Generalising the BETA Type System

Søren Brandt and Jørgen Lindskov Knudsen

Department of Computer Science
University of Aarhus
DK-8000 Aarhus C, Denmark
Email: {sbrandt,jlknudsen}@daimi.aau.dk

Abstract. The type system of object-oriented programming languages should enable the description of models that originate from object-oriented analysis and design. In this paper, the BETA type system is generalised, resulting in direct language support for a number of new modelling aspects. The increased expressive power is obtained from a synergy between general block structure and the generalised type hierarchy, and not from syntactic additions to the language.
The type hierarchy described in this paper is a superset of the class hierarchy. In order to regain an orthogonal and internally consistent language, we investigate the impact of the new type hierarchy on other parts of the language. The resulting increase in expressive power serves to further narrow the gap between statically and dynamically typed languages, adding among other things more general generics, immutable references, and attributes with types not known until runtime.
Keywords: language design, type systems, object-oriented modelling, constraints, BETA.

1 Introduction

The type system of an object-oriented language should enable the description of important aspects of models that originate from object-oriented analysis and design. For example, an object-oriented model may describe issues like "the manufacturer of my car" and "the wheel of a car". "The manufacturer of my car" is an immutable reference from "my car" to the actual manufacturer, implying that 1) the manufacturer is not a part of the car, 2) the manufacturer existed before the car, and 3) the reference can never change in the entire lifetime of the car. Here, both "the manufacturer" and "my car" are concrete objects.

Another example is "the wheel of a car", that refers to a part object ("the wheel") of some other object ("a car"), but without determining the concrete car object. The type "the wheel of a car" is therefore a concrete type only relative to a concrete car object, but includes enough information to imply that this specific type does not describe the wheel of a truck.

A common understanding of types in object-oriented languages is as predicates on classes. For example, in BETA [Madsen et al. 93b], C++ [Stroustrup 93], and Eiffel [Meyer 92], a type is the name of a class `C`. Interpreted as a type, the class name `C` is a predicate that evaluates to true on the set of subclasses of `C`.

Other languages, such as Sather [Omohundro 93] and Emerald [Black et al. 87], separate the type and class hierarchies, but still interpret types as predicates on classes: The predicate evaluates to true for classes that *conform* to the type. In any case, a typed reference can only refer to instances of classes on which the type predicate evaluates to true.

Even though the BETA type system is very expressive as compared to most statically typed OO languages, it is at times found to be more restrictive than necessary. The quest for flexible yet static type systems is a search for good compromises between the ultimate freedom of expression in a dynamically typed language, and the safe but constraining rigidity of completely static type systems. The generalisation of the BETA type system presented in this paper is an attempt to gain more flexibility without sacrificing the level of static type checking supported, and to allow expression of a number of models arising in object-oriented analysis and design.

The BETA type system is generalised in two directions: Firstly, by allowing type expressions that do not uniquely name a class, but instead denote a closely related set of classes. Secondly, by allowing types that cannot be interpreted as predicates on classes, but must be more generally interpreted as predicates on objects. We then investigate the impact this generalisation has on other parts of the language. The result is a large increase in expressive power.

The type system described in this paper originates from the development of MetaBETA [Brandt & Schmidt 96], a reflective extension of BETA, featuring a dynamic meta-level interface accessible to programs at runtime. The challenge of adding a dynamic meta-level interface to a statically typed language without circumventing the type system lead to the generalisation of the BETA type system described in this paper. It is a major ingredient in the MetaBETA approach, allowing dynamic reflection in a statically typed language.

This paper assumes some basic knowledge of the BETA language. For readers not familiar with the language, a short BETA primer is included in Appendix A. Section 2 describes current BETA type checking, Section 3 generalises the BETA type system, and Section 4 investigates the effect of the generalised type system on object creation operators. Section 5 shows some examples of the expressive power gained from the generalisations, Section refpatvarextension generalises the BETA concept of dynamic pattern references, and Section 7 describes the semantics of generalised attribute declarations. Finally, Section 8 describes some limitations on the generalised type system, and Sections 9 and 10 point to future work and presents our conclusions.

2 BETA type checking

The type checking rules of BETA are heavily influenced by BETA's general support for localisation in the form of pattern (class) nesting. Simula [Dahl et al. 84] originally introduced this property, inspired by the general block structure in the Algol languages. However, class nesting in Simula is more restricted than in

BETA. This section describes aspects of the BETA type system needed for the purposes of this paper. Other aspects are described in [Madsen et al. 93a].

2.1 Terminology

BETA is a block structured language allowing general pattern nesting: Objects are instances of patterns and have attributes that are references to either other objects or patterns. These references may be either dynamic of static.

Dynamic object references may refer to different objects at different points in time. However, dynamic object references cannot refer to arbitrary objects: They are subject to *qualification* constraints[1]. Likewise, dynamic pattern references may, subject to qualifications, refer to different patterns at different points in time. The BETA declaration:

```
aCircle: ^Circle;
```

declares `aCircle` as a dynamic object reference (^) qualified by `Circle`, i.e., it is only allowed to refer to instances of `Circle` or instances of subpatterns of `Circle`. Statically it can only be assumed that attributes defined for the `Circle` pattern are available in the object referred to by `aCircle`. At runtime, the actual object referred to by `aCircle` may have several other attributes (since it might be an instance of a subpattern of `Circle`), but these cannot be accessed through the `aCircle` reference, since they are not statically known. An unconstrained declaration can be made using `Object` as qualification:

```
o: ^Object;
```

`o` can refer to any object, but only operations defined for all object types are allowed on object references qualified by `Object`.

Most BETA type checking is done at compile time. However, by allowing attributes to have a virtual type that can be specialised in subpatterns, BETA supports covariant[2] pattern hierarchies, and runtime type checks can therefore not be completely avoided in the general case[3]. To enforce strong typing, BETA therefore in some cases reverts to runtime type checking on destructive assignments that cannot be statically accepted or rejected.

For example, writing `o[]->aCircle[]`, the value of the `Object` reference `o` may be assigned to the `Circle` reference `aCircle`. The compiler is unable to accept or reject this assignment, since it cannot statically deduce whether the assignment is type correct: It knows nothing of the actual type of the object

[1] The *qualification* of a BETA dynamic reference corresponds to the *type* of an Eiffel reference or a C++ pointer.

[2] Covariance means that a subpattern may specialise inherited attributes. Hence, the pattern and its attributes are simultaneously specialised — they are covariant. For an interesting discussion on covariance, see [Shang 95].

[3] Usually, static typing of hierarchical type systems is ensured by enforcing contravariant or nonvariant relationships between super/sub-types in a type hierarchy [Black et al. 87, Omohundro 93].

referred to by `o`. To handle this problem, the compiler inserts a runtime type check.

In summary, type checking in BETA is based on qualified (typed) attribute declarations. A qualification limits the possible values of a dynamic reference, and is usually the name of a pattern. A qualified reference can only refer to instances of the qualification or instances of subpatterns of the qualification. Hence, the qualification tells the compiler what operations are applicable to any object that can potentially be referred.

2.2 Formal Notation

In BETA, the term "attribute" encompasses all variables and procedures in a BETA program. To describe the qualification constraints on attributes, this section introduces some formal notation. The aim of this notation is to enable precise description of the semantics of the different kinds of attributes in a BETA program:

```
dor: ^Circle;           (* Dynamic Object Reference  *)
sor: @Circle;           (* Static Object Reference   *)
dpr: ##Circle;          (* Dynamic Pattern Reference *)
pd:  Circle(# ... #);   (* Pattern Declaration       *)
so:  @Circle(# ... #);  (* Singular Object           *)
```

The notation is summarised in Figure 1, where `attr` and `q` are path expressions, o is an object, and $\mathtt{p}_{sub}$ and $\mathtt{p}_{super}$ are patterns. The details of the notation are described in the following.

