

40.3 Structures

A structure descriptor has the structure lowtag type code, making `structurep` a fast operation. A structure data-block has the following format:

```
-------------------------------------------------------
|   length (24 bits) | Structure header type (8 bits) |
-------------------------------------------------------
|   structure type name (a symbol)                    |
-------------------------------------------------------
|   structure slot 0                                  |
-------------------------------------------------------
|   ... structure slot length - 2                     |
-------------------------------------------------------
```

The header word contains the structure length, which is the number of words (other than the header word.) The length is always at least one, since the first word of the structure data is the structure type name.

40.4 Fixnums

A fixnum has one of the following formats in 32 bits:

```
-------------------------------------------------------
|        30 bit 2's complement even integer   | 0 0 0 |
-------------------------------------------------------
```

or

```
-------------------------------------------------------
|        30 bit 2's complement odd integer    | 1 0 0 |
-------------------------------------------------------
```

Effectively, there is one tag for immediate integers, two zeros. This buys one more bit for fixnums, and now when these numbers index into simple-vectors or offset into memory, they point to word boundaries on 32-bit, byte-addressable machines. That is, no shifting need occur to use the number directly as an offset.

This format has another advantage on byte-addressable machines when fixnums are offsets into vector-like data-blocks, including structures. Even though we previously mentioned data-blocks are dual-word aligned, most indexing and slot accessing is word aligned, and so are fixnums with effectively two tag bits.

Two tags also allow better usage of special instructions on some machines that can deal with two low-tag bits but not three.

Since the two bits are zeros, we avoid having to mask them off before using the words for arithmetic, but division and multiplication require special shifting.

40.5 Other-immediates

As for fixnums, there are two different three-bit lowtag codes for other-immediate, allowing 64 other-immediate types:

```
-----------------------------------------------------------------
|   Data (24 bits)        | Type (8 bits with low-tag)   | 1 0 |
-----------------------------------------------------------------
```

The type-code for an other-immediate type is considered to include the two lowtag bits. This supports the concept of a single "type code" namespace for all descriptors, since the normal lowtag codes are disjoint from the other-immediate codes.

For other-pointer objects, the full eight bits of the header type code are used as the type code for that kind of object. This is why we use two lowtag codes for other-immediate types: each other-pointer object needs a distinct other-immediate type to mark its header.

The system uses the other-immediate format for characters, the `symbol-value` unbound trap value, and header-words for data-blocks on the heap. The type codes are laid out to facilitate range checks for common subtypes; for example, all numbers will have contiguous type codes which are distinct from the contiguous array type codes. See section 40.7 for details.

40.6 Data-Blocks and Header-Word Format

Pointers to data-blocks have the following format:

```
-----------------------------------------------------------------
|      Dual-word address of data-block (29 bits)       | 1 1 1 |
-----------------------------------------------------------------
```

The word pointed to by the above descriptor is a header-word, and it has the same format as an other-immediate:

```
-----------------------------------------------------------------
|   Data (24 bits)        | Type (8 bits with low-tag) | 0 1 0 |
-----------------------------------------------------------------
```

This is convenient for scanning the heap when GC'ing, but it does mean that whenever GC encounters an other-immediate word, it has to do a range check on the low byte to see if it is a header-word or just a character (for example). This is easily acceptable performance hit for scanning.

The system interprets the data portion of the header-word for non-vector data-blocks as the word length excluding the header-word. For example, the data field of the header for ratio and complex numbers is two, one word each for the numerator and denominator or for the real and imaginary parts.

For vectors and data-blocks representing Lisp objects stored like vectors, the system (usually) ignores the data portion of the header-word:

```
-----------------------------------------------------------------
| Unused Data (24 bits)   | Type (8 bits with low-tag) | 0 1 0 |
-----------------------------------------------------------------
|           Element Length of Vector (30 bits)           | 0 0 |
-----------------------------------------------------------------
```

Using a separate word allows for much larger vectors, and it allows `length` to simply access a single word without masking or shifting. Similarly, the header for complex arrays and vectors has a second word, following the header-word, the system uses for the fill pointer, so computing the length of any array is the same code sequence.

For normal Lisp vectors, the data portion MUST be zero. For hash tables, a vector is used to store information about the hash key and value, and the data portion is non-zero to indicate to GC that this is the key/value vector for the hash table. GENCGC uses this to determine scavenge the key/value pairs correctly. Cheney GC also uses this to determine if rehashing (for EQ hash tables) is needed.

40.7 Data-Blocks and Other-immediates Typing

These are the other-immediate types. We specify them including all low eight bits, including the other-immediate tag, so we can think of the type bits as one type – not an other-immediate major type and a subtype. Also, fetching a byte and comparing it against a constant is more efficient than wasting even a small amount of time shifting out the other-immediate tag to compare against a five bit constant. (The current values can be obtained from the generated internals.h file.)

```
                                                          HEX
Number   (< 36)
  bignum                                           10     0A
    ratio                                          14     0E
    single-float                                   18     12
    double-float                                   22     16
    double-double-float                            26     1A
    complex                                        30     1E
    (complex single-float)                         34     22
    (complex double-float)                         38     26
    (complex double-double-float)                  42     2A

Array   (<= 46 code 118)
   Simple-Array   (<= 46 code 118)
         simple-array                              46     2E
      Vector  (<= 50 code 118)
         simple-string                             50     32
         simple-bit-vector                         54     36
         simple-vector                             58     3A
         (simple-array (unsigned-byte 2) (*))      62     3E
         (simple-array (unsigned-byte 4) (*))      66     42
         (simple-array (unsigned-byte 8) (*))      70     46
         (simple-array (unsigned-byte 16) (*))     74     4A
         (simple-array (unsigned-byte 32) (*))     78     4E
         (simple-array (signed-byte 8) (*))        82     52
         (simple-array (signed-byte 16) (*))       86     56
         (simple-array (signed-byte 30) (*))       90     5A
         (simple-array (signed-byte 32) (*))       94     5E
         (simple-array single-float (*))           98     62
         (simple-array double-float (*))           102    66
         (simple-array double-double-float (*))    106    6A
         (simple-array (complex single-float) (*)  110    6E
         (simple-array (complex double-float) (*)  114    72
         (simple-array (complex double-double) (*) 118    76
      complex-string                               122    7A
      complex-bit-vector                           126    7E
      (array * (*))   -- general complex vector.   130    82
   complex-array                                   134    86

code-header-type                                   138    8A
function-header-type                               142    8E
closure-header-type                                146    92
funcallable-instance-header-type                   150    96
byte-code-function-header-type                     154    9A
byte-code-closure-header-type                      158    9E
closure-function-header-type                       162    A2
return-pc-header-type (a.k.a LRA)                  166    A6
value-cell-header-type                             170    AA
```

symbol-header-type	174	AE
base-character-type	178	B2
system-area-pointer-type (header type)	182	B6
unbound-marker	186	BA
weak-pointer-type	190	BE
instance-header-type	194	C2
fdefn-type	198	C6
scavenger-hook-type	202	CA

40.8 Strings

All strings in the system are C-null terminated. This saves copying the bytes when calling out to C. The only time this wastes memory is when the string contains a multiple of eight characters, and then the system allocates two more words (since Lisp objects are dual-word aligned) to hold the C-null byte. Since the system will make heavy use of C routines for systems calls and libraries that save reimplementation of higher level operating system functionality (such as pathname resolution or current directory computation), saving on copying strings for C should make C call out more efficient.

The length word in a string header, see "Data-Blocks and Header-Word Format", section 40.6, counts only the characters truly in the Common Lisp string. Allocation and GC will have to know to handle the extra C-null byte, and GC already has to deal with rounding up various objects to dual-word alignment.

