The Python Compiler for CMU Common Lisp

Robert A. MacLachlan
Carnegie Mellon University

Abstract

The Python compiler for CMU Common Lisp has been under development for over five years, and now forms the core of a production quality public domain Lisp implementation. Python synthesizes the good ideas from Lisp compilers and source transformation systems with mainstream optimization and retargetability techniques. Novel features include strict type checking and source-level debugging of compiled code. Unusual attention has been paid to the compiler's user interface.

1 Design Overview

CMU Common Lisp is a public domain implementation of Common Lisp. The intended application of this implementation is as a high-end Lisp development environment. Rather than maximum peak performance or low memory use, the main design goals are the ease of attaining reasonable performance, of debugging, and of developing programs at a high level of abstraction.

CMU Common Lisp is portable, but does not sacrifice efficiency or functionality for ease of retargeting. It currently runs on three processors: MIPS R3000, SPARC, and IBM RT PC. Python has 50,000 lines of machine-independent code. Each back-end is 8,000 lines of code (usually derived from an existing back-end, not written from scratch.) Porting takes 2–4 wizard-months.

An unusual amount of effort was devoted to user-interface aspects of compilation, even in comparison to commercial products. One of the theses of this paper is that many of Lisp's problems are best seen as user-interface problems.

In a large language like Common Lisp, efficient compiler algorithms and clever implementation can still be defeated in the details. In 1984, Brooks and Gabriel[5] wrote of Common Lisp that:

> Too many costs of the language were dismissed with the admonition that "any good compiler" can take care of them. No one has yet written nor is likely without tremendous effort a compiler that does a fraction of the tricks expected of it. At present, the "hope" for all Common Lisp implementors is that an outstandingly portable Common Lisp compiler will be written and widely used.

Python has a lot of operation-specific knowledge: there are over a thousand special-case rules for 640 different operations, 14 flavors of function call and three representations for multiple values. The + function has 6 transforms, 8 code generators and a type inference method. This functionality is not without a cost; Python is bigger and slower than commercial CL compilers.

2 Type Checking

Python brings strong typing to Common Lisp. Type declarations are optional in Common Lisp, but Python's checking of any type assertions that do appear is both precise and eager:

Precise -- All type assertions are tested using the exact type asserted. All type assertions (both implicit and declared) are treated equally.

Eager — Run-time type checks are done at the location of the first type assertion, rather than being delayed until the value is actually used.

Explicit type declarations are neither trusted nor ignored; declarations simply provide additional assertions which the type checker verifies (with a run-time test if necessary) and then exploits. This eager type checking allows the compiler to use specialized variable representations in safe code.

Because declared types are precisely checked, declarations become more than an efficiency hack. They provide a mechanism for introducing moderately complex consistency assertions into code. Type declarations are more powerful than `assert` and `check-type` because they make it easier for the compiler to propagate and verify assertions.

2.1 Run-Time Errors

Debuggers for other languages have caught up to Lisp, but Lisp programs often remain easier to debug because run-time errors are signalled sooner after the inconsistency arises and contain more useful information.

Debugger capabilities are important, but the *time* that the debugger is invoked is also very important. There is only so much a debugger can do after a program has already crashed and burned. Strict type checking improves the quality and timeliness of run-time error messages.

The efficiency penalty of this run-time type checking would be prohibitive if Python did not both minimize run-time checks through compile-time type inference and apply optimizations to minimize the cost of the residual type checks.

2.2 Compile-Time Type Errors

In addition to proving that a type check is unnecessary, type inference may also prove that the asserted and derived types are inconsistent. In the latter case, the code fragment can never be executed without a type error, so a compile-time warning is appropriate. Compiling this example:

```
(defmacro addf (x n)
  `(+ ,x (the single-float ,n)))

(defun type-error (x)
  (addf x 3))
```

Gives this compiler warning:

```
In: defun type-error
  (addf x 3)
-> +
==>
  (the single-float 3)
Warning: This is not a single-float:
  3
```

In addition to detecting static type errors, flow analysis detects simple dynamic type errors like:

```
(defun foo (x arg)
  (ecase x
    (:register-number
     (and (integerp arg) (>= x 0)))
    ...))

In: defun foo
  (>= x 0)
-> if <
==>
  x
Warning: This is not a (or float rational):
  :register-number
```

Johnson[8] shows both the desirability and feasibility of compile-time type checking as a way to reduce the brittleness (or increase the robustness) of Lisp programs developed using an exploratory methodology.