`attr:`	The attribute denoted by the path expression `attr`.
object(`attr:`)	The object referred to by the object reference `attr:`.
location(`attr:`)	The object of which `attr:` is an attribute
pattern(`attr:`)	The pattern referred to by the pattern reference `attr:`.
`q##`	The path expression `q` interpreted as a qualification.
qual(`attr:`)	The qualification of the attribute `attr:`.
pattern-of(`o`)	The pattern of which the object `o` is an instance
extension(`q##`)	The extension of the qualification `q##`.
$\mathtt{p}_{sub} \leq \mathtt{p}_{super}$	$\mathtt{p}_{sub}$ is a sub-pattern of $\mathtt{p}_{super}$.

Fig. 1. Formal notation used in this paper.

Adding a trailing colon to a path expression is used to reflect that we are talking about the attribute itself, and not its value. For example, to denote the `aCircle` attribute itself, we shall write `aCircle:`.

The **object**() function returns the object referred to by an object reference attribute. For example, **object**(`aCircle:`) is the object currently referred to by the attribute `aCircle:`.

location(`attr:`) returns the object that contains the `attr:` attribute. For example, **location**(`aCalc2.clear:`) is **object**(`aCalc2:`) in Figure 9 of Appendix A.

The **pattern**() function returns the pattern referred to by a pattern reference attribute. For example, **pattern**(`dpr:`) denotes the pattern currently referred to by `dpr:`.

To avoid ambiguities, we shall use `a.b.c##` to denote the qualification interpretation of a path expression `a.b.c`. If, for example, `a.b.c` denotes a pattern, the expression `a.b.c` could be taken to mean "the result of creating and executing a new instance of **pattern**(`a.b.c:`)". Likewise, in a later section we will allow qualification expressions denoting object reference attributes, and in that case the BETA expression `a.b.c` would mean "evaluate the do-part of **object**(`a.b.c:`)". The `a.b.c##` notation avoids these ambiguities.

The **qual**() function returns the qualification of an attribute, and is defined for all kinds of BETA attributes: For the `dor:`, `sor:`, and `dpr:` attributes, **qual**() returns the qualification expression to the right of, respectively, `^`, `@`, or `##`. Hence, **qual**(`dor:`) = **qual**(`sor:`) = **qual**(`dpr:`) = `Circle##`. Pattern declarations such as `pd` are fixed points for **qual**(), and **qual**(`pd:`) therefore returns the `pd` pattern itself. For singular object declarations such as **object**(`so:`), **qual**() returns the otherwise anonymous pattern of which **object**(`so:`) is the only instance. Exactly what constitutes a pattern will be described in Section 2.3.

The **pattern-of**() function returns the pattern of which an object is an instance. For example, after evaluating:

```
    myCircle: Circle (# ... #); (* Circle subpattern *)
    aCircle: ^Circle;
do &myCircle[]->aCircle[];      (* Object instantiation *)
```

pattern-of(**object**(`aCircle:`)) is the `myCircle` pattern.

The notation $\mathtt{p}_{sub} \leq \mathtt{p}_{super}$ means that the pattern $\mathtt{p}_{sub}$ is a subpattern of $\mathtt{p}_{super}$. Hence, used as a qualification, all objects that qualify to $\mathtt{p}_{sub}$ also qualify to $\mathtt{p}_{super}$. The set of objects qualifying to a pattern `p` is denoted **extension**(`p`). Hence, we shall define the $\leq$ relation on patterns by:

$$\mathtt{p}_{sub} \leq \mathtt{p}_{super} \overset{def}{\Leftrightarrow} \mathbf{extension}(\mathtt{p}_{sub}) \subseteq \mathbf{extension}(\mathtt{p}_{super}) \tag{1}$$

The meaning of **extension**(`p`) will be defined in Section 2.3. For example, `myCircle` $\leq$ `Circle`, since all instances of `myCircle` are in the extension of `Circle`.

With the notation introduced, we may now formally express the general qualification constraints on dynamic object and pattern references. Consider the declarations:

```
dor: ^Q;
dpr: ##Q;
```

The qualification constraint on the dynamic object reference `dor:` is:

$$\textbf{pattern-of}(\textbf{object}(\texttt{dor:})) \leq \textbf{qual}(\texttt{dor:}) \tag{2}$$

meaning that the object referred to by `dor:` must at all times be an instance of a subpattern of the qualification of `dor:`, in this case `Q##`.
The qualification constraint on the dynamic pattern reference `dpr:` is:

$$\textbf{pattern}(\texttt{dpr:}) \leq \textbf{qual}(\texttt{dpr:}) \tag{3}$$

meaning that the pattern referred to by `dpr:` must at all times be a subpattern of the qualification of `dpr:`, in this case `Q##`.

2.3 The Impact of Block Structure

BETA is a block structured language allowing general pattern nesting: Objects are instances of patterns and have attributes that may themselves be patterns. This influences the type system since the apparently "same" pattern attribute of different objects in fact denotes different patterns. This is illustrated by the example in Figure 2, which declares a pattern `Window` with the nested (class) pattern `Line` and the nested (method) pattern `drawline`. Furthermore, two instances of `Window`, `w1` and `w2`, are declared.

The patterns `w1.Line` and `w2.Line` are different, since the draw method in instances of each of these patterns draws a line in separate windows. This has consequences also when using `w1.Line` and `w2.Line` as qualifications, as illustrated in Figure 2: References qualified by `w1.Line` cannot refer to instances of `w2.Line` and vice versa.

```
Window:
  (# drawline:
       (# p1,p2: @Point;
       enter (p1,p2) do ...
       #);
     Line:
       (# p1,p2: @Point;
          draw: (# do (p1,p2)->drawline #);
       #);
  #);
```

```
   w1,w2: @Window;
   l1: ^w1.Line; l2: ^w2.Line;
do &w1.Line[]->l1[];  (* OK *)
   &w2.Line[]->l2[];  (* OK *)
   l1[]->l2[];     (* ERROR *)
   l2[]->l1[];     (* ERROR *)
```

Fig. 2. Nested pattern example

Two important concepts in understanding the BETA type system and its relation to block structure, is patterns and object descriptors. An object descriptor, as shown in Figure 3, is a source code entity, whereas a pattern is a corresponding runtime entity.

In the BETA grammar, the syntactic category <`ObjectDescriptor`> matches source code of the form shown in Figure 3. Every occurrence of the syntactic

```
Super
(# Decl1; Decl2; ... Decln (* attribute-part *)
enter In                   (* enter-part *)
do Imp1; Imp2; ... Impm       (* do-part *)
exit Out                    (* exit-part *)
#)
```

Fig. 3. Syntactic category <`ObjectDescriptor`>

category <`ObjectDescriptor`> in a program source uniquely defines an object descriptor. In addition to being the meat of pattern declarations, (see Figure 7 in Appendix A), object descriptors occur in singular object declarations, as well as nested directly in the do-part of other object descriptors (as exemplified by Figure **??** in Appendix A).

