

Using an Object-Oriented Predicative Style to Solve the Invoicing Case Study

Richard F. Paige

*Department of Computer Science, York University,
Toronto, Ontario, Canada, M3J 1P3. paige@cs.yorku.ca*

Abstract. We apply the predicative programming formal method of [2] in specifying and designing a solution to the invoicing case study. The method is used in an object-oriented style: first-class citizens of the descriptions are classes and objects. We discuss some of the advantages to using the predicative method in its object-oriented style, as well as its limitations in this problem domain.

1 Introduction

This paper constitutes a formal specification and design for the invoicing system [3]. The particular method used in solving the problem is predicative programming, a program design calculus due to Hehner [2]. Because predicative programming may be unfamiliar to some readers, we include a short section on its introduction, emphasizing its notations for describing state, and for specifying operations on a system.

While predicative programming is not considered to be an object-oriented method, according to most standard definitions, it can be used in such a style. In our solution to the invoice case study, we apply predicative programming in an object-oriented style. We do this because the use of classes and objects has been shown in practice to be appropriate for describing requirements, system specifications, and designs. Furthermore, the requirements, as we understand and formalize them, lend themselves naturally to an object-oriented style of description. Section 2.2 introduces the object-oriented style, which is new.

We apply the predicative programming style with two goals in mind: to help understand the problem clearly; and, to eventually allow us to refine our system descriptions to programs. While we do not have the space to present detailed refinements here, it has been shown that predicative programming is well-suited for refinement [2].

In the process of solving the case study, we also describe how using the predicative method and its object-oriented style has helped us (or hindered us). We clearly describe our design decisions with respect to the requirements, and explain how the decisions are reflected in the descriptions of system components.

2 Predicative Programming

Predicative programming is a program design calculus like Morgan's refinement calculus [5], but unlike the latter treats programs as specifications. In this approach, programs

and specifications are predicates on pre- and post-state (final values of variables are annotated with a prime; initial values of variables are undecorated). The weakest predicate specification is \top (“true”), and the strongest specification is \perp (“false”). Refinement is just boolean implication.

Definition 1. A predicative specification P is refined by a specification Q if $\forall \sigma, \sigma' . (P \Leftarrow Q)$, where σ and σ' denote the pre-state and post-state, respectively.

The refinement relation enjoys various properties that allow specifications to be refined by parts, steps, and cases. As well, specifications can be combined using the familiar operators of boolean theory, along with all the usual program combinators, as well as combinators for parallelism and communication through channels. One particular novelty with predicative programming is that recursive programs can be developed rather than iterative programs, using recursive refinement rules. It has been suggested that this can simplify the process of developing certain programs [2], since it eliminates the need to construct invariants before developing loops.

Predicative programming can be used to describe objects and classes. In order to describe such entities, we need to introduce the predicative notation for types, i.e., bunches.

2.1 Bunches and Types

A bunch is a collection of values, and can be written as in this example: $2, 3, 5$. A bunch consisting of a single element is identical to the element. Some bunches are worth naming, such as *null* (the empty bunch), *nat* (the bunch of natural numbers), *int* (the bunch of integers), *char* (the bunch of characters) and so on. More interesting bunches can be written with the aid of the solution quantifier \S , sometimes pronounced “those”, as in the example

$$\S i : int \cdot i^2 = 4$$

And we use the asymmetric notation $m, ..n$ for $\S i : int \cdot m \leq i < n$. Bunches can also be used as a type system, as in

$$\mathbf{var} \ x : nat$$

perhaps with restrictions for easy implementation. Any bunch (even an element, or *null*) can be used as a type. Bunches can also be used in arithmetic expressions, where the arithmetic operators distribute over bunch union (comma):

$$nat = 0, nat + 1$$

We have twice previously used a colon in expressions involving bunches; more generally, $A : B$ is a boolean expression saying that A is a subbunch of B . For example,

$$\begin{aligned} 2 &: nat \\ nat &: int \end{aligned}$$

When we use a bunch as a type, like in the declaration of x above, we can thereafter use the variable to represent zero *or more* elements of the specified type. So, for our example declaration of $x : nat$, x can stand for zero, one, two, or any other quantity of natural numbers. So x could be the element 1, or the bunch 5, 6, 7. The subbunch relation $x : nat$ holds true in both cases.

