Predicate Dispatching in the Common Lisp Object System

by

Aaron Mark Ucko

S.B. in Theoretical Mathematics, S.B. in Computer Science,
both from the Massachusetts Institute of Technology (2000)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2001

© Massachusetts Institute of Technology 2001. All rights reserved.

Author .....
Department of Electrical Engineering and Computer Science
May 11, 2001

Certified by .....
Howard E. Shrobe
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Predicate Dispatching in the Common Lisp Object System

by

Aaron Mark Ucko

Submitted to the Department of Electrical Engineering and Computer Science
on May 11, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

I have added support for predicate dispatching, a powerful generalization of other dispatching mechanisms, to the Common Lisp Object System (CLOS). To demonstrate its utility, I used predicate dispatching to enhance Weyl, a computer algebra system which doubles as a CLOS library. My result is Dispatching-Enhanced Weyl (DEW), a computer algebra system that I have demonstrated to be well suited for both users and programmers.

Thesis Supervisor: Howard E. Shrobe
Title: Principal Research Scientist

Acknowledgments

I would like to thank the MIT Artificial Intelligence Laboratory for funding my studies and making this work possible. I would also like to thank Gregory Sullivan and Jonathan Bachrach for helping Dr. Shrobe supervise my project. Finally, I would like to thank John Aspinall, Michael Ernst, Tom Knight, John Lawrence, and Gail Zacharias for various useful suggestions.

Contents

1 Introduction **11**

1.1 Background 11

1.1.1 Predicate Dispatching 11

1.1.2 The Common Lisp Object System 12

1.1.3 Weyl 13

1.2 Organization 14

2 Related Work **15**

2.1 Predicate Dispatching 15

2.2 Other Dispatching Approaches 16

3 System Specification **17**

3.1 Syntax 17

3.2 Semantics 17

3.3 Differences from the Reference System [EKC98] 18

3.4 Other Known Issues 19

4 Basic Examples **21**

5 Design Considerations **25**

5.1 General Design Principles 25

5.2 Predicates as Qualifiers 25

5.3 Syntax 26

5.4 Modularity 27

5.5 Ambiguity . . . 28
5.6 Coverage . . . 28
5.7 Communication via `&aux` Variables . . . 29
5.8 Lazy Predicate Checking . . . 29
5.9 Mutation and Specificity . . . 30

6 Implementation **31**
6.1 Overview . . . 31
6.2 Representations . . . 32
6.3 Analyzing Implication . . . 34
6.4 Portability . . . 35

7 Symbolic Mathematics **37**
7.1 Popular Software . . . 37
7.1.1 Macsyma . . . 37
7.1.2 Maple . . . 38
7.1.3 Mathematica . . . 38
7.2 Example: Symbolic Integration . . . 39
7.3 Example: The Struve H Function . . . 40
7.4 Current Limitations . . . 40

8 Contributions **41**

9 Future Directions **43**

A Base system source **45**
A.1 pd-package.lisp . . . 45
A.2 predicate-dispatch.lisp . . . 45
A.3 predicate-classes.lisp . . . 48
A.4 normalize-or.lisp . . . 53
A.5 pc-build.lisp . . . 58
A.6 xcond.lisp . . . 70

B Source for DEW applications **71**

B.1 pd-integration.lisp 71

B.2 struve.lisp 75

C Other code **77**

C.1 Struve.m 77

List of Figures

4-1 Example hierarchy of expression types 22

6-1 Predicate class hierarchy (only the leaves are concrete) 33

6-2 Expression analysis class hierarchy (only the leaves are concrete) . . . 33

Chapter 1

Introduction

1.1 Background

1.1.1 Predicate Dispatching

Quite a few programming languages allow procedures to have multiple implementations. In Common Lisp [Ste90]'s terminology, each implementation is a *method* of the same *generic function*, and deciding which to use in a particular situation is *dispatching*. The details of dispatching vary from language to language, but it generally involves attempting to determine the most specific applicable method, where the definitions of *applicability* and *specificity* depend on the language.

One traditional approach is type-based dispatching. In a system using that approach, every method has a tuple of types; a method applies to a tuple of arguments iff every argument is an instance of the corresponding type (or a subtype thereof). Also, one method is more specific than another iff its specializers are pointwise subtypes of the other method's. Even within this approach, there is a fair bit of variation; for instance, some languages (such as Common Lisp [Ste90], Dylan [App92], and Cecil [CCG98]) consider all mandatory arguments' dynamic types, but others (including C++ [Str97] and Java [GJS96]) distinguish the first argument syntactically and semantically, considering only statically-declared types for the others. (In some cases, C++ doesn't even look at the distinguished argument's dynamic type.)

Some languages with better support for dynamic types extend the system further by introducing predicate classes [Cha93], also called "modes" and "classifiers"; in these languages, of which Cecil [CCG98] is a good example, every object effectively has a (potentially dynamically changing) set of predicate types in addition to its normal type.

In another common approach, which ML [MTH90], Haskell [PJHA+99], and related languages use, every method has a pattern which determines applicability, allowing relatively fine-grained control. However, "specificity" in these languages is simply a matter of textual ordering; a developer can inadvertently shadow a method by defining it after another method with more general applicability.

To put it briefly, predicate dispatching [EKC98] is the best of both worlds. Like pattern-matching, it allows fine-grained control of applicability, but like type-based dispatching, it bases specificity on mathematical relationships rather than textual ordering. Specifically, the idea is that every definition of a procedure has an associated predicate expression, whose truth value can depend not just on the types of the arguments but also on the types of their contents, and even on the result of arbitrary boolean expressions in the base language. (Predicate dispatching is more powerful than predicate classes because the expressions can refer to multiple arguments.) It is cleaner than pattern matching in that the system considers logical implication rather than textual ordering when choosing between multiple applicable procedures. Furthermore, other techniques do not allow applicability to depend on relationships between arguments, and lack efficient ways to specify disjunctions of (simple or compound) tests.

1.1.2 The Common Lisp Object System

In the words of John Foderaro [Fod91], Lisp is "a programmable programming language" in that it is extremely extensible by design. One way in which this property manifests itself is that most popular dialects, including ANSI Common Lisp, support a powerful macro facility. Because Lisp has such a uniform syntax, programs in such dialects can define new special forms as easily as new functions. (By contrast, most

other languages would require modifying the implementation, making such programs less portable and harder to build.)

ANSI Common Lisp's macro system is by no means its only avenue of extensibility; the language also includes a powerful object system, imaginatively named the Common Lisp Object System (CLOS) [BDG+88]. Even without extensibility, CLOS is very rich; among other things, it supports multiple inheritance, generic functions which dynamically dispatch on all of their arguments, dynamic class redefinition, and several means of method combination. On top of all that, CLOS has a Meta-Object Protocol (MOP) [KdRB91], which promotes classes, methods, and generic functions to first-class "meta-objects," making it possible to obtain a wide range of custom behavior by subclassing the standard meta-object classes.

1.1.3 Weyl

One interesting piece of software written in CLOS is Weyl [Zip93], a computer algebra system that is based on the principles of category theory and that tags every value with an appropriate domain. The traditional design of computer algebra systems consists of a pile of predefined code and an interface for interactive use; if users can write additional code at all, they typically have to use a proprietary extension language which is quirky or provides little general-purpose functionality. The popular programs Mathematica [Wol99] and Maple [Kam99] both have such extension languages.

However, Weyl's author, Richard Zippel, wrote the base code as a set of publically available CLOS classes and methods, providing full Common Lisp as an extension language and allowing the system to double as a class library for Lisp software that manipulates mathematical objects. It differs from Macsyma [Mac95], a historically important algebra system written in Lisp, by using Lisp as its extension language and by operating at a higher (domain-based) level having been designed to be useful as a library.

Unfortunately, Weyl is still not quite as extensible as it could be: many interesting operations (including, for instance, integration) have special cases that do not

correspond to a simple intersection of operand types. As a consequence, code implementing such an operation has to check for such cases directly, which is less than ideal as far as extensibility goes: if a user of the system finds an additional case interesting, that user has to copy the code that checks for special cases of the operation and insert a test for the relevant case. With predicate dispatching, on the other hand, the user would simply be able to define a method whose predicate corresponds to the case.

1.2 Organization

Chapter 2 discusses related work. Chapter 3 describes the interface to my code. Chapter 4 presents some examples that motivate and clarify the remaining material. Chapter 5 discusses the design issues I considered. Chapter 6 describes my implementation of predicate dispatching in more detail. Chapter 7 discusses its application to symbolic mathematics. Chapter 8 lists my contributions. Chapter 9 discusses future directions.

Chapter 2

Related Work

Discussion of other computer algebra systems appears in Section 7.1.

2.1 Predicate Dispatching

Ernst *et al.* [EKC98] introduce the notion of predicate dispatching; their paper defines its semantics in terms of an abstract syntax, and demonstrates its generality with a number of examples (whose equivalents in my system's syntax appear in Chapter 4). The authors also advertise a small implementation (Güd) which supports their core syntax and some useful syntactic sugar.

Chambers and Chen [CC99] present an algorithm for producing efficient dispatch trees for predicate-dispatched multimethods. Their work is in the context of the Vortex optimizing compiler for Cecil, and takes advantage of static type information which I do not have available; however, much is still applicable, and could be used to improve the performance of my system.

Bachrach and Burke [BB] discuss building dispatch trees at runtime, limiting the contents of the trees to methods which have actually been applicable so far. Their work is also in terms of compilation, and most relevant to the case of per-call-site dispatching trees, but could potentially benefit my system as well.

2.2 Other Dispatching Approaches

Languages which do not support predicate dispatching may still support some other kind of dispatching; here are some of the more interesting approaches.

CLOS [BDG+88] has a relatively rich type-based system even without the MOP. Unlike many popular languages, it is multiply-dispatched, considering all mandatory arguments' dynamic types. In addition, it supports `eql` specializers, which restrict applicability to cases where the argument in question is a particular object, and forms of method combination that let multiple applicable methods cooperate with each other to a limited degree.

Dylan's [App92] actual dispatching system is purely type-based. However, its definition of "type" is fairly broad; the language includes a rich, but unfortunately inextensible, type system supporting multiple inheritance, singleton types (akin to `eql` specializers), union types (retroactive supertypes), and limited types (with constraints on range or element type).

Cecil [CCG98] is multiply-dispatched and has a prototype-based object system that automatically gives it something resembling `eql` specifiers. (It does not prevent people from extending objects used in that fashion, though.) In addition, it supports predicate classes [Cha93], which allow objects to have dynamically changing sets of predicate types in addition to their static types; dispatching can consider predicate types.

Mathematica [Wol99], meanwhile, is an sophisticated example of a pattern-based language: It does not require all the patterns (and bodies) to appear in the same place, so users can extend system functions. It has an elaborate system of precedence which makes textual order relatively unimportant (though still, alas, relevant in some cases). It even allows users to conditionalize applicability on predicate expressions. On the other hand, it does not directly support any sort of class-based subtyping.

Chapter 3

System Specification

Note: Readers not already familiar with predicate dispatching may wish to read the next chapter first.

3.1 Syntax

The syntax for methods of a predicate-dispatched generic function is similar to that for methods of standard CLOS generic functions:

(**defpdmethod** *name lambda-list predicates . body*)

Name, *lambda-list*, and *body* are exactly as in normal `defmethod`; in particular, *lambda-list* can contain specializers (restricting mandatory arguments to particular types or values), which allow the system to take advantage of native CLOS dispatch-optimizing mechanisms. *Predicates* is a (possibly empty) literal list of Lisp expressions.

3.2 Semantics

The predicates are evaluated as if they appeared in the body, except that assignment to normal arguments (as opposed to `&aux` variables) leads to undefined behavior. (However, a method's predicates effectively share `&aux` variables with each other and its body, so assigning to them is a valid way to pass information; see Section 5.7.)

For a predicate-dispatched method to be considered applicable to a given vector of arguments, each of the method's predicates must return true (non-`nil`) given the argument values. The system considers a method to be more specific than another method iff it can determine that the first method's predicates logically imply the second's; it honors `and`, `or`, `not`, `eql` (where one argument is constant), `typep` (where the type is constant), and accessors. It considers conventionally-defined methods less specific than methods defined with `defpdmethod`, which is not necessarily correct.

The system does not guarantee when or how often it will evaluate a method's predicates, save that it will evaluate them in order and stop as soon as it finds one that evaluates to `nil`.

3.3 Differences from the Reference System [EKC98]

Although I based my design on Ernst *et al.*'s reference system, it differs in some respects:

- My system extends an existing language (Lisp), whereas theirs defines a new language for specifying predicates.
- My system does not check for ambiguity. (See Section 5.5.)
- My system does not ensure ahead of time that there is always an applicable method. (See Section 5.6.)
- My system does not allow predicates to bind variables for bodies; instead, it allows them to assign to `&aux`-bound variables. (See Section 5.7.)
- My system does not (directly) handle pattern matching; however, it would be possible to add support cleanly. (Again, see Section 5.7.)
- My system does not have named predicate types; macros can do the same job.

