Multi-Methods in a Statically-Typed Programming Language

Warwick B. Mugridge, John Hamer, John G. Hosking
Department of Computer Science, University of Auckland,
Private Bag, Auckland, New Zealand
rick@cs.aukuni.ac.nz jham1@cs.aukuni.ac.nz john@cs.aukuni.ac.nz

ABSTRACT:

Multivariant functions in Kea are a statically-typed form of the multi-methods of CLOS (Keene, 1989) but encapsulation is retained. Multivariants permit fine typing distinctions to be made, allow despatching to be avoided in some cases, and may be used to avoid some restrictions of the contravariance rule.

Once multivariant functions are introduced by example, the semantics of the despatch of multivariants are provided, based on the generation of despatching variants. Three issues arise with despatching: redundancy, ambiguity, and exhaustiveness of a (partially-ordered) set of variants with respect to a function call. It is shown that the approach taken here is consistent with separate compilation.

KEYWORDS: object-oriented, multi-methods, static-typing, polymorphism, contravariance

1. Introduction

A form of multi-methods is introduced in the context of Kea[1], a statically-typed object-oriented and functional programming language which is currently being extended to include higher-order and (implicitly) polymorphic functions. Multivariant functions in Kea are a statically-typed form of the multi-methods of CLOS, in which despatching depends on the class of all arguments to a function (Keene, 1989). Unlike CLOS, however, Kea retains a notion of encapsulation. Multivariants permit fine typing distinctions to be made, allow despatching to be avoided in some cases, and may be used to avoid some restrictions of the contravariance rule (Cook, 1989).

In the Simula despatching model, an object is the implicit first argument to a procedure (or function) call; the class of this object determines the procedure that is executed (Dahl and Nygaard, 1966). This model is

[1] *Kea* was previously known as *Class Language*; a kea is an inquisitive New Zealand alpine bird.

inherited by Smalltalk and most other object-oriented languages (Goldberg and Robson, 1983). In class-based approaches, methods are associated with classes in a class hierarchy or partially-ordered set, and encapsulation is provided in some form, giving the benefits of abstract data types.

The Common Lisp Object System (CLOS) introduced the multiple-despatch model, in which the selection of the method to be executed depends on the class of all arguments to the message, not just the object (Keene, 1989). CLOS provides for multiple despatch with generic functions in a dynamically-typed setting but where encapsulation has been ignored.

The advantages of a statically-typed programming language are well known. The most important is that many errors can be detected at compile-time; such errors have to be found at run-time in a dynamically-typed language like Smalltalk or CLOS, sometimes long after a program is "complete". However, the disadvantages of an inflexible typing system, and/or the need to supply type information, lead many to prefer dynamically-typed languages. An important aim is to find typing systems which do not place unnecessary demands or restrictions on a programmer; automatic type inference and bounded parametric polymorphism are steps towards this goal (Cardelli and Wegner, 1985).

Section 2 of this paper briefly introduces Kea to provide a context for multivariant functions. Section 3 introduces multivariant functions by example and shows that there is not a clear distinction between overloading and inclusion polymorphism. Multivariants may be used to avoid some of the restrictions of the contravariance rule. Section 4 defines the semantics of despatching multivariant functions with a scheme for the automatic generation of *despatching* variants. Three issues that arise with despatching are defined: redundancy, ambiguity, and exhaustiveness of a (partially-ordered) set of variants with respect to a function call. Section 5 raises compilation issues, including provision for separate compilation. The final section concludes the paper and suggests future work.

2. Introduction to Kea

Kea inherits from the object-oriented paradigm the notions of information-hiding, abstract data types, inclusion polymorphism, method overriding, and multiple inheritance. In addition, it introduces dynamic classification (Hamer et al, 1989; Hamer, 1990a; Hosking et al, 1990). From the functional language paradigm, Kea inherits higher-order and polymorphic functions, type inference, and lazy evaluation.

A class consists of a signature and an implementation. The signature consists of public features of the class. A public feature of a component object is referenced using the "^" operator. The implementation consists of expressions for public and private (non-public) features. A feature of a class is typed; its expression is evaluated when a value is required, such as in the evaluation of another expression. An object is created on demand with the pseudo-function *new*. The arguments to *new* are lazily evaluated; they pass information to the new object (as object parameters) from the context in which it was created.

A class inherits the signatures and implementations of all its generalisation classes (superclasses); it may extend either. A class inherits a feature only once from a superclass even when there are several inheritance paths. Generalisation relationships between classes define a (partial) type ordering, similar to Trellis/Owl (Halbert and O'Brien, 1987). There is no notion of inheritance without a type relationship, in contrast to Smalltalk (Goldberg and Robson, 1983).