In predicative programming, we write functions in a standard way, as in the example $\lambda n : nat \cdot n + 1$. The domain of a function is obtained using the Δ operator. If the function body does not use its variables, we may write just the domain and body with an arrow between. For example, $2 \rightarrow 3$ is a function that maps 2 to 3, which we could have written $\lambda n : 2 \cdot 3$, with an unused variable.

When the domain of a function is an initial segment of the natural numbers, we sometimes use a list notation, as in $[3; 5; 2; 5]$. The empty list is $[nil]$. We also use the asymmetric notation $[m; ..n]$ for a list of integers starting with m and ending before n . List length is $\#$, and list catenation is $+$.

Function formation distributes over bunch union, and so a function whose body is a union is equal to a union of functions.

$$(\lambda v : D \cdot A, B) = (\lambda v : D \cdot A), (\lambda v : D \cdot B)$$

A union of functions applied to an argument gives the union of the results

$$(f, g) x = fx, gx$$

A function f is included in a function g according to the *function inclusion law*.

$$(f : g) = ((\Delta g : \Delta f) \wedge (\forall x : \Delta g \cdot fx : gx))$$

Thus we can prove

$$(f : A \rightarrow B) = ((A : \Delta f) \wedge (\forall a : A \cdot fa : B))$$

Using inclusion both ways round, we find function equality is as usual:

$$(f = g) = ((\Delta f = \Delta g) \wedge (\forall x : \Delta f \cdot fx = gx))$$

By defining *list* as

$$list = \lambda T : \Delta list \cdot 0, ..\#(list T) \rightarrow T$$

then *list T* consists of all lists whose items are of type T .

The selective union $f | g$ of functions f and g is a function that behaves like f when applied to an argument in the domain of f , and otherwise behaves like g .

$$\begin{aligned} \Delta(f | g) &= \Delta f, \Delta g \\ (f | g)x &= \mathbf{if} x : \Delta f \mathbf{ then } f x \mathbf{ else } g x \end{aligned}$$

One of the uses of a selective union is to write a (selective) list update. For example, if $L = [2; 5; 3; 4]$ then

$$2 \rightarrow 6 | L = [2; 5; 6; 4]$$

Another use is to create a record structure, as in

$$\text{"name"} \rightarrow \text{"Smith"} \mid \text{"age"} \rightarrow 33$$

which is included in

$$\text{"name"} \rightarrow \text{list char} \mid \text{"age"} \rightarrow \text{nat}$$

2.2 Object-oriented style

With the bunch notion of types, classes can be described as records, where each attribute of the class may be a state component or a functional method. There are three components to a predicative class description: an interface specification, where the types of attributes and functional methods are provided; a class definition, where the bodies of functional methods are provided; and finally, specifications of state-change methods. For example, a stack ADT can be specified as follows. A stack has one attribute, *contents*, which is a list of elements of type T . It also has three methods, *push*, *pop*, and *top*. The former two methods are state-changing methods that concatenate or remove elements from the top of the stack; we explain how this kind of method is specified in a moment. *top* is a functional method. In describing a stack class, we first specify its interface.

$$\text{StackInterface} = \text{"contents"} \rightarrow \text{list } T \mid \text{"top"} \rightarrow T$$

The interface of the class describes a *contents* list attribute, and the interface of the *top* method. To specify the definition of the *top* method, we refine the bunch *StackInterface* to include only those values of *top* that satisfy the usual definition of the operation.

$$\text{Stack} = \{s : \text{StackInterface} \cdot s\text{"top"} = s\text{"contents"}(\#s\text{"contents"} - 1)$$

The *Stack* class is a bunch. The bunch contains those elements of *StackInterface* in which the *top* attribute is the value of the last element in the *contents* list.

An object is an instance of a class. Since a class is just a type, object creation is just variable declaration. To declare an object of class *Stack*, we write

$$\text{var } s : \text{Stack}$$

and access the *contents* field of object s by dereferencing s , as $s\text{"contents"}$. In general, the dereferenced field may be any functional method or attribute of the object. To assign a value to field *contents*, we just carry out a record field assignment, written either as $s\text{"contents"} := \text{value}$, or as the overriding $\text{"contents"} \rightarrow \text{value} \mid s$.

According to the declaration of the *Stack*, above, s is an object of type *Stack*. But because types are bunches, we can also use s to represent an arbitrary number of *Stacks*, as we saw in Section 2. We will make use of this capability in the case study.