3.4 Other Known Issues

- Efficiency could be better.
- The system assumes that all method definitions appear at toplevel.
- It can get specificity wrong when code mixes defmethod and defpdmethod, because it treats anything with a predicate as more specific than anything without.
- Invoking a predicate-dispatched generic function can make CMU Common Lisp [Mac92] repeatedly print "`Note: Deleting unused function NEXT-METHOD-P.`"
- The system does not work properly on all CLOS implementations; see Section 6.4.

Chapter 4

Basic Examples

Except where otherwise noted, I took the examples here from [EKC98].

Here is how one might merge two lists into a list of pairs:

```
(defpdmethod zip (l1 l2)
    ((consp l1)
      (consp l2))
    (cons (cons (car l1) (car l2))
          (zip (cdr l1) (cdr l2))))

(defpdmethod zip (l1 l2)
    ((or (null l1) (null l2)))
    nil)
```

These two methods cover all possible pairs of lists with no overlap: the first applies when neither is empty (as predicates are implicitly `and`ed together), and the second applies when either is.

This example is actually simple enough for standard CLOS's multiple dispatching to be able to handle it:

```
(defmethod zip2 ((l1 cons) (l2 cons))
    (cons (cons (car l1) (car l2))
          (zip2 (cdr l1) (cdr l2))))

(defmethod zip2 (l1 l2)
    nil)
```

(The first method is more specific, and so overrides the second when both apply.)

However, CLOS's standard dispatching cannot handle everything predicate dispatching can. One simple case it cannot handle is dispatching based on parity, as in the Collatz "hailstone" function [Lag85]:

```
(defpdmethod hailstone (n)
    ((evenp n))
    (/ n 2))

(defpdmethod hailstone (n)
    ((oddp n))
    (+ (* n 3) 1))

(defun hsseq (n)
    (if (= n 1)
        '(1)
        (cons n (hsseq (hailstone n)))))
```

(This example does not appear in [EKC98].)

In general, predicates are useful when one wants to pick a method based on the contents of objects rather than just their types. The following code defines a hierarchy of expression types (shown in Figure 4-1), and then defines the `constant-fold` operation and gives special cases for adding and multiplying two constants:

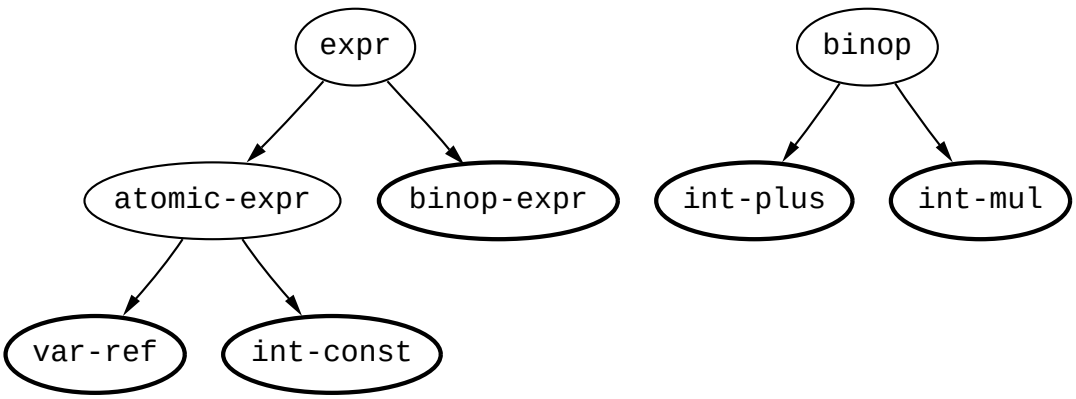

Figure 4-1: Example hierarchy of expression types

```
(defclass expr () ())

(defclass atomic-expr (expr) ())
(defclass var-ref (atomic-expr) ()) ;; would have slots in practice.
```

```
(defclass int-const (atomic-expr)
    ((value :reader value-of)))

(defclass binop () ())
(defclass int-plus (binop) ())
(defclass int-mul (binop) ())

(defclass binop-expr (expr)
    ((op :reader op-of)
     (arg1 :reader arg1-of)
     (arg2 :reader arg2-of)))

(defpdmethod constant-fold (e)
    () ;; default method.
    e)

(defpdmethod constant-fold ((e binop-expr))
    ((typep (op-of e) 'int-plus)
     (typep (arg1-of e) 'int-const)
     (typep (arg2-of e) 'int-const))
    (make-instance 'int-const :value (+ (value-of (arg1-of e))
                                        (value-of (arg2-of e)))))

(defpdmethod constant-fold ((e binop-expr))
    ((typep (op-of e) 'int-mul)
     (typep (arg1-of e) 'int-const)
     (typep (arg2-of e) 'int-const))
    (make-instance 'int-const :value (* (value-of (arg1-of e))
                                        (value-of (arg2-of e)))))
```

In principle, one could also have explicit types like "sum of two constants," but that would lead to an explosion of types. Predicates make this sort of code easier to understand and easier to extend with more elaborate cases such as the following, which handles the case of sums where one term is zero:

```
(defpdmethod constant-fold ((e binop-expr) &aux (a2 (arg2-of e)))
    ((typep (op-of e) 'int-plus)
     (typep (arg1-of e) 'int-const)
     (zerop (value-of (arg1-of e))) ;; guarded by previous check
     (not (typep a2 'int-const))) ;; avoid possible ambiguity
    a2)
```

```
(defpdmethod constant-fold ((e binop-expr) &aux (a1 (arg1-of e)))
  ((typep (op-of e) 'int-plus)
   (not (typep a1 'int-const)) ;; avoid possible ambiguity
   (typep (arg2-of e) 'int-const)
   (zerop (value-of (arg2-of e))) ;; guarded by previous check
  a1)
```

Note that these methods only apply when the non-zero terms are not integer constants. (They could be variable references or compound expressions, say.) Without this restriction, present in the original example, there would be no unique most specific method for a sum where one term is zero and the other term is an integer constant; *zero* is more specific than *some integer constant*, but *some integer constant* is more specific than *anything*. On the other hand, this is really a case of harmless ambiguity, as both methods would give the same result on such input; Section 5.5 discusses this issue further.

As I mentioned, predicate dispatch also generalizes pattern matching. Patterns whose elements have fixed positions translate directly into predicates, as in the above code for `constant-fold`. However, the system can also accommodate patterns with variably-positioned elements with a bit of external help. The following code for rewriting products containing two sines illustrates that approach:

```
(defpdmethod linearize ((expr ge-times) &aux (terms (terms-of expr)) mi)
  ((setf mi (match-or-nil terms '(* sin? * sin? *))))
  (let ((x (arg-of (second mi))) (y (arg-of (fourth mi))))
    (make-ge-times (domain-of expr)
                   (append (list 1/2 (- (cos (- x y)) (cos (+ x y))))
                           (first mi) (third mi) (fifth mi))))))
```

In this example, `mi` stands for "match information"; because it is an `&aux`-bound variable, the predicate's assignment to it is visible in the body. (`match-or-nil` is a fictitious function that returns a list of subexpressions on success and `nil` on failure; I take advantage of the fact that `setf` returns the last value supplied.)

Chapter 5

Design Considerations

5.1 General Design Principles

When developing my system, I tried to satisfy several principles, which I list in decreasing order of priority:

- Copying Ernst *et al.*'s reference system [EKC98].
- Making the interface simple, and consistent with the rest of CLOS.
- Making the code portable across CLOS implementations.
- Keeping the implementation simple enough to complete.
- Making the code reasonably efficient.

Unfortunately, the fact that I was dealing with an interpreted language interfered with some of these goals; in particular, I had to give up early detection of ambiguity or insufficient coverage.

5.2 Predicates as Qualifiers

In standard CLOS, methods have two attributes that affect dispatching: specializers and qualifiers. Specializers are associated with mandatory arguments and determine

applicability; qualifiers are associated with entire methods and affect method combination. (For instance, standard method combination uses the qualifiers `:before`, `:after`, and `:around`.) CLOS assumes that those are the only relevant attributes; if I define a method with the same specializers and qualifiers as another method, the system will discard the original method.

Therefore, I had three options. First, I could treat predicates as specializers, which would require arbitrarily assigning them to mandatory arguments and dealing with the one-specializer-per-argument limit. Second, I could treat predicates as qualifiers, even though only specializers are supposed to affect applicability. Finally, I could treat predicates as a third distinguishing attribute, hacking everything necessary to honor it at the inevitable cost of some portability. I chose to treat predicates as qualifiers for the sake of simplicity; unfortunately, that approach turned out to break on one implementation I tried.[1] (It works on at least three others, though.)

5.3 Syntax

Although the MOP is fairly versatile, it provides no hooks into `defmethod` that would allow qualifiers to appear after lambda lists. Because lambda lists provide useful context for predicates, I introduced a wrapper allowing predicates to appear in between lambda lists and bodies. Because my wrapper took a different syntax from `defmethod`, a different name was in order; I chose `defpdmethod`, where `pd` of course stands for "predicate dispatched." (`defpmethod` was also a possibility, and easier to pronounce but also easier to typo.) As for its precise syntax, I considered supporting a new lambda-list keyword (perhaps `&when` or `&predicate`) or a `declare`-like construct, but decided to take a simpler approach because I saw no particularly compelling reason to add another context-specific syntax extension.

In keeping with that philosophy (simplicity of interface), I also chose to allow developers to represent predicates as arbitrary Lisp expressions rather than making them learn a special language for predicates or deal with syntactic restrictions; after

[1] Allegro Common Lisp 6.0 [Fra00]; see Section 6.4.

all, Lisp already has reasonable support for type-checking, boolean combination, etc. As a side bonus, this design made it easy to write a prototype that dealt with everything but specificity ranking. One consequence of this decision is that the structure of predicates is implicit rather than explicit; if the system's internal representation for a particular sort of test improves, developers will not necessarily have to modify their code to take advantage of the improvement. On the other hand, the analyzer is not especially clever, so developers may inadvertently write code that it handles poorly. (It may end up being unable to determine implication relations even when they exist, whereas it would have been able to had one or both predicates been written differently. Note that this issue just affects determination of relative specificity, which is uncomputable in the general case anyway.)

The only other major syntactic issue is that `defpdmethod` takes a list of predicates (which it implicitly intersects) rather than a single predicate. Given that `t` and explicit intersection are both legal, this is purely a surface issue; my only reason for taking a list was that such an arrangement worked better for my prototype, which approximately judged specificity by counting predicates. (That approach was extremely crude, but allowed me to test some parts of the system early on.)

5.4 Modularity

In order to avoid interfering with the existing environment, I put everything in its own package (`predicate-dispatch`). Moreover, I created a new generic function type (`gf-with-predicate-dispatching`) and a new method type (`predicate-dispatched-method`), and specialized my method redefinitions on them. Because CLOS's internals are object-oriented, this addition took very little work; unfortunately, it requires more of a MOP than some implementations have. (See Section 6.4 for details.) That issue does not terribly concern me, since I need to assume a proper MOP elsewhere anyway.

5.5 Ambiguity

Ernst *et al.* [EKC98] ensure that there is always a unique most specific method. In many cases, particularly in the domain of mathematics, their concern is unwarranted. In such cases, anything that applies is correct, and anything other than the default method is a win, so overlapping non-default methods cause no problems. For instance, one interesting problem is integration, a partial treatment of which appears in Section 7.2. If the user-visible code were a generic function rather than a wrapper and it had an additional method for integrals with numerical limits, that method would be ambiguous with methods specialized on particular sorts of integrand; however, that would not be a problem because the system would end up getting the same answer either way.

On the other hand, there are also cases where ambiguity really is a problem, so catching it can be useful. For instance, consider the case of a generic function for transferring funds between bank accounts, where certain types of account require special treatment. If I wanted to transfer money from an account of type X to an account of type Y, then neither a method specialized only on a source of type X nor a method specialized only on a destination of type Y would be appropriate.

The best solution would probably be to give each predicate-dispatched generic function a flag indicating whether it can have ambiguous methods; for safety's sake, they would be illegal by default. My system performs no ambiguity checking as of yet, however.

5.6 Coverage

Ernst *et al.* also ensure that there is an applicable method at every call site, which they can readily do because their system is compiler-based. Although my system could also benefit from such checks, working them in would be difficult; some sort of stand-alone checker that developers could pass their code through might make more sense.

5.7 Communication via `&aux` Variables

Common Lisp allows programmers to replace `let*` around a method body with `&aux` in its lambda list. I extended this feature for predicate-dispatched methods; a method's predicates effectively share bindings of `&aux` variables with the method body and each other (but not with other methods). I considered special-casing disjunctions *à la* Ernst *et al.* [EKC98], but decided against it; the exception makes sense in their system because they let predicates bind variables, whereas I merely let them assign to (shared) variables that are already bound. One use of this extension is for performance; binding `&aux` variables to expressions which appear in multiple predicates, or in the body and some predicate, avoids making the system recompute them.