Python does less interprocedural type inference, so compile-time type checking serves mainly to reduce the number of compile-debug cycles spent fixing trivial type errors. Not all (or even most) programs can be statically checked, so run-time errors are still possible. Some errors can be detected without testing, but testing is always necessary.

Once the basic `subtypep` operation has been implemented, it can be used to detect any inconsistencies during the process of type inference, minimizing the difficulty of implementing compile-time type error checking. The importance of static type checking may be a subject of debate, but many consider it a requirement for developing reliable software. Perhaps compile-time type errors can be justified on these unquantifiable grounds alone.

2.3 Style Implications

Precise, eager type checking lends itself to a new coding style, where:

- Declarations are specified to be as precise as possible: **(integer 3 7)** instead of **fixnum**, **(or node null)** instead of no declaration.

- Declarations are written during initial coding rather than being postponed until tuning, since declarations now enhance debuggability rather than hurting it.

- Declarations are used with more confidence, since their correctness is tested during debugging.

3 Accountability to Users

A Lisp implementation hides a great deal of complexity from the programmer. This makes it easier to write programs, but harder to attain good efficiency.[7] Unlike in simpler languages such as C, there is not a straightforward *efficiency model* which programmers can use to anticipate program performance.

In Common Lisp, the cost of generic arithmetic and other conceptually simple operations varies widely depending on both the actual operand types and on the amount of compile-time type information. Any accurate efficiency model would be so complex as to place unrealistic demands on programmers — the workings of the compiler are (rightly) totally mysterious to most programmers.

A solution to this problem is to make the compiler accountable for its actions[15] by establishing a dialog with the programmer. This can be thought of as providing an automated efficiency model. In a compiler, accountability means that efficiency notes:

- are needed whenever an inefficient implementation is chosen for non-obvious reasons,

- should convey the seriousness of the situation,

- must explain precisely what part of the program is being referred to, and

- must explain why the inefficient choice was made (and how to enable efficient compilation.)

3.1 Efficiency Note Example

Consider this example of "not good enough" declarations:

```
(defun eff-note (s i x y)
  (declare (string s) (fixnum i x y))
  (aref s (+ i x y)))
```

which gives these efficiency notes:

```
In: defun eff-note
  (+ i x y)
==>
  (+ (+ i x) y)
Note:
Forced to do inline (signed-byte 32) arithmetic (cost 3).
Unable to do inline fixnum arithmetic (cost 2) because:
The first argument is a (integer -1073741824
                              1073741822),
not a fixnum.
  (aref s (+ i x y))
-> let*
==>
  (kernel:data-vector-ref array kernel:index)
Note:
Forced to do full call.
Unable to do inline array access (cost 5) because:
   The first argument is a base-string,
   not a simple-base-string.
```

4 Source-Level Debugging

In addition to compiler messages, the other major user interface that Lisp implementations offer to the programmer is the debugger. Historically, debugging was done in interpreted code, and Lisp debuggers offered excellent functionality when compared to other programming environments. This was more due to the use of a dynamic-binding interpreter than to any great complexity of the debuggers themselves.

There has been a trend toward compiling Lisp programs during development and debugging. The first-order reason for this is that in modern implementations, compiled code is so much faster than the interpreter that interpreting complex programs has become impractical. Unfortunately, compiled debugging makes interpreter-based debugging tools like ***evalhook*** steppers much less available. It is usually not even possible to determine a more precise error location than "somewhere in this function."

One solution is to provide source-level debugging of compiled code. Although the idea has been around for a while[18], source level debugging is not yet available in the popular stock hardware implementations. To provide useful source-level debugging, CMU Common Lisp had to overcome these implementation difficulties:

- Excessive space and compile-time penalty for dumping debug information.

- Excessive run-time performance penalty because debuggability inhibits important optimizations like register allocation and untagged representation.

- Poor correctness guarantees. Incorrect information is worse than none, and even very rare incorrectness destroys user confidence in the tool.

The Python symbolic debugger information is compact, allows debugging of optimized code and provides precise semantics. Source-based debugging is supported by a bidirectional mapping between locations in source code and object code. Due to careful encoding, the space increase for source-level debugging is less than 35%. Precise variable liveness information is recorded, so potentially incorrect values are not displayed, giving what Zellweger[22] calls *truthful* behavior.