Classification expresses sufficient conditions for object class membership; for example, a rectangle with equal sides can be treated as a square. A classification attribute specifies a *cluster*: a set of mutually-exclusive subclasses (Smith and Smith, 1977). If class A has a cluster {B, C}, classification ensures that any object of class A will also belong to either class B or C (but not both). In this way, clusters constrain types; for example, the presence of cluster {B, C} means that no class can inherit from both B and C. Multiple classification is achieved with independent classification attributes, so that an object can be classified to several independent subclasses.

Dynamic classification of an object permits its type to be elaborated at run-time. An object may be explicitly classified as also belonging to other classes, based on the evaluation of its classification attributes. This process of classification is carried out on demand, whenever a possible classification may lead to code which can affect the current evaluation of an expression. Classification is lazy in that it is only carried out to the extent that is necessary. Classification need not mirror inheritance, allowing for "classification leaps" down the class inheritance structure. Hosking et al (1990) provides further details.

2.1 Higher-Order and Polymorphic Functions in the List Classes

Kea is currently being extended to include higher-order, (implicitly) polymorphic functions with multiple despatch. The use of higher-order and polymorphic functions in class *List* and its subclasses is shown in Fig. 2.1. The three classes here together define a data structure for a list of integers. The class *List* specifies the signatures of the public functions available (corresponding to "virtuals" in Simula (Dahl and Nygaard, 1966)), as well as defining the constructor function *cons*.

Class *EmptyList* provides code for the empty list case. Class *ListNode* defines the non-empty list case with object parameters *head* and *tail*. The classification feature *defaultEmpty* in class *List* specifies that the two subclasses of *List* are mutually exclusive. An object of class *List* will be classified to class *EmptyList*, as defined by the expression for *defaultEmpty*, on the first access to a public function of the object (other than *cons*). Thus, class *List* is not an abstract class; the default classification makes "new List" operationally equivalent to "new EmptyList".

```
class List.
 public    cons, filter, map, fold, append.
 classification defaultEmpty: [EmptyList, ListNode]
     := EmptyList.
  cons(front: integer) := new ListNode(front, self).
  filter(keep: integer -> boolean): List.
  map(trans: integer -> integer): List.
  fold(accum: (integer, Any) -> Any, identity: Any): Any.
  append(other: List): List.
 end List.
```

```
class EmptyList.
 generalisation List.
  filter(keep) := self.
  map(trans) := self.
  fold(accum, identity) := identity.
  append(other) := other.
 end EmptyList.
```

```
class ListNode.
 generalisation List.
 parameter head: integer.
           tail: List.
 public head, tail.
  filter(keep)
     := tail^filter(keep)^cons(head) if keep(head)
     | tail^filter(keep).
  map(trans) := tail^map(trans)^cons(trans(head)).
  fold(accum, identity)
     := accum(head, tail^fold(accum, identity)).
  append(other) := tail^append(other)^cons(head).
 end ListNode.
```

Figure 2.1 The *List* Classes[2]

Functions in Kea may be higher-order and/or polymorphic, as is usual in functional languages (Field and Harrison, 1988). For example, the higher-order function *map* in class *ListNode* in Fig. 2.1 is inferred to be of type "ListNode → (integer → integer) → ListNode". This function returns the list resulting from applying the function *trans* to each of the elements of the provided list. The *map* function call in Fig. 2.2 returns the increment of each of the integers in a list, using an anonymous function. Similarly, the function *filter* is used in Fig. 2.2 to select the positive integers from a list.

```
ints := new List^cons(-1)^cons(2).
positives := ints^filter(lambda(i: integer) => i > 0).
increment := ints^map(lambda(i: integer) => i + 1).
sum := ints^fold(add, 0).
add(i: integer, total: integer) := i + total.
```

Figure 2.2 Using functions *filter*, *map*, and *fold*

2 The symbol "|" is read as "or" in function expressions.

Parametric polymorphism for functions is implicit, both in bounded and unbounded forms (Cardelli and Wegner, 1985). For example, the function *fold* in Fig. 2.1 has the type *Any* specified for some of the parameters. The type of this function is inferred to be "List → (integer → x → x) → x → x", in which *x* is a type variable. The function *fold* is used in Fig. 2.2 to sum the elements of an *List*; the actual type of the function application here is "List → (integer → integer → integer) → integer → integer".

3. Multivariant Functions by Example

Kea's multivariant functions are related to the multi-methods of CLOS (Keene, 1989). As with multi-methods, the code chosen for execution (during *despatching*) depends on the type of all function call arguments, rather than just the type of the primary object (self). Kea, however, is statically typed; multivariant functions and their calls are statically checked for type-correctness. The selection of the appropriate function *variant* can be made at compile-time if the types of function call arguments are suitable, as discussed below.