Another, more interesting, use is communication; since the variables are shared, an assignment in one predicate is visible in later predicates and the body. Thanks to this feature, usefully supporting pattern-matching (for example) would not require modifying my system. Somebody could simply write a helper function that takes a pattern and whatever it should match against and returns either `nil` (indicating no match) or a description of the match that a predicate could assign to an `&aux` variable.

5.8 Lazy Predicate Checking

Letting predicates pass information to their associated method bodies led to certain architectural constraints. In particular, it made it impossible for `compute-applicable-methods` (a MOP function I specialize) to check predicates; my definition of `compute-applicable-methods` instead sorts methods based on implication (arbitrarily ordering methods with logically independent predicates) and leaves the checking up to the wrapper `make-method-lambda` uses. That arrangement turned out to have an unexpected advantage: when the wrapper goes through the sorted list, it can stop checking predicates as soon as it finds a hit. (Invoking `call-next-method` or `next-method-p`

within the body would force it to resume until it found another hit, but still not necessarily go all the way through the list.) In addition, it does not interfere with applicable-method caching, so it is also a performance win in that respect.

5.9 Mutation and Specificity

Allowing predicates to modify variables means that the implication checker ought to consider ordering. To take a contrived example, if a is an auxilary variable,

(**and** (*eq* *a* 2) (**setf** *a* 2))

implies

(**and** (*evenp* *a*) (**setf** *a* 2))

but

(**and** (*evenp* *a*) (**setf** *a* 2))

implies

(**and** (**setf** *a* 2) (*eq* *a* 2)).

As it happens, my system is too conservative about predicates encapsulating raw code for that problem to arise; it sees no implication relationship between any of the above predicates. A better way to avoid having to worry about ordering would be to impose usage restrictions (checked at method definition time): predicates may not assign to `&aux` variables which already have assignments or dereference unassigned `&aux` variables, where the default of `nil` does not count as an assignment.

Chapter 6

Implementation

6.1 Overview

`defpdmethod` is a macro which expands into a call to `defmethod`, using the code in `pc-build.lisp` (Section A.5) to convert the supplied specifiers and predicates into an object of class `predicate-qualifier` (described in Section 6.2), which it supplies to `defmethod` as a qualifier. It also contains some code to deal with sharing `&aux` variables. Like all of the top-level code, it appears in `predicate-dispatch.lisp` (Section A.2), which you can read for more details.

`defmethod` in turn calls down to `make-method-lambda`, which mostly generates a lot of boilerplate to check predicates; although putting all that code in every method no doubt leads to some memory bloat, my semantics for `&aux` leave little choice. As discussed in Section 5.8, this code checks predicates on demand, so it should win on time even if it loses on space.

Invoking a predicate-dispatched generic function triggers some other parts of the code. First, the system calls the new definition of `compute-applicable-methods-using-classes`, and perhaps also `compute-applicable-methods`. Lacking places to store `&aux`-variable values, neither can usefully determine eligibility; however, they still do useful work by sorting the predicate-blind list of applicable methods by implication. The system determines whether a predicate p implies a predicate q by constructing $\neg p \vee q$ and using the code in `normalize-or.lisp` (Section A.4) to attempt to simplify

it to `*true*`.

Once the system has the list, it uses `compute-effective-method` to turn it into an effective method form. My implementation differs from the standard one in not supporting standard method combination (`:around`, `:before`, and `:after`), and in passing the most specific method along with the other methods; my architecture requires the second change so that the code `make-method-lambda` issues can check its predicate.

6.2 Representations

As mentioned in Section 5.2, I turn predicates into qualifiers. Specifically, I turn them into instances of `predicate-qualifier`, which contains an object descended from class `predicate` (which I shall describe shortly) and an integer indicating how many `&aux` variables to allocate space for. Class `predicate` is the abstract ancestor of all predicate types; Figure 6-1 shows the entire hierarchy, which contains nine concrete and three abstract classes. For instance, my system would turn the predicate in

(**defpdmethod** *∗test∗* ((*foo standard-object*) (*bar standard-class*))
  ((*eql* (*class-of foo*) *bar*))
  t)

into the object

```
#<pq 0 #<and #<proj 0 #<type STANDARD-OBJECT>>
             #<proj 1 #<type STANDARD-CLASS>>
             #<Interpreted Function (LAMBDA
                                      (PREDICATE-DISPATCH::AUXV FOO BAR
                                       &REST #:G1004)
                                     (DECLARE
                                      (IGNORE #:G1004
                                       PREDICATE-DISPATCH::AUXV))
                                     (EQL (CLASS-OF FOO) BAR))
               {480DBF91}>>
```

The code that constructs predicates from expressions starts out by analyzing them in terms of another hierarchy, which Figure 6-2 presents. In addition, as mentioned in Section 5.4, I subclass `standard-generic-function` with `gf-with-predicate-`

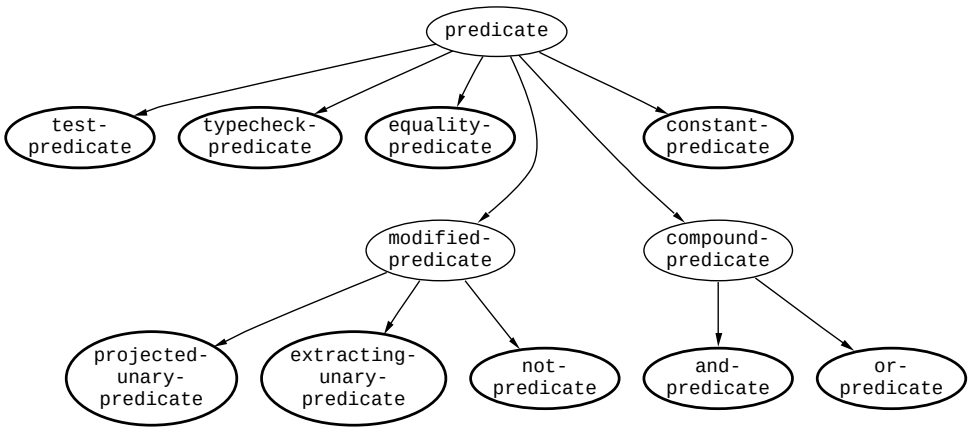

Figure 6-1: Predicate class hierarchy (only the leaves are concrete)

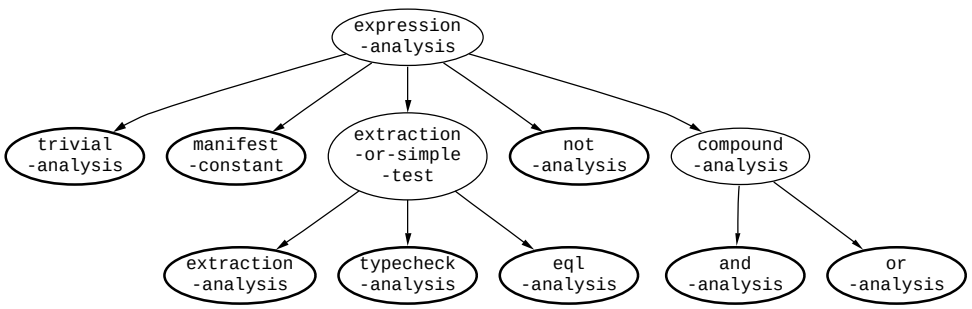

Figure 6-2: Expression analysis class hierarchy (only the leaves are concrete)

`dispatching` and `standard-method` with `predicate-dispatched-method` to avoid interfering with existing CLOS code; aside from that, I just work with standard types. (There are some places where symbols can only take on values from a small finite set; I'll discuss them in the next section.)

6.3 Analyzing Implication

The above explanation should suffice for most of my code. However, `normalize-or.lisp` (Section A.4) deserves additional discussion; it is relatively complicated because it deals with implication. ($p \Rightarrow q$ is equivalent to $\neg p \vee q$; proving that the latter expression is always true allows me to conclude that the former is.)

My system starts out by rewriting the given predicate in disjunctive normal form, which may entail duplicating some subpredicates. This process yields a predicate with three levels: from the bottom up, the levels are simple terms (which may be negated, but are not compound), purely conjunctive predicates (PCPs) (conjunctions of terms), and full predicates (disjunctions of purely conjunctive predicates).

The code, in turn, has five levels. At the top, `normalize-predicate` maps full predicates to full predicates, using a loop that builds its result up from purely conjunctive predicates. That loop, in turn, contains an inner loop that passes the PCP to be added along with each existing PCP to `compute-safe-patch`, which determines their union's maximal PCPs. `compute-safe-patch` contains more (implicit) loops that get its information from `analyze-term`, which takes a simple term and a PCP and returns an analysis (described later in this section). `analyze-term` contains a final implicit loop, which passes pairs of simple terms to `compare-terms` and gets back relations (also described later in this section).

There are five possible analyses of a simple term with respect to a PCP:

extra: Not (known to be) inconsistent *or* redundant with the given PCP.

fatal-mismatch: Inconsistent with some term of the PCP, and not (known to be) complementary with it.

match: Equivalent to some term of the PCP.

mismatch: Complementary to some term of the PCP, and not (known to be) inconsistent with any other terms.

weak: Redundant given the PCP, but not (known to be) equivalent to any term.

There are also seven possible relations between the simple terms p and q: `same` ($p \Leftrightarrow q$), `opposite` ($p \Leftrightarrow \neg q$), `forward` ($p \Rightarrow q$), `backward` ($q \Rightarrow p$), `exclusive` ($p \Rightarrow \neg q$), `comprehensive` ($\neg p \Rightarrow q$), and `nil` (no known relation).

6.4 Portability

In general, I tried to write my code so that it would work on any CLOS implementation with support for the full MOP. However, I was only able to get it to work on three: CMU Common Lisp [Mac92] (2.4.x and 2.5.x), Allegro Common Lisp [Fra00] (5.0 but not 6.0), and GNU Common Lisp [Sch01] (for which I had to build Portable CommonLoops [BKK$^+$86]).

Allegro Common Lisp 6.0 seems unwilling to deal with passing predicates off as qualifiers: it only accepts non-standard qualifiers in conjunction with non-standard method combinations, and I was unable to find a usage of `define-method-combination` that would satisfy it. (A method combination definition involves classifying methods into a statically defined set of groups, and Allegro objects to putting two methods with identical specializers in the same group.)

The other two implementations I tried (CLISP 2000–03–06 [HS00] and Poplog 15.53 [Slo90]), meanwhile, would not let me subclass `standard-generic-function` because it was a `built-in-class` rather than a `standard-class`. (Neither seems to have heard of `funcallable-standard-class`.)

Chapter 7

Symbolic Mathematics

One application area in which predicate dispatching can be particularly useful is symbolic mathematics, which exhibits a lot of special cases that do not correspond well to combinations of types.

7.1 Popular Software

The two leading programs for symbolic mathematics (as opposed to numerical computation) are Maple [Kam99] and Mathematica [Wol99]; Macsyma [Mac95] is also historically significant. Although all three work well for symbolic calculations, their extension languages leave something to be desired; as such, none is a great platform for programs that deal with mathematical objects. (As you may recall, Section 1.1.3 discusses Weyl, which takes a more balanced approach.)

7.1.1 Macsyma

Macsyma started out as part of a U.S. Department of Energy project in the late 1960s, but has survived to this day because it continues to be useful for many problems. It is a Lisp program, but its interface uses a separate extension language with an infix syntax akin to typical mathematical notation. Needless to say, it supports anonymous functions, albeit with dynamic scoping. It lacks dispatching *per se*, but compensates

by providing a pattern-substitution facility.

Although Macsyma is a good program, it suits users much better than developers. For one thing, although it is written in Lisp, it predates object-oriented programming by well over a decade, so developers cannot cleanly extend its built-in functions. (They can define pattern rules, but those are somewhat arcane and serve a somewhat different purpose. On the other hand, developers can take advantage of CLOS when adding wholly new functions.) Also, it relies too much on global parameters to work well as a library.

7.1.2 Maple

In the early 1980s, researchers at the University of Waterloo in Canada set out to produce a computer algebra system that would run reasonably on relatively inexpensive hardware. (Previous systems effectively required dedicated mainframes.) Its extension language is similar to Macsyma's, but a bit more sophisticated; it supports not only anonymous functions but also modules, lexical scope, and optional type-checking. It supports dispatching only in that a few built-in functions will dispatch on the type of a single argument; however, like Macsyma it has a separate pattern-substitution facility.