General block structure allows pattern nesting to an arbitrary level, and a nested BETA pattern is therefore a closure defined by a a unique object descriptor and an *origin* object. The origin of a pattern is significant for two purposes: Firstly, instances of the pattern may need access to attributes of the origin object [4]. Secondly, the origin affects the set of objects qualifying to a pattern when used as a variable qualification, and is therefore significant for type checking purposes. A pattern `p` is uniquely identified by the pair:

$$(\mathbf{origin}(\mathrm{p}), \mathbf{descriptor}(\mathrm{p}))$$

where **origin**(p) is the object of which `p` is an attribute, and **descriptor**(p) is the object descriptor for `p`. Due to inheritance, a pattern may have a super pattern, **super**(p), uniquely identified by the pair:

$$(\mathbf{origin}(\mathbf{super}(\mathrm{p})), \mathbf{descriptor}(\mathbf{super}(\mathrm{p})))$$

leading to a chain of patterns starting in `p` and terminating with the `Object` pattern. For patterns declared at the outermost block-level, no origin reference is needed, since there is no surrounding object.

[4] The *static link* between activation records of languages, such as Pascal, where procedures may be declared local to other procedures is a special case of the BETA origin reference.

Direct pattern instances An object is a *direct instance* of a pattern iff the object was created as an instance of that pattern. A direct instance of a pattern `p` contains a reference to **descriptor**(`p`) and also a reference to each of the origin objects found in the pattern chain starting in `p`. For example, the `w1.Line` pattern corresponds to the pair (**object**(`w1:`), `Window.Line`). As illustrated in Figure 4, the `l1` instance of `w1.Line` therefore has a reference to the object descriptor corresponding to `Window.Line`, and an origin reference to the `w1` object. The `Window` instance `w1` has a reference to the object descriptor for `Windows`, but no origin reference, since the `Window` pattern is declared at the outermost block-level. That objects `l1` and `l2` are instances of different patterns follows from their origin references pointing to different objects, although `l1` and `l2` share the descriptor of `Window.Line` objects.

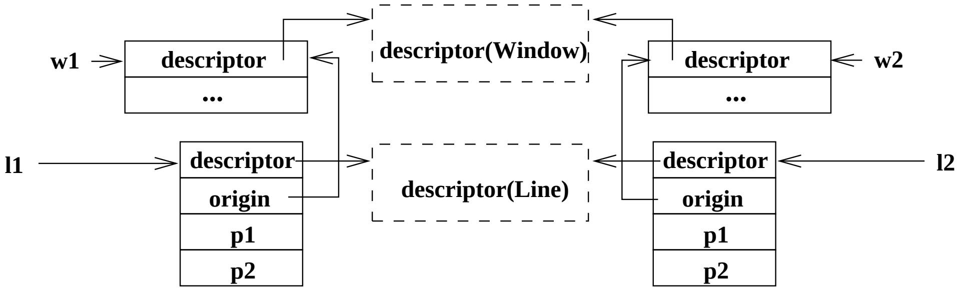

Fig. 4. Lines have an origin reference to the `Window` instance in which their pattern is nested.

The extension of a pattern Finally, we may now define the extension of a pattern as follows: An object `o` that is a direct instance of a pattern `p` is in the extension of a pattern `q`, i.e., `o` $\in$ **extension**(`q`), iff `q` can be found in the chain of patterns starting in `p`.

3 A Generalised Qualification Concept

In this section, we introduce a generalised qualification concept. The BETA type system is generalised by 1) allowing type expressions that do not uniquely name a class, and 2) by allowing types that are interpreted as predicates on objects. The generalisation is obtained without adding new syntax to the language.

In most situations, current BETA *only* allows qualification expressions that uniquely name a pattern. But logically there is no reason why a qualification should always be equivalent to a specific pattern. In fact, current BETA does allow one special kind of qualification that does not uniquely name a pattern: Recall that at runtime, a BETA pattern `p` is a closure defined by an origin object and an object descriptor. By naming the object descriptor but leaving the origin unspecified, a qualification that corresponds to a *set* of patterns results. In current BETA, this is allowed in the case of dynamic object references, as exemplified by `aShape` below:

```
Window: (# Shape: (# ... #)#); aShape: ^Window.Shape;
```

Seen as a predicate on patterns, the `Window.Shape` qualification evaluates to true on any pattern `Shape` nested inside *some* `Window` object `w`, and subpatterns of these.

Figure 5 illustrates the process of pattern specialisation as a stepwise narrowing of the pattern extension. But no matter the degree of specialisation, the

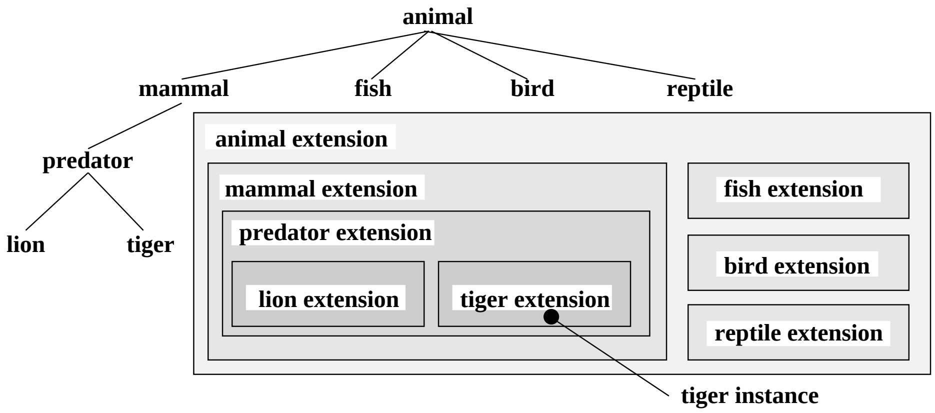

Fig. 5. Classification hierarchy and corresponding extension sets

extension of a pattern continues being unlimited, since any pattern has an unlimited number of instances. Hence BETA, like all other OO languages, only allows the expression of qualifications with unlimited extensions.

In principle we can imagine qualification extensions of any size, finite or unlimited. By explicitly listing a number of object names, any qualification with finite extension can be specified. As a tentative syntax for such qualifications, consider `w: ^[w1,w2,w3];`, with the intended meaning that the dynamic object reference `w` is allowed to refer to any of the objects `w1`, `w2`, or `w3`. However, not wanting to change the syntax of BETA, we restrict our attention to lists of length one, i.e., allowing the use of a single object name in place of a qualification.

3.1 Taxonomy of Generalised Qualifications

The previous section introduced the idea of generalising the set of qualifications allowed. This section divides qualifications into 8 main categories along three binary axes.

Syntactically, a BETA qualification expression is a path expression, i.e., a dot-separated list of names: A qualification `q` is generally of the syntactic form $\mathtt{n}_1.\mathtt{n}_2.\ldots.\mathtt{n}_m$, where each $\mathtt{n}_i$ is a name that denotes either a pattern attribute or an object reference attribute. Qualification expressions appear in several situations:

– As qualifications in variable declarations:

```
dor: ^q;  (* dynamic object reference *)
dpr: ##q; (* dynamic pattern reference *)
```

– As values assigned to dynamic pattern references: `do q##->dpr##`
– As pattern specification in part object declarations: `sor: @q;`
– As super specification in pattern declarations:

```
p: q(# ... #);    (* pattern declaration *)
pv:< q(# ... #); (* virtual pattern declaration *)
```

– As super specification in singular object declarations and method executions:

```
   sor: @q(# ... #); (* singular object  *)
do q(# ... #)        (* method execution *)
```

The first axis along which we classify qualification expressions `q` distinguishes *pattern qualifications* and *object qualifications*: If $\mathtt{n}_m$ is the name of a pattern reference attribute, `q` is called a pattern qualification. If $\mathtt{n}_m$ is the name of an object reference attribute, `q` is called an *object qualification*.