3.1 Overloading

Multivariant functions provide for overloading, where different variants have unrelated parameter types. For example, in Fig. 3.1 the function *div* accepts either integers or reals. The applicable variant (and hence the result type) can be determined at compile-time for a call of the function *div*. Thus the expression for *aList* in Fig. 3.1 is incorrectly typed because the *List* function *cons* requires an integer parameter; it is rejected at compile-time. Compile-time selection of the appropriate variant for a function call means that there need be no despatch at run-time.

```
div(r1: real, r2: real) := r1 / r2.              % real
div(r: real, i: integer) := r / toReal(i).       % real
div(i: integer, r: real) := toReal(i) / r.       % real
div(i1: integer, i2: integer) := i1 div i2.      % integer
```

```
anInt := div(4,2).                          % integer
aReal := div(4, 2.0).                       % real
aList := new List^cons(div(2.0, 4)).        % Type error
```

Figure 3.1 Overloaded Function

Overloading also arises naturally from the coincidental matching of function names from unrelated classes.

3.2 Despatching

Type information about the parameters of a function call may not be sufficient to select the appropriate function variant at compile-time. For example, consider the function *equal* in the classes *Point* and

ColorPoint in Fig. 3.2 (adapted from Canning et al, 1989). The types of the two variants are "Point → Point → boolean" and "ColorPoint → ColorPoint → boolean". With a function call in which the types of the arguments are only known (statically) to be of type *Point*, a selection must be made at run-time between the two relevant variants, based on the type of the actual parameters. For example, if the actual parameters to the function call are both of class *ColorPoint*, the variant in class *ColorPoint* is dynamically selected.

Encapsulation is enforced: a function within a class may access any parameters and functions of an object of that class. However, access is only permitted to public functions of the arguments of a function.

```
class Point.
 public x, y, move, equal.
 parameter x, y.
  x: float.
  y: float.
  equal(p: Point) := x = p^x and y = p^y.
 end Point.
```

```
class ColorPoint.
 generalisation Point.
 public color.
 parameter color: Color.
  equal(p: ColorPoint) := x = p^x and y = p^y and color = p^color.
 end ColorPoint.
```

Figure 3.2 Despatching

Cook (1989) points out that the contravariance rule[3] is violated in Eiffel, a statically-typed language which uses the Simula despatching model (Meyer, 1988). Multivariant functions in Kea allow the benefits of subclassing to be retained without violating this rule. The contravariance rule is satisfied because the function *equal* in class *ColorPoint* does not completely override the inherited function; instead, it provides code to handle the case when the object and the function parameter are both of type *ColorPoint*.

The two uses of *equal* in the expression for *consistent* in Fig. 3.3 provide the same result; if either the object or the parameter (or both) are of type *Point*, the function variant in class *Point* is called.

```
a := new Point(x := 0.0, y := 0.0).
b := new ColorPoint (x := 0.0, y := 0.0, colour := red).

consistent := a^equal(b) = b^equal(a).
```

Figure 3.3 Consistency of Result from *equal*

---

[3] The contravariance rule specifies that a function f of type "AA → B" is a subtype of function g of type "A → BB" (i.e. $f \leq g$) if and only if A ≤ AA and B ≤ BB. That is, the subtype may "narrow" the result type but can only "widen" the parameter type.

Functions that access objects need not be defined within a class; in this case access is only permitted to publics of those objects supplied as parameters. This is illustrated with the function *firmEqual* in Fig. 3.4 which provides a different notion of equality: an object of class *Point* can not be equal to an object of class *ColorPoint*. The order of variants defines the sequence in which they are considered during despatching.

```
firmEqual(p: ColorPoint, q: ColorPoint)
   := p^x = q^x and p^y = q^y and p^colour = q^colour.
firmEqual(p: ColorPoint, q: Point) := false.
firmEqual(p: Point, q: ColorPoint) := false.
firmEqual(p: Point, q: Point) := p^x = q^x and p^y = q^y.
```

Figure 3.4 A Different Notion of Equality

3.3 Overloading and Despatching

The need for despatching may depend on the particular function call. For example, consider the function *addList* in Fig. 3.5, which extends the *List* classes of Fig. 2.1. This function takes two lists and adds them element by element; the resulting list is the length of the shortest of the two lists. The types of the three variants in Fig. 3.5 are "EmptyList → List → EmptyList", "ListNode → EmptyList → EmptyList", and "ListNode → ListNode → ListNode" respectively.

```
class List.
...
   cons(front: integer) := new ListNode(front, self).
...
   addList(other: List): List.
 end List.
```

```
class EmptyList.
 ...
  addList(other) := self.
 end EmptyList.
```

```
class ListNode.
 ...
 parameter head: integer.
           tail: List.
...
  addList(other: EmptyList) := other.
  addList(other: ListNode)
     := tail^addList(other^tail)^cons(head + other^head).
 end ListNode.
```