Again like Macsyma, Maple is good for users but not all that great for developers. Because its (C) source is not available, developers can extend the system only in its own language, which is decent enough but not quite the same as anything else. Also, it lacks true dispatching; the closest it has is pattern substitution, with the caution "It is the responsibility of the user to make sure that the pattern is not overlapping." Its effective monolinguality also detracts from its utility as a library.

7.1.3 Mathematica

Mathematica is the youngest popular program under consideration, dating back only to the late 1980s. Whereas Macsyma and Maple both combine pattern-matching with monolithic functions, Mathematica expresses everything in terms of rewrite rules. As

such, it essentially amounts to a huge term-rewriting system, which is an interesting design choice. In order to get the most out of that design, it defines relatively complex heuristics for ranking rules in terms of specificity, though still does not appear to tackle actual implication.

As with Maple, Mathematica's source is unreleased C, so the system is extensible only in its own language and relatively unsuitable as a library for other software. Also, its rewrite-rule-focused language, though surprisingly versatile (and clearly Turing-complete), is not the right tool for every problem; as such, even examples intended to show it off resort to various kludges. (For instance, page 228 of [Mae96b] suggests faking `call-next-method` by conditionalizing on a global variable which the extending method temporarily sets to false.) It also suffers from a lack of non-structural subtyping.

7.2 Example: Symbolic Integration

Note that all three of the above systems, despite their differences in design, support some form of pattern matching. The reason for this is that there are quite a few mathematical functions and operators which have simple values only in certain special cases; a lot of the time, the special cases correspond poorly to intersections of natural types. One particularly good example of this phenomenon is symbolic integration.

`pd-integration.lisp` (Section B.1) contains my implementation of integration in Dispatching-Enhanced Weyl (DEW). Its external interface is `int`, which calls down to `integral` via either `definite-integral` or `indefinite-integral`, depending on whether the caller specified limits. `integral` does the actual work; by default, it constructs an object of type `ge-integral` (where `ge` is Weyl's abbreviation for "general expression"), but there are also a number of specialized methods which yield more useful results, such as the one that handles exponentials with constant bases:

```
(defpdmethod integral ((expr ge-expt) var
                       &aux (base (base-of expr)) (exp (exponent-of expr)))
  ((free? base var)
   (linear? exp var))
```

```
(if (= base 1)
    (/ exp (deriv base var))
     (/ (make-ge-expt (domain-of expr) base exp) (log base) (deriv base var))))
```

Note that the *only* method here that can get by without predicates is the one that simply distributes integration over addition:

```
(defpdmethod integral ((expr ge-plus) var) ()
   (make-ge-plus (domain-of expr)
                 (mapcar (lambda (exp)
                           (integral exp var))
                         (terms-of expr))))
```

This issue may help explain why standard Weyl lacks support for integration.

7.3 Example: The Struve H Function

Programming in Mathematica [Mae96b] uses the Struve function $H_\nu(z)$ to demonstrate how to implement support for a new special function in Mathematica (which I target because it is the most sophisticated of the three programs discussed in Section 7.1). Maeder's code deals with easy-to-compute special cases, series expansion, numerical evaluation, differentiation, and formatting. Most of this code translates well into DEW; my translation appears in Section B.2. I had to omit symbolic series expansion because Weyl does not (yet?) support it, and differentiation and formatting because it was not always clear how to treat symbolic functions of multiple arguments.

7.4 Current Limitations

Although DEW has the makings of an excellent computer algebra system, it is not quite there yet. Its most serious problem is that its mathematical library is much smaller than other systems'. Actual pattern-matching sugar would also come in handy in some cases, such as simplifying products where the relevant terms may be mixed between irrelevant terms. One final issue is that DEW only deals with Lisp's prefix syntax, which some mathematicians find awkward.

Chapter 8

Contributions

I have portably extended CLOS with predicate dispatching, making its advanced functionality available to a wide range of users. I have additionally contributed to the computer algebra community by making this functionality available to Weyl, yielding an even more interesting system.

Chapter 9

Future Directions

I plan to add examples and test cases to make sure my code works in a wide range of situations, and to improve my existing examples, particularly integration. As it happens, integration has the problem that it is often necessary to fall back on heuristics, which may require backtracking if chosen poorly; however, it may be possible to handle that with judicious use of `call-next-method`.

If time permits, I will add other features to the general predicate dispatch code. Here's what I have in mind, in decreasing order of priority:

- Write code to produce custom dispatching trees, which should take care of the majority of the issues listed in Section 3.4. Figure out if adding new methods requires rebuilding the trees from scratch.
- Provide a reasonable syntax for specifying predicates by means of patterns.
- Enhance the predicate type system. (Numeric comparison could turn out to be useful, for instance.)

I might also give DEW an alternate front-end supporting conventional (infix) algebraic notion for input, since that is much more conventional for mathematics.

Appendix A

Base system source

A.1 pd-package.lisp

```
(make-package :predicate-dispatch :use '(:lisp
                                         #+allegro :mop
                                         #+(and CLOS (not POPLOG)) :clos
                                         #+(and PCL (not CMU)) :pcl))
(in-package :predicate-dispatch)

#+CMU (shadowing-import '(mop:compute-applicable-methods
                          mop:compute-applicable-methods-using-classes
                          mop:compute-discriminating-function
                          mop:compute-effective-method
                          mop:find-class
                          ;;mop:find-method-combination
                          mop::funcallable-standard-class
                          mop:generic-function-method-combination
                          mop:generic-function-name
                          mop:make-method-lambda
                          mop:method-function
                          mop:method-qualifiers
                          mop:method-specializers))

(export '(defpdmethod gf-with-predicate-dispatching
          predicate-dispatched-method))

(provide 'pd-package)
```

A.2 predicate-dispatch.lisp

```
(require 'pd-package)
(in-package :predicate-dispatch)
```

```
(require 'pc-build)

;; Known issues with this code:
;;  * It's not particularly efficient.
;;  * It assumes all method definitions appear at toplevel.
;;  * It has bogus semantics for specificity.  (Mostly fixed, but bogus
;;    behavior can still occur when code mixes defmethod and defpdmethod.)
;;  * CMUCL spews "Note: Deleting unused function NEXT-METHOD-P."

(defclass gf-with-predicate-dispatching (standard-generic-function)
    ()
    (:metaclass funcallable-standard-class)
    (:default-initargs :method-class (find-class 'predicate-dispatched-method)))

(defmethod update-instance-for-different-class :before
    (old (new gf-with-predicate-dispatching) &rest junk)
    ;; AFAICT, change-class ignores default initargs.
    (declare (ignore junk))
    (unless (slot-boundp old 'method-class)
        (setf (slot-value new 'method-class)
              (find-class 'predicate-dispatched-method))))

(defclass predicate-dispatched-method (standard-method)
    ())

;; Turn the predicate list into a qualifier.  This approach has two
;; major advantages:
;;  * There's a portable way to pull the predicate list back out.
;;  * It doesn't yield "redefinitions" with identical specializers and
;;    qualifiers but different predicates.

(defmacro defpdmethod (name lambda-list pred-list &rest body)
    (ensure-generic-function name
                             :generic-function-class
                             'gf-with-predicate-dispatching)
    (let ((auxtail (member '&aux lambda-list))
          (pred (build-predicate lambda-list pred-list)))
        (if auxtail
            (do ((auxvars (cdr auxtail) (cdr auxvars))
                 (index 0 (1+ index))
                 (let-clauses nil))
                ((null auxvars)
                 `(defmethod ,name ,pred ,(ldiff lambda-list auxtail)
                      (let ,let-clauses ,@body)))
                (push `(,(car-or-identity (car auxvars)) (aref auxv ,index))
                      let-clauses))
```

```
        `(defmethod ,name ,pred ,lambda-list ,@body))))

(defmethod predicate-of ((method standard-method))
  (declare (ignore method))
  *true*)

(defmethod pq-of ((method predicate-dispatched-method))
  (find-if #'predicate-qualifier? (method-qualifiers method)))

(defmethod predicate-of ((method predicate-dispatched-method))
  (predicate-of (pq-of method)))

(defun sort-methods (methods)
  (stable-sort (copy-list methods) #'implies? :key #'predicate-of))

(defmethod compute-applicable-methods-using-classes
  ((gf gf-with-predicate-dispatching) classes)
  (multiple-value-bind (methods memoizable)
      (call-next-method)
    (values (sort-methods methods) memoizable)))

(defmethod compute-applicable-methods ((gf gf-with-predicate-dispatching) args)
  (sort-methods (call-next-method)))

(defmethod make-method-lambda ((gf gf-with-predicate-dispatching)
                               (method predicate-dispatched-method)
                               lambda-expression
                               environment)
  `(lambda (args remaining-methods &optional auxv)
     (,(call-next-method gf method
         `(lambda (&rest args)
            (let ((next-auxv nil))
              (labels ((find-next-method ()
                         (if remaining-methods
                             (multiple-value-bind (applies? av)
                                 (let ((pq (pq-of (car remaining-methods))))
                                   (evaluate-predicate
                                     (predicate-of pq) args
                                     (make-array (auxv-count-of pq)
                                                 :initial-element nil)))
                               (cond (applies? (setf next-auxv av)
                                               t)
                                     (t (pop remaining-methods)
                                        (find-next-method))))
                             nil)))
                (cond (auxv
```

```
                                (apply #'(lambda ,(cadr lambda-expression)
                                           (labels ((next-method-p ()
                                                      (or next-auxv (find-next-method)))
                                                    (call-next-method (&rest cnm-args)
                                                      (unless (next-method-p)
                                                        (error "No next method for ~A."
                                                               ',(generic-function-name
                                                                  gf)))
                                                      (funcall (method-function
                                                                (car remaining-methods))
                                                               (or cnm-args args)
                                                               (cdr remaining-methods)
                                                               next-auxv)))
                                             ,@(cddr lambda-expression)))
                                         args))
                               ((find-next-method)
                                (funcall (method-function (car remaining-methods))
                                         args (cdr remaining-methods) next-auxv))
                               (t (error "No applicable method for ~A on ~S."
                                         ',(generic-function-name gf) args))))))
                    environment)
            args (cdr remaining-methods))))

(defmethod compute-effective-method ((gf gf-with-predicate-dispatching)
                                     method-combination
                                     methods)
  (declare (ignore method-combination))
  `(call-method ,(car methods) ,methods))

(provide 'predicate-dispatch)
```