4.1 Debugger Example

This program demonstrates some simple source-level debugging features:

```
(defstruct my-struct
  (slot nil :type (or integer null)))

(defun source-demo (structs)
  (dolist (struct structs)
    (when (oddp (my-struct-slot struct))
      (return struct))))

(source-demo
 (list (make-my-struct :slot 0)
      (make-my-struct :slot 2)
      (pathname "foo")))
```

Resulting in a run-time error and debugger session:

```
Type-error in source-demo:
   #p"foo" is not of type my-struct

Debug (type H for help)

; The usual call frame printing,
(source-demo (#s(my-struct slot 0)
                    #s(my-struct slot 2)
                    #p"foo"))

; But also local variable display by name,
6] l
struct = #p"foo"
structs = (#s(my-struct slot 0)
           #s(my-struct slot 2)
           #p"foo")

; evaluation of local variables in expressions, and
6] (my-struct-slot (cadr structs))
2

; display of the error location.
6] source 1

(oddp (#:***here*** (my-struct-slot struct)))

; and you can jump to editing the exact error location.
6] edit
```

4.2 The Debugger Toolkit

The implementation details of stack parsing, debug info and code representation are encapsulated by an interface exported from the `debug-internals` package. This makes it easy to layer multiple debugger user interfaces onto CMU CL. The interface is somewhat similar to Zurawski[23]. Unlike SmallTalk, Common Lisp does not define a standard representation of activation records, etc., so the toolkit must be defined as a language extension.

5 Efficient Abstraction

Powerful high-level abstractions help make program maintenance and incremental design/development easier[20, 21]; this is why Common Lisp has such a rich variety of abstraction tools:

- Global and local functions, first-class functional values,

- Dynamic typing (ad-hoc polymorphism), generic functions and inheritance,

- Macros, embedded languages, and other language extensions.

Unfortunately, inefficiency often forces programmers avoid appropriate abstraction. When the difficulty of efficient abstraction leads programmers to adopt a low-level programming style, they have forfeited the advantages of using a higher-level language like Lisp In order to produce a high-quality Lisp development environment, it is not sufficient for the C-equivalent subset of Lisp to be as efficient as C — the higher level programming tools characteristic of Lisp must also be efficiently implemented.

Python increases the efficiency of abstraction in several ways:

- Extensive partial evaluation at the Lisp semantics level,
- Efficient compilation of the full range of Common Lisp features: local functions, closures, multiple values, arrays, numbers, structures, and
- Block compilation and inline expansion (synergistic with partial evaluation and untagged representations.)

5.1 Partial Evaluation

Partial evaluation is a generalization of constant folding[9]. Instead of only replacing completely constant expressions with the compile-time result of the expression, partial evaluation transforms arbitrary code using information on parts of the program that are constant. Here are a few sample transformations:

- Operation is constant:
  (funcall #'eql x y) $\Leftrightarrow$ (eql x y)
- Variables whose bindings are invariant:
  (let ((x (+ a b))) x)
  $\Rightarrow$
  (+ a b)
- Unused expression deletion:
  (+ (progn (* a b) c) d)
  $\Rightarrow$
  (+ c d)
- Conditionals that are constant:
  (if t x y) $\Rightarrow$ x
  (if (consp *a-cons*) x y) $\Rightarrow$ x
  (if p (if p x y) z) $\Rightarrow$ (if p x z)

The canonicalization of control flow and environment during **ICR Conversion** aids partial evaluation by making the value flow apparent.

Source transformation systems have demonstrated that partial evaluation can make abstraction efficient by eliminating unnecessary generality[15, 4], but only a few non-experimental compilers (such as Orbit[9]) have made comprehensive use of this technique. Partial evaluation is especially effective in combination with inline expansion[19, 13].