40.9 Symbols and NIL

Symbol data-block has the following format:

```
-------------------------------------------------------
|     5 (data-block words)     | Symbol Type (8 bits) |
-------------------------------------------------------
|                       Value Descriptor              |
-------------------------------------------------------
|  Hash Value (x86/amd64/sparc) Unused (other arch.)  |
-------------------------------------------------------
|                        Property List                |
-------------------------------------------------------
|                          Print Name                 |
-------------------------------------------------------
|                           Package                   |
-------------------------------------------------------
```

All of these slots are self-explanatory given what symbols must do in Common Lisp.

The issues with nil are that we want it to act like a symbol, and we need list operations such as CAR and CDR to be fast on it. CMU Common Lisp solves this by putting nil as the first object in static space, where other global values reside, so it has a known address in the system:

```
-------------------------------------------------------  <-- space
|     6 (data-block words)     |         0            |      start
-------------------------------------------------------
|     0 (data-block words)     | Symbol Type (8 bits) |
-------------------------------------------------------  <-- nil
|                           Value/CAR                 |
-------------------------------------------------------
|                         Hash Value/CDR              |
-------------------------------------------------------
|                         Property List               |
-------------------------------------------------------
|                           Print Name                |
-------------------------------------------------------
|                        Package                      |
-------------------------------------------------------
|                          ...                        |
-------------------------------------------------------
```

In addition, we make the list typed pointer to nil actually point past the header word of the nil symbol data-block. This has usefulness explained below. The value and hash-value of nil are nil. Therefore, any reference to nil used as a list has quick list type checking, and CAR and CDR can go right through the first and second words as if nil were a cons object.

When there is a reference to nil used as a symbol, the system adds offsets to the address the same as it does for any symbol. This works due to a combination of nil pointing past the symbol header-word and the chosen list and other-pointer type tags. The list type tag is four less than the other-pointer type tag, but nil points four additional bytes into its symbol data-block.

40.10 Array Headers

The array-header data-block has the following format:

```
----------------------------------------------------------------
| Header Len (24 bits) = Array Rank +6   | Array Type (8 bits) |
----------------------------------------------------------------
|              Fill Pointer (30 bits)                    | 0 0 |
----------------------------------------------------------------
|              Fill Pointer p (29 bits) -- t or nil    | 1 1 1 |
----------------------------------------------------------------
|              Available Elements (30 bits)              | 0 0 |
----------------------------------------------------------------
|              Data Vector (29 bits)                   | 1 1 1 |
----------------------------------------------------------------
|              Displacement (30 bits)                    | 0 0 |
----------------------------------------------------------------
|              Displacedp (29 bits) -- t or nil        | 1 1 1 |
----------------------------------------------------------------
|              Range of First Index (30 bits)            | 0 0 |
----------------------------------------------------------------
                              .
                              .
                              .
```

The array type in the header-word is one of the eight-bit patterns from "Data-Blocks and Other-immediates Typing", section 40.6, indicating that this is a complex string, complex vector, complex bit-vector, or a multi-dimensional array. The data portion of the other-immediate word is the length of the array header data-block. Due to its format, its length is always six greater than the array's number of dimensions. The following words have the following interpretations and types:

Fill Pointer: This is a fixnum indicating the number of elements in the data vector actually in use. This is the logical length of the array, and it is typically the same value as the next slot. This is the second word, so LENGTH of any array, with or without an array header, is just four bytes off the pointer to it.

Fill Pointer P: This is either T or NIL and indicates whether the array uses the fill-pointer or not.

Available Elements: This is a fixnum indicating the number of elements for which there is space in the data vector. This is greater than or equal to the logical length of the array when it is a vector having a fill pointer.

Data Vector: This is a pointer descriptor referencing the actual data of the array. This a data-block whose first word is a header-word with an array type as described in "Data-Blocks and Header-Word Format", section 40.6 and "Data-Blocks and Other-immediates Typing", section 40.7

Displacement: This is a fixnum added to the computed row-major index for any array. This is typically zero.

Displacedp: This is either t or nil. This is separate from the displacement slot, so most array accesses can simply add in the displacement slot. The rare need to know if an array is displaced costs one extra word in array headers which probably aren't very frequent anyway.

Range of First Index: This is a fixnum indicating the number of elements in the first dimension of the array. Legal index values are zero to one less than this number inclusively. IF the array is zero-dimensional, this slot is non-existent.

... (remaining slots): There is an additional slot in the header for each dimension of the array. These are the same as the Range of First Index slot.

40.11 Bignums

Bignum data-blocks have the following format:

```
-------------------------------------------------------
|      Length (24 bits)        | Bignum Type (8 bits) |
-------------------------------------------------------
|             least significant bits                  |
-------------------------------------------------------
                                .
                                .
                                .
```

The elements contain the two's complement representation of the integer with the least significant bits in the first element or closer to the header. The sign information is in the high end of the last element.

40.12 Code Data-Blocks

A code data-block is the run-time representation of a "component". A component is a connected portion of a program's flow graph that is compiled as a single unit, and it contains code for many functions. Some of these functions are callable from outside of the component, and these are termed "entry points".

Each entry point has an associated user-visible function data-block (of type `function`). The full call convention provides for calling an entry point specified by a function object.

Although all of the function data-blocks for a component's entry points appear to the user as distinct objects, the system keeps all of the code in a single code data-block. The user-visible function object is actually a pointer into the middle of a code data-block. This allows any control transfer within a component to be done using a relative branch.

Besides a function object, there are other kinds of references into the middle of a code data-block. Control transfer into a function also occurs at the return-PC for a call. The system represents a return-PC somewhat similarly to a function, so GC can also recognize a return-PC as a reference to a code data-block. This representation is known as a Lisp Return Address (LRA).

It is incorrect to think of a code data-block as a concatenation of "function data-blocks". Code for a function is not emitted in any particular order with respect to that function's function-header (if any). The code following a function-header may only be a branch to some other location where the function's "real" definition is.

The following are the three kinds of pointers to code data-blocks:

Code pointer (labeled A below): A code pointer is a descriptor, with other-pointer low-tag bits, pointing to the beginning of the code data-block. The code pointer for the currently running function is always kept in a register (CODE). In addition to allowing loading of non-immediate constants, this also serves to represent the currently running function to the debugger.

LRA (labeled B below): The LRA is a descriptor, with other-pointer low-tag bits, pointing to a location for a function call. Note that this location contains no descriptors other than the one word of immediate data, so GC can treat LRA locations the same as instructions.

Function (labeled C below): A function is a descriptor, with function low-tag bits, that is user callable. When a function header is referenced from a closure or from the function header's self-pointer, the pointer has other-pointer low-tag bits, instead of function low-tag bits. This ensures that the internal function data-block associated with a closure appears to be uncallable (although users should never see such an object anyway).

Information about functions that is only useful for entry points is kept in some descriptors following the function's self-pointer descriptor. All of these together with the function's header-word are known as the "function header". GC must be able to locate the function header. We provide for this by chaining together the function headers in a NIL terminated list kept in a known slot in the code data-block.