A.3 predicate-classes.lisp

```
(require 'pd-package)
(in-package :predicate-dispatch)

;; proposed schema for normalized predicates:
;;
;; full multipredicate = PCM | or(PCM{2,}) | constant
;; full unary predicate [predicate class] = PCU | or(PCU{2,}) | constant
;;
;; PCM = SM | and(SM{2,})
;;  SM = test | not(test) | projected-unary(PCU)
;; PCU = PNU | and(PNU{2,})
;; PNU = PEU | not(PEU)
```

```
;; PEU = SU | extracting-unary(SU)
;;  SU = typecheck | equality | test
;;
;; abbreviations stand for (Purely Conjunctive)/Simple Multi-arg/Unary and
;; Possibly Negated/Extracting Unary.

(defclass predicate () ; abstract
    ((normal? :initarg :normal?
              :reader normal?
              :initform nil)))

(defclass test-predicate (predicate)
    ((test :initarg :test
           :reader test-of)
     (pass-auxv :initarg :pass-auxv
                :reader pass-auxv?
                :initform t)))

(defclass constant-predicate (predicate)
    ((value :initarg :value
            :reader value-of)))

(defclass typecheck-predicate (predicate)
    ((target-type :initarg :target
                  :reader target-of)))

(defclass equality-predicate (predicate)
    ((target-value :initarg :target
                   :reader target-of)))

(defclass modified-predicate (predicate) ; abstract
    ((base-predicate :initarg :base
                     :reader base-of)))

(defclass projected-unary-predicate (modified-predicate)
    ((argument-index :initarg :index
                     :reader index-of)))

(defclass extracting-unary-predicate (modified-predicate)
    ((accessor-chain :initarg :accessors
                     :reader accessors-of)))

(defclass not-predicate (modified-predicate)
    ())

(defclass compound-predicate (predicate) ; abstract
    ((subpreds :initarg :subpreds
```

```
                :reader subpreds-of)))

(defclass and-predicate (compound-predicate)
    ())

(defclass or-predicate (compound-predicate)
    ())

(defclass predicate-qualifier ()
    ((predicate :initarg :predicate
                :reader predicate-of)
     (auxv-count :initarg :auxv-count
                 :reader auxv-count-of)))

(defun predicate-qualifier? (x)
    (typep x 'predicate-qualifier))

(defconstant *true* (make-instance 'constant-predicate :value t))
(defconstant *false* (make-instance 'constant-predicate :value nil))

(defmethod evaluate-predicate ((predicate test-predicate) args auxv)
    (values (apply (test-of predicate)
                   (if (pass-auxv? predicate) (cons auxv args) args))
            auxv))

(defmethod evaluate-predicate ((predicate constant-predicate) args auxv)
    (declare (ignore args))
    (values (value-of predicate) auxv))

(defmethod evaluate-predicate ((predicate typecheck-predicate) args auxv)
    (values (typep (car args) (target-of predicate)) auxv))

(defmethod evaluate-predicate ((predicate equality-predicate) args auxv)
    (values (eql (car args) (target-of predicate)) auxv))

(defmethod evaluate-predicate ((predicate projected-unary-predicate) args auxv)
    (evaluate-predicate (base-of predicate)
                        (list (nth (index-of predicate) args)) auxv))

(defmethod evaluate-predicate ((predicate extracting-unary-predicate) args
                               auxv)
    (do ((arg (car args) (funcall (car accessors) arg))
         (accessors (accessors-of predicate) (cdr accessors)))
        ((null accessors) (evaluate-predicate (base-of predicate)
                                              (list arg) auxv))))

(defmethod evaluate-predicate ((predicate and-predicate) args auxv)
```

```
    (values (every #'(lambda (subpred) (evaluate-predicate subpred args auxv))
                   (subpreds-of predicate))
            auxv))

(defmethod evaluate-predicate ((predicate or-predicate) args auxv)
    (values (some #'(lambda (subpred) (evaluate-predicate subpred args
                                                          ;(copy-seq auxv)
                                                          auxv))
                  (subpreds-of predicate))
            auxv))

(defmethod evaluate-predicate ((predicate not-predicate) args auxv)
    (values (not (evaluate-predicate (base-of predicate) args auxv)) auxv))

(defun make-or (subpreds &optional normal?)
    (make-instance 'or-predicate :subpreds subpreds :normal? normal?))

(defun make-and (subpreds &optional normal?)
    (make-instance 'and-predicate :subpreds subpreds :normal? normal?))

(defun make-not (base &optional normal?)
    (make-instance 'not-predicate :base base :normal? normal?))

(defun constant-predicate? (predicate)
    (typep predicate 'constant-predicate))

(defun or-predicate? (predicate)
    (typep predicate 'or-predicate))

(defmethod normalize-predicate (predicate)
    predicate)

(defmethod normalize-predicate ((predicate not-predicate))
    (if (normal? predicate)
        predicate
        (normalize-predicate
          (let ((base (normalize-predicate (base-of predicate))))
            (typecase base
                (not-predicate (base-of base))
                (and-predicate (make-or (mapcar #'make-not (subpreds-of base))))
                (or-predicate (make-and (mapcar #'make-not (subpreds-of base))))
                (projected-unary-predicate (make-instance 'projected-unary-predicate
                                                          :index (index-of base)
                                                          :base (make-not (base-of
                                                                           base))))
                (constant-predicate (make-instance 'constant-predicate
                                                   :value (not (value-of base))))
```

```
                (otherwise (make-not base t)))))))

(require 'normalize-or) ;; split off due to size

(defun flattened-and-subpredicates (p)
  (if (typep p 'and-predicate)
      (apply #'append (mapcar #'flattened-and-subpredicates (subpreds-of p)))
      (list p)))

(defun safe-index-of (p)
  (and (typep p 'projected-unary-predicate)
       (index-of p)))

(defmethod normalize-predicate ((predicate and-predicate))
  (cond ((normal? predicate) predicate)
        ((null (subpreds-of predicate)) *true*)
        ((null (cdr (subpreds-of predicate)))
         (normalize-predicate (car (subpreds-of predicate))))
        (t (let ((subpreds (mapcar #'normalize-predicate
                                   (flattened-and-subpredicates predicate))))
             (let ((first-or (find-if #'or-predicate? subpreds)))
               (if first-or ;; distribute!
                   (normalize-predicate
                    (make-or
                     (mapcar #'(lambda (x)
                                 (make-and
                                  (substitute x first-or subpreds :count 1)))
                             (subpreds-of first-or))))
                   (do ((ht (make-hash-table))
                        (rsp (reverse subpreds) (cdr rsp)))
                       ((null rsp)
                        (let ((terms (gethash nil ht)))
                          (maphash #'(lambda (index preds)
                                       (if index
                                           (push (make-instance
                                                  'projected-unary-predicate
                                                  :base (normalize-predicate
                                                         (make-and preds))
                                                  :index index)
                                                 terms)))
                                   ht)
                          (make-and terms t)))
                     (if (typep (car rsp) 'projected-unary-predicate)
                         (push (base-of (car rsp))
                               (gethash (index-of (car rsp)) ht))
                         (push (car rsp) (gethash nil ht)))))))))))
```

```
(defmethod normalize-predicate ((predicate projected-unary-predicate))
    (if (normal? predicate)
        predicate
        (flet ((make-cousin (new-base &optional normal?)
                            (make-instance 'projected-unary-predicate
                                           :base new-base
                                           :index (index-of predicate)
                                           :normal? normal?)))
            (let ((base (normalize-predicate (base-of predicate))))
                (typecase base
                    (or-predicate (normalize-predicate
                                    (make-or (mapcar #'make-cousin (subpreds-of base)))))
                    (constant-predicate base)
                    (otherwise (make-cousin base t)))))))

(defmethod normalize-predicate ((predicate extracting-unary-predicate))
    (if (normal? predicate)
        predicate
        (normalize-predicate
          (flet ((make-cousin (new-base &optional normal?)
                              (make-instance 'extracting-unary-predicate
                                             :base new-base
                                             :accessors (accessors-of predicate)
                                             :normal? normal?)))
            (let ((base (normalize-predicate (base-of predicate))))
                (typecase base
                    (or-predicate (make-or (mapcar #'make-cousin (subpreds-of base))))
                    (and-predicate (make-and (mapcar #'make-cousin (subpreds-of base))))
                    (constant-predicate base)
                    (not-predicate (make-not (make-cousin (base-of base))))
                    (otherwise (make-cousin base t)))))))))

(defmethod implies? (pred1 pred2)
    ;; takes advantage of simplification done in normalize-or.lisp
    (let ((norm (normalize-predicate (make-or (make-not pred1) pred2))))
        (and (constant-predicate? norm) (value-of norm))))

(provide 'predicate-classes)
```

A.4 normalize-or.lisp

```
(require 'pd-package)
(in-package :predicate-dispatch)

;; Abstract transformations:
;; * p v ~p -> *true*
```

```
;; * (p+ ^ q*) v p+ -> p+
;; * (p ^ q+) v (~p ^ q+) -> q+
;; * (p ^ q* ^ r+) v (~p ^ q*) -> (q* ^ r+) v (~p ^ q*)
;; * (p ^ q* ^ r+) v (~p ^ q* ^ s+)
;;   -> (q* ^ r+ ^ s+) v (p ^ q* ^ r+) v (~p ^ q* ^ s+)
;;   [may be useful in setting the stage for other transformations]

;; prove complete?

;; interesting test cases:
;; * (p ^ q) v (p ^ ~q) v (~p ^ q) v (~p ^ ~q)
;; * (p ^ q ^ r) v ~p v ~q v ~r
;; * (p ^ q) v (~q ^ r) v (~r ^ ~p) v (p ^ ~q ^ ~r) v (~p ^ q ^ r)

(defun flattened-or-subpredicates (p)
    (if (typep p 'or-predicate)
        (apply #'append (mapcar #'flattened-or-subpredicates (subpreds-of p)))
        (list p)))

(defun extra-flattened-and-subpredicates (p)
    (cond ((typep p 'and-predicate)
             (apply #'append (mapcar #'extra-flattened-and-subpredicates
                                     (subpreds-of p))))
          ((and (typep p 'projected-unary-predicate)
                (typep (base-of p) 'and-predicate))
           (mapcar #'(lambda (x) (make-instance 'projected-unary-predicate
                                                :base x :index (index-of p)))
                   (subpreds-of (base-of p))))
          (t (list p))))

(defun maybe-make-eup (base accessors)
    (if accessors (make-instance 'extracting-unary-predicate
                                 :base base :accessors accessors)
        base))

(defun maybe-flip (orig pairs flip?)
    (if flip?
        (or (cdr (assoc orig pairs))
            (car (rassoc orig pairs)))
        orig))

(defun adjust-comparison (orig neg-x neg-y)
    (maybe-flip (maybe-flip orig
                            '((same . opposite)
                              (forward . comprehensive)
                              (backward . exclusive))
```

```
                    neg-x)
                '((same . opposite)
                  (forward . exclusive)
                  (backward . comprehensive))
                neg-y))

(defun compare-terms (x y &optional neg-x neg-y)
  (adjust-comparison
    (cond ((typep x 'projected-unary-predicate)
           (and (typep y 'projected-unary-predicate)
                (= (index-of x) (index-of y))
                (compare-terms (base-of x) (base-of y) neg-x neg-y)))
          ((typep y 'projected-unary-predicate) nil)
          ;;
          ((and (typep x 'equality-predicate)
                (typep y 'equality-predicate))
           (if (eql (target-of x) (target-of y)) 'same 'exclusive))
          ;; no appropriate auxv...pass nil instead, and ignore errors.
          ((typep x 'equality-predicate)
           (ignore-errors
             (if (evaluate-predicate y (target-of x) nil)
                 'forward
                 'exclusive)))
          ((typep y 'equality-predicate)
           (ignore-errors
             (if (evaluate-predicate x (target-of y) nil)
                 'backward
                 'exclusive)))
          ;;
          ((typep x 'not-predicate)
           (compare-terms (base-of x) y (not neg-x) neg-y))
          ((typep y 'not-predicate)
           (compare-terms x (base-of y) neg-x (not neg-y)))
          ;;
          ((and (typep x 'extracting-unary-predicate)
                (typep y 'extracting-unary-predicate))
           (do ((x-acc (accessors-of x) (cdr x-acc))
                (y-acc (accessors-of y) (cdr y-acc)))
               ((or (null x-acc) (null y-acc)
                    (not (eql (car x-acc) (car y-acc))))
                (and (or (null x-acc) (null y-acc))
                     (compare-terms (maybe-make-eup (base-of x) x-acc)
                                    (maybe-make-eup (base-of y) y-acc)
                                    neg-x neg-y)))))
          ;;
```

```
            ((and (typep x 'typecheck-predicate)
                  (typep y 'typecheck-predicate))
              (let ((tx (target-of x))
                    (ty (target-of y)))
                ;; In a more static environment, we'd want to compare the sets
                ;; of concrete subtypes.  As it is, we have to be conservative.
                (cond ((eql tx ty) 'same)
                      ((subtypep tx ty) 'forward)
                      ((subtypep ty tx) 'backward)
                      (t nil))))
            ;;
            ((and (typep x 'test-predicate)
                  (typep y 'test-predicate))
              (if (eql (test-of x) (test-of y))
                  'same
                  nil))
            ;;
            (t nil))
      neg-x neg-y))

(defun analyze-term (x yy)
  (let ((comparisons (remove nil (mapcar #'(lambda (y) (compare-terms x y))
                                         yy))))
    (cond ((null comparisons) 'extra)
          ((member 'exclusive comparisons) 'fatal-mismatch)
          ((member 'opposite comparisons) 'mismatch)
          ((member 'same comparisons) 'match)
          ;; ((member 'forward comparisons) 'match)
          ((member 'backward comparisons) 'weak)
          ;; "comprehensive" and "forward" don't help us.
          (t 'extra))))

(defun annotate-term (x yy)
  (cons x (analyze-term x yy)))

(defun keep? (x y)
  (or (member 'extra y :key #'cdr)
      (member 'weak x :key #'cdr)))

(defun compute-safe-patch (left right)
  ;; Return values:
  ;; (1) Safe patch if appropriate (single mismatch), nil otherwise.
  ;; (2) Does left cover any extra territory?
  ;; (3) Does right?
  (let* ((fleft (extra-flattened-and-subpredicates left))
         (fright (extra-flattened-and-subpredicates right))
```

```
        (xleft (mapcar #'(lambda (l) (annotate-term l fright)) fleft))
        (mismatches (count 'mismatch xleft :key #'cdr)))
    (if (or (member 'fatal-mismatch xleft :key #'cdr)
            (> mismatches 1))
        (values nil t t) ;; We lose.
        (let* ((xright (mapcar #'(lambda (r) (annotate-term r fleft)) fright))
               (keep-right (keep? xright xleft)))
          (if (zerop mismatches)
              (values nil
                      (or (not keep-right)
                          (keep? xleft xright))
                      keep-right)
              (values
               ;; Merge useful terms into a patch.  Keep weak terms on the left
               ;; despite their redundancy because something may depend on them
               ;; wrt short-circuiting.
               ;; Return t instead of an empty patch.
               (or (mapcar #'car
                           (delete 'mismatch
                                   (nconc xleft
                                          (delete-if
                                           #'(lambda (x)
                                               (member (cdr x) '(match weak)))
                                           xright))
                                   :key #'cdr))
                   t)
               (keep? xleft xright)
               keep-right))))))

(defmethod normalize-predicate ((predicate or-predicate))
  (cond ((normal? predicate) predicate)
        ((null (subpreds-of predicate)) *false*)
        (t
         (let ((subpreds (mapcar #'normalize-predicate
                                 (flattened-or-subpredicates predicate))))
           (if (some #'value-of (remove-if-not #'constant-predicate? subpreds))
               *true*
               ;; can mutate here because nothing else refers to subpreds.
               (do ((queue (delete-if #'constant-predicate? subpreds)
                           (cdr queue))
                    (new-subpreds nil (if insert-head
                                          (cons (car queue) survivors)
                                          survivors))
                    (survivors nil nil)
                    (insert-head t (and (not insert-head) (null survivors))))
```

```
                ((null queue) (if (cdr new-subpreds)
                                  (make-or new-subpreds t)
                                  (car new-subpreds)))
                ;; (format t "Queue:~:W~%NewS: ~:W~%" queue new-subpreds)
                (dolist (subpred new-subpreds)
                  (multiple-value-bind (patch keep-left keep-right)
                      (compute-safe-patch (car queue) subpred)
                    ;; (format t "Patch:~:W~%KeepL:~:W~%KeepR:~:W~%" patch
                    ;;  keep-left keep-right)
                    (if patch
                        (if (eq patch t)
                            (return-from normalize-predicate *true*);; we win!
                            (push (normalize-predicate (make-and patch))
                                  (cdr queue))))
                    (if keep-left
                        (setf insert-head t))
                    (if keep-right
                        (push subpred survivors)))))))))
```

```
(provide 'normalize-or)
```