5.2 Partial Evaluation Example

Consider this analog of `find` which searches in a linked list threaded by a specified successor function:

```
(declaim (inline find-in))
(defun find-in (next element list &key (key #'identity)
                                  (test #'eql test-p)
                                  (test-not nil not-p))
  (when (and test-p not-p)
    (error "Both :Test and :Test-Not supplied."))
  (if not-p
     (do ((current list (funcall next current)))
         ((null current) nil)
       (unless (funcall test-not (funcall key current)
                                element)
         (return current)))
     (do ((current list (funcall next current)))
         ((null current) nil)
       (when (funcall test (funcall key current) element)
         (return current)))))
```

When inline expanded and partially evaluated, this transformation results:

```
(find-in #'foo-next 3 foo-list
         :key #'foo-offset :test #'<)
⇒
(do ((current foo-list (foo-next current)))
    ((null current) nil)
  (when (< (foo-offset current) 3)
    (return current)))
```

5.3 Meta-Programming and Partial Evaluation

Meta-programming is writing programs by developing domain-specific extensions to the language. It is an extremely powerful abstraction mechanism that Lisp makes comparatively easy to use. However, macro writing is still fairly difficult to master, and efficient macro writing is tedious because the expansion must be hand-optimized.

Partial evaluation makes meta-programming easier by making inline functions efficient enough so that macros can be reserved for cases where functional abstraction is insufficient. Such optimization also makes it simpler to write macros because unnecessary conditionals and variable bindings can be introduced into the expansion without hurting efficiency.

5.4 Block Compilation

Block compilation assumes that global functions will not be redefined, allowing the compiler to see across function boundaries[18].

In CMU Common Lisp, full source-level debugging is still supported in block-compiled functions. The only effect that block compilation has on debuggability is that the entire block must be recompiled when any function in it is redefined. Often block compilation is first used during tuning, so full-block recompilation is rarely necessary during development.

Block compilation reduces the basic call overhead because the control transfer is a direct jump and argument syntax checking is safely eliminated. Python also does keyword and optional argument parsing at compile-time.

Another advantage of block compilation is that locally optimized calling conventions can be used. In particular, numeric arguments and return values are passed in non-pointer registers.

In addition to reductions in the call overhead itself, block compilation is important because it extends type inference and partial evaluation across function boundaries. Block compilation allows some of the benefits of whole-program compilation[3] without excessive compile-time overhead or complete loss of incremental redefinition.

Python provides an extension which declares that only certain functions in a block compilation are *entry points* that can be called from outside the compilation unit. This is a syntactic convenience that effectively converts the non-entry **defuns** into an enclosing **labels** form. Whichever syntax is used, the entry point information allows greatly improved optimization because the compiler can statically locate all calls to block-local functions.

6 Numeric Efficiency

Efficient compilation of Lisp numeric operations has long been a concern[16, 6]. Even when type inference is sufficient to show that a value is of an appropriate numeric type, efficient implementation is still difficult because the use of tagged pointers to represent all objects is incompatible with efficient numeric representations.

Python offers a combination of several features which make efficient numeric code much easier to attain:

- Untagged numbers can be allocated both in registers and on the stack.
- Type inference and the wide palette of code generation strategies reduce the number of declarations needed to get efficient code.
- Block compilation increases the range over which efficient numeric representations can be used.
- Efficiency notes provide reliable feedback about numeric operations that were not compiled efficiently.

In addition to open-coding operations on single and double **floats** and **fixnums**, Python also implements full-word integer arithmetic on signed and unsigned 32 bit operands. Word-integer arithmetic is used to implement **bignum** operations, resulting in unusually good **bignum** performance.

7 Implementation

Python's structure is broadly characterized by the compilation phases and the data structures (or *intermediate representations*) that they manipulate. Two major intermediate representations are used:

- The Implicit Continuation Representation (ICR) represents the lisp-level semantics of the source code during the initial phases. ICR is roughly equivalent to a subset of Common Lisp, but is represented as a flow-graph rather than a syntax tree. The main ICR data structures are **nodes**, **continuations** and **blocks**. Partial evaluation and semantic analysis are done on this representation. Phases which only manipulate ICR comprise the "front end".
- 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. All values are stored in Temporary Names (TNs). All operations are Virtual OPerations (VOPs), and their operands are either TNs or compile-time constants.

When compared to the intermediate representations used by most Lisp compilers, ICR is lower level than an abstract syntax tree front-end representation, and VMR is higher level than a macro-assembler back-end representation.

8 Compilation Phases

Major phases are briefly described here. The phases from "local call analysis" to "type check generation" all interact; they are generally repeated until nothing new is discovered.

ICR conversion Convert the source into ICR, doing macroexpansion and simple source-to-source transformation. All names are resolved at this time, so later source transformations can't introduce spurious name conflicts. See section 9.

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 entry stubs for functions that are subject to general function call.

Find components Find the flow graph components and compute a depth-first ordering. Delete unreachable code. Separate top-level code from run-time code, and determine which components are top-level components.