The second axis distinguishes *full qualifications* and *partial qualifications*: If all $\mathtt{n}_i$, $i < m$, are names of object reference attributes, `q` is called a full qualification. If some $\mathtt{n}_i$, $i < m$ is the name of a pattern attribute, `q` is called a partial qualification. Partial qualifications do not uniquely identify an object attribute.

These two axes result in four qualification categories as depicted in Figure 6.

	$\mathtt{n}_1.\ldots.\mathtt{n}_{m-1}$: only objects	$\mathtt{n}_1.\ldots.\mathtt{n}_{m-1}$: at least one pattern
$\mathtt{n}_m$: object	full object qualification	partial object qualification
$\mathtt{n}_m$: pattern	full pattern qualification	partial pattern qualification

Fig. 6. Qualification categories for $\mathtt{q}=\mathtt{n}_1.\mathtt{n}_2.\ldots.\mathtt{n}_m$

The third and last axis distinguishes *static qualifications* from *dynamic qualifications*: If all $\mathtt{n}_i$ are names of static pattern attributes or static object reference attributes, `q` is called a *static qualification*. Otherwise, `q` is called a *dynamic qualification*. A static qualification has a static extension, whereas the extension of a dynamic qualification depends on the time of evaluation.

Note that the classification of qualification expressions only depends on the statically known kind of the attributes named in the expression, and can therefore easily be known by the compiler.

Example 1. Full pattern qualifications such as `Window` and `aWindow.Shape` uniquely name a pattern. Partial pattern qualifications such as `Window.Shape` correspond to sets of related patterns. Full object qualifications such as `aWindow` or `aShape` correspond to qualifications with a single element extension, i.e., they name an object. Finally, for an example of a partial object qualification consider the following code:

```
Person: (# name: @Text #);
aPersonName: ^Person.name;
```

`aPersonName` is here constrained by a static partial object qualification to refer to the name of *some* person, but without restricting the specific person. □

3.2 Qualification Hierarchies

Inheritance is the basis for building pattern hierarchies such as the one shown in Figure 5. BETA pattern hierarchies is a generalisation of class hierarchies in languages such as Eiffel and C++. With the introduction of generalised qualifications, another hierarchy has emerged: The qualification hierarchy.

A pattern name is a special case of qualification (the full pattern qualification), and the set of patterns is thus a subset of the set of qualifications. The inheritance relation equips the set of patterns with a partial order as defined by Equation 1 in Section 2.2. Likewise, the qualification hierarchy is partially ordered: For full pattern qualifications, the partial order corresponds to the inheritance relation. In general, we shall define the partial order on qualifications by means of their extension. This is a straightforward generalisation of Equation 1:

$$\mathtt{q} \leq \mathtt{q'} \overset{def}{\Leftrightarrow} \mathbf{extension}(\mathtt{q}) \subseteq \mathbf{extension}(\mathtt{q'})$$

where q and q' are both qualification expressions. For example, the extension of a full object qualification o is a single-element set:

$$\begin{array}{ll} & \mathtt{o} \leq \mathtt{q} \\ \overset{def}{\Leftrightarrow} & \mathbf{extension}(\mathtt{o}) \subseteq \mathbf{extension}(\mathtt{q}) \\ \Leftrightarrow & \{\mathtt{o}\} \subseteq \mathbf{extension}(\mathtt{q}) \\ \Leftrightarrow & \mathtt{o} \in \mathbf{extension}(\mathtt{q}) \end{array}$$

The special value NONE[5] means *no object*. By definition, NONE qualifies to any qualification, and is therefore the bottom ($\perp$) of the qualification hierarchy:

$$\forall \mathtt{q} : \ \mathtt{NONE} \leq \mathtt{q}$$

In current BETA, the pattern hierarchy is the basis for type checking. In generalised BETA, the qualification hierarchy takes over, and hence allows more general qualifications.

We have introduced a qualification hierarchy which is different from the pattern hierarchy. However, as opposed to the type and class hierarchies of, e.g., Emerald, the qualification and pattern hierarchies are not separate, since the pattern hierarchy is embedded in the qualification hierarchy. Another difference from the Emerald type hierarchy is that our qualification hierarchy contains qualifications that uniquely identify single objects.

4 Generalised Semantics for Object Creation Operators

The BETA operators & and @ are used to create new objects, i.e., new instances of BETA patterns: & is used for dynamic object creation in imperative code, and @ is used in declarations to denote the creation of static part objects as part of their location:

```
      Person: (# left_arm, right_arm: @arm #);
      aPerson: ^Person;
   do &Person[]->aPerson[];
```

Here, a new person object is created using the & operator. A reference to the new person object is then assigned to the dynamic object reference aPerson. Along with the new Person object, a number of static part objects, the limbs, are created. Creation of the limbs happens automatically, since they are declared using the @ operator, which also binds the newly created limbs to the identifiers left_arm and right_arm.

The & operator takes a single qualification parameter q. The @ operator takes two parameters: A qualification parameter q, and an identifier parameter id. Current BETA requires that the q parameter given to & and @ denotes a pattern, i.e., q must be a full pattern qualification. Given a qualification q, the semantics of both operators is to create a new instance of the pattern denoted by q. In addition, @ binds the newly created object to the identifier parameter id.

To what extent can these semantics be generalised to allow other kinds of qualification parameters to & and @? Given a qualification with a finite extension, the "create a new object" semantics do not apply anymore, since all objects contained in a finite extension are by definition already there, and any new object would therefore not be part of that extension. We therefore generalise the

[5] NONE corresponds to the null pointer in C++, and the special value Void in Eiffel.

semantics of & from "create new object" into:

$$\&(\mathtt{q}) = \begin{cases} \text{new } \mathtt{q} & \text{if } \mathtt{q} \text{ is a full pattern qualification} \\ \mathtt{o} & \text{if } \mathtt{q} \text{ is a full object qualification and } \mathbf{extension}(\mathtt{q}) = \{\mathtt{o}\} \\ \mathit{error} & \text{otherwise} \end{cases} \quad (4)$$

That is, applying & to a qualification q always returns an object that qualifies to q. But in case of q being a full object qualification, the returned object is no longer new[6].

The object creation operator, &, is evaluated each time the statement of which it is part is executed. The static object reference operator, @, is evaluated once, namely at creation time of its location. The result of evaluating @(id, q) is to bind the value &(q)[] to the identifier id. Afterwards, the binding of id is not allowed to change.

5 Examples

Before considering all possible declarations in more detail, we show some examples of the expressive power gained from the introduction of generalised qualifications and object creation operators.

Example 2. The following code demonstrates one way of binding an immutable reference to an already existing object:

```
   aWindow: ^Window;
   Foo: (# sor: @aWindow #);
   aFoo: ^Foo;
do &Foo[]->aFoo[];
```

When an instance of Foo is created, the static object reference sor: is bound to the value &(aWindow##)[], where aWindow is a dynamic full object qualification. According to Equation 4, this means that aFoo.sor: is bound to refer the object referred to by aWindow: at the time when aFoo is created. I.e., aFoo.sor: becomes an immutable reference to **object**(aWindow:). □

Generalised qualifications allow us to think of full object qualifications as specialisations of qualifications with unlimited extensions. This results in virtual patterns that can be specialised into full object qualifications.