A.5 pc-build.lisp

```
(require 'pd-package)
(in-package :predicate-dispatch)

(require 'predicate-classes)
(require 'xcond)

;; I'd love to use macroexpand-all from the walker bundled with PCL,
;; but it's not completely safe.  (Tagbody's tags *cannot* result from
;; macro expansion.)

(defun build-predicate (lambda-list predicate-bodies)
  (multiple-value-bind (ll-analysis ll-predicates auxv-count)
      (analyze-lambda-list-for-analysis lambda-list)
    (make-instance 'predicate-qualifier
                   :predicate (normalize-predicate
                               (make-and
                                (append ll-predicates
                                        (mapcar #'(lambda (body)
                                                    (build-pred-internal
                                                     (analyze-expr body
                                                                   ll-analysis)
                                                     lambda-list))
                                                predicate-bodies))))))
```

```
                                :auxv-count auxv-count)))

(defclass expression-analysis ()
    ((arguments-used :initarg :args-used
                     :reader args-used)))

(defclass trivial-analysis (expression-analysis)
    ((code :initarg :code
           :reader code-of)
     (substitute? :initarg :substitute?
                  :reader substitute?
                  :initform nil)))

(defclass manifest-constant (expression-analysis)
    ((value :initarg :value
            :reader value-of)
     (arguments-used :initform nil)))

(defclass extraction-or-simple-test (expression-analysis)
    ((argument-index :initarg :index
                     :reader index-of)
     (unary-chain :initarg :chain
                  :reader chain-of)))

(defclass extraction-analysis (extraction-or-simple-test)
    ())

(defclass typecheck-analysis (extraction-or-simple-test)
    ((target-type :initarg :target
                  :reader target-of)))

(defclass eql-analysis (extraction-or-simple-test)
    ((target-value :initarg :target
                   :reader target-of)))

(defclass not-analysis (expression-analysis)
    ((base-analysis :initarg :base
                    :reader base-of)))

(defclass compound-analysis (expression-analysis)
    ((terms :initarg :terms
            :reader terms-of)))

(defclass and-analysis (compound-analysis)
    ())

(defclass or-analysis (compound-analysis)
```

```
  ())

(defun mkxtriv (analysis var code)
  (make-instance 'trivial-analysis
                 :args-used (adjoin var (args-used analysis))
                 :code code))

(defun mktriventry (arg)
  (cons arg (make-instance 'trivial-analysis :args-used (list arg) :code arg)))

(defun null-triv (expr)
  (make-instance 'trivial-analysis :args-used nil :code expr))

(defun car-or-identity (x)
  (if (consp x)
      (car x)
      x))

(defun extract-arg-name (ll-term key?)
  (cond ((atom ll-term) ll-term)
        ((and (consp (car ll-term)) key?) (second ll-term))
        (t (car ll-term))))

(defun analyze-lambda-list-for-analysis (lambda-list)
  (do ((ll lambda-list (cdr ll))
       (index 0 (1+ index))
       (partial-result (list (mktriventry 'auxv)))
       (ll-predicates nil)
       (auxv-count 0)
       (last-key nil))
      ((null ll) (values partial-result (reverse ll-predicates) auxv-count))
    (let* ((carll (car ll))
           (arg (extract-arg-name carll (eq last-key '&key))))
      (assert (symbolp arg))
      (cond ((eq carll '&allow-other-keys) nil) ; do nothing
            ((member carll '(&optional &rest &key &aux)) (setf last-key carll))
            ((member carll lambda-list-keywords)
             (error "Unsupported lambda list keyword ~S" carll))
            ((null last-key)
             (push (cons arg (make-instance 'extraction-analysis
                                            :index index :chain nil
                                            :args-used (list arg)))
                   partial-result)
             (if (and (consp carll) (cdr carll))
                 (push (build-pred-internal
                        (analyze-specializer (second carll) arg index
```

```
                                                    partial-result)
                                lambda-list)
                              ll-predicates)))
                    ((eq last-key '&rest) (push (mktriventry arg) partial-result))
                    ((eq last-key '&aux)
                      (if (consp carll)
                          (push (build-pred-internal
                                  (analyze-expr `(progn (setf (aref auxv ,auxv-count)
                                                              ,(second carll))
                                                        t)
                                                                partial-result)
                                lambda-list)
                              ll-predicates))
                      (push (cons arg (make-instance 'trivial-analysis
                                                     :args-used `(auxv ,arg)
                                                     :code `(aref auxv ,auxv-count)
                                                     :substitute? t))
                            partial-result)
                      (incf auxv-count))
                    (t ; &optional or &key
                      (if (or (atom carll) (null (cdr carll)))
                          (push (mktriventry arg) partial-result)
                           (let ((analysis (analyze-expr (second carll) partial-result)))
                               (push (cons arg (mkxtriv analysis arg
                                                        `(or ,arg ,(second carll))))
                                     partial-result)
                               (if (cddr carll)
                                   (push (mktriventry (third carll)) partial-result)))))))))

(defun analyze-specializer (specializer arg index ll-analysis)
    (cond ((symbolp specializer) (make-instance 'typecheck-analysis
                                                :index index :chain nil
                                                :target specializer
                                                :args-used (list arg)))
          ((and (listp specializer)
                (= (length specializer) 2)
                (eq (car specializer) 'eql))
            (analyze-expr `(eql ,arg ,(cadr specializer)) ll-analysis))
          (t (error "Unsupported specializer ~S for ~S" specializer arg))))

(defun analyze-expr (expr ll-analysis &optional macros symbol-macros)
    (multiple-value-bind (expansion expanded?)
           (macroexpand-1 expr)
       (xcond ((assoc expr symbol-macros)
               => #'(lambda (a) (analyze-expr (cdr a) ll-analysis macros
```

```
                                                symbol-macros)))
                ((and (consp expr)
                      (assoc (car expr) macros))
                 => #'(lambda (a) (analyze-expr (apply (cdr a) (cdr expr))
                                                ll-analysis macros symbol-macros)))
                ((and expanded? (or (atom expr)
                                    (not (or (special-operator-p (car expr))
                                             (member (car expr)
                                                     '(and eql not or typep))))))
                 (analyze-expr expansion ll-analysis macros symbol-macros))
                ((assoc expr ll-analysis) => #'cdr)
                ((constantp expr) (make-instance 'manifest-constant
                                                 :value (eval expr)))
                ((atom expr) (null-triv expr))
                ((consp (car expr));; must be a lambda form
                 (analyze-exprs expr ll-analysis macros symbol-macros))
                ((assoc (car expr) *analysis-helpers*)
                 => #'(lambda (a) (funcall (cdr a) expr ll-analysis macros
                                           symbol-macros)))
                (t (let* ((aa (mapcar #'(lambda (x) (analyze-expr x ll-analysis
                                                                  macros
                                                                  symbol-macros))
                                      (cdr expr)))
                          (aa1 (car aa)))
                     (if (or (cdr aa)
                             (special-operator-p (car expr))
                             (not (typep aa1 'extraction-analysis)))
                         (make-instance 'trivial-analysis :code expr
                                        :args-used
                                        (reduce #'union aa :key #'args-used))
                         (make-instance 'extraction-analysis
                                        :index (index-of aa1)
                                        :chain (cons (symbol-function (car expr))
                                                     (chain-of aa1))
                                        :args-used (args-used aa1)))))))))

(defun analyze-exprs (ee l m s &optional whole (force-trivial? t))
  (if (or force-trivial? (cdr ee))
      (make-instance 'trivial-analysis
                     :args-used (reduce #'union ee
                                        :key #'(lambda (e)
                                                 (args-used (analyze-expr
                                                             e l m s))))
                     :code whole)
      (analyze-expr (car ee) l m s)))
```

```
;; force trivial analysis
(defun analyze-first-arg (e l m s)
    (analyze-exprs (list (second e)) l m s e))

(defun analyze-second-arg (e l m s)
    (analyze-exprs (list (third e)) l m s e))

(defun analyze-block (e l m s)
    (analyze-exprs (cddr e) l m s e))

(defun analyze-eval-when (e l m s)
    (analyze-exprs (cddr e) l m s e))

(defun analyze-flet/labels (e l m s)
    (let* ((function-names (mapcar #'car (second e)))
           (m2 (remove-if #'(lambda (x) (member (car x) function-names)) m))
           (m-for-functions (case (car e)
                              (flet m)
                              (labels m2)))
           (functions-analysis (analyze-exprs (mapcar #'(lambda (x)
                                                          `(lambda ,@(cdr x)))
                                                      (second e))
                                              l m-for-functions s))
           (body-analysis (analyze-locally `(locally ,@(cddr e)) l m2 s)))
      (make-instance 'trivial-analysis
                     :args-used (union (args-used functions-analysis)
                                       (args-used body-analysis))
                     :code e)))

(defun analyze-function (e l m s)
    (if (and (consp (cadr e))
             (eq (caadr e) 'lambda))
        (analyze-lambda (cadr e) l m s)
        (null-triv e)))

(defun analyze-lambda (e l m s)
    (let ((new-bound-vars (mapcar #'car (analyze-lambda-list-for-analysis
                                          (second e)))))
      (flet ((shadowed? (a) (member (car a) new-bound-vars)))
        (analyze-locally (if (stringp (third e))
                             (cdddr e)
                             (cddr e))
                         (remove-if #'shadowed? l)
                         m
                         (remove-if #'shadowed? s)
                         e))))
```

```
(defun analyze-let/let* (e l m s)
  (let ((new-bound-vars (mapcar #'car-or-identity (second e))))
    (flet ((shadowed? (a) (member (car a) new-bound-vars))
           (for-bindings (x x*) (case (car e)
                                  (let x)
                                  (let* x*))))
      (let* ((l2 (remove-if #'shadowed? l))
             (s2 (remove-if #'shadowed? s))
             (bindings-analysis (analyze-exprs
                                 (apply #'append
                                        (mapcar #'cdr
                                                (remove-if #'atom (second e))))
                                 (for-bindings l l2) m (for-bindings s s2)))