A code data-block has the following format:

```
A -->
****************************************************************
|  Header-Word count (24 bits)    |   Code-Type (8 bits)       |
----------------------------------------------------------------
|  Number of code words (fixnum tag)                           |
----------------------------------------------------------------
|  Pointer to first function header (other-pointer tag)        |
----------------------------------------------------------------
|  Debug information (structure tag)                           |
----------------------------------------------------------------
|  First constant (a descriptor)                               |
----------------------------------------------------------------
|  ...                                                         |
----------------------------------------------------------------
|  Last constant (and last word of code header)                |
----------------------------------------------------------------
|  Some instructions (non-descriptor)                          |
----------------------------------------------------------------
|     (pad to dual-word boundary if necessary)                 |

B -->
****************************************************************
|  Word offset from code header (24)   |   Return-PC-Type (8)  |
----------------------------------------------------------------
|  First instruction after return                              |
----------------------------------------------------------------
|  ... more code and LRA header-words                          |
----------------------------------------------------------------
|     (pad to dual-word boundary if necessary)                 |

C -->
****************************************************************
|  Offset from code header (24)  |   Function-Header-Type (8)  |
----------------------------------------------------------------
|  x86/amd64/sparc: Address of start of instructions for       |
|  function (non-descriptor)                                   |
|  other architectures:                                        |
|  Self-pointer back to previous word (with other-pointer tag) |
----------------------------------------------------------------
|  Pointer to next function (other-pointer low-tag) or NIL     |
----------------------------------------------------------------
```

```
|  Function name (a string or a symbol)                        |
----------------------------------------------------------------
|  Function debug arglist (a string)                           |
----------------------------------------------------------------
|  Function type (a list-style function type specifier)        |
----------------------------------------------------------------
|  Start of instructions for function (non-descriptor)         |
----------------------------------------------------------------
|  More function headers and instructions and return PCs,      |
|  until we reach the total size of header-words + code        |
|  words.                                                      |
----------------------------------------------------------------
```

The following are detailed slot descriptions:

Code data-block header-word: The immediate data in the code data-block's header-word is the number of leading descriptors in the code data-block, the fixed overhead words plus the number of constants. The first non-descriptor word, some code, appears at this word offset from the header.

Number of code words: The total number of non-header-words in the code data-block. The total word size of the code data-block is the sum of this slot and the immediate header-word data of the previous slot. header-word.

Pointer to first function header: A NIL-terminated list of the function headers for all entry points to this component.

Debug information: The DEBUG-INFO structure describing this component. All information that the debugger wants to get from a running function is kept in this structure. Since there are many functions, the current PC is used to locate the appropriate debug information. The system keeps the debug information separate from the function data-block, since the currently running function may not be an entry point. There is no way to recover the function object for the currently running function, since this data-block may not exist.

First constant ... last constant: These are the constants referenced by the component, if there are any.

LRA header word: The immediate header-word data is the word offset from the enclosing code data-block's header-word to this word. This allows GC and the debugger to easily recover the code data-block from an LRA. The code at the return point restores the current code pointer using a subtract immediate of the offset, which is known at compile time.

Function entry point header-word: The immediate header-word data is the word offset from the enclosing code data-block's header-word to this word. This is the same as for the return-PC header-word.

Address of start of instructions for function: This is implemented on x86, amd64, and sparc only. In a non-closure function, this address allows the call sequence to always indirect through the second word in a user callable function. See section "Closure Format". With a closure, indirecting through the second word also gets you the start of instructions of a function. This pointer is a raw address, not a descriptor.

Self-pointer back to header-word: In a non-closure function, this self-pointer to the previous header-word allows the call sequence to always indirect through the second word in a user callable function. See section "Closure Format". With a closure, indirecting through the second word gets you a function header-word. The system ignores this slot in the function header for a closure, since it has already indirected once, and this slot could be some random thing that causes an error if you jump to it. This pointer has an other-pointer tag instead of a function pointer tag, indicating it is not a user callable Lisp object.

Pointer to next function: This is the next link in the thread of entry point functions found in this component. This value is NIL when the current header is the last entry point in the component.

Function name: This function's name (for printing). If the user defined this function with DEFUN, then this is the defined symbol, otherwise it is a descriptive string.

Function debug arglist: A printed string representing the function's argument list, for human readability. If it is a macroexpansion function, then this is the original DEFMACRO arglist, not the actual expander function arglist.

Function type: A list-style function type specifier representing the argument signature and return types for this function. For example,

```
(function (fixnum fixnum fixnum) fixnum)
```

or

```
(function (string &key (:start unsigned-byte)) string)
```

This information is intended for machine readablilty, such as by the compiler.

40.13 Closure Format

A closure data-block has the following format:

```
----------------------------------------------------------------
|  Word size (24 bits)           |  Closure-Type (8 bits)      |
----------------------------------------------------------------
|  Pointer to function header (other-pointer low-tag)          |
----------------------------------------------------------------
|                               .                              |
|                       Environment information                |
|                               .                              |
----------------------------------------------------------------
```

A closure descriptor has function low-tag bits. This means that a descriptor with function low-tag bits may point to either a function header or to a closure. The idea is that any callable Lisp object has function low-tag bits. Insofar as call is concerned, we make the format of closures and non-closure functions compatible. This is the reason for the self-pointer in a function header. Whenever you have a callable object, you just jump through the second word, offset some bytes, and go.

40.14 Function call

Due to alignment requirements and low-tag codes, it is not possible to use a hardware call instruction to compute the LRA. Instead the LRA for a call is computed by doing an add-immediate to the start of the code data-block.

An advantage of using a single data-block to represent both the descriptor and non-descriptor parts of a function is that both can be represented by a single pointer. This reduces the number of memory accesses that have to be done in a full call. For example, since the constant pool is implicit in an LRA, a call need only save the LRA, rather than saving both the return PC and the constant pool.

40.15 Memory Layout

CMUCL has four spaces, read-only, static, dynamic-0, and dynamic-1. Read-only contains objects that the system never modifies, moves, or reclaims. Static space contains some global objects necessary for the system's runtime or performance (since they are located at a known offset at a known address), and the system never moves or reclaims these. However, GC does need to scan static space for references to moved objects. Dynamic-0 and dynamic-1 are the two heap areas for stop-and-copy GC algorithms.

What global objects are at the head of static space???

```
NIL
```

```
eval::*top-of-stack*
lisp::*current-catch-block*
lisp::*current-unwind-protect*
FLAGS (RT only)
BSP (RT only)
HEAP (RT only)
```

In addition to the above spaces, the system has a control stack, binding stack, and a number stack. The binding stack contains pairs of descriptors, a symbol and its previous value. The number stack is the same as the C stack, and the system uses it for non-Lisp objects such as raw system pointers, saving non-Lisp registers, parts of bignum computations, etc.

40.16 System Pointers

The system pointers reference raw allocated memory, data returned by foreign function calls, etc. The system uses these when you need a pointer to a non-Lisp block of memory, using an other-pointer. This provides the greatest flexibility by relieving contraints placed by having more direct references that require descriptor type tags.

A system area pointer data-block has the following format:

```
-------------------------------------------------------
|     1 (data-block words)        | SAP Type (8 bits) |
-------------------------------------------------------
|             system area pointer                     |
-------------------------------------------------------
```

"SAP" means "system area pointer", and much of our code contains this naming scheme. We don't currently restrict system pointers to one area of memory, but if they do point onto the heap, it is up to the user to prevent being screwed by GC or whatever.

40.17 Weak Pointers

A weak-pointer data-block has the following format:

```
-------------------------------------------------------
|  4 (data-block words) |  Weak pointer Type (8 bits) |
-------------------------------------------------------
|                 weak-pointer-value                  |
-------------------------------------------------------
|                 weak-pointer-broken                 |
-------------------------------------------------------
|                 mark-bit (T or NIL)                 |
-------------------------------------------------------
|                   next                              |
-------------------------------------------------------
```

The mark-bit is used when gencgc is available. It's used to note if this weak pointer has been visited before so that scavenging weak-pointers isn't an $O(n^2)$ process.

The last slot is an internal slot used by the C runtime to chain all the weak pointers together for GC.

Chapter 41

Memory Management

41.1 Stacks and Globals

41.2 Heap Layout

41.3 Garbage Collection

Chapter 42

Interface to C and Assembler

42.1 Linkage Table

The linkage table feature is based on how dynamic libraries dispatch. A table of functions is used which is filled in with the appropriate code to jump to the correct address.

For CMUCL, this table is stored at target-foreign-linkage-space-start. Each entry is target-foreign-linkage-entry-size bytes long.