Example 3. Consider the relation between a child and each of its parents. These relations are immutable references bound when the child comes into existence. In generalised BETA, we can express this as follows:

[6] The decision to disallow partial qualifications as parameter to & is more or less arbitrary. It could be argued that a better choice would be to create a new object along with the objects necessary to provide the environment of the new object. For example, &Window.Shape[], could create a new Window object w, followed by the creation of an instance of w.Shape.

```
Person:
  (# FatherBinding:< Person; father: @FatherBinding;
     MotherBinding:< Person; mother: @MotherBinding;
  #);
makeChild:
  (# newfather, newmother: ^Person
  enter (newfather[], newmother[])
  exit &Person
         (# FatherBinding:: newfather;
            MotherBinding:: newmother;
         #)[]
  #);
```

Here, a `Person` is defined to have an immutable reference to each parent. Due to `Person` having `Person` part objects, `Person` is an abstract pattern of which it is impossible to create direct instances. However, by further binding `FatherBinding` and `MotherBinding` into full object qualifications, it becomes possible to create a `Person`, and at the same time bind the parent references. Current BETA forces us to describe the parent references by dynamic references, which is unfortunate since people do not tend to change biological parents. Conversely, generalised BETA is able to directly model that parent references are immutable and that the parents existed before the child. To solve the chicken-and-egg problem of creating the very first `Person` instance, we can create a person with no parents:

```
do (NONE,NONE)->makeChild->adam[];
```

□

To be used as super specification in a pattern declaration, a qualification expression must be a full pattern qualification: A partial qualification would be ambiguous, and a full object qualification would correspond to the action of specialising an object. Likewise, in a part object declaration, the pattern specification must eventually evaluate to a full qualification for the part object to be unambiguously defined. Notice however, that an abstract pattern may declare part objects of only partially known type, as long as concrete subpatterns specify a full qualification for the part.

Example 4. Consider an abstract `CodeDisplay` pattern:

```
Grammar: (# PrettyPrinter:< (# ... #); ... #);
CodeDisplay:
  (# PrettyPrinter:< Grammar.PrettyPrinter;
     pp: @PrettyPrinter;
  #);
```

A `CodeDisplay` uses a prettyprinter to actually display the code, but the exact type of the pretty-printer depends on the type of `Grammar` being used. `CodeDisplay` is an abstract pattern, because the type of the `pp` part object is ambiguous. However, on creation of a `CodeDisplay`, the actual grammar used is known, and can be used to disambiguate the type of `pp`:

```
aGrammar: @Grammar;
aCodeDisplay: @CodeDisplay
  (# PrettyPrinter:: aGrammar.PrettyPrinter #);
```

□

We may also make good use of partial pattern qualifications in expressing generics: In BETA, the type parameter of a generic class is expressed by a nested virtual class [Madsen & Møller-Pedersen 89].

Example 5. Consider the declaration of a generic list pattern:

```
List:
  (# Element:< Object; (* Element type *)
     insert: (* Insert element *)
       (# new: ^Element
       enter new[] do ...
       #);
  #);
```

To create a list of persons, the generic `List` class is specialised by final binding the `Element` virtual:

```
aPersonList: @List(# Element:: Person #);
```

But to create a generic `Shape` list, we must use a partial pattern qualification:

```
ShapeList: List (# Element::< Window.Shape #)
```

which is illegal in current BETA, since the `Element` virtual in `ShapeList` does not uniquely name a pattern. In generalised BETA, however, the example is perfectly legal. Further specialising `ShapeList`, we can now create lists that can contain shapes from specific windows only, or use `ShapeList` directly to create a list of all shapes, regardless of the window to which they belong:

```
w1,w2: @Window;
w1shapes: @ShapeList(# Element:: w1.Shape #);
w2shapes: @ShapeList(# Element:: w2.Shape #);
allShapes: @ShapeList;
```

In current BETA, a list containing all `Shape` objects can only be declared as a list that can contain any object, thereby losing all static information on the type of objects in the list. □

Example 6. Consider the declaration of a dynamic text reference: `aText: ^Text;` One might be interested in declaring a dynamic text reference that is only allowed to refer to text objects that are actually person names:

```
Person: (# name: @Text; ... #);
aPersonName: ^Person.name;
```

Clearly, `aPersonName` can only refer to text objects, but the type system now also expresses and enforces that the text referred is the name of a person. □

6 Generalised Dynamic Pattern References

Dynamic pattern references were originally introduced in BETA to support patterns as first-class values [Agesen et al. 89], e.g., allowing patterns as method parameters. In practice, dynamic pattern references are used in several ways:

- As function and method pointers.
- As dynamic class references.
- As qualification references, allowing explicit runtime type checks.

In Section 3 we argued that patterns are a special case of the more general qualification concept. It is therefore natural to introduce a corresponding generalisation of dynamic pattern references, turning them into dynamic *qualification references*. However, to avoid introducing new terminology, we retain the term dynamic pattern reference.

In current BETA, a dynamic pattern reference value is obtained by appending the `##` operator to object names or pattern names: The expression `o##` returns **pattern-of**(`o`), i.e., a reference to the pattern of which the object `o` is an instance. Likewise, the value of the expression `p##` is a reference to the pattern `p`. In both cases, a reference to a full pattern qualification is obtained. These semantics give rise to the following irregularities:

1. If **object**(`so:`) is a singular object, the BETA expression `so##` returns a reference to the otherwise anonymous pattern **pattern-of**(**object**(`so:`)) of which **object**(`so:`) is the only instance. This breaks the anonymity of **pattern-of**(**object**(`so:`)), and using `so##`, it is then possible to create new instances of the pattern. This would destroy the singularity of **object**(`so:`), and singularity can therefore not be guaranteed by the compiler.
2. There is currently no way to obtain a reference to qualifications other than full pattern qualifications. For example, partial qualifications are currently not first-class values, and can therefore not be passed as method parameters. Consider the `copy` method below, intended to copy an object unless it is of a specific type:

```
copy:
  (# o,ocopy: ^Object; dont_copy: ##Object;
  enter (o[],dont_copy##)
  do (if not (o##<=dont_copy##) then
        o[]->performCopy->ocopy[];
     if);
  exit ocopy[]
  #);
do (anObject[],Window.Shape##)->copy->aCopy[];
```

As illustrated, a desired use of `copy` might be to copy all objects that do not qualify to `Window.Shape##`. Unfortunately, `Window.Shape##` is a partial pattern qualification, and the example is therefore illegal, since only references to full pattern qualifications are currently allowed.

To remove these irregularities, we change and generalise the `##` operator into:

The value of the expression $\mathtt{n}_1.\mathtt{n}_2.\ldots.\mathtt{n}_m$`##` is the qualification denoted by the path expression $\mathtt{n}_1.\mathtt{n}_2.\ldots.\mathtt{n}_m$ at evaluation time.

Note that for dynamic qualification expressions, each evaluation of the expression may result in a new qualification. For full object qualifications `o`, the above definition changes the value of `o##` from **pattern-of**(**object**(`o:`)), to the full object qualification with extension {**object**(`o:`)}. The expression `&(o##)[]` thus becomes equivalent to `o[]`, and the singular object irregularity has disappeared. Likewise, the `copy` example is now fully legal.

7 Attribute Declaration Semantics

Previous sections generalised the notions of qualifications and object creation operators. In this section, we list the semantics for all attribute types in light of the new qualification concept. As will be seen, very few surface changes have been made to the semantics of attribute declarations. The real changes stem from generalisations to the qualification hierarchy, and the resulting changes to the object creation operators. To describe the semantics of attribute declarations, we consider the following attribute properties:

- Is destructive assignment allowed?
- Initial value.
- Possible values.