             (body-analysis (analyze-locally `(locally ,@(cddr e)) l2 m s2)))
        (make-instance 'trivial-analysis
                       :args-used (union (args-used bindings-analysis)
                                         (args-used body-analysis))
                       :code e)))))

(defun analyze-locally (e l m s &optional (whole e) (force-trivial? t))
  ;; deal with "special" declarations
  (do ((exprs (cdr e) (cdr exprs))
       (specials nil)
       (l2 l)
       (s2 s))
      ((or (null exprs)
           (atom (car exprs))
           (not (eq (caar exprs) 'declare)))
       (flet ((shadowed? (a) (member (car a) specials)))
         (analyze-exprs exprs
                        (remove-if #'shadowed? l2)
                        m
                        (remove-if #'shadowed? s2)
                        whole
                        force-trivial?)))
    (dolist (decl (cdar exprs))
      (if (eq (car decl) 'special)
          (mapc #'(lambda (v) (pushnew v specials))
                (cdr decl))))))

(defun analyze-macrolet (e l m s)
  ;; loses on &environment.  Eit.
  (do ((m2 m)
       (clauses (second e) (cdr clauses)))
      ((null clauses) (analyze-locally `(locally ,@(cddr e)) l m2 s e nil))
```

```
        (let* ((clause (car clauses))
               (name (first clause))
               (ll (second clause))
               (body (cddr clause))
               (qargs (gensym)))
          (push (cons name (eval `(lambda (&rest ,qargs)
                                    (destructuring-bind ,(cons (gensym) ll)
                                        (cons ,name ,qargs)
                                      ,@body))))
                m2))))

(defun analyze-setq (e l m s)
  (labels ((every-other (l)
             (if (and (consp l) (consp (cdr l)))
                 (cons (cadr l) (every-other (cddr l)))
                 nil)))
    (analyze-exprs (every-other (cdr e)) l m s) e))

(defun analyze-symbol-macrolet (e l m s)
  (do ((s2 s)
       (clauses (second e) (cdr clauses)))
      ((null clauses) (analyze-locally `(locally ,@(cddr e))
                                       (set-difference l s2 :key #'car)
                                       m s2 e nil))
    (let* ((clause (car clauses))
           (name (first clause))
           (expansion (second clause)))
      (push (cons name expansion) s2))))

(defun analyze-tagbody (e l m s)
  (analyze-exprs (remove-if #'atom (cdr e)) l m s e))

;;;

(defun analyze-and/or (e l m s)
  (let ((subanalyses (mapcar #'(lambda (x) (analyze-expr x l m s))
                             (cdr e))))
    (make-instance (case (car e)
                     (and 'and-analysis)
                     (or 'or-analysis))
                   :terms subanalyses
                   :args-used (reduce #'union subanalyses :key #'args-used))))

(defun analyze-eql (e l m s)
  (flet ((make-eql (ea mc)
           (make-instance 'eql-analysis
```

```
                                        :index (index-of ea)
                                        :chain (chain-of ea)
                                        :args-used (args-used ea)
                                        :target (value-of mc))))
        (let* ((subanalyses (mapcar #'(lambda (x) (analyze-expr x l m s))
                                    (cdr e)))
               (types (mapcar #'type-of subanalyses)))
          (cond ((equal types '(extraction-analysis manifest-constant))
                 (make-eql (first subanalyses) (second subanalyses)))
                ((equal types '(manifest-constant extraction-analysis))
                 (make-eql (second subanalyses) (first subanalyses)))
                (t (make-instance 'trivial-analysis
                                  :args-used (reduce #'union subanalyses
                                                     :key #'args-used)
                                  :code e))))))

(defun analyze-not (e l m s)
  (let ((base (analyze-expr (second e) l m s)))
    (if (typep base 'extraction-analysis)
        (make-instance 'extraction-analysis
                       :index (index-of base)
                       :chain (cons #'not (chain-of base))
                       :args-used (args-used base))
        (make-instance 'not-analysis
                       :base base))))

(defun analyze-typep (e l m s)
  (let ((subanalyses (mapcar #'(lambda (x) (analyze-expr x l m s))
                             (cdr e))))
    (if (and (typep (first subanalyses) 'extraction-analysis)
             (typep (second subanalyses) 'manifest-constant))
        (make-instance 'typecheck-analysis
                       :index (index-of (first subanalyses))
                       :chain (chain-of (first subanalyses))
                       :args-used (args-used (first subanalyses))
                       :target (value-of (second subanalyses)))
        (make-instance 'trivial-analysis
                       :args-used (reduce #'union subanalyses
                                          :key #'args-used)
                       :code e))))

(defconstant *analysis-helpers*
  ;; omitted (potentially evaluate any "argument"):
  ;;   catch, if, multiple-value-{call,prog1}, progn, progv, throw,
  ;;   unwind-protect
  ;; quote is also omitted because constantp takes care of it.
```

```
    `((block . ,#'analyze-block)
      (eval-when . ,#'analyze-eval-when)
      (flet . ,#'analyze-flet/labels)
      (function . ,#'analyze-function)
      (go . ,#'(lambda (e l m s) (declare (ignore l m s)) (null-triv e)))
      (labels . ,#'analyze-flet/labels)
      (lambda . ,#'analyze-lambda)
      (let . ,#'analyze-let/let*)
      (let* . ,#'analyze-let/let*)
      (load-time-value . ,#'analyze-first-arg)
      (locally . ,#'analyze-locally)
      (macrolet . ,#'analyze-macrolet)
      (return-from . ,#'analyze-second-arg)
      (setq . ,#'analyze-setq)
      (symbol-macrolet . ,#'analyze-symbol-macrolet)
      (tagbody . ,#'analyze-tagbody)
      (the . ,#'analyze-second-arg)
      ;;
      (and . ,#'analyze-and/or)
      (eql . ,#'analyze-eql)
      (not . ,#'analyze-not)
      (or . ,#'analyze-and/or)
      (typep . ,#'analyze-typep)))

(defmethod build-pred-internal ((analysis trivial-analysis) lambda-list)
  (do ((ll (cons 'auxv lambda-list) (cdr ll))
       (used (args-used analysis))
       (code (code-of analysis))
       (index -1 (1+ index)) ;; starts at -1 to account for auxv.
       (auxv-count 0)
       rnorm ropt rest rkey ignores last-ll-key remaining-optionals smlcl)
      ((null ll) (if (and (not rest) (null rkey))
                     (setf rest (gensym)
                           ignores (cons rest ignores)))
       (make-instance 'test-predicate
                      :test (eval `(lambda (,@(reverse rnorm)
                                            ,@(and ropt
                                                   (cons '&optional
                                                         (reverse ropt)))
                                            ,@(and rest `(&rest ,rest))
                                            ,@(and rkey `(&key
                                                          ,@(reverse rkey)
                                                          &allow-other-keys)))
                                    ,@(and ignores
                                           `((declare (ignore ,@ignores))))
```

```
                                        ,(if smlcl
                                             `(symbol-macrolet ,smlcl ,code)
                                             code)))))
        (let* ((carll (car ll))
               (arg (extract-arg-name carll (eq last-ll-key '&key)))
               (used? (member arg used))
               (arg2 (and (consp carll) (third carll)))
               (used2? (member arg2 used)))
          (cond ((eq carll '&allow-other-keys) nil) ; do nothing (always on)
                ((member carll '(&optional &rest &key &aux))
                 (setf last-ll-key carll)
                 (if (eq carll '&optional)
                     (setf remaining-optionals (cdr ll))))
                ((member carll lambda-list-keywords)
                 (error "Unsupported lambda list keyword ~S" carll))
                ((and (null last-ll-key) (null (cdr used)) (eq arg (car used))
                      (>= index 0))
                 (return-from build-pred-internal
                     (make-instance 'projected-unary-predicate
                                    :index index
                                    :base (make-instance
                                           'test-predicate
                                           :test (eval `(lambda (,arg)
                                                                ,(code-of analysis)))
                                           :pass-auxv nil))))
                ((null last-ll-key) (push arg rnorm) ; don't want specializer!
                                    (if (not used?) (push arg ignores)))
                ((and (eq last-ll-key '&rest) used?) (setf rest arg))
                ((eq last-ll-key '&aux)
                 (if used? (push `(,arg (aref auxv ,auxv-count)) smlcl))
                 (incf auxv-count))
                ((and (eq last-ll-key '&key) (or used? used2?))
                 (if used2? (progn (push carll rkey)
                                   (if (not used?) (push arg ignores)))
                     (push (list arg (second carll)) rkey)))
                ((and (eq last-ll-key '&optional) (or used? used2?))
                 (mapc #'(lambda (x) (push (car-or-identity x) ropt))
                       (ldiff remaining-optionals ll))
                 (setf remaining-optionals (cdr ll))
                 (if used2? (progn (push carll ropt)
                                   (if (not used?) (push arg ignores)))
                     (push (list arg (second carll)) ropt)))))))

(defmethod build-pred-internal ((analysis manifest-constant) ll)
  (declare (ignore ll))
```

```
    (make-instance 'constant-predicate :value (value-of analysis)))

(defmethod build-pred-internal ((analysis extraction-analysis) ll)
    (declare (ignore ll))
    (let* ((chain0 (chain-of analysis))
           (not? (eql (car chain0) #'not))
           (chain (if not? (cdr chain0) chain0))
           (test (make-instance 'test-predicate
                                :test (or (car chain) #'identity)
                                :pass-auxv nil))
           (subbase (cond ((null chain) test)
                          ((null (cdr chain)) test)
                          (t (make-instance 'extracting-unary-predicate
                                            :accessors (cdr chain)
                                            :base test))))
           (base (make-instance 'projected-unary-predicate
                                :index (index-of analysis)
                                :base subbase)))
        (if not?
            (make-instance 'not-predicate :base base)
            base)))

(defmethod build-pred-internal ((analysis extraction-or-simple-test) ll)
    (declare (ignore ll))
    (let ((base (make-instance (if (typep analysis 'typecheck-analysis)
                                   'typecheck-predicate
                                   'equality-predicate)
                               :target (target-of analysis)))
          (chain (chain-of analysis)))
        (make-instance 'projected-unary-predicate
                       :index (index-of analysis)
                       :base (if chain
                                 (make-instance 'extracting-unary-predicate
                                                :accessors chain :base base)
                                 base))))

(defmethod build-pred-internal ((analysis not-analysis) ll)
    (make-not (build-pred-internal (base-of analysis) ll)))

(defmethod build-pred-internal ((analysis compound-analysis) ll)
    (funcall (typecase analysis
                (and-analysis #'make-and)
                (or-analysis #'make-or))
             (mapcar #'(lambda (a) (build-pred-internal a ll))
                     (terms-of analysis))))
```

```
(provide 'pc-build)
```