At startup, the table is initialized with default values in os_foreign_linkage_init. On x86 platforms, the first entry is code to call the routine resolve_linkage_tramp. All other entries jump to the first entry. The function resolve_linkage_tramp looks at where it was called from to figure out which entry in the table was used. It calls lazy_resolve_linkage with the address of the linkage entry. This routine then fills in the appropriate linkage entry with code to jump to where the real routine is located, and returns the address of the entry. On return, resolve_linkage_tramp then just jumps to the returned address to call the desired function. On all subsequent calls, the entry no longer points to resolve_linkage_tramp but to the real function.

This describes how function calls are made. For foreign data, lazy_resolve_linkage stuffs the address of the actual foreign data into the linkage table. The lisp code then just loads the address from there to get the actual address of the foreign data.

For sparc, the linkage table is slightly different. The first entry is the entry for call_into_c so we never have to look this up. All other entries are for resolve_linkage_tramp. This has the advantage that resolve_linkage_tramp can be much simpler since all calls to foreign code go through call_into_c anyway, and that means all live Lisp registers have already been saved. Also, to make life simpler, we lie about closure_tramp and undefined_tramp in the Lisp code. These are really functions, but we treat them as foreign data since these two routines are only used as addresses in the Lisp code to stuff into a lisp function header.

On the Lisp side, there are two supporting data structures for the linkage table: *linkage-table-data* and *foreign-linkage-symbols*. The latter is a hash table whose key is the foreign symbol (a string) and whose value is an index into *linkage-table-data*.

linkage-table-data is a vector with an unlispy layout. Each entry has 3 parts:

- symbol name
- type, a fixnum, 1 = code, 2 = data
- library list - the library list at the time the symbol is registered.

Whenever a new foreign symbol is defined, a new *linkage-table-data* entry is created. *foreign-linkage-symbols* is updated with the symbol and the entry number into *linkage-table-data*.

The *linkage-table-data* is accessed from C (hence the unlispy layout), to figure out the symbol name and the type so that the address of the symbol can be determined. The type tells the C code how to fill in the entry in the linkage-table itself.

Chapter 43

Low-level debugging

Chapter 44

Core File Format

Chapter 45

Fasload File Format

45.1 General

The purpose of Fasload files is to allow concise storage and rapid loading of Lisp data, particularly function definitions. The intent is that loading a Fasload file has the same effect as loading the source file from which the Fasload file was compiled, but accomplishes the tasks more efficiently. One noticeable difference, of course, is that function definitions may be in compiled form rather than S-expression form. Another is that Fasload files may specify in what parts of memory the Lisp data should be allocated. For example, constant lists used by compiled code may be regarded as read-only.

In some Lisp implementations, Fasload file formats are designed to allow sharing of code parts of the file, possibly by direct mapping of pages of the file into the address space of a process. This technique produces great performance improvements in a paged time-sharing system. Since the Mach project is to produce a distributed personal-computer network system rather than a time-sharing system, efficiencies of this type are explicitly *not* a goal for the CMU Common Lisp Fasload file format.

On the other hand, CMU Common Lisp is intended to be portable, as it will eventually run on a variety of machines. Therefore an explicit goal is that Fasload files shall be transportable among various implementations, to permit efficient distribution of programs in compiled form. The representations of data objects in Fasload files shall be relatively independent of such considerations as word length, number of type bits, and so on. If two implementations interpret the same macrocode (compiled code format), then Fasload files should be completely compatible. If they do not, then files not containing compiled code (so-called "Fasdump" data files) should still be compatible. While this may lead to a format which is not maximally efficient for a particular implementation, the sacrifice of a small amount of performance is deemed a worthwhile price to pay to achieve portability.

The primary assumption about data format compatibility is that all implementations can support I/O on finite streams of eight-bit bytes. By "finite" we mean that a definite end-of-file point can be detected irrespective of the content of the data stream. A Fasload file will be regarded as such a byte stream.

45.2 Strategy

A Fasload file may be regarded as a human-readable prefix followed by code in a funny little language. When interpreted, this code will cause the construction of the encoded data structures. The virtual machine which interprets this code has a *stack* and a *table*, both initially empty. The table may be thought of as an expandable register file; it is used to remember quantities which are needed more than once. The elements of both the stack and the table are Lisp data objects. Operators of the funny language may take as operands following bytes of the data stream, or items popped from the stack. Results may be pushed back onto the stack or pushed onto the table. The table is an indexable stack that is never popped; it is indexed relative to the base, not the top, so that an item once pushed always has the same index.

More precisely, a Fasload file has the following macroscopic organization. It is a sequence of zero or more groups concatenated together. End-of-file must occur at the end of the last group. Each group begins with a series of seven-bit ASCII characters terminated by one or more bytes of all ones `#xFF`; this is called the *header*. Following the bytes which terminate the header is the *body*, a stream of bytes in the funny binary language. The body of necessity begins with a byte other than `#xFF`. The body is terminated by the operation `FOP-END-GROUP`.

The first nine characters of the header must be `FASL FILE` in upper-case letters. The rest may be any ASCII text, but by convention it is formatted in a certain way. The header is divided into lines, which are grouped into paragraphs. A paragraph begins with a line which does *not* begin with a space or tab character, and contains all lines up to, but not including, the next such line. The first word of a paragraph, defined to be all characters up to but not including the first space, tab, or end-of-line character, is the *name* of the paragraph. A Fasload file header might look something like this:

```
FASL FILE >SteelesPerq>User>Guy>IoHacks>Pretty-Print.Slisp
Package Pretty-Print
Compiled 31-Mar-1988 09:01:32 by some random luser
Compiler Version 1.6, Lisp Version 3.0.
Functions: INITIALIZE DRIVER HACK HACK1 MUNGE MUNGE1 GAZORCH
   MINGLE MUDDLE PERTURB OVERDRIVE GOBBLE-KEYBOARD
   FRY-USER DROP-DEAD HELP CLEAR-MICROCODE
    %AOS-TRIANGLE %HARASS-READTABLE-MAYBE
Macros:    PUSH POP FROB TWIDDLE
```

one or more bytes of `#xFF`

The particular paragraph names and contents shown here are only intended as suggestions.

45.3 Fasload Language

Each operation in the binary Fasload language is an eight-bit (one-byte) opcode. Each has a name beginning with "`FOP-`". In the following descriptions, the name is followed by operand descriptors. Each descriptor denotes operands that follow the opcode in the input stream. A quantity in parentheses indicates the number of bytes of data from the stream making up the operand. Operands which implicitly come from the stack are noted in the text. The notation "⇒ stack" means that the result is pushed onto the stack; "⇒ table" similarly means that the result is added to the table. A construction like "$n(1)$ $value(n)$" means that first a single byte n is read from the input stream, and this byte specifies how many bytes to read as the operand named *value*. All numeric values are unsigned binary integers unless otherwise specified. Values described as "signed" are in two's-complement form unless otherwise specified. When an integer read from the stream occupies more than one byte, the first byte read is the least significant byte, and the last byte read is the most significant (and contains the sign bit as its high-order bit if the entire integer is signed).

Some of the operations are not necessary, but are rather special cases of or combinations of others. These are included to reduce the size of the file or to speed up important cases. As an example, nearly all strings are less than 256 bytes long, and so a special form of string operation might take a one-byte length rather than a four-byte length. As another example, some implementations may choose to store bits in an array in a left-to-right format within each word, rather than right-to-left. The Fasload file format may support both formats, with one being significantly more efficient than the other for a given implementation. The compiler for any implementation may generate the more efficient form for that implementation, and yet compatibility can be maintained by requiring all implementations to support both formats in Fasload files.

Measurements are to be made to determine which operation codes are worthwhile; little-used operations may be discarded and new ones added. After a point the definition will be "frozen", meaning that existing operations may not be deleted (though new ones may be added; some operations codes will be reserved for that purpose).

0: `FOP-NOP`

No operation. (This is included because it is recognized that some implementations may benefit from alignment of operands to some operations, for example to 32-bit boundaries. This operation can be used to pad the instruction stream to a desired boundary.)

1: `FOP-POP` ⇒ table

One item is popped from the stack and added to the table.