BETA supports four attribute kinds, described in turn below.

Static Object References: `sor: @q`
The only interesting property of a static object reference is its initial value, since destructive assignment is not allowed. We use the notation **crt**(`sor:,expr`) to denote the value of `expr` at the time when **location**(`sor:`) is created.

The initial value of `sor:` is given by

$$\textbf{object}(\texttt{sor:}) = \textbf{crt}(\texttt{sor:},\&\textbf{qual}(\texttt{sor:})\texttt{[]})$$

with the value of $\&\textbf{qual}(\texttt{sor:})\texttt{[]}$ defined by Equation 4 in Section 4. The above equation is the same as in current BETA, with the new possibilities resulting from Equation 4.

Static Pattern References: `spr:q` or `spr: q(# ... #)`

In current BETA, static pattern references always declare new patterns. This is consistent with the declaration of static object references, that always create new objects. However, we have generalised static object references to allow them to be bound to already existing objects, and generalised dynamic pattern references to be able to refer to any kind of qualification. As a combination, static pattern references can be bound to any kind of qualification, including existing ones.

Again, the behaviour of static pattern reference `spr:` is essentially captured in a single equation:

$$\textbf{pattern}(\texttt{spr:}) = \textbf{crt}(\texttt{spr:},\textbf{qual}(\texttt{spr:}))$$

We may split static pattern reference declarations into two cases:

- `spr: q(# ... #)`, declaring the pattern $(\textbf{location}(\texttt{spr:}), \texttt{q(\# ... \#)})$.
- `spr: q`, where `spr:` gets bound to $\textbf{crt}(\texttt{spr:},\texttt{q\#\#})$.

The last case is new, and allows static pattern references to be bound to existing qualifications. For example, this allows a dynamic qualification to be evaluated and then bound to a static pattern reference, leading to attributes whose qualification is not known until runtime. As will be demonstrated later, the result is a great flexibility while retaining the need to do runtime type checks only on potentially dangerous assignments, as opposed to the need in dynamically typed languages to do an implicit type-check (method lookup) on each message send.

Dynamic Object References: `dor: ^q;`

The initial value of $\textbf{object}(\texttt{dor:})$ is `NONE`, with possible values given by a generalisation of Equation 2 in Section 2.2:

$$\textbf{object}(\texttt{dor:}) \leq \textbf{qual}(\texttt{dor:})$$

Dynamic Pattern References: `dpr: ##q`

The initial value of $\textbf{pattern}(\texttt{dpr:})$ is `NONE`, with possible values given by Equation 3 from Section 2.2:

$$\textbf{pattern}(\texttt{dpr:}) \leq \textbf{qual}(\texttt{dpr:})$$

8 Restrictions on Qualifications

We have described a generalised concept of qualified, (typed), attributes, adding considerably to the expressive power of the BETA language. However, not all the described qualifications on dynamic references can be efficiently enforced.

As described in Section 2, the BETA type system is not completely statically checkable, and therefore needs a minimal number of runtime type checks. Generalising the type system should not make it inherently more difficult to check. For this reason, we choose to allow only qualifications that can be enforced by a combination of compile time checking and occasional (constant-time) runtime checks on potentially dangerous assignments.

In general, we only allow a qualification `q` on a dynamic reference `dr:`, if it is checkable by only monitoring assignments to `dr:`. If the qualification itself is dynamic, (as explained in Section 3.1), not only the attribute itself must be monitored, but also its qualification, resulting in complex relationships between the allowed values of dynamic attributes. Although it is possible to implement a runtime system that maintains constraint graphs in order to enforce dynamic qualifications, we choose to disallow dynamic qualifications to avoid severe unpredictable runtime overheads. Thus, if the declaration of a dynamic reference attribute results in a qualification equation, (see Section 7), with a dynamic right-hand side, the declaration will be declared illegal, and rejected by the compiler.

Example 7. Consider the declaration of a dynamic object reference attribute `aShape` with a dynamic qualification:

```
Window: (# Shape: (# ... #) #);
aWindow: ^Window;
aShape: ^aWindow.Shape;
```

Here, the qualification constraint **object**(`aShape:`) $\leq$ `aWindow.Shape##` must be enforced, which requires that we monitor the value of **object**(`aShape:`) *and* the value of **object**(`aWindow:`), since **extension**(`aWindow.Shape`) changes if **object**(`aWindow:`) changes. In the general case, knowing where to do runtime checking therefore requires either global information at compile time, or the maintenance of a constraint graph at runtime. We do not find any of these options acceptable, and hence choose to disallow this kind of qualification. □

In fact, the current Mjølner BETA compiler does allow the above kind of dynamic full pattern qualification. However, the dynamic full pattern qualification `aWindow.Shape` is interpreted as the static partial pattern qualification `Window.Shape`. and therefore only the less restrictive **object**(`aShape:`) $\leq$ `Window.Shape` constraint is enforced. The following scenario is thus accepted without compile time or runtime errors:

```
do &Window[]->aWindow[];
   &aWindow.Shape[]->aShape[]; (* OK *)
   &Window[]->aWindow[]; (* object(aShape:) illegal!! *)
```

Fortunately we can disallow this type of dynamic full pattern declaration with virtually no loss of expressive power, since we can convert the dynamic qualification into a static qualification:

```
   aDynWin: ^Window;
do &Window[]->aDynWin[];
   (# aStatWin: @aDynWin;
      aShape: ^aStatWin.Shape;
   do &aStatWin.Shape[]->aShape[]; (* OK *)
      &Window[]->aDynWin[];        (* OK *)
      &Window[]->aStatWin[]; (* compile time error *)
   #)
```

Note that `aStatWin.Shape` is a static qualification whose extension cannot change, and therefore does not need to be monitored at runtime: Assignment to `aDynWin` does not change the extension of `aStatWin.Shape`, and assignment to `aStatWin` is a compile-time error.

However, if we were content with the knowledge that **object**(`aShape:`) qualifies to the `Shape` pattern of *some* window, and do not care about the exact window, we should have used a partial pattern qualification:

```
   aDynWin: ^Window;
   aShape: ^Window.Shape;
do &Window[]->aDynWin[];
   &aDynWin.Shape[]->aShape[]; (* OK *)
   &Window[]->aDynWin[];       (* OK *)
```

Example 8. Another illegal example is the declaration of object references with dynamic partial object qualifications:

```
Vehicle: (# owner: ^Person #);
aVehicleOwner: ^Vehicle.owner;
```

Again, the problem is that the qualification itself is dynamic: In this case, a `Vehicle` changing owner may result in the breaking of the qualification constraint on `aVehicleOwner`, since the previous owner may thereby stop being the owner of any vehicle. □

Static Binding of Dynamic Qualifications The decision to disallow dynamic qualifications for dynamic references is not a real limitation. In general, if we wish to specify something like:

```
   anObject: ^dynqual
do o[]->anObject[]; ...
```

where `dynqual` is some dynamic qualification expression, we can transform it into:

```
o[]->(# statqual: dynqual; anObject: ^statqual;
     enter anObject[] do ...
     #)
```

which enforces a static qualification on the `anObject` reference. This technique is called *static binding of dynamic qualification* and can be used to resolve all dynamic qualifications.