ICR optimize A grab-bag of all the non-flow ICR optimizations. Fold constant functions, propagate types and eliminate code that computes unused values. Source transform calls to some known global functions by replacing the function with a computed inline expansion. Merge blocks and eliminate `if-if` constructs. Substitute let variables. See section 10.

Constant propagation This phase uses global flow analysis to propagate information about lexical variable values (and their types), eliminating unnecessary type checks and tests.

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. Print compile-time type warnings.

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

Environment analysis Determine which distinct environments need to be allocated, and what context needs to be closed over by each environment. We detect non-local exits and and mark locations where dynamic state must be restored. This is the last pure ICR pass.

TN allocation Iterate over all defined functions, determining calling conventions and assigning TNs to local variables. Use type and policy information to determine which VMR translation to use for known functions, and then create TNs for expression evaluation temporaries. Print efficiency notes when a poor translation is chosen.

Control analysis Linearize the flow graph in a way that minimizes the number of branches. The block-level structure of the flow graph is frozen at this point (see section 12.3.)

Stack analysis Discard stack multiple values that are unused due to a local exit making the values receiver unreachable. The phase is only run when TN allocation actually uses the stack values representation.

VMR conversion Convert the ICR into VMR by translating nodes into VOPs. Emit simple type checks. See section 12.

Copy propagation Use flow analysis to eliminate unnecessary copying of TN values.

Representation selection Look at all references to each TN to determine which representation has the lowest cost. Emit appropriate VOPs for coercions to and from that representation. See section 13.

Register allocation Use flow analysis to find the live set at each call site and to build the conflict graph Find a legal register allocation, attempting to minimize unnecessary moves. Registers are saved across calls according to precise lifetime information, avoiding unnecessary memory references. See section 14.

Code generation Convert the VMR to machine instructions by calling the VOP code generators (see section 15.1.)

Instruction-level optimization
Some optimizations (such as instruction scheduling) must be done at the instruction level. This is not a general peephole optimizer.

Assembly and dumping
Resolve branches and convert to an in-core function or an object file with load-time fixup information.

9 The Implicit Continuation Representation (ICR)

ICR is a flow graph representation that directly supports flow analysis. The conversion from source into ICR throws away large amounts of syntactic information, eliminating the need to represent most environment manipulation special forms. Control special forms such as `block` and `go` are directly represented by the flow graph, rather than appearing as pseudo-expressions that obstruct the value flow. This elimination of syntactic and control information removes the need for most "beta transformation" optimizations[17].

The set of special forms recognized by ICR conversion is exactly that specified in the Common Lisp standard; all macros are really implemented using macros. The full Common Lisp **lambda** is implemented with a simple fixed-arg lambda, greatly simplifying later compiler phases.

9.1 Continuations

A continuation represents a place in the code, or alternatively, the destination of an expression result and a transfer of control[1]. This is the *Implicit* Continuation Representation because the environment is not directly represented (it is later recovered using flow analysis.) To make flow analysis more tractable, continuations are not first-class — they are an internal compiler data structure whose representation is quite different from the function representation.

9.2 Node Types

In ICR, computation is represented by these **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 value receiver 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. The common information primarily concerns the order of evaluation and the destinations and properties of results. This control and value flow is indicated in the node by references to continuations.

9.3 Blocks

The **block** structure represents a basic block, in the control flow 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.

9.4 Source Tracking

Source location information is recorded in each node. In addition to being required for source-level debugging, source information is also needed for compiler error messages, since it is very difficult to reconstruct anything resembling the original source from ICR.

10 Properties of ICR

ICR is a flow-graph representation of de-sugared Lisp. It combines most of the desirable properties of CPS with direct support for data flow analysis.

The ICR transformations are effectively source-to-source, but because ICR directly reflects the control semantics of Lisp (such as order of evaluation), it has different properties than an abstract syntax tree:

- Syntactically different expressions with the same control flow are canonicalized:

```
(tagbody (go 1)
        2 y
          (go 3)
        1 x
          (go 2)
        3 z)
⇔
(progn x y z nil)
```

- Use/definition information is available, making substitutions easy.

- All lexical and syntactic information is frozen in during ICR conversion (alphatization[17]), so transformations can be made with no concern for variable name conflicts, scopes of declarations, etc.