We are not limited to specifying static collections of objects: objects can be created dynamically (either by a system, or by the environment in which the system is operating) and added to a system. This is demonstrated in the second part of the case study in Section 5, where new orders can be added to the invoice system.

To describe state-change methods, each method of a class is a function external to the class: each function takes an instance of the class as argument, and results in a (possibly changed) instance of the class. Suppose f is to be a state-change method of class C . Then we define $f : C \rightarrow C$, which takes an object of class C as argument, and returns the changed object. To call the method f , applied to an object c of class C , we write $c.f$ which we define as sugar for the specification

$$c := f(c)$$

In practice, a programmer would likely implement the specification as a state change to the object c . In effect, this model of state-change methods is the same as that used in compilers for object-oriented languages: for a method of a class, there is one copy of the method which can be applied to any object of that class.

With this formulation of classes, inheritance (both interface and class) is easy to define: interface inheritance is catenation of class interfaces; while class inheritance is catenation of class definitions. Derived classes are then subclasses (and subbunches) of their parent classes. State-change methods associated with base classes are inherited by derived classes in the following sense. The derived classes can use the state-change methods within their own state-change methods, because of bunch typing rules.

3 The Invoice Problem

In studying the problem, we follow the suggestions of Jackson [4], and consider the phenomena of the *application domain* and the phenomena that are shared between the application domain and the *machine* (which is the system that we want to build). Some phenomena of the application domain include: warehouse, products, quantities, orders, stocks, customers, and so on. Phenomena shared between the application domain and the machine include: orders, products, quantities, and stocks. In our design, we describe the shared phenomena in a system specification. Detailed design involves implementing the specification, typically in a programming language.

The general method associated with the predicative programming style requires us to answer the following questions, with respect to the shared phenomena.

- *What information should be represented in the machine?* This includes description of phenomena solely in the machine, and may include description of shared phenomena.
- *How should this information be represented in an object-oriented style?* This corresponds with the usual step in most object-oriented design methods: identification of classes and objects, and their interrelationships.
- *What are the operations that the system should be able to perform?* The operations will be described, according to the style, as features of classes (i.e., functional or state-change methods).

3.1 What information should be represented?

We identify two important shared phenomena from the requirements: the *orders*, and the *stock*. Two questions immediately raise themselves at this point.

1. **What constitutes an order?** From the requirements, an order has exactly one reference to a product in a certain quantity. As well, since the focus of the system is to invoice orders (i.e., change their status from “pending” to “invoiced”), an order must also include its status. Furthermore, a reference to a product may be on two or more different orders. Therefore, our data description of orders must somehow maintain the distinctiveness of orders that reference the same product.
2. **What is stock?** Stock is a quantity of some product that is available to be invoiced.

The predicative method suggests that an order be represented as a *class*, while stock should be represented as a function from product to available quantity. This will be expressed in our formalization, starting in Section 4.

We have not distinguished between part 1 and part 2 of the case study in describing information representation: our analysis of the requirements, and the predicative style, suggests that the same information should be represented in both part 1 and part 2. Only the system operations should change. An object-oriented or object-oriented style is therefore appropriate for describing such reusable system components.

3.2 What are the system operations?

In answering this question, we are concerned with the issue of how the system can be used (directly or indirectly) by clients. The operations of the invoicing system will change the system state. For the first part of the case study, we identify only one operation, *UpdateSystem*, which changes the status of an order from “pending” “to invoiced”. In the second part of the study, we require several further operations:

- an operation to add a new order to the system
- an operation to remove an order (e.g., after it has been invoiced, or after cancellation by a customer) from the system
- an operation to add more stock to the system (e.g., after invoicing all of the quantity currently in stock)
- an operation to add a new stock item to the system

In formalization, when applying the predicative method, we will need to ask more detailed questions, such as “When can an order be added to the system?” and “When can the order be invoiced?”

The next section describes a formalization of the information representation and the system operations, for the first part of the case study. Section 5 describes the system operations for the second part of the study.