Figure 3.5 Function *addList*

Consider the example function calls in Fig. 3.6 (with result types shown as comments). Despatching is not needed for the expressions of *v1*, *v2*, and *v3*, as adequate type information is available to select the appropriate variant statically. In addition, the specific type of these expressions is determined. For

example, the function call in the expression for *v3* is "ListNode → ListNode → x"; given that the third variant is selected statically, the type variable *x* is determined to be *ListNode*.

```
empty := new EmptyList.                          % EmptyList
one := empty^cons(1).                            % ListNode
positives := one^filter(lambda(x) => x > 0).     % List
v1 := empty^addList(one).                        % EmptyList
v2 := one^addList(empty).                        % EmptyList
v3 := one^addList(one).                          % ListNode
v4 := one^addList(positives).                    % List
v5 := positives^addList(positives).              % List
```

Figure 3.6 Overloaded and Despatching Function Calls

Despatching is required for the calls to *addList* in the expressions of *v4* and *v5*. The expressions are both of type *List*; this type is based on the types of the variants involved in the selection.

4. Semantics of Despatching Multivariant Functions

We define the semantics of multivariant functions through their translation to a lazy functional language. *Despatching* variants are generated during this translation; these define the selection between variants that is carried out at run-time. For functions within a class, the object is made explicit as *self*, the first parameter. For example, the function *addList* from Fig. 3.5 is translated to the code shown in Fig. 4.1 (in which redundant conformance tests have been removed).

```
addList1(self, p1) := self.   % EmptyList → List → EmptyList
addList2(self, p1) := p1.     % ListNode → EmptyList → EmptyList
addList3(self, p1)            % ListNode → ListNode → ListNode
     := cons(addList5(tail(self), tail(p1)), head(self) + head(p1)).
addList4(self, p1)            % ListNode → List → List
     := addList2(self, p1) if conforms(p1)(EmptyList)
     | addList3(self, p1).
addList5(self, p1)            % List → List → List
     := addList1(self, p1) if conforms(self)(EmptyList)
     | addList4(self, p1).
```

Figure 4.1 Generated Code for *addList*

Two *depatching* variants $addList_4$ and $addList_5$ have been generated to select between other variants. For example, the variant $addList_4$ (called by *v4* in Fig. 3.6) selects at run-time between the variants $addList_2$ and $addList_3$ depending on the type of the second actual parameter; this variant is of type "ListNode → List → List". The function *conforms* takes an object and returns a function which in turn takes a class identifier as parameter; the latter function returns true if the object is of that class.

To assist in defining the semantics of multivariant despatching, we informally introduce a "first pass" translation. The results of this translation are used to define the generation of despatching variants. Three important properties of sets of variants are defined: redundancy, ambiguity, and exhaustiveness. We stress that the aim here is to define the semantics of despatching; an implementation will use rather different techniques. For example, *conforms* information and the results of unary functions are cached in the current system (Hamer, 1990b).

4.1 The "First Pass" Translation

The first pass takes a Kea program and produces:

- Function variants in a functional form in which the object is included as an explicit first parameter (self) to encapsulated functions. This means that the implicit parameter can be treated the same as other parameters in Section 4.2.
- The partial order of the variants of each function;
- Functions for object creation and access to object parameters; and
- Subtyping and classification information.

We ignore here a number of issues in this translation: signatures, checking that encapsulation is respected, and checking the constraints imposed by classification attributes.

For example, the definition of the function *addList*, which appears in the classes *EmptyList* and *ListNode* in Fig. 3.5, is translated to the functional form shown in Fig. 4.2. Function calls are later resolved to specific function variants, as defined in Section 4.2.

```
addList1(self: EmptyList, other: List) := self.
addList2(self: ListNode, other: EmptyList) := other.
addList3(self: ListNode, other: ListNode)
     := cons(addList(tail(self), tail(other)), head(self) + head(other)).
```

Figure 4.2 Function *addList* in Functional Form

Information is gathered about the class relationships and clusters. Each class has zero or more clusters, where a *cluster* is a set of classes corresponding to a single classification attribute. For example, clusters(List) = {{EmptyList, ListNode}}. The subtype relation $<$ is the transitive closure of immediate subclass; the relation $\leq$ is the reflexive transitive closure.

A multivariant function f is defined as a triple (P, V, Θ), where P is the number of arguments of the function, V is the set of variants with name f and P arguments, and Θ specifies the partial order of the variants in V. For the purposes of the following discussion, we are only concerned with the type of the variants.

The partial order is defined as follows. Let $v_1, v_2 \in V$ where v_1 appears in class C_1 and v_2 appears in class C_2. The set Θ contains the element $v_1 << v_2$ iff: $C_1 = C_2$ and v1 appears before v2; or $C_1 < C_2$. For example, the function *addList* in Fig. 4.2 is defined as the triple (2, {addList$_1$, addList$_2$, addList$_3$}, {addList$_2$ << addList$_3$}). For the purpose of defining Θ, function variants which are declared outside of classes are treated as being defined within a class T, where for all $C \in$ the user-defined classes, $C \leq T$.