- During ICR conversion this syntactic scoping information is discarded and the environment is made implicit, causing further canonicalization:

```
(let ((fun (let ((val init))
             #'(lambda () ...val...))))
  ... (funcall fun) ...
⇔
(let ((val init))
  (let ((fun #'(lambda () ...val...)))
    ... (funcall fun) ...
```

- All the expressions that compute a value (uses of a continuation) are directly connected to the receiver of that value, so syntactic close delimiters don't hide optimization opportunities. As far as optimization of + is concerned, these forms are identical:

```
(+ 3 (progn (gork) 42))
(+ 3 42)
(+ 3 (unwind-protect 42 (gork)))
```

  As the last example shows, this holds true even when the value position is not tail-recursive.

11 Type Inference

Type inference is important for efficiently compiling generic arithmetic and other operations whose definition is based on ad-hoc polymorphism. These operations are generally only open-coded when the operand types are known at compile-time; insufficient type information can cause programs to run thirty times slower. Almost all prior work on Lisp type inference[4, 14, 9, 17] has concentrated on reducing the number of type declarations required to attain acceptable performance (especially for generic arithmetic operations.)

Type inference is also important because it allows compile-time type error messages and reduces the cost of run-time type checking.

11.1 Type Inference Techniques

In order to support type inferences involving complex types, type specifiers are parsed into an internal representation. This representation supports `subtypep`, type simplification, and the union, intersection and difference of arbitrary numeric subranges, `member` and `or` types. `function` types are used to represent function argument signatures. Baker[2] describes an efficient and relatively simple decision procedure for `subtypep`, but it doesn't support accountable type inference, since inferred types can't be inverted into intelligible type specifiers for reporting back to the user.

One important technique is to separately represent the results of forward and backward type inference[4, 8]. This separates provable type inferences from assertions, allowing type checks to be emitted only when needed.

Dynamic type inference is done using data flow analysis tc solve a variant of the constant propagation problem. Wegman[19] offers an example of how constant propagation can be used to do Lisp type inference.

12 The Virtual Machine Representation (VMR)

VMR preserves the block structure of ICR, but annotates the nodes with a target dependent Virtual Machine (VM) Representation. VMR is a generalized tuple representation derived from PQCC[11]. There are only two major objects in VMR: VOPs (Virtual OPerations) and TNs (Temporary Names). All instructions are generated by VOPs and all run-time values are stored in TNs.

12.1 Values and Types

A TN holds a single value. When the number of values is known, multiple TNs are used to hold multiple values, otherwise the values are forced onto the stack. TNs are used to represent user variables as well as expression evaluation temporaries (and other implicit values.)

A *primitive type* is a type meaningful at the implementation level; it is an abstraction of the precise Common Lisp type. Examples are `fixnum`, `base-char` and `single-float`. During VMR conversion the primitive type of a value is used to determine both where where the value can be stored and which type-specific implementations of an operation can be applied to the value. A primitive type may have multiple possible representations (see **Representation Selection**).

VMR conversion creates as many TNs as necessary, annotating them only with their primitive type. This keeps VMR conversion from needing any information about the number or kind of registers, etc. It is the responsibility of **Register Allocation** to efficiently map the allocated TNs onto finite hardware resources.

12.2 Virtual OPerations

The PQCC VM technology differs from a conventional virtual machine in that it is not fixed. The set of VOPs varies depending on the target hardware.

The default calling mechanisms and a few primitives are implemented using standard VOPs that must implemented by each VM. In PQCC, it was a rule of thumb that a VOP should translate into about one instruction. VMR uses a number of VOPs that are much more complex (e.g. function call) in order to hide implementation details from VMR conversion.

Calls to primitive functions such as `+` and `car` are translated to VOP equivalents using declarative information in the particular VM definition; VMR conversion makes no assumptions about which operations are primitive or what operand types are worth special-casing.

12.3 Control flow

In ICR, control transfers are implicit in the structure of the flow graph. Ultimately, a linear instruction sequence must be generated. The **Control Analysis** phase decides what order to emit blocks in. In effect, this amounts to deciding which arm of each conditional is favored by allowing to drop through; the right decision gives optimizations such as loop rotation. A poor ordering would result in unnecessary unconditional branches.

In VMR all control transfers are explicit. Conditionals are represented using conditional branch VOPs that take single target label and a not-p flag indicating whether the sense of the test is negated. An unconditional `branch` VOP is emitted afterward if the other path isn't a drop-through.