4 Case 1: Invoicing Orders

We now refine the informal descriptions of information representation, and operations, given in the preceding section. From our answer to the question *What constitutes an order?*, we construct a model of the shared phenomena of orders as a class.

$$\text{Order} = \text{“product”} \rightarrow \text{PRODUCT} \mid \text{“quantity”} \rightarrow \text{nat} \mid \\ \text{“status”} \rightarrow \text{STATUS} \mid \text{“id”} \rightarrow \text{ID}$$

Each attribute of the class corresponds with an attribute identified from the requirements, with one exception: the attribute *id*, which we discuss momentarily. The bunch types *PRODUCT* and *ID* are left unspecified. In applying the predicative method, we analyze the requirements and determine that they say nothing about what constitutes a product. *ID* will be described shortly. The bunch *STATUS* can be defined directly from the requirements. It is a bunch of two values:

$$STATUS = PENDING, INVOICED$$

It is useful at this time to briefly contrast the definition of *Order* with a Z definition. To describe exactly one order in Z, we could write the state schema

<i>Order</i>
<i>product</i> : <i>PRODUCT</i>
<i>quantity</i> : \mathbb{N}
<i>status</i> : <i>STATUS</i>
<i>id</i> : <i>ID</i>

where *PRODUCT*, *STATUS*, and \mathbb{N} would be sets (and not bunches). In a Z specification, operations on *Orders* would then be written—as Z operation schemas—and the state of the invoicing system would be modeled, most likely via a set of *Orders*.

The definition of *Order* includes an attribute *id*. This attribute arose in response to the question: **can the same product and the same quantity be referenced on different orders?** The *id* field (of type *ID*) is used to uniquely identify orders; without it, there would be no way of distinguishing two identical, but distinct orders. We require that a system invariant states that all orders have distinct *id* attributes. We maintain this invariant through operations: in each operation that adds an order to the system, we ensure that the operation maintains the invariant. An alternative approach would be to add a functional method to our specification of the system that states the invariant. Then, an order cannot be added to the system if it violates the invariant. This latter approach is akin to that used in Z: schema invariants can be used to express invariant system constraints. We do not use the Z-like approach here, because there will be only one operation (*AddNewOrder* in the second part of the case study, Section 5) that could potentially violate the invariant. For reasons of conciseness, we include a verification of the invariant in the operation itself. We could use the more Z-like approach without difficulty.

We now address the question: **what is stock?** According to the requirements, stock is a quantity of some product that is available to be invoiced. The stock in the system is described using a function from *PRODUCT*s to natural numbers, where the range of this function represents the quantity of *PRODUCT*s in stock. A function is used for several reasons: so that it is easy to describe lookups; because it is, in our opinion, an appropriate way to describe the shared phenomena of stocks in the problem domain; and, because it makes the addition of extra stock (or new products) easy to describe.

$$stock : PRODUCT \rightarrow nat$$

The next question that arises is: **what is the invoicing system?** We have identified its two key components—orders, and stock—but we must now describe the whole. The

predicative method suggests that the invoicing system itself be modeled as an object with two attributes: *orders*, of class *Order*, and a *stock* entity.

$$\text{var } System : \text{“orders”} \rightarrow Order \mid \text{“stock”} \rightarrow (PRODUCT \rightarrow nat)$$

The *System* is an object, and not a class, because it is unique, according to our interpretation of the requirements and our formalization. The predicative method allows specification of a unique *System* entity, by declaration.

Another question arises at this point, due to our formalization: **is there a limit on the size of the system?** According to our formalization, the *System* can contain an arbitrary number of *Orders*, limited only by the cardinality of the *ID* bunch. The *stock* is limited only by the cardinality of the *PRODUCT* bunch in its domain. By studying the requirements, we find no hint of limits that are required on the number of orders or on the number of stocked products, so the formalization seems reasonable.

Notice that because the attribute *orders* is of type *Order*, which is a bunch, the system can use the *orders* attribute to represent more than one order. Equally, we could have defined the *orders* field of the *System* object as a set, which would allow representing multiple orders as well. We use bunches instead of sets because bunches are the standard in predicative programming.

After modeling the system data, we turn to modeling operations and the invoicing process. In doing so, we address the following questions.