2: `FOP-PUSH` *index*(4) ⇒ stack

Item number *index* of the table is pushed onto the stack. The first element of the table is item number zero.

3: `FOP-BYTE-PUSH` *index*(1) ⇒ stack
Item number *index* of the table is pushed onto the stack. The first element of the table is item number zero.

4: `FOP-EMPTY-LIST` ⇒ stack
The empty list (`()`) is pushed onto the stack.

5: `FOP-TRUTH` ⇒ stack
The standard truth value (`T`) is pushed onto the stack.

6: `FOP-SYMBOL-SAVE` *n*(4) *name*(*n*) ⇒ stack & table
The four-byte operand *n* specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the default package, and the resulting symbol is both pushed onto the stack and added to the table.

7: `FOP-SMALL-SYMBOL-SAVE` *n*(1) *name*(*n*) ⇒ stack & table
The one-byte operand *n* specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the default package, and the resulting symbol is both pushed onto the stack and added to the table.

8: `FOP-SYMBOL-IN-PACKAGE-SAVE` *index*(4) *n*(4) *name*(*n*) ⇒ stack & table
The four-byte *index* specifies a package stored in the table. The four-byte operand *n* specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the specified package, and the resulting symbol is both pushed onto the stack and added to the table.

9: `FOP-SMALL-SYMBOL-IN-PACKAGE-SAVE` *index*(4) *n*(1) *name*(*n*) ⇒ stack & table
The four-byte *index* specifies a package stored in the table. The one-byte operand *n* specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the specified package, and the resulting symbol is both pushed onto the stack and added to the table.

10: `FOP-SYMBOL-IN-BYTE-PACKAGE-SAVE` *index*(1) *n*(4) *name*(*n*) ⇒ stack & table
The one-byte *index* specifies a package stored in the table. The four-byte operand *n* specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the specified package, and the resulting symbol is both pushed onto the stack and added to the table.

11: `FOP-SMALL-SYMBOL-IN-BYTE-PACKAGE-SAVE` *index*(1) *n*(1) *name*(*n*) ⇒ stack & table
The one-byte *index* specifies a package stored in the table. The one-byte operand *n* specifies the length of the print name of a symbol. The name follows, one character per byte, with the first byte of the print name being the first read. The name is interned in the specified package, and the resulting symbol is both pushed onto the stack and added to the table.

12: `FOP-UNINTERNED-SYMBOL-SAVE` *n*(4) *name*(*n*) ⇒ stack & table
Like `FOP-SYMBOL-SAVE`, except that it creates an uninterned symbol.

13: `FOP-UNINTERNED-SMALL-SYMBOL-SAVE` *n*(1) *name*(*n*) ⇒ stack & table
Like `FOP-SMALL-SYMBOL-SAVE`, except that it creates an uninterned symbol.

14: `FOP-PACKAGE` ⇒ table
An item is popped from the stack; it must be a symbol. The package of that name is located and pushed onto the table.

15: `FOP-LIST` *length*(1) ⇒ stack
The unsigned operand *length* specifies a number of operands to be popped from the stack. These are made into a list of that length, and the list is pushed onto the stack. The first item popped from the stack becomes the last element of the list, and so on. Hence an iterative loop can start with the empty list and

perform "pop an item and cons it onto the list" *length* times. (Lists of length greater than 255 can be made by using `FOP-LIST*` repeatedly.)

16: `FOP-LIST*` *length*(1) ⇒ stack
This is like `FOP-LIST` except that the constructed list is terminated not by `()` (the empty list), but by an item popped from the stack before any others are. Therefore *length*+1 items are popped in all. Hence an iterative loop can start with a popped item and perform "pop an item and cons it onto the list" *length*+1 times.

17-24: `FOP-LIST-1`, `FOP-LIST-2`, ..., `FOP-LIST-8`
`FOP-LIST-`*k* is like `FOP-LIST` with a byte containing *k* following it. These exist purely to reduce the size of Fasload files. Measurements need to be made to determine the useful values of *k*.

25-32: `FOP-LIST*-1`, `FOP-LIST*-2`, ..., `FOP-LIST*-8`
`FOP-LIST*-`*k* is like `FOP-LIST*` with a byte containing *k* following it. These exist purely to reduce the size of Fasload files. Measurements need to be made to determine the useful values of *k*.

33: `FOP-INTEGER` $n(4)$ *value*(n) ⇒ stack
A four-byte unsigned operand specifies the number of following bytes. These bytes define the value of a signed integer in two's-complement form. The first byte of the value is the least significant byte.

34: `FOP-SMALL-INTEGER` $n(1)$ *value*(n) ⇒ stack
A one-byte unsigned operand specifies the number of following bytes. These bytes define the value of a signed integer in two's-complement form. The first byte of the value is the least significant byte.

35: `FOP-WORD-INTEGER` *value*(4) ⇒ stack
A four-byte signed integer (in the range -2^{31} to $2^{31}-1$) follows the operation code. A LISP integer (fixnum or bignum) with that value is constructed and pushed onto the stack.

36: `FOP-BYTE-INTEGER` *value*(1) ⇒ stack
A one-byte signed integer (in the range -128 to 127) follows the operation code. A LISP integer (fixnum or bignum) with that value is constructed and pushed onto the stack.

37: `FOP-STRING` $n(4)$ *name*(n) ⇒ stack
The four-byte operand n specifies the length of a string to construct. The characters of the string follow, one per byte. The constructed string is pushed onto the stack.

38: `FOP-SMALL-STRING` $n(1)$ *name*(n) ⇒ stack
The one-byte operand n specifies the length of a string to construct. The characters of the string follow, one per byte. The constructed string is pushed onto the stack.

39: `FOP-VECTOR` $n(4)$ ⇒ stack
The four-byte operand n specifies the length of a vector of LISP objects to construct. The elements of the vector are popped off the stack; the first one popped becomes the last element of the vector. The constructed vector is pushed onto the stack.

40: `FOP-SMALL-VECTOR` $n(1)$ ⇒ stack
The one-byte operand n specifies the length of a vector of LISP objects to construct. The elements of the vector are popped off the stack; the first one popped becomes the last element of the vector. The constructed vector is pushed onto the stack.

41: `FOP-UNIFORM-VECTOR` $n(4)$ ⇒ stack
The four-byte operand n specifies the length of a vector of LISP objects to construct. A single item is popped from the stack and used to initialize all elements of the vector. The constructed vector is pushed onto the stack.

42: `FOP-SMALL-UNIFORM-VECTOR` $n(1)$ ⇒ stack
The one-byte operand n specifies the length of a vector of LISP objects to construct. A single item is popped from the stack and used to initialize all elements of the vector. The constructed vector is pushed onto the stack.

43: `FOP-INT-VECTOR` *len*(4) *size*(1) *data*($\lceil len * count/8 \rceil$) ⇒ stack
The four-byte operand n specifies the length of a vector of unsigned integers to be constructed. Each integer is *size* bits long, and is packed according to the machine's native byte ordering. *size* must be a directly supported i-vector element size. Currently supported values are 1,2,4,8,16 and 32.

44: `FOP-UNIFORM-INT-VECTOR` *n*(4) *size*(1) *value*(@ceiling<*size*/8>) ⇒ stack
The four-byte operand n specifies the length of a vector of unsigned integers to construct. Each integer is *size* bits big, and is initialized to the value of the operand *value*. The constructed vector is pushed onto the stack.

45: `FOP-LAYOUT`
Pops the stack four times to get the name, length, inheritance and depth for a layout object.

46: `FOP-SINGLE-FLOAT` *data*(4) ⇒ stack
The *data* bytes are read as an integer, then turned into an IEEE single float (as though by `make-single-float`).

47: `FOP-DOUBLE-FLOAT` *data*(8) ⇒ stack
The *data* bytes are read as an integer, then turned into an IEEE double float (as though by `make-double-float`).