Example 9. As a more complete example of using this technique, consider a `List` pattern, where the qualification of the types of objects in the `List` can be controlled dynamically:

```
List:
  (# elementType: ##Object;
     insert:
       (# elmType: elementType; new: ^elmType;
       enter new[] do ...
       #);
     changeElementType:
       (# newElementType: ##Object;
       enter newElementType##
       do scan
          (#
          do (if not (current##<=newElementType##) then
                current[]->delete
             if)
          #);
          newElementType##->elementType##
       #);
  #);
```

We may now create a `List` object to contain `Window.Shape` objects:

```
   winShapeList: @List;
do Window.Shape##->winShapeList.changeElementType;
   &aWindow.Shape[]->winShapeList.insert;
```

If we later choose to restrict the same `List` object to contain only `aWindow Shape`s, we can dynamically choose to do so by:

```
do aWindow.Shape##->winShapeList.changeElementType;
```

causing all `Shapes` that do not qualify to `aWindow.Shape##` to be removed from the list. At all times during the lifetime of `winShapeList`, the elements in the list will automatically conform to the qualification, given by the `elementType` attribute.

We can also create a `List` that can only contain instances of a pattern unknown until runtime, for example because it is loaded dynamically:

```
   aList: @List;
   aPattern: ##Object;
do 'aPatternName'->loadPattern->aPattern##;
   aPattern##->aList.changeElementType;
   &aPattern[]->aList.insert;
```

It should be noted that runtime type checks are clearly still needed to enforce the typing in these examples, but considering the dynamic nature of the examples, avoiding runtime checking would also be a surprising achievement. However, the example does ensure that the element type is enforced, although it is not known until runtime. □

The generalisation of BETA qualifications has resulted in a type-system with an increased expressive power. Several of the new possibilities are easy to implement, and add considerably to the expressive power of the BETA language, without adding new syntax.

Static attributes do not require continuous monitoring, since they are bound to a value that is acceptable at creation time, and afterward are not allowed to change. Therefore, any qualification expression is acceptable as the type of a static attribute.

For dynamic references, we have chosen to disallow attribute declarations with dynamic qualifications, since their enforcement requires either global compile-time knowledge, or the maintenance of a constraint-graph at runtime. Neither of these options are acceptable in a language with support for separate compilation and efficient execution.

9 Future Work

Static type systems are continuously under attack from dynamic type systems. Dynamic type systems play an important role in early software development such as prototyping and explorative programming. Conversely, static type systems show their strength in industrial software production. This tension has resulted in an almost religious war amongst language designers for at least the last decade, but it seems that it is now time to join forces: The last few years have seen much research in type inference of dynamic type systems, extracting type information from a given program. The information may then be utilised as an aid to the programmer, giving him a better understanding of the types implicitly specified in the program, but also as a help to the compiler and runtime system in order to optimise the runtime efficiency of the program [Chambers & Ungar 90, Oxhøj et al. 92, Plevyak & Chien 95, Agesen 95].

The work reported in this paper only deals with this tension in a limited way, by allowing more flexible type declarations without compromising the level of static type-checking already supported. However, industrial strength type systems must look for better ways to support early software development. One way to go is to investigate the possibilities for introducing dynamic typing components into static type systems.

10 Conclusion

This paper examined the current BETA type system and proposed generalisations that can be introduced without extensive runtime overhead, and without changing the language syntax. The paper investigated the impact of these generalisations on other parts of the language, in order to regain an orthogonal and internally consistent language.

We have investigated the tension between BETA block structure and the type system, giving rise to the introduction of the concepts of full and partial pattern qualifications. Partial pattern qualifications are types that do not uniquely relate to a single pattern, resulting in increased flexibility in expressing dynamic references and generics.

Analysis of BETA qualification expressions revealed that any conceivable path expression can be given a perfectly logical type-interpretation, giving rise to increased expressive power without changing the language syntax. We have examined the generalised qualifications, finding that the BETA language has been augmented with several new and powerful mechanisms, including creation-time bound immutable object references, creation-time bound immutable pattern references, virtual object binding, and more general generics.

Finally, we have examined the runtime overhead of the new facilities, revealing that a particular kind of qualifications, the dynamic qualifications, impose severe runtime overhead. Albeit dynamic qualifications are very powerful, we have excluded them from this proposal due to their runtime overhead. Further investigations into these qualifications are needed in order to fully understand the ramifications. Fortunately, in most cases where dynamic qualifications seem the obvious choice, it is found that the type constraints needed can be specified by either using the newly introduced partial qualifications, or by re-binding dynamic qualifications to static pattern references.

In general terms, we have extended the flexibility of the BETA type system without imposing any additional runtime overhead. Object relations that naturally arise in object-oriented analysis and design have been given direct language language support, without adding new language concepts or syntax. The flexibility has been gained from a more general interpretation of existing concepts.

11 Acknowledgements

The authors would like to thank Erik Ernst, René Wenzel Schmidt, and the anonymous ECOOP referees who read earlier drafts of this paper and provided many valuable comments.

References

[Agesen 95] O. Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95)*, Aarhus, Denmark, August 1995.

[Agesen et al. 89] O. Agesen, S. Frølund, and M. Olsen. Persistent and Shared Objects in BETA. Master's thesis, Department of Computer Science, University of Aarhus, April 1989.

[Black et al. 87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[Brandt & Schmidt 96] S. Brandt and R. W. Schmidt. The Design of a Meta-Level Architecture for the BETA Language. In C. Zimmermann, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection.* CRC Press Inc, Boca Raton, Florida, 1996.

[Chambers & Ungar 90] C. Chambers and D. Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[Dahl et al. 84] O. Dahl, B. Myrhaug, and K. Nygaard. Simula 67 Common Base Language. Pub. 725, Norwegian Computing Center, Oslo, 1984.

[Madsen & Møller-Pedersen 89] O. L. Madsen and B. Møller-Pedersen. Virtual Classes – A Powerful Mechanism in Object-Oriented Programming. In *Proceedings of the Fourth Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *Sigplan Notices*, October 1989.

[Madsen et al. 93a] O. Madsen, B.Magnusson, and B. Pedersen. Strong typing of Object-Oriented Languages Revisited. In J. Knudsen, O. Madsen, B. Magnusson, and M. Löfgren, editors, *Object-Oriented Environments.* Prentice Hall, September 1993.

[Madsen et al. 93b] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison Wesley, Reading, MA, June 1993.

[Meyer 92] B. Meyer. *Eiffel, The Language.* Prentice Hall, 1992.

[Omohundro 93] S. Omohundro. The Sather Programming Language. *Doctor Dobb's Journal*, pages 42 – 48, October 1993.

[Oxhøj et al. 92] N. Oxhøj, J. Palsberg, and M. I. Schwartzbach. Making type inference practical. In *Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP'92)*, Utrecht, The Netherlands, June 1992.

[Plevyak & Chien 95] J. B. Plevyak and A. A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *Proceedings of the Ninth Conference on*

Object-Oriented Programming Systems, Languages and Applications (OOPSLA'94), Portland, OR, October 1995.
[Shang 95] D. L. Shang. Covariant Deep Subtyping Reconsidered. *ACM SIGPLAN Notices*, 30(5):21 – 28, May 1995.
[Stroustrup 93] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, 1993.

A A BETA Primer

This section briefly introduces the BETA language, and may be skipped by readers familiar with BETA. For a comprehensive description of the language, readers are referred to [Madsen et al. 93b].