13 Representation selection

Some types of object (such as `single-float`) have multiple possible representations[6]. Multiple representations are useful mainly when there is a particularly efficient untagged representation. In this case, the compiler must choose between the normal tagged pointer representation and an alternate untagged representation. Representation selection has two subphases:

- TN representations are selected by examining all the TN references and choosing the representation with the lowest cost.
- Representation conversion code is inserted where the representation of a value is changed.

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

14 Register allocation

Consists of two passes, one which does an initial register allocation, a post-pass that inserts spilling code to satisfy violated register allocation constraints.

The interface to the register allocator is derived from the extremely general storage description model developed for the PQCC project[12]. The two major concepts are:

storage base (SB): A storage resource such as a register file or stack frame. It is composed of same-size units which are jointly allocated to create larger values. Storage bases may be fixed-size (for registers) or variable in size (for memory.)

storage class (SC): A subset of the locations in an underlying SB, with an associated element size. Examples are `descriptor-reg` (based on the `register` SB), and `double-float-stack` (based on the `number-stack` SB, element size 2, offsets dual-word aligned.)

In addition to allowing accurate description of any conceivable set of hardware storage resources, the SC/SB model is synergistic with representation selection. Representations *are* SCs. The set of legal representations for a primitive type is just a list of SCs. This is above and beyond the advantages noted in [6] of having a general-purpose register allocator as in the TNBind/Pack model[10].

15 Virtual Machine Definition

The VM description for an architecture has three major parts:

- The SB and SC definition for the storage resources, with their associated primitive types.
- The instruction set definition used by the assembler, assembly optimizer and disassembler.
- The definitions of the VOPs.

15.1 VOP definition example

VOPs are defined using the **define-vop** macro. Since classes of operations (memory references, ALU operations, etc.) often have very similar implementations, **define-vop** provides two mechanisms for sharing the implementation of similar VOPs: inheritance and VOP variants. It is common to define an "abstract VOP" which is then inherited by multiple real VOPs. In this example, **single-float-op** is an abstract VOP which is used to define all dyadic single-float operations on the MIPS R3000 (only the + definition is shown.)

```
(define-vop (single-float-op)
  (:args (x :scs (single-reg))
         (y :scs (single-reg)))
  (:results (r :scs (single-reg)))
  (:variant-vars operation)
  (:policy :fast-safe)
  (:note "inline float arithmetic")
  (:vop-var vop)
  (:save-p :compute-only)
  (:generator 2
    (note-this-location vop :internal-error)
    (inst float-op operation :single r x y)))

(define-vop (+/single-float single-float-op)
  (:variant '+))
```

16 Evaluation

For typical benchmarks, CMU CL run-times are modestly better than commercial implementations. When average speeds of the non-I/O, non-system Gabriel benchmarks were compared to a commercial implementation, CMU unsafe code was 1.2 times faster, and safe code was 1.8 times faster. Applications that make heavy use of Python's unique features (such as partial evaluation and numeric representation) can be 5x or more faster than commercial implementations.

Generally, poorly tuned programs show greater performance improvements under CMU CL than highly tweaked benchmarks, since Python makes a greater effort to consistently implement all language features efficiently and to provide efficiency diagnostics.

The non-efficiency advantages of CMU CL are at least as important, but harder to quantify. Commercial implementations do weaker type checking than CMU CL, and do not support source level debugging of optimized code. It is also easier to write programs when the programmer can assume a high level of optimizations.

The main cost of this additional functionality is in compile time: compilations using Python may be as much as 3x longer than they are for the quickest commercial Common Lisp compilers. However, on modern workstations with at least 24 megabytes of memory incremental recompilation times are still quite acceptable. A substantial part of the slowdown results from poor memory system performance caused by Python's large code size (3.4 megabytes) and large data structures (0.3–3 megabytes.)

17 Summary

- Strict type checking and source-level debugging make development easier.
- It is both possible and useful to give compile-time warnings about type errors in Common Lisp programs.
- Optimization can reduce the overhead of type checking on conventional hardware.
- The difficulty of producing efficient numeric and array code in Common Lisp is largely due to the compiler's opacity to the user. The lack of a simple efficiency model can be compensated for by making the compiler accountable to the user for its optimization decisions.
- Partial evaluation is a practical programming tool when combined with inline expansion and block compilation. Lisp-level optimization is important because it allows programs to be written at a higher level without sacrificing efficiency.
- It is possible to get most of the advantages of continuation passing style without the disadvantages.
- The clear ICR/VMR division helps portability. It is probably a mistake to convert a Lisp-based representation directly to assembly language.
- A very general model of storage allocation (such as the one used in PQCC) is extremely helpful when defining untagged representations for Lisp values. This is above and beyond the utility of a general register allocator noted in [6].