For each class, there is a function to create new objects of that class. There is also a function to access each object parameter. For example, consider the class *ColorPoint* from Fig. 3.2. The function *new_ColorPoint*, shown in Fig. 4.3, creates an object of class *ColorPoint*. An object consists of a pair (M, P), where *M* is a class membership function and *P* is a sequence of actual object parameters.[4] The general function *conforms* selects M from the pair; it is used in depatching code. The argument to *conforms* is used when a class has one or more classification attributes.

```
new_ColorPoint(p1, p2, p3) := o where o :=
     (conformsCP(o), (p1, p2, p3)).
conformsCP(o) := lambda(c) => c = ColorPoint or c = Point.
x((c, (p1, ...))) := p1.
y((c, (p1, p2, ...))) := p2.
color((c, (p1, p2, p3))) := p3.

conforms((c,p)) := c.
```

Figure 4.3 Object Creation and Object Parameters

4.2 Despatching Function Calls

The second phase of translation involves the generation of despatching variants. Three issues arise in depatching function calls: redundancy, ambiguity, and exhaustiveness. A variant is redundant if it can never be selected; redundancy points to a programmer error. A set of variants is ambiguous with respect to a function call if different total orderings of the partial order of the relevant variants lead to different results. A set of variants is exhaustive with respect to a function call if the variants cover all possible subtypes of the arguments of the function call.

---

4 The treatment of object parameters is simplified here; under multiple inheritance they are partially ordered.

These properties of variant sets are defined in this section, along with the generation of despatching variants. These definitions do not take account of polymorphic or recursive function variants; see Mugridge et al (1991a) for further details.

An Example.

Consider the classes in subtype relations $B \leq D$, $B \leq A$, and $C \leq A$ and where clusters(A) ={{B, C}}. The function f is defined as the triple (2, V_f, Θ_f), where V_f = {f_1: D → D → T, f_2: A → B → T, f_3: B → D → T, f_4: C → B → T, f_5: C → C → T}, and Θ_f = {f_3 << f_1, f_4 << f_2, f_4 << f_5, f_5 << f_2}. The function g is defined as the triple (1, V_g, Θ_g), where V_g = {g_1: D → T, g_2: A → T}, and Θ_g = {}. This situation is illustrated in Fig. 4.4. In the remainder of this section, we consider a function call, such as f': A → B → x, as being a variant type in which the result type x is unknown.

The following problems arise:

- The variant f_2 is **redundant**; due to the variants f_3 and f_4 and the cluster, it can never be selected. A function call f': A → B → x will match either f_3 and f_4 because an object of class A must also be either of class B or class C due to the cluster.

- The variants g_1 and g_2 are **ambiguous** with respect to a function call g': B → x. Both variants apply but there is no order defined between them in Θ_g.

- The variant set V_f is not **exhaustive** with respect to the function call f': A → C → x. It would be with the addition of the variant f_6: B → C → T.

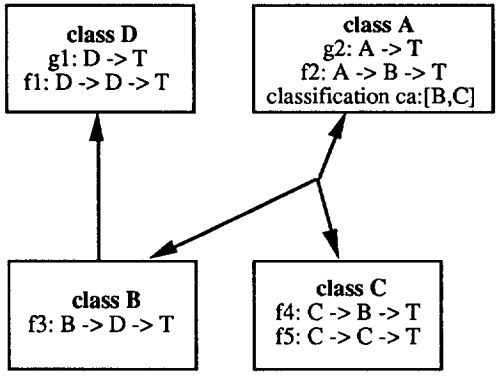

Figure 4.4 Redundancy, Ambiguity, and Exhaustiveness

Definitions.

Cover: The *cover* of a variant v: $t_1 \rightarrow \ldots \rightarrow t_n \rightarrow w$ is the set $\{s_1 \rightarrow \ldots \rightarrow s_n \mid s_1 \leq t_1, \ldots, s_n \leq t_n\}$. The *cover* of a set of variants V is $\{c \mid c \in \text{cover}(v') \text{ and } v' \in V\}$.

ExtendedCover: We can extend the variants in a cover by considering clusters. If $\{s_{i1}, \ldots, s_{in}\} \in \text{clusters}(s_i)$ and a cover of a variant contains all the classes $s_{i1}, \ldots, s_{in}$ in argument position i, then the variant must also cover the class s_i in that argument position.

The *extendedCover* of a set of variants V is the set C where C contains:

- all variants in cover(V);
- $S \rightarrow t \rightarrow U$ if C contains $S \rightarrow t_i \rightarrow U, \ldots, S \rightarrow t_m \rightarrow U$ where $\{t_1, \ldots, t_m\} \in \text{clusters}(t)$, S is a type $s_1 \rightarrow \ldots \rightarrow s_i$ and U is a type $u_1 \rightarrow \ldots \rightarrow u_j$.