- **What are the system operations for this part of the case study?** We do not have to take into account data flows. Therefore, we are only worried about the operations for the transformation of an order’s status. There is only one system operation: *UpdateSystem*, which, given an order, changes its status from *PENDING* to *INVOICED*. Invoicing is therefore an atomic, mutually exclusive operation: no two invoicing operations can occur simultaneously. If we wanted to model a more complex system where this is allowed, then predicative programming would be well-suited to the task: the method can be used for describing monitors or semaphores, which could be used by concurrent processes (also describable in predicative notation) to access the system stock and orders state.
- **When can an order be invoiced?** The answer comes directly from the requirements: whenever the quantity of the ordered product that is requested is in stock. The requirements suggest that an order cannot be invoiced if the quantity requested is not in stock. Successful invoicing of an order will also have a side-effect: it changes the stock available, providing the invoice can be made, by decreasing the amount of product in stock by the amount ordered.
- **How do we ensure that invoicing can actually occur?** A request for invoicing cannot be satisfied if the requested stock is not available. But the system, in this part of the case study, is not able to add stock or add orders. Therefore, it is assumed that the environment will initialize the stock and available orders in some way. According to the requirements, the machine that we describe in this part does not perform any stock maintenance (beyond that associated with updates after an order), nor should it add or remove orders.
- **Can orders be placed for products that are not in stock?** A product that is not in stock may have quantity 0, or may simply not be in the domain of the *stock*

component of the *System*. Our formalization suggests that an order can indeed be placed for a product that is not in stock. We assume that the mechanisms for adding stock will be used at some point. However, such an order cannot be invoiced, at least until sufficient stock has been added.

The operation *UpdateSystem*, identified from the requirements, changes an order *o* from *PENDING* to *INVOICED*, providing that the quantity of product requested in the order is in stock. Before describing the *UpdateSystem* operation, we provide a piece of sugar. We let *amount* stand for the stocked amount of a product that is requested on an order *o*.

$$amount = System\text{“stock”}(o\text{“product”})$$

The *UpdateSystem* operation is then as follows.

$$\begin{aligned}
 UpdateSystem &= \lambda o : Order \cdot \\
 &\quad \mathbf{if} (o : System\text{“orders”} \wedge amount \geq o\text{“quantity”}) \mathbf{then} \\
 &\quad \quad \text{“orders”} \rightarrow UpdateOrder(o, System\text{“orders”}) \mid \\
 &\quad \quad \text{“stock”} \rightarrow (o\text{“product”} \rightarrow (amount - o\text{“quantity”}) \mid System \\
 &\quad \mathbf{else} System
 \end{aligned}$$

UpdateSystem takes one order *o*, and invoices it (changes the *status* of *o* to *INVOICED*), providing that sufficient quantity of the requested product is in stock. Simultaneously, the *stock* is updated to remove the requested amount from stock.

To write such an operation in Z, we could either write *UpdateSystem* as a function on objects of type *Order* (where *Order* is the state schema as given earlier), or, we might expect that we could just write an operation schema, such as

$ \begin{array}{l} UpdateSystem \\ \Delta Order \\ \Delta System \\ \hline Order \in System.orders \\ System.stock(Order.product) \geq Order.quantity \\ \dots \end{array} $
--

(we have omitted postcondition details). Note that in this case *Order* represents a single order. But *UpdateSystem* should be applicable to any object of class *Order*. So the Z schema as given is insufficient: we really need to talk about the set of all *Orders* in our system, not just a single order. A convention for solving this problem is given in [1]; Hall defines syntactic devices for describing updates of a system, after changing a single object within that system.

Returning to the predicative formulation of the problem, we now concern ourselves with using *UpdateSystem*. To update the system by changing an order *o* from *PENDING* to *INVOICED*, we would use the syntax *System.UpdateSystem*, which is sugar for

$$System := UpdateSystem(o)$$

The *UpdateOrder* function takes two *Order* parameters, *o* and *systemOrders*, where *o* is in *systemOrders*. It replaces *o* in *systemOrders* with an identical order, except with the *status* field changed to *PENDING*.

$$\begin{aligned} \text{UpdateOrder} &= \lambda o : \text{Order} \cdot \lambda \text{systemOrders} : \text{Order} \cdot \\ &\quad \S i : \text{Order} \cdot (i : \text{systemOrders} \wedge \neg i : o) \vee (i = \text{"status"} \rightarrow \text{INVOICED} \mid o) \end{aligned}$$

UpdateOrders results in the bunch of all orders in *systemOrders*, except *o* itself, which is replaced by an order identical to *o*, but with changed *status* attribute.

5 Case 2: Updating the invoicing system

In this version of the invoicing system, new orders can be created, orders can be cancelled, and new stock can be added. The advantage of using the object-oriented predicative style now becomes apparent to us: addition of new system operations to handle the new phenomena of the application domain is a straightforward process. We do not have to alter the descriptions of state components; the description of the *System*, as well as its components *orders* and *stock*, remain the same as in the first part. The state components do not have to change because they are bunches, and as such can represent an arbitrary number of orders and stocked products. However, reusing the formalization from Case 1 means that a number of questions have to be answered before proceeding.