48: `FOP-STRUCT` *n*(4) ⇒ stack
The four-byte operand n specifies the length structure to construct. The elements of the vector are popped off the stack; the first one popped becomes the last element of the structure. The constructed vector is pushed onto the stack.

49: `FOP-SMALL-STRUCT` *n*(1) ⇒ stack
The one-byte operand n specifies the length structure to construct. The elements of the vector are popped off the stack; the first one popped becomes the last element of the structure. The constructed vector is pushed onto the stack.

50-52: Unused

53: `FOP-EVAL` ⇒ stack
Pop an item from the stack and evaluate it (give it to `EVAL`). Push the result back onto the stack.

54: `FOP-EVAL-FOR-EFFECT`
Pop an item from the stack and evaluate it (give it to `EVAL`). The result is ignored.

55: `FOP-FUNCALL` *nargs*(1) ⇒ stack
Pop *nargs*+1 items from the stack and apply the last one popped as a function to all the rest as arguments (the first one popped being the last argument). Push the result back onto the stack.

56: `FOP-FUNCALL-FOR-EFFECT` *nargs*(1)
Pop *nargs*+1 items from the stack and apply the last one popped as a function to all the rest as arguments (the first one popped being the last argument). The result is ignored.

57: `FOP-CODE-FORMAT` *implementation*(1) *version*(1)
This FOP specifiers the code format for following code objects. The operations `FOP-CODE` and its relatives may not occur in a group until after `FOP-CODE-FORMAT` has appeared; there is no default format. The *implementation* is an integer indicating the target hardware and environment. See `compiler/generic/vm-macs.lisp` for the currently defined implementations. *version* for an implementation is increased whenever there is a change that renders old fasl files unusable.

58: `FOP-CODE` *nitems*(4) *size*(4) *code*(*size*) ⇒ stack
A compiled function is constructed and pushed onto the stack. This object is in the format specified by the most recent occurrence of `FOP-CODE-FORMAT`. The operand *nitems* specifies a number of items to pop off the stack to use in the "boxed storage" section. The operand *code* is a string of bytes constituting the compiled executable code.

59: `FOP-SMALL-CODE` *nitems*(1) *size*(2) *code*(*size*) ⇒ stack
A compiled function is constructed and pushed onto the stack. This object is in the format specified by the most recent occurrence of `FOP-CODE-FORMAT`. The operand *nitems* specifies a number of items to pop off the stack to use in the "boxed storage" section. The operand *code* is a string of bytes constituting the compiled executable code.

60 `FOP-FDEFINITION`
Pops the stack to get an fdefinition.

61 `FOP-SANCTIFY-FOR-EXECUTION`
A code component is popped from the stack, and the necessary magic is applied to the code so that it can be executed.

62: `FOP-VERIFY-TABLE-SIZE` *size*(4)
If the current size of the table is not equal to *size*, then an inconsistency has been detected. This operation is inserted into a Fasload file purely for error-checking purposes. It is good practice for a compiler to output this at least at the end of every group, if not more often.

63: `FOP-VERIFY-EMPTY-STACK`
If the stack is not currently empty, then an inconsistency has been detected. This operation is inserted into a Fasload file purely for error-checking purposes. It is good practice for a compiler to output this at least at the end of every group, if not more often.

64: `FOP-END-GROUP`
This is the last operation of a group. If this is not the last byte of the file, then a new group follows; the next nine bytes must be "`FASL FILE`".

65: `FOP-POP-FOR-EFFECT` stack ⇒
One item is popped from the stack.

66: `FOP-MISC-TRAP` ⇒ stack
A trap object is pushed onto the stack.

67: `FOP-DOUBLE-DOUBLE-FLOAT` *double-double-float*(8) ⇒ stack
The next 8 bytes are read, and a double-double-float number is constructed.

68: `FOP-CHARACTER` *character*(3) ⇒ stack
The three bytes are read as an integer then converted to a character. This FOP is currently rather useless, as extended characters are not supported.

69: `FOP-SHORT-CHARACTER` *character*(1) ⇒ stack
The one byte specifies the code of a Common Lisp character object. A character is constructed and pushed onto the stack.

70: `FOP-RATIO` ⇒ stack
Creates a ratio from two integers popped from the stack. The denominator is popped first, the numerator second.

71: `FOP-COMPLEX` ⇒ stack
Creates a complex number from two numbers popped from the stack. The imaginary part is popped first, the real part second.

72 `FOP-COMPLEX-SINGLE-FLOAT` *real(4) imag(4)* ⇒ stack
Creates a complex single-float number from the following 8 bytes.

73 `FOP-COMPLEX-DOUBLE-FLOAT` *real(8) imag(8)* ⇒ stack
Creates a complex double-float number from the following 16 bytes.

74: `FOP-FSET`
Except in the cold loader (Genesis), this is a no-op with two stack arguments. In the initial core this is used to make DEFUN functions defined at cold-load time so that global functions can be called before top-level forms are run (which normally installs definitions.) Genesis pops the top two things off of the stack and effectively does (SETF SYMBOL-FUNCTION).

75: `FOP-LISP-SYMBOL-SAVE` $n(4)$ $name(n)$ ⇒ stack & table
Like `FOP-SYMBOL-SAVE`, except that it creates a symbol in the LISP package.

76: `FOP-LISP-SMALL-SYMBOL-SAVE` $n(1)$ $name(n)$ ⇒ stack & table
Like `FOP-SMALL-SYMBOL-SAVE`, except that it creates a symbol in the LISP package.

77: `FOP-KEYWORD-SYMBOL-SAVE` $n(4)$ $name(n)$ ⇒ stack & table
Like `FOP-SYMBOL-SAVE`, except that it creates a symbol in the KEYWORD package.

78: `FOP-KEYWORD-SMALL-SYMBOL-SAVE` $n(1)$ $name(n)$ ⇒ stack & table
Like `FOP-SMALL-SYMBOL-SAVE`, except that it creates a symbol in the KEYWORD package.

79-80: Unused

81: `FOP-NORMAL-LOAD`
This FOP is used in conjunction with the cold loader (Genesis) to read top-level package manipulation forms. These forms are to be read as though by the normal loaded, so that they can be evaluated at cold load time, instead of being dumped into the initial core image. A no-op in normal loading.

82: `FOP-MAYBE-COLD-LOAD`
Undoes the effect of `FOP-NORMAL-LOAD`.

83: `FOP-ARRAY` $rank(4)$ ⇒ stack
This operation creates a simple array header (used for simple-arrays with rank /= 1). The data vector is popped off of the stack, and then *rank* dimensions are popped off of the stack (the highest dimensions is on top.)

84: `FOP-SINGLE-FLOAT-VECTOR` $length(4)$ $data$(n) ⇒ stack
Creates a *(simple-array single-float (*))* object. The number of single-floats is *length.*

85: `FOP-DOUBLE-FLOAT-VECTOR` $length(4)$ $data$(n) ⇒ stack
Creates a *(simple-array double-float (*))* object. The number of double-floats is *length.*

86: `FOP-COMPLEX-SINGLE-FLOAT-VECTOR` $length(4)$ $data$(n) ⇒ stack
Creates a *(simple-array (complex single-float) (*))* object. The number of complex single-floats is *length.*

87: `FOP-COMPLEX-DOUBLE-FLOAT-VECTOR` $length(4)$ $data$(n) ⇒ stack
Creates a *(simple-array (complex double-float) (*))* object. The number of complex double-floats is *length.*

88: `FOP-DOUBLE-DOUBLE-FLOAT-VECTOR` $length(4)$ $data$(n) ⇒ stack
Creates a *(simple-array double-double-float (*))* object. The number of double-double-floats is *length.*

89: `FOP-COMPLEX-DOUBLE-DOUBLE-FLOAT` $data(32)$ ⇒ stack
Creates a *(complex double-double-float)* object from the following 32 bytes of data.