Relevant. Let a function f be (n, V, Θ). A variant $v \in V$ is *relevant* to a function call f' iff $\text{cover}(v) \cap \text{cover}(f') \neq \emptyset$.

Exhaustive: A set of variants E is *exhaustive* with respect to a variant v iff $\text{cover}(v) \subseteq \text{extendedCover}(E)$.

Redundant. There are two sources of redundancy. The simplest case is where a set of function variants from the same class (i.e. with the same first argument) cover all the argument types of a later variant from that class. The second case, illustrated in the example above, arises where a set of function variants from the classes in a cluster cover all the argument types of a variant from the cluster's class. These forms of redundancy are defined as follows:

(1) Direct Redundance: Let a function f be (n, V, Θ), $v \in V$, $V' = \{v' \in V \mid v' << v \in \Theta$ and v and v' have the same first argument type$\}$. The variant v is *redundant* if V' is exhaustive with respect to v.

(2) Extended Redundance: Let a function f be (n, V, Θ) and $v \in V$. The variant v is *redundant* if there exists a cluster C such that $V' = \{v' \in V \mid v' << v \in \Theta$ and the type of the first argument of v' is $C_i \in C\}$ and V' is exhaustive with respect to v.

Ambiguous: Let a function f be (n, V, Θ). The variant set V is *ambiguous* with respect to a function call f' iff there exists distinct variants $v_i, v_j \in \text{cover}(V)$ such that there is no ordering defined between v_i and v_j in Θ and $\text{cover}(v_i) \cap \text{cover}(v_j) \cap \text{cover}(f') \neq \emptyset$.

Despatch Variant: The generation scheme *despatch* generates code to select between an exhaustive set of variants as follows:

despatch($\{v_i: t_{1,1} \rightarrow ... \rightarrow t_{1,m} \rightarrow w_1, \ ... \ , v_n: t_{n,1} \rightarrow ... \rightarrow t_{n,m} \rightarrow w_n\}$) =

```
v0(p1, ...,pm) := v1(p1, ... , pm) if conforms(p1)(t1,1) and ...
                                    and conforms(pm)(t1,m)
                | ...
                | vn(p1, ... , pm) if conforms(p1)(tn,1) and ...
                                    and conforms(pm)(tn,m).
```

Function Call Despatching: Let a function f be (n, V, Θ) and f': $t_1 \rightarrow ... \rightarrow t_n \rightarrow x$ be an application of function f. Code can be generated for f' iff R, the set of variants relevant to f', is exhaustive and not ambiguous with respect to f'. In this case, a call is made to the despatching variant generated by despatch(R).

Examples.

• The cover(f_2) = {A → B, B → B, C → B} and extendedCover({f_3, f_4}) = {B → D, B → B, C → B, A → B}. As cover(f_2) ⊆ extendedCover({f_3, f_4}), {f_3, f_4} is exhaustive with respect to f_2, and hence the variant f_2 is redundant.

• The cover of the function call g': B → x is {B}. The cover(g_1) = {D, B} and cover(g_2) = {A, B, C}. There is no ordering defined between g_1 and g_2 in Θ_g and yet cover(g_1) ∩ cover(g_2) ∩ cover(g') = {B}. Hence g_1 and g_2 are ambiguous with respect to g'.

• The cover of the function call f': A → C → x is {A → C, B → C, C → C}. However, B → C ∉ extendedCover(V_f) and so V_f is not exhaustive with respect to f'.

• Consider the function call f': A → B → x. The relevant variant set R = {f_2, f_3, f_4}, which is exhaustive and not ambiguous with respect to f'. A call is made to the the despatching variant v' defined as follows:

```
v'(p1, p2) := f3(p1, p2) if conforms(p1)(B)
            | f4(p1, p2) if conforms(p1)(C)
            | f2(p1, p2).      % This case is redundant
```

5. Compilation Issues

Problems with variant redundancy, ambiguity, and non-exhautiveness must be signalled during compilation. We now consider two issues: handling exhaustiveness and ambiguity at runtime, and separate compilation of a Kea program.

5.1 Runtime Checks

When a set of variants is not exhaustive with respect to a function call, a compilation error should result. As it is convenient to develop a partially-completed program, a better approach is to give a warning and extend the variant set so that it is exhaustive with respect to the function call. The extra variant produces an error message at run-time. For example, a program that calls the function *tail* with an argument that is only known to be of type *List* results in a warning and leads to the new variant shown in Fig. 5.1.

```
tail(self: List): List
    := exception("function tail can only be applied to a ListNode").
```

Figure 5.1 Automatically-Generated Error-Checking Variant

A warning could also be given when two or more variants are ambiguous with respect to a subset of the cover of a function call; an extra variant can be generated which gives an error if the ambiguity arises at runtime.