18 Acknowledgments

Scott Fahlman's infinite patience and William Lott's machine-level expertise made this project possible.

This research was sponsored by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] APPEL, A. W., AND JIM, T. Continuation-passing, closure-passing style. In *ACM Conference on the Principles of Programming Languages* (1989), ACM, pp. 293–302.

[2] BAKER, H. G. A decision procedure for common lisp's subtypep predicate. to appear in Lisp and Symbolic Computation, Sept. 1990.

[3] BAKER, H. G. The nimble project — an applications compiler for real-time common lisp. In *Proceedings of the InfoJapan '90 Conference* (Oct. 1990), Information Processing Society of Japan.

[4] BEER, R. The compile-time type inference and type checking of common lisp programs: a technical summary. Tech. Rep. TR-88-116, Case Western Reserve University, 1988.

[5] BROOKS, R. A., AND GABRIEL, R. P. A critique of common lisp. In *ACM Conference on Lisp and Functional Programming* (1984), pp. 1–8.

[6] BROOKS, R. A., GABRIEL, R. P. AND STEELE JR., G. L. An optimizing compiler for lexically scoped lisp. In *ACM Symposium on Compiler Construction* (1982), ACM, pp. 261–275.

[7] GABRIEL, R. P. Lisp: Good news, bad news, how to win big. to appear, 1991.

[8] JOHNSON, P. M. *Type Flow Analysis for Exploratory Software Development.* PhD thesis, University of Massachusetts, Sept. 1990.

[9] KRANZ, D., KELSEY, R., REES, J., ET AL. Orbit: An optimizing compiler for scheme. In *ACM Symposium on Compiler Construction* (1986), ACM, pp. 219–233.

[10] LEVERETT, B. W. *Register Allocation in Optimizing Compilers.* UMI Research Press, 1983.

[11] LEVERETT, B. W., CATTEL, R. G. G., HOBBS, S. O., NEWCOMER, J. M., REINER, A. H., SCHATZ, B. R., AND WULF, W. A. An overview of the production quality compiler-compiler project. *Computer 13*, 8 (Aug. 1980), 38–49.

[12] REINER, A. Cost minimization in register assignment for retargetable compilers. Tech. Rep. CMU-CS-84-137, Carnegie Mellon University School of Computer Science, June 1984.

[13] SCHEIFLER, R. W. An analysis of inline substitution for a structured programming language. *Communications of the ACM 20*, 9 (Sept. 1977), 647–654.

[14] SHIVERS, O. *Control-Flow Analysis in Higher-Order Languages.* PhD thesis, Carnegie Mellon University School of Computer Science, May 1991.

[15] STEELE, B. S. K. An accountable source-to-source transformation system. Tech. Rep. AI-TR-636, MIT AI Lab, June 1981.

[16] STEELE JR., G. L. Fast arithmetic in maclisp. In *Proceedings of the 1977 MACSYMA User's Conference* (1977), NASA.

[17] STEELE JR., G. L. Rabbit: A compiler for scheme. Tech. Rep. AI-TR-474, MIT AI Lab, May 1978.

[18] TEITELMAN, W., ET AL. *Interlisp Reference Manual.* Xerox Palo Alto Research Center, 1978.

[19] WEGMAN, M. N., AND ZADECK, K. Constant propagation with conditional branches. Tech. Rep. CS-89-36, Brown University, Department of Computer Science, 1989.

[20] WINOGRAD, T. Breaking the complexity barrier again. In *ACM SIGPLAN-SIGIR Interface Meeting* (1973), ACM, pp. 13–22.

[21] WINOGRAD, T. Beyond programming languages. *Communications of the ACM 22*, 7 (Aug. 1979) 391–401.

[22] ZELLWEGER, P. T. Interactive source-level debugging of optimized programs. Tech. Rep. CSL-84-5, Xerox Palo Alto Research Center, May 1984.

[23] ZURAWSKI, L. W., AND JOHNSON, R. E. Debugging optimized code with expected behavior. to appear, Apr. 1991.