90: `FOP-COMPLEX-DOUBLE-DOUBLE-FLOAT-VECTOR` $length(4)$ $data$(n) ⇒ stack
Creates a *(simple-arra (complex double-double-float) (*))* object. The number of complex double-double-floats is *length.*

91-139: Unused

140: `FOP-ALTER-CODE` $index(4)$
This operation modifies the constants part of a code object (necessary for creating certain circular function references.) It pops the new value and code object are off of the stack, storing the new value at the specified index.

141: `FOP-BYTE-ALTER-CODE` *index*(1)
Like `FOP-ALTER-CODE`, but has only a one byte offset.

142: `FOP-FUNCTION-ENTRY` *index*(4) ⇒ stack
Initializes a function-entry header inside of a pre-existing code object, and returns the corresponding function descriptor. *index* is the byte offset inside of the code object where the header should be plunked down. The stack arguments to this operation are the code object, function name, function debug arglist and function type.

143: `FOP-MAKE-BYTE-COMPILED-FUNCTION` *size*(1) ⇒ stack
Create a byte-compiled function. *FIXME:* describe what's on the stack.

144: `FOP-ASSEMBLER-CODE` *length*(4) ⇒ stack
This operation creates a code object holding assembly routines. *length* bytes of code are read and placed in the code object, and the code object descriptor is pushed on the stack. This FOP is only recognized by the cold loader (Genesis.)

145: `FOP-ASSEMBLER-ROUTINE` *offset*(4) ⇒ stack
This operation records an entry point into an assembler code object (for use with `FOP-ASSEMBLER-FIXUP`). The routine name (a symbol) is on stack top. The code object is underneath. The entry point is defined at *offset* bytes inside the code area of the code object, and the code object is left on stack top (allowing multiple uses of this FOP to be chained.) This FOP is only recognized by the cold loader (Genesis.)

146: Unused

147: `FOP-FOREIGN-FIXUP` *len*(1) *name*(*len*) *offset*(4) ⇒ stack
This operation resolves a reference to a foreign (C) symbol. *len* bytes are read and interpreted as the symbol *name*. First the *kind* and the code-object to patch are popped from the stack. The kind is a target-dependent symbol indicating the instruction format of the patch target (at *offset* bytes from the start of the code area.) The code object is left on stack top (allowing multiple uses of this FOP to be chained.)

148: `FOP-ASSEMBLER-FIXUP` *offset*(4) ⇒ stack
This operation resolves a reference to an assembler routine. The stack args are (*routine-name*, *kind* and *code-object*). The kind is a target-dependent symbol indicating the instruction format of the patch target (at *offset* bytes from the start of the code area.) The code object is left on stack top (allowing multiple uses of this FOP to be chained.)

149: `FOP-CODE-OBJECT-FIXUP` ⇒ stack
FIXME: Describe what this does!

150: `FOP-FOREIGN-DATA-FIXUP` ⇒ stack
FIXME: Describe what this does!

151-156: Unused

157: `FOP-LONG-CODE-FORMAT` *implementation*(1) *version*(4)
Like FOP-CODE-FORMAT, except that the version is 32 bits long.

158-199: Unused

200: `FOP-RPLACA` *table-idx*(4) *cdr-offset*(4)

201: `FOP-RPLACD` *table-idx*(4) *cdr-offset*(4)
These operations destructively modify a list entered in the table. *table-idx* is the table entry holding the list, and *cdr-offset* designates the cons in the list to modify (like the argument to `nthcdr`.) The new value is popped off of the stack, and stored in the `car` or `cdr`, respectively.

202: `FOP-SVSET` *table-idx*(4) *vector-idx*(4)
Destructively modifies a `simple-vector` entered in the table. Pops the new value off of the stack, and stores it in the *vector-idx* element of the contents of the table entry *table-idx.*

203: `FOP-NTHCDR` *cdr-offset*(4) ⇒ stack
Does `nthcdr` on the top-of stack, leaving the result there.

204: `FOP-STRUCTSET` *table-idx*(4) *vector-idx*(4)
Like `FOP-SVSET`, except it alters structure slots.

205-254: Unused

255: `FOP-END-HEADER`
Indicates the end of a group header, as described above.

Appendix A

Glossary

assert (a type) In Python, all type checking is done via a general type assertion mechanism. Explicit declarations and implicit assertions (e.g. the arg to + is a number) are recorded in the front-end (implicit continuation) representation. Type assertions (and thus type-checking) are "unbundled" from the operations that are affected by the assertion. This has two major advantages:

- Code that implements operations need not concern itself with checking operand types.
- Run-time type checks can be eliminated when the compiler can prove that the assertion will always be satisfied.

See also *restrict*.

back end The back end is the part of the compiler that operates on the *virtual machine* intermediate representation. Also included are the compiler phases involved in the conversion from the *front end* representation (or *ICR*).

bind node This is a node type the that marks the start of a *lambda* body in *ICR*. This serves as a placeholder for environment manipulation code.

IR1 The first intermediate representation, also known as *ICR*, or the Implicit Continuation Represenation.

IR2 The second intermediate representation, also known as *VMR*, or the Virtual Machine Representation.

basic block A basic block (or simply "block") has the pretty much the usual meaning of representing a straight-line sequence of code. However, the code sequence ultimately generated for a block might contain internal branches that were hidden inside the implementation of a particular operation. The type of a block is actually `cblock`. The `block-info` slot holds an `VMR-block` containing backend information.

block compilation Block compilation is a term commonly used to describe the compile-time resolution of function names. This enables many optimizations.

call graph Each node in the call graph is a function (represented by a *flow graph.*) The arcs in the call graph represent a possible call from one function to another. See also *tail set*.

cleanup A cleanup is the part of the implicit continuation representation that retains information scoping relationships. For indefinite extent bindings (variables and functions), we can abandon scoping information after ICR conversion, recovering the lifetime information using flow analysis. But dynamic bindings (special values, catch, unwind protect, etc.) must be removed at a precise time (whenever the scope is exited.) Cleanup structures form a hierarchy that represents the static nesting of dynamic binding structures. When the compiler does a control transfer, it can use the cleanup information to determine what cleanup code needs to be emitted.

closure variable A closure variable is any lexical variable that has references outside of its *home environment*. See also *indirect value cell*.

closed continuation A closed continuation represents a `tagbody` tag or `block` name that is closed over. These two cases are mostly indistinguishable in *ICR*.

home Home is a term used to describe various back-pointers. A lambda variable's "home" is the lambda that the variable belongs to. A lambda's "home environment" is the environment in which that lambda's variables are allocated.

indirect value cell Any closure variable that has assignments (`setq`s) will be allocated in an indirect value cell. This is necessary to ensure that all references to the variable will see assigned values, since the compiler normally freely copies values when creating a closure.

set variable Any variable that is assigned to is called a "set variable". Several optimizations must special-case set variables, and set closure variables must have an *indirect value cell.*

code generator The code generator for a *VOP* is a potentially arbitrary list code fragment which is responsible for emitting assembly code to implement that VOP.

constant pool The part of a compiled code object that holds pointers to non-immediate constants.

constant TN A constant TN is the *VMR* of a compile-time constant value. A constant may be immediate, or may be allocated in the *constant pool.*

constant leaf A constant *leaf* is the *ICR* of a compile-time constant value.

combination A combination *node* is the *ICR* of any fixed-argument function call (not `apply` or `multiple-value-call`.)

top-level component A top-level component is any component whose only entry points are top-level lambdas.

top-level lambda A top-level lambda represents the execution of the outermost form on which the compiler was invoked. In the case of `compile-file`, this is often a truly top-level form in the source file, but the compiler can recursively descend into some forms (`eval-when`, etc.) breaking them into separate compilations.

component A component is basically a sequence of blocks. Each component is compiled into a separate code object. With *block compilation* or *local functions*, a component will contain the code for more than one function. This is called a component because it represents a connected portion of the call graph. Normally the blocks are in depth-first order (*DFO*).

component, initial During ICR conversion, blocks are temporarily assigned to initial components. The "flow graph canonicalization" phase determines the true component structure.

component, head and tail The head and tail of a component are dummy blocks that mark the start and end of the *DFO* sequence. The component head and tail double as the root and finish node of the component's flow graph.

local function (call) A local function call is a call to a function known at compile time to be in the same *component.* Local call allows compile time resolution of the target address and calling conventions. See *block compilation.*

conflict (of TNs, set) Register allocation terminology. Two TNs conflict if they could ever be live simultaneously. The conflict set of a TN is all TNs that it conflicts with.

continuation The ICR data structure which represents both:

- The receiving of a value (or multiple values), and
- A control location in the flow graph.