5.2 Separate Compilation

Provision is made for separate compilation. Consider the set of classes shown in Fig. 5.2, in which the classes *A* and *B* have been compiled within a library and the classes *C* and *D* appear in new code that uses the library.

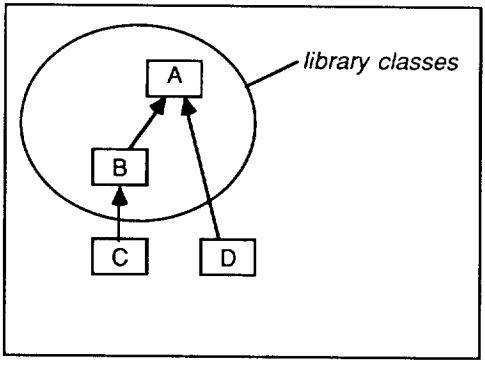

Figure 5.2 Library and Added Classes

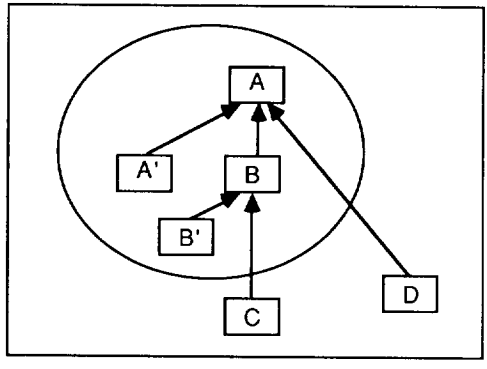

Figure 5.3 After Class Migration

The set of classes in a library is translated so that a class is not used to create objects if it is (or may become) a superclass of other classes. The translation, shown in Fig. 5.3, introduces an empty class *A'* as a subclass of *A*. Any object which previously would have been created as an object of class *A* is instead

created as an object of class *A'*. This process of "class migration" automatically introduces an abstract class.[5]

Class migration opens the way for separate compilation by permitting subclasses to be introduced later. When a function call to a variant is compiled, the position of the call is added to a list of all calls to that variant. If a new subclass is introduced later, such as class *C* in Fig. 5.3, new despatch code is generated and all function calls are redirected to the new (despatching) variant which has been generated to take account of variants in new subclasses.

```
f(self: A) := h^g.
f(self: B) := g^h.
```

Fig 5.4 The Functions in the Library

For example, consider the two variants of the function *f* defined in the library, as shown in Fig 5.4. The generated code, along with a despatching variant, is shown in Fig. 5.5; this despatching variant is based on the assumption that *A'* and *B* are the only subclasses of *A*.

```
f1(self) := g(h(self)).                  % A'
f2(self) := h(g(self)).                  % B
f0(self) := f2(self) if conforms(self)(B)       % A
         | f1(self).
```

Fig 5.5 Generated Code for the Library Variants

Later compilation of classes *C* and *D* with the library makes the original despatching variant f_0 incorrect. Provision must be made for the new variants shown in Fig. 5.6. New despatch code is generated, taking account of the new variants, as shown in Fig. 5.7.

```
f(self: C) := g^g.
f(self: D) := h^h.
```

Fig 5.6 The Functions in Classes *C* and *D*

All calls to f_0 are redirected to $f_{0'}$ and all calls to f_2 are redirected to $f_{2'}$. This is handled by re-linking the list of function calls from f_2 so that they are linked to the variant $f_{2'}$. A "code linking" phase runs through the lists and resolves the addresses. The links makes it possible to eliminate the code of unused variants in a "garbage collection" phase; this is important when using a small portion of a large library.

---

5 The original motivation for class migration was "type loss" (Mugridge, et al, 1990b).

```
f3(self) := g(g(self)).                    % C
f4(self) := h(h(self)).                    % D
f0'(self) := f3(self) if conforms(self)(C)          % A
           | f4(self) if conforms(self)(D)
           | f0(self).
f2'(self) := f3(self) if conforms(self)(C)          % B
           | f2(self).
```

Fig 5.7 The New Generated Code

Adding classes later can invalidate previously acceptable function calls. For example, if the subclass *C* were introduced without a variant for function *f*, the set of variants would be no longer exhaustive with respect to function calls to f_0. This is handled as in Section 5.1.

6. Conclusions and Future Work

Multivariant functions in Kea generalise the notion of despatching in statically-typed object-oriented languages; the ideas are also relevant to procedural object-oriented languages, such as Eiffel (Meyer, 1988). Multivariants are a statically-typed form of the multi-methods of CLOS (Keene, 1989) but where encapsulation is retained. As despatching can be avoided when there is adequate type information about arguments to a function call, there need be no unnecessary overhead on function calls. In addition, multivariant functions avoid the restrictions on subtyping imposed by the contravariance rule.