A.1 Patterns

A BETA program execution consists of a collection of objects. An object is an instance of a pattern. The pattern construct unifies programming language concepts such as class, generic class, method, process, coroutine, and exception. This results in a syntactically small language, but is also a point of confusion for programmers with background in more traditional languages because classes and methods in BETA have the same syntax. In this paper, we use the convention that patterns with names beginning with an upper-case letter correspond to classes, whereas patterns with names beginning with a lower-case letter correspond to methods[7].

A pattern declaration has the form shown in Figure 7, where `P` is the name

```
P: Super
   (# Decl1; Decl2; ... Decln      (* attribute-part *)
   enter In                            (* enter-part *)
   do Imp1; Imp2; ... Impm                (* do-part *)
   exit Out                              (* exit-part *)
   #)
```

Fig. 7. Structure of a pattern declaration

of the pattern and `Super` is an optional superpattern for `P`. The attribute-part is a list of declarations of reference attributes, part objects, and nested patterns. The most important forms taken by `Decli` are the following:

– `R1: ^Q` where `Q` is a pattern, declares `R1` as a *dynamic reference* to instances of (subpatterns of) `Q`. `R1` is similar to a pointer in C++, and may thus refer

[7] Occasionally, the same pattern is used as both a method and a class, although such cases do not occur in this paper.

to different objects at different points in time. The value of `R1` is changed through destructive assignments of the form `newvalue[]->R1[]`.
- `R2: @Q` where `Q` is a pattern, declares `R2` as a *static reference* to an instance of the pattern `Q`. `R2` is also called a *static part object*. “Static” means that `R2` is an immutable reference that cannot be changed by destructive assignment. It is bound to a new instance of `Q` when the object containing the `R2` attribute is created.
- `R3: Q (# ... #)`. `R3` declares and names a nested pattern `R3` with super pattern `Q`. `R3` can be used as a nested class or as a method pattern.
- `R4:< Q`. Declares `R4` as a new *virtual* pattern that can be specialised in subpatterns of `P`. The operators `::<` and `::` are used to specialise a virtual pattern.
- `R5: ##Q`. Declares `R5` as a *dynamic pattern reference*, allowed to refer to the pattern `Q`, or any subpattern of `Q`. Dynamic pattern references are in practice used as dynamic method and function references, class references, and dynamic qualification references (described later).

The enter-part, `In`, describes input parameters to instances of `P` (formal parameters), the do-part describes the actions performed by executing instances of `P`, and the exit-part, `Out`, describes the output parameters (return values).

```
Calc:
  (# add:                     (* add is non-virtual *)
       (# a,b,c: @Integer;
       enter (a,b)
       do a+b->c; c->display;
       exit c
       #);
     clear:<                  (* clear is virtual   *)
       (#
       do 0->display; INNER; (* INNER used for specialisation *)
       #);
     display: @               (* display is a part object *)
       (# value: @Integer;
       enter value
       do value->screen.putint;
       #);
  #);
```

Fig. 8. An example BETA pattern.

Figure 8 shows a BETA fragment defining a simple calculator. The `Calc` pattern is used as a class. The `add` pattern is a non-virtual pattern, serving as a method for instances of the surrounding `Calc` pattern. The `clear` pattern is a virtual pattern, serving as a virtual method which can be specialised in subpatterns of `Calc`. Finally, `display` is a static part object modelling the display of the calculator. Creation of a `Calc` instance and invocation of its `add` method is done by:

```
    aCalc: ^Calc; value: @Integer;
do &Calc[]->aCalc[]; (1,2)->&aCalc.add->value;
```

First, an instance of `Calc` is instantiated, using the object creation operator `&`. Object expressions followed by the box (`[]`) operator means "object reference". Thus, evaluation of `&Calc[]` creates a new calculator object and returns an object reference which is assigned to the dynamic object reference `aCalc`. `&aCalc.add` creates an instance of the `add` pattern and executes its do-part. The arrow (`->`) exiting `(1,2)` assigns the actual parameter list `(1,2)` to the formal enter list `(a,b)` of the method object, while the arrow exiting `&aCalc.add` assigns the formal exit list `(c)` to the actual exit list, in this case one-element list `(value)`. Syntactic sugar allows the `&` sign to be omitted in the case of method executions. Thus, we may instead write:

```
    aCalc: ^Calc; value: @Integer;
do &Calc[]->aCalc[]; (1,2)->aCalc.add->value;
```

For readability, we shall use the syntactically sugared method call syntax.

A.2 Specialisation

In BETA, a virtual pattern can be *specialised* in a subpattern, not overridden. Execution of an object always begins at the top of the inheritance hierarchy, and control is transfered down the specialisation chain at each `INNER` imperative. The `INNER` imperative has no effect in the most specific pattern.

For example, the virtual pattern, `Calc.clear`, may be extended in subpatterns of `Calc`:

```
    Calc2: Calc                   (* subpattern of Calc *)
      (# clear::<                 (* further binding *)
           (# do INNER; 'Clear'->screen.putline #);
      #);
    aCalc2: @Calc2;
do aCalc2.clear;
```

`Calc2` inherits from `Calc`, and extends (`::<`) the `clear` virtual pattern to report whenever the screen is cleared. The execution of `aCalc2.clear` begins at the do-part of `Calc.clear` (in Figure 8), and then, at the `INNER` imperative, control is transferred to the do-part of `Calc2.clear`. The `INNER` imperative in `Calc2.clear` has no effect, since `clear` has not been further extended. When the do-part of `Calc2.clear` terminates, control returns to the do-part of `Calc.clear` which returns directly to the caller of `aCalc2.clear`. Thus, execution of `aCalc2.clear` first clears the display, and then writes `Clear` to the terminal screen.

Specialisation of virtual patterns using `::<` is called *further binding*. After further binding, the `clear` pattern is still virtual, and can be further extended in subpatterns of `Calc2`. Using a *final binding* (`::`) the pattern is extended and at the same time converted to a non-virtual pattern which cannot be further extended.

It is possible to specialise a method without creating a new pattern: Firstly, assuming that `aCalc2` is the only instance of `Calc2` we need, `aCalc2` could be declared as a *singular object*, as shown in Figure 9a.

```
   aCalc2: @Calc
     (# clear:: (* final binding *)
          (#
          do INNER;
             'Clear'->screen.putline;
          #);
     #);
do aCalc2.clear;
```

```
do aCalc2.clear
     (#
     do 'Very '->screen.puttext
     #);
```

Fig. 9. (a) Singular object declaration. (b) Call-spot specialisation.

This ensures that `aCalc2` will be the only object of its kind, and avoids name space pollution with the superfluous `Calc2` name. The example demonstrates that instance creation and specialisation may happen simultaneously. Secondly, specialisation and execution can happen simultaneously. For example, the method `aCalc2.clear` can be specialised directly at the call-spot, as shown in Figure 9b, where the display is cleared, and `Very Clear` is written to the terminal screen.

A.3 Dynamic Pattern References

An example usage of the *dynamic pattern reference* attribute kind is shown below:

```
   calcP: ##Calc; aCalc: ^Calc;
do Calc2##->calcP##; &calcP[]->aCalc[];
```

The dynamic pattern reference `calcP` is allowed to refer to the `Calc` pattern or subpatterns of `Calc`. Above, the `Calc2` pattern is assigned to `calcP`, and then an instance of the `Calc2` pattern now referred to by `calcP` is created and assigned to `aCalc`. Dynamic pattern references may be compared using relational operators `<=`, `>=`, and `=`, in order to check inheritance relationships.

This article was processed using the LaTeX macro package with LLNCS style