- **Can data flows arrive at the system concurrently?** That is, can new orders, cancellation of orders, stock updates, et cetera, arrive at the system simultaneously? According to our *System* formalization, operations that change the state of the system must be guaranteed atomic access, otherwise the system will enter an inconsistent state. All operations that we identify must therefore be atomic and mutually exclusive; that is, orders may be added or deleted, or stock changed, but in the process of making such a change, no other operation may be active. Notice that we could describe a system that permits concurrent attempts to change state by describing a mutual exclusion mechanism. Formalizing the *stock* and *orders* using predicative notation helps to clarify this requirement: stock cannot be changed (i.e., via functional overriding) by an operation unless the operation has exclusive access to the stock function. And similarly, an order cannot be updated unless an update operation has exclusive access.
- **What is the initial state of the system?** An initial state for the system must be provided. This is ensured by describing a new system operation, *InitSystem*. Predicative notation clarifies this point: because we formalize orders as a bunch, addition of the very first order cannot be carried out unless the bunch of orders is originally *null*—which isn't guaranteed unless there has been initialization of *orders* at some point.
- **How do we maintain order identity uniqueness?** Recall that a constraint identified in Case 1 is that all order identities are unique. This is so that customers can make multiple references to the same product of the same quantity. Order uniqueness is ensured by assigning a unique *id* field to each order, and by ensuring that the system operation to add a new order maintains the identity invariant. Predicative

programming requires us to have this constraint; without it, bunches—like sets—will not allow representation of multiple instances of the same element (and therefore will not allow multiple references to the same product and quantity on different orders). Formalizing in predicative notation clearly suggests that *id* uniqueness must be an invariant of all state-change operations.

To initialize the system, the operation *InitSystem* is used. It sets the attributes of the *System* to their respective zeros. Since *System* is a record of bunches, it is initialized by assigning to its fields the value of the empty bunch.

$$\text{InitSystem} = \text{"stock"} \rightarrow \text{null} \mid \text{"orders"} \rightarrow \text{null} \mid \text{System}$$

To add a new order to the system, an order is constructed (in the environment), and the *AddNewOrder* operation is invoked.

$$\begin{aligned} \text{AddNewOrder} = \lambda o : \text{Order} \cdot \\ \text{if } \neg o \text{"id"} : \S i : \text{System} \text{"orders"} \text{"id"} \text{ then} \\ \text{"orders"} \rightarrow (\text{System} \text{"orders"} , o) \mid \text{System} \\ \text{else System} \end{aligned}$$

If the *id* field of the new order is possessed by an order already present in the system, the *AddNewOrder* operation cannot change the system, as suggested by our answer to the question **how do we maintain order uniqueness?**, above.

Cancelling an order occurs by passing an order *id* as a parameter, and removing the corresponding order from the system, providing that the order is present. This raises another interesting question: **when can an order be cancelled?** One constraint is that the order is present in the system. But the system can contain two types of orders: those that are *PENDING*, and those that are *INVOICED*. An order that is *INVOICED* has, presumably, been sent to the customer. Cancellation of an invoiced order seems impossible! (Though the customer might obtain satisfaction by returning their invoiced order—a feature that seems to be outside of the problem frame for our system). Therefore, a second constraint on cancellation is that the order is *PENDING*, and not *INVOICED*.

$$\text{CancelOrder} = \lambda id : \text{ID} \cdot \text{"orders"} \rightarrow (\text{System} \text{"orders"} \ominus \text{cancelOrders}) \mid \text{System}$$

where

$$\text{cancelOrders} = \S j : \text{System} \text{"orders"} \cdot j \text{"id"} = id \wedge j \text{"status"} = \text{PENDING}$$

The bunch difference operator, \ominus , is defined as $A \ominus B = \S i : A \cdot \neg i : B$. It is the bunch of all *i* in *A* that are not in *B*. Note that difference is defined even if *B* and *A* do not coincide.