Cardelli and Wegner (1985) distinguish overloading (ad hoc polymorphism) and universal polymorphism (parameteric and inclusion polymorphism). However, multivariant functions show that the distinction is not so clear; whether overloading or inclusion polymorphism is involved can depend on the function calls concerned.

Encapsulation of multivariant functions is provided. Functions in Kea may be organised within classes (i.e. based on the object: the implicit first argument), where access is available to all functions of the class, both public and private. Functions that are written outside of classes may only access public functions. Hence the first argument to a function is still given special status, as in many object-oriented programming languages: Smalltalk (Goldberg and Robson, 1983), Eiffel (Meyer, 1988), and Trellis/Owl (Halbert and O'Brien, 1987). This is in comparison with CLOS, which discards the notion of encapsulation altogether in introducing multi-methods (Keene, 1989). Further work is needed in considering other ways to integrate encapsulation and multi-methods.

As Kea is currently defined, all function argument types must be specified. We are considering the introduction of further type inference so as to eliminate the need for explicit typing where it is unnecessary. For example, the signatures defined in class *List* in Fig. 2.1 could be inferred automatically. A related area

of investigation is into "type loss", which prevents the full potential of static typing from being realised (Mugridge et al, 1991b). Unfortunately, bounded parametric polymorphism (Cardelli and Wegner, 1985) only avoids some forms of "type loss".

Multivariant functions provide a weak form of selection when compared to the pattern-matching of functional languages like Hope (Field and Harrison, 1988; Mugridge et al, 1990). It would be convenient to introduce a form of pattern-matching into Kea. We are considering the definition of patterns (consisting only of public functions) in a class and using those patterns in variants. For example, the function *addList* in class *ListNode*, from Fig. 3.5, is recoded in Fig. 6.1 to use a possible form of pattern-matching.

```
class ListNode.
 pattern (head, tail).
 ...
  addList(other: EmptyList) := other.
  addList((h,t): ListNode) := tail^addList(t)^cons(head + h).
 end ListNode.
```

Figure 6.1 Pattern-Matching in Function *addList*

Acknowledgements

The authors acknowledge the financial assistance provided by the Building Research Association of New Zealand, the University of Auckland Research Committee, and the New Zealand University Grants Committee.

References

Canning P S, Cook W R, Hill W L, Olthoff W G, 1989. Interfaces for strongly-typed object-oriented programming, OOPSLA'89, ACM SIGPLAN Notices, **24** (10) October, 1989, pp457-467.

Cardelli L, Wegner P, 1985. On understanding types, data abstraction, and polymorphism, *Computing Surveys*, **17**(4), pp471-522.

Cook W, 1989. A proposal for making Eiffel type safe, in Cook S (Ed), *ECOOP 89*, Cambridge University Press, pp57-70.

Dahl O J, Nygaard K, 1966. Simula - an Algol-based simulation language, *CACM* **9** (9), pp671-678.

Field A J, Harrison P G, 1988. *Functional Programming*, Addison-Wesley.

Goldberg A, Robson D, 1983. *Smalltalk 80: The Language and its Implementation*, Addison-Wesley.

Halbert D C, O'Brien P D, 1987. Using types and inheritance in object-oriented programming, *IEEE Software*, September 1987, pp71-79.

Hamer J, Hosking J G, Mugridge W B, 1989. Knowledge-based systems for representing codes of practice, Report 48, Department of Computer Science, University of Auckland, New Zealand.

Hamer J, 1990a. Expert Systems for codes of practice, PhD Thesis, Department of Computer Science, University of Auckland, New Zealand.

Hamer J, 1990b. Class Language runtime system: detailed specification, BRANZ Contract 85-024, Technical Report No. 9, Department of Computer Science, University of Auckland, New Zealand.

Hosking J G, Hamer J, Mugridge W B, 1990. Integrating functional and object-oriented programming, Procs. Pacific Tools 80 Conference, Sydney, Australia, November 1990.

Keene S E, 1989. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley, 1989.

Meyer B, 1988. *Object-Oriented Software Construction*, Prentice Hall.

Mugridge W B, Hosking J G, Hamer J, 1990. Functional extensions to an object-oriented programming language, Report No. 49, Department of Computer Science, University of Auckland, New Zealand.

Mugridge W B, Hamer J, Hosking J G, 1991a. The semantics of multivariant functions, in preparation.

Mugridge W B, Hamer J, Hosking J G, 1991b. Type loss in statically-typed object-oriented proghramming languages, in preparation.

Smith J M, Smith D C P, 1977. Database abstractions: aggregation and generalization, *ACM Trans. on Database Systems*, **2** (2), 1977, pp105-133.