In the Implicit Continuation Representation, the environment is implicit in the continuation's BLOCK (hence the name.) The ICR continuation is very similar to a CPS continuation in its use, but its representation doesn't much resemble (is not interchangeable with) a lambda.

cont A slot in the *node* holding the *continuation* which receives the node's value(s). Unless the node ends a *block*, this also implicitly indicates which node should be evaluated next.

cost Approximations of the run-time costs of operations are widely used in the back end. By convention, the unit is generally machine cycles, but the values are only used for comparison between alternatives. For example, the VOP cost is used to determine the preferred order in which to try possible implementations.

CSP, CFP See *control stack pointer* and *control frame pointer.*

Control stack The main call stack, which holds function stack frames. All words on the control stack are tagged *descriptors.* In all ports done so far, the control stack grows from low memory to high memory. The most recent call frames are considered to be "on top" of earlier call frames.

Control stack pointer The allocation pointer for the *control stack.* Generally this points to the first free word at the top of the stack.

Control frame pointer The pointer to the base of the *control stack* frame for a particular function invocation. The CFP for the running function must be in a register.

Number stack The auxiliary stack used to hold any *non-descriptor* (untagged) objects. This is generally the same as the C call stack, and thus typically grows down.

Number stack pointer The allocation pointer for the *number stack.* This is typically the C stack pointer, and is thus kept in a register.

NSP, NFP See *number stack pointer*, *number frame pointer.*

Number frame pointer The pointer to the base of the *number stack* frame for a particular function invocation. Functions that don't use the number stack won't have an NFP, but if an NFP is allocated, it is always allocated in a particular register. If there is no variable-size data on the number stack, then the NFP will generally be identical to the NSP.

Lisp return address The name of the *descriptor* encoding the "return pc" for a function call.

LRA See *lisp return address.* Also, the name of the register where the LRA is passed.

Code pointer A pointer to the header of a code object. The code pointer for the currently running function is stored in the `code` register.

Interior pointer A pointer into the inside of some heap-allocated object. Interior pointers confuse the garbage collector, so their use is highly constrained. Typically there is a single register dedicated to holding interior pointers.

dest A slot in the *continuation* which points the the node that receives this value. Null if this value is not received by anyone.

DFN, DFO See *Depth First Number*, *Depth First Order.*

Depth first number Blocks are numbered according to their appearance in the depth-first ordering (the `block-number` slot.) The numbering actually increases from the component tail, so earlier blocks have larger numbers.

Depth first order This is a linearization of the flow graph, obtained by a depth-first walk. Iterative flow analysis algorithms work better when blocks are processed in DFO (or reverse DFO.)

Object In low-level design discussions, an object is one of the following:

- a single word containing immediate data (characters, fixnums, etc)
- a single word pointing to an object (structures, conses, etc.)

These are tagged with three low-tag bits as described in the section 40 This is synonymous with *descriptor.* In other parts of the documentation, may be used more loosely to refer to a *lisp object.*

Lisp object A Lisp object is a high-level object discussed as a data type in the Common Lisp definition.

Data-block A data-block is a dual-word aligned block of memory that either manifests a Lisp object (vectors, code, symbols, etc.) or helps manage a Lisp object on the heap (array header, function header, etc.).

Descriptor A descriptor is a tagged, single-word object. It either contains immediate data or a pointer to data. This is synonymous with *object*. Storage locations that must contain descriptors are referred to as descriptor locations.

Pointer descriptor A descriptor that points to a *data block* in memory (i.e. not an immediate object.)

Immediate descriptor A descriptor that encodes the object value in the descriptor itself; used for characters, fixnums, etc.

Word A word is a 32-bit quantity.

Non-descriptor Any chunk of bits that isn't a valid tagged descriptor. For example, a double-float on the number stack. Storage locations that are not scanned by the garbage collector (and thus cannot contain *pointer descriptors*) are called non-descriptor locations. *Immediate descriptors* can be stored in non-descriptor locations.

Entry point An entry point is a function that may be subject to "unpredictable" control transfers. All entry points are linked to the root of the flow graph (the component head.) The only functions that aren't entry points are *let* functions. When complex lambda-list syntax is used, multiple entry points may be created for a single lisp-level function. See *external entry point*.

External entry point A function that serves as a "trampoline" to intercept function calls coming in from outside of the component. The XEP does argument syntax and type checking, and may also translate the arguments and return values for a locally specialized calling calling convention.

XEP An *external entry point*.

lexical environment A lexical environment is a structure that is used during VMR conversion to represent all lexically scoped bindings (variables, functions, declarations, etc.) Each `node` is annotated with its lexical environment, primarily for use by the debugger and other user interfaces. This structure is also the environment object passed to `macroexpand`.

environment The environment is part of the ICR, created during environment analysis. Environment analysis apportions code to disjoint environments, with all code in the same environment sharing the same stack frame. Each environment has a "*real*" function that allocates it, and some collection `let` functions. Although environment analysis is the last ICR phase, in earlier phases, code is sometimes said to be "in the same/different environment(s)". This means that the code will definitely be in the same environment (because it is in the same real function), or that is might not be in the same environment, because it is not in the same function.

fixup Some sort of back-patching annotation. The main sort encountered are load-time *assembler fixups*, which are a linkage annotation mechanism.

flow graph A flow graph is a directed graph of basic blocks, where each arc represents a possible control transfer. The flow graph is the basic data structure used to represent code, and provides direct support for data flow analysis. See component and ICR.

foldable An attribute of *known functions*. A function is foldable if calls may be constant folded whenever the arguments are compile-time constant. Generally this means that it is a pure function with no side effects.

FSC

full call

function attribute function "real" (allocates environment) meaning function-entry more vague (any lambda?) funny function GEN (kill and...) global TN, conflicts, preference GTN (number) IR ICR VMR ICR conversion, VMR conversion (translation) inline expansion, call kill (to make dead) known function LAMBDA leaf let call lifetime analysis, live (tn, variable) load tn LOCS (passing, return locations) local call local TN, conflicts, (or just used in one block) location (selection) LTN (number) main entry mess-up (for cleanup) more arg (entry) MV non-local exit non-packed SC, TN non-set variable operand (to vop) optimizer (in icr optimize) optional-dispatch pack, packing, packed pass (in a transform) passing locations (value) conventions (known, unknown) policy (safe, fast, small, ...) predecessor block primitive-type reaching definition REF representation selection for value result continuation (for function) result type assertion (for template) (or is it restriction) restrict a TN to finite SBs a template operand to a primitive type (boxed...) a tn-ref to particular SCs

return (node, vops) safe, safety saving (of registers, costs) SB SC (restriction) semi-inline side-effect in ICR in VMR sparse set splitting (of VMR blocks) SSET SUBPRIMITIVE successor block tail recursion tail recursive tail recursive loop user tail recursion

template TN TNBIND TN-REF transform (source, ICR) type assertion inference top-down, bottom-up assertion propagation derived, asserted descriptor, specifier, intersection, union, member type check type-check (in continuation) UNBOXED (boxed) descriptor unknown values continuation unset variable unwind-block, unwinding used value (dest) value passing VAR VM VOP

XEP