We turn now to the facilities for entering quantities in the stock. One question immediately appears: **can we add new stock items?** By examining the stock formalization, we see that it is permissible, by extending the function *stock*, using selective union. The requirements do not seem to prohibit adding new items. Therefore, we express such functionality in our formalization. Stock can be added by applying *AddStock*. A quantity *q* of product *p* is supplied. If *p* is already recognized by the invoicing system, its

in-stock quantity is updated. Otherwise, a new stock entry is added, by extending the function *stock*.

$$\begin{aligned}
 \text{AddStock} &= \lambda p : \text{PRODUCT} \cdot \lambda q : \text{nat} \cdot \\
 &\quad \mathbf{if} \ p : \Delta \text{stock} \ \mathbf{then} \\
 &\quad \quad \text{"stock"} \rightarrow (p \rightarrow \text{System} \text{"stock"}(p) + q \mid \text{System} \text{"stock"}) \mid \text{System} \\
 &\quad \mathbf{else} \ \text{"stock"} \rightarrow (p \rightarrow q \mid \text{System} \text{"stock"}) \mid \text{System}
 \end{aligned}$$

To add new stock, the call *System.AddStock(p, q)* is used. This is sugar for

$$\text{System} := \text{AddStock}(p, q)$$

Finally, one last question arises: **should we be able to delete items from stock?** Our formalization suggests that we already have some means to do this: by invoicing. Each invoice operation removes a quantity from stock. Once that quantity has been reduced to 0, the *PRODUCT* is effectively removed from stock (subject to future additions of quantity). The requirements do not suggest that deleting stock items is a part of the domain of the machine. Therefore, we omit such an operation from our formalization.

6 Discussion

The predicative programming method of [2] is well-suited to describing any kind of computation, and any kind of system, be it real-time, concurrent, object-oriented, or interactive. The object-oriented style introduced herein can be used to describe many important object concepts: classes, class interfaces, instantiation, inheritance (which we have not needed to use for this problem), assembly, overriding, and (limited) encapsulation. It is also useful for reuse: specifications for base-class operations can be reused in derived classes (though we do not show this here).

In using the predicative object-oriented style to solve the invoicing problem, we noted the following.

- Because we use bunch types as classes to represent orders and multiple orders, we needed a way to uniquely identify a possibly arbitrary number of orders which are otherwise indistinguishable (in terms of product and quantity). To do this, we added an *id* field to the class *Order*. It is not reasonable to disallow different orders for the same product and in the same quantity.
- Using the predicative style allowed concise representation of the status-change: functional overriding was used to write the change of an order from *PENDING* to *INVOICED*.
- Moving from a system where new orders did not have to be processed to a system where they did, required no change in data representation; this was an artifact of using the object-oriented style, as well as bunches as types. Also, we could use a consistent style of specification: functional overriding was used in describing the methods to add new stock, or to add or delete an order from the system.

In the second part of the case study, we were tempted to describe the system in more detail, i.e., in terms of how orders or objects are created (i.e., how quantities, products, and identities are determined by the system or the user). However, we decided that such details were beyond the scope of the study: the problem was to specify an invoicing system, and not how it is used. We therefore left the specification of order creation undescribed (as is usual practice in predicative programming and other similar methods, e.g., Z): operations can be invoked by the system, or by the environment, as needed.

The style of specification supported by the predicative method is very similar to that used with Z. A specifier defines system state (in our case, classes and objects), and operations upon that state. Our approach differs somewhat from the usual Z ‘style’ in that we have emphasized using an object-oriented style of specification. Our object-oriented style differs from styles for Z [1], primarily because we describe classes as bunches, which are just types. Specifying objects, hierarchies, and operations on objects then uses standard predicative notations, and does not require conventions to deal with the effects of operations on a system that may contain many object instances.

In applying the predicative method, we were forced to think hard about the representation of the orders and the stock. From analysis of the requirements, we determined that the orders and the stock could be represented by bunches (of, respectively, type *Order* and type $PRODUCT \rightarrow nat$). We considered an ordered representation of *Orders* (e.g., on *id* fields), but decided that the requirements did not describe any need for an ordering mechanism. Certainly, representing *Orders* as functions from *id* to an appropriate record type would make describing order updates much simpler. However, when implementing the system, different ways of arranging orders or of adding orders to the system may be required. Therefore, we thought it best to interpret the requirements literally, and have an unordered bunch representation of orders (which can easily be data transformed to an ordered representation, as we discuss in the next section).

A final motivation for using the predicative programming method is because it is a program design calculus—for both imperative and functional programs. Thus, given the specification of the system above, we can apply standard predicative refinement rules [