A.6 xcond.lisp

```
(require 'pd-package)
(in-package :predicate-dispatch)

;; supports => a la Scheme cond.
(defmacro xcond (&rest forms)
    (and forms
         (let ((form (car forms)))
             (cond ((atom form) (error "Bad cond form ~A" form))
                   ((null (cdr form)) (let ((fresh-sym (gensym)))
                                        `(let ((,fresh-sym ,(car form)))
                                           (if ,fresh-sym
                                               ,fresh-sym
                                               (xcond ,@(cdr forms))))))
                   ((and (= (length form) 3) (eq (second form) '=>))
                    (let ((fresh-sym (gensym)))
                      `(let ((,fresh-sym ,(car form)))
                         (if ,fresh-sym
                             (funcall ,(third form) ,fresh-sym)
                             (xcond ,@(cdr forms))))))
                   (t `(if ,(car form)
                           (progn ,@(cdr form))
                           (xcond ,@(cdr forms))))))))

(provide 'xcond)
```

Appendix B

Source for DEW applications

B.1 pd-integration.lisp

```
(in-package :weyli)
(require 'predicate-dispatch)
(use-package 'predicate-dispatch)

(defclass ge-integral (general-expression)
    ((expr :initarg :expr
           :accessor expression-of)
     (var :initarg :var
          :accessor variable-of)
     (lower :initarg :lower
            :initform nil
            :accessor lower-bound-of)
     (upper :initarg :upper
            :initform nil
            :accessor upper-bound-of)))

(defmethod print-object ((int ge-integral) stream)
    (format stream "int ~A d~A" (expression-of int) (variable-of int))
    (if (lower-bound-of int)
        (format stream " from ~A to ~A" (lower-bound-of int)
                (upper-bound-of int))))

(defpdmethod integral (expr var) ()
    (let ((d (typecase expr
               (domain-element (domain-of expr))
               (t *general*))))
      (make-instance 'ge-integral :expr expr :var var :domain d)))

(defun definite-integral (expr var lower upper)
    (let* ((ge-var (coerce var *general*))
```

```
        (indef (integral expr ge-var)))
      (if (typep indef 'ge-integral)
        (make-instance 'ge-integral :expr expr :var var
                       :domain (domain-of indef)
                       :lower lower :upper upper)
         (- (substitute (coerce upper *general*) ge-var indef)
            (substitute (coerce lower *general*) ge-var indef)))))

(defun constant-of-integration ()
  (coerce (gensym "C") *general*))

(defun indefinite-integral (expr var)
  (+ (integral expr (coerce var *general*))
     (constant-of-integration)))

(defun first-domain (&rest args)
  (cond ((null args) nil)
        ((typep (car args) 'domain-element)
          (domain-of (car args)))
        (t (apply #'first-domain (cdr args)))))

(defun int (expr var &optional lower upper)
  (if lower
      (weyli::definite-integral expr var lower upper)
       (weyli::indefinite-integral expr var)))

(defmethod symbol-of ((sym symbol))
  sym)

(defun free? (expr var)
  ;;(zerop (deriv expr var))
  (not (depends-on? expr var))
  )

(defun make-free-in? (var)
  (lambda (expr) (free? expr var)))

;; useful for a limited version of the chain rule
(defun linear? (expr var)
  (let ((d (deriv expr var)))
    (and (not (zerop d))
         (free? d var))))

(defun same-var? (var1 var2)
  (and (or (symbolp var1) (typep var1 'ge-variable))
       (or (symbolp var2) (typep var2 'ge-variable))
       (eq (symbol-of var1) (symbol-of var2))))
```

```
(export 'int)

;;; useful specializations follow

(defpdmethod integral (expr var)
    ((free? expr var))
    (* expr var))

(defpdmethod integral (expr var)
    ((same-var? expr var))
    (* 1/2 expr expr))

(defpdmethod integral ((expr ge-plus) var) ()
    (make-ge-plus (domain-of expr)
                  (mapcar (lambda (exp)
                              (integral exp var))
                          (terms-of expr))))

(defpdmethod integral ((expr ge-times) var
                       &aux (free? (make-free-in? var))
                       (terms (terms-of expr)))
    ((member-if free? terms))
    (let ((domain (domain-of expr)))
        (make-ge-times domain
                       (cons (integral (make-ge-times domain
                                                      (remove-if free? terms))
                                       var)
                             (remove-if-not free? terms)))))

(defpdmethod integral ((expr ge-times) var)
    ((= (length (terms-of expr)) 1))
    (integral (car (terms-of expr)) var))

(defpdmethod integral ((expr ge-expt) var
                       &aux (base (base-of expr)) (exp (exponent-of expr)))
    ((free? exp var)
      (linear? base var))
    (let ((exp1 (+ exp 1)))
        (if (zerop exp1)
            ;; XXX – should take absolute value of log.
            (/ (make-ge-log (domain-of expr) base) (deriv base var))
             (/ (make-ge-expt (domain-of expr) base exp1) exp1 (deriv base var)))))

(defpdmethod integral ((expr ge-expt) var
                       &aux (base (base-of expr)) (exp (exponent-of expr)))
    ((free? base var)
```

```
    (linear? exp var))
   (if (= base 1)
       (/ exp (deriv base var))
        (/ (make-ge-expt (domain-of expr) base exp) (log base) (deriv base var))))

(defpdmethod integral ((expr ge-application) var
                       &aux (first-arg (first (args-of expr))))
   ((equal (name-of (funct-of expr)) "sin")
    (linear? first-arg var))
   (- (/ (make-ge-cos (domain-of expr) first-arg) (deriv first-arg var))))

(defpdmethod integral ((expr ge-application) var
                       &aux (first-arg (first (args-of expr))))
   ((equal (name-of (funct-of expr)) "cos")
    (linear? first-arg var))
   (/ (make-ge-sin (domain-of expr) first-arg) (deriv first-arg var)))

(defpdmethod integral ((expr ge-application) var
                       &aux (first-arg (first (args-of expr))))
   ((equal (name-of (funct-of expr)) "tan")
    (linear? first-arg var))
   (- (/ (make-ge-log (domain-of expr)
                      (make-ge-cos (domain-of expr) first-arg))
         (deriv first-arg var))))

(defpdmethod integral ((expr ge-application) var
                       &aux (first-arg (first (args-of expr))))
   ((equal (name-of (funct-of expr)) "log")
    (linear? first-arg var))
   (/ (- (* first-arg (make-ge-log (domain-of expr) first-arg))
         first-arg)
      (deriv first-arg var)))

(defpdmethod integral ((expr ge-application) var
                       &aux (first-arg (first (args-of expr))))
   ((equal (name-of (funct-of expr)) "sinh")
    (linear? first-arg var))
   (/ (make-ge-cosh (domain-of expr) first-arg) (deriv first-arg var)))

(defpdmethod integral ((expr ge-application) var
                       &aux (first-arg (first (args-of expr))))
   ((equal (name-of (funct-of expr)) "cosh")
    (linear? first-arg var))
   (/ (make-ge-sinh (domain-of expr) first-arg) (deriv first-arg var)))

(defpdmethod integral ((expr ge-application) var
```

```
                              &aux (first-arg (first (args-of expr))))
    ((equal (name-of (funct-of expr)) "tanh")
      (linear? first-arg var))
    (/ (make-ge-log (domain-of expr)
                    (make-ge-cosh (domain-of expr) first-arg))
       (deriv first-arg var)))

;; ...
```

B.2 struve.lisp

```
(in-package :weyl)
(require 'predicate-dispatch)
(use-package 'predicate-dispatch)

;; This code assumes that the Bessel and gamma functions have been defined.

(defpdmethod struve-h (r z)
    (let ((d (or (first-domain r z) *general*)))
        (make-ge-funct d (make-function d 'struve-h 2) r z)))

(defpdmethod struve-h ((r ratio) z)
    ((> r 0)
      (not (zerop z))
      (= (denominator r) 2))
    ;; Weyl doesn't support symbolic sums. :-/
    (do ((m 0 (1+ m))
         (sum 0 (+ sum (/ (* (gamma (+ m 1/2))
                             (expt (/ z 2) (+ (* -2 m) r -1)))
                          (gamma (+ r 1/2 (* -1 m)))))))
        ((> m r) (+ (bessel-y r z) (/ sum pi)))))

(defpdmethod struve-h ((r ratio) z)
    ((< r 0)
      (not (zerop z))
      (= (denominator r) 2))
    (* (expt -1 (- -1/2 r)) (bessel-j (- r) z)))

(defpdmethod struve-h (nu z)
    ((zerop z))
    0)

(defpdmethod struve-h ((nu number) (z number))
    ((or (floatp nu) (floatp z)))
    ;; don't give inexact results for exact inputs.
    (do ((s 0 (+ s (/ zf g1 g2)))
```

```
        (so -1 s)
        (z2 (- (expt (/ z 2) 2)))
        (k1 3/2 (1+ k1))
        (k2 (+ nu 3/2) (1+ k2))
        (g1 (gamma 3/2) (* g1 k1))
        (g2 (gamma (+ nu 3/2)) (* g2 k2))
        (zf ((expt (/ z 2)) (+ nu 1)) (* zf z2)))
      ((= s so) s))
  )

;; not sure how to work derivative or print form in, given issues with
;; mixing defmethod and defpdmethod. :-/ (Subclassing ge-application
;; also seems to be out, given Weyl's treatment of it.)
```

Appendix C

Other code

C.1 Struve.m

```
(* :Title: Struve *)


(* :Context: ProgrammingInMathematica`Struve` *)

(* :Author: Roman E. Maeder *)

(* :Summary:
   Definitions for the Struve functions
 *)

(* :Copyright: © 1989-1996 by Roman E. Maeder *)

(* :Package Version: 2.0 *)

(* :Mathematica Version: 3.0 *)

(* :History:
   2.0 for Programming in Mathematica, 3rd ed.
   1.1 for Programming in Mathematica, 2nd ed.
   1.0 for Programming in Mathematica, 1st ed.
*)

(* :Keywords: Struve *)

(* :Sources:
   Roman E. Maeder. Programming in Mathematica, 3rd ed. Addison-Wesley, 1996.
*)

(* :Discussion:
   See Section 8.4 of "Programming in Mathematica"
```

```
*)

BeginPackage["ProgrammingInMathematica`Struve`"]

StruveH::usage = "StruveH[nu, z] gives the Struve function."

Begin["`Private`"]

SetAttributes[ StruveH, {NumericFunction, Listable} ]

(* special values *)

StruveH[r_Rational?Positive, z_] /; Denominator[r] == 2 :=
        BesselY[r, z] +
        Sum[Gamma[m + 1/2] (z/2)^(-2m + r - 1)/Gamma[r + 1/2 - m], {m, 0, r-1/2}]/Pi

StruveH[r_Rational?Negative, z_] /; Denominator[r] == 2 :=
        (-1)^(-r-1/2) BesselJ[-r, z]

(* Series expansion *)

StruveH/: Series[StruveH[nu_?NumberQ, z_], {z_, 0, ord_Integer}] :=
        (z/2)^(nu + 1) Sum[ (-1)^m (z/2)^(2m)/Gamma[m + 3/2]/Gamma[m + nu + 3/2],
                            {m, 0, (ord-nu-1)/2} ] + O[z]^(ord+1)

(* numerical evaluation *)

StruveH[_, 0] := 0

StruveH[nu_?NumericQ, z_?NumericQ] /; Precision[{nu, z}] < Infinity :=
        Module[{s = 0, so = -1, z2 = -(z/2)^2, k1 = 3/2, k2 = nu + 3/2, g1, g2, zf},
                zf = (z/2)^(nu+1); g1 = Gamma[k1]; g2 = Gamma[k2];
                While[so != s,
                        so = s; s += zf/g1/g2;
                        g1 *= k1; g2 *= k2; zf *= z2; k1++; k2++
                ]; s
        ]

(* derivatives *)

StruveH/: Derivative[0, n_Integer?Positive][StruveH] :=
  Function[{nu, z},
    D[ (StruveH[nu-1, z] - StruveH[nu+1, z] + (z/2)^nu/Sqrt[Pi]/Gamma[nu + 3/2])/2,
       {z, n-1} ]
  ]

(* interpretation and formatting for traditional form *)
```

```
StruveH/:
MakeBoxes[StruveH[nu_, z_], form:TraditionalForm] :=
    RowBox[{SubscriptBox["H", MakeBoxes[nu, form]], "(", MakeBoxes[z, form], ")"}]

MakeExpression[ RowBox[{SubscriptBox["H", nu_], "(", z_, ")"}],
                form:TraditionalForm ] :=
  MakeExpression[ RowBox[{"StruveH", "[", RowBox[{nu, ",", z}], "]"}], form ]

End[]

Protect[StruveH]

EndPackage[]
```

Bibliography

[App92] Apple Computer. *Dylan, an Object-Oriented Dynamic Language*, April 1992.

[BB] Jonathan Bachrach and Glenn Burke. Partial dispatch: Optimizing dynamically-dispatched multimethod calls with compile-time types and runtime feedback. Work in progress.

[BDG+88] Daniel Gureasko Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. *Common Lisp Object System specification: X3J13 document 88-002R*, volume 23 of *ACM SIGPLAN Notices*. ACM Press, New York, NY, September 1988.

[BKK+86] D[aniel] G[ureasko] Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Common Loops, merging Lisp and object-oriented programming. *ACM SIGPLAN Notices*, 21(11):17–29, November 1986.

[CC99] Craig Chambers and Weimin Chen. Efficient multiple and predicate dispatching. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '99)*, Denver, CO, November 1999.

[CCG98] Craig Chambers and the Cecil Group. *The Cecil Language: Specification and Rationale*. University of Washington Department of Computer Science, Seattle, WA, December 1998. Version 3.0.

[Cha93] Craig Chambers. Predicate classes. In *ECOOP '93 Conference Proceedings*, pages 268–296, Kaiserslautern, Germany, July 1993.

[EKC98] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 186–211, Brussels, Belgium, July 1998.

[Fod91] John K. Foderaro. Introduction to the special Lisp section. *Communications of the ACM*, 34(9):27, September 1991. As quoted by Paul Graham.

[Fra00] Franz Inc., Berkeley, CA. *Allegro CL user guide*, version 6.0 edition, 2000.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, 1996.

[HS00] Bruno Haible and Michael Stoll. *clisp — Common Lisp language interpreter and compiler*, March 2000.

[Kam99] Ernic Kamerich. *A Guide to Maple*. Springer-Verlag, 1999.

[KdRB91] Gregor Kiczales, James des Rivières, and Daniel G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge, MA, 1991.

[Lag85] Jeffrey C. Lagarias. The $3x + 1$ problem and its generalizations. *The American Mathematical Monthly*, 92:3–23, 1985.

[Mac92] Robert A. MacLachlan, editor. CMU Common Lisp user's manual. Technical Report CMU-CS-92-161, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1992. October 31, 1997 Net Version.

[Mac95] Macsyma, Inc. *Macsyma mathematics and system reference manual*. Macsyma, Inc., Arlington, MA, fifteenth edition, 1995.

[Mae96a] Roman E. Maeder. *The Mathematica Programmer II*. Academic Press, New York, NY, 1996.

[Mae96b] Roman E. Maeder. *Programming in Mathematica*. Addison Wesley, Reading, MA, third edition, 1996.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[PJHA+99] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. *Haskell 98: A Non-strict, Purely Functional Language*, February 1999.

[Sch01] William Schelter. GNU Common Lisp 2.3.8, January 2001.

[Slo90] A. Sloman. *The Sussex University POPLOG System*. University of Sussex, Sussex, England, June 1990.

[Ste90] Guy L. Steele, Jr. *Common Lisp: the Language*. Digital Press, Bedford, MA, second edition, 1990.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, third edition, 1997.

[Wol99] Stephen Wolfram. *The Mathematica Book*. Cambridge University Press and Wolfram Research, Inc., New York, NY, and Champaign, IL, fourth edition, 1999.

[Zip93] Richard Zippel. The Weyl computer algebra substrate. In Alfonso Miola, editor, *Design and Implementation of Symbolic Computation Systems, International Symposium, DISCO '93*, volume 722 of *Lecture Notes in Computer Science*, pages 303–318, Gmunden, Austria, September 1993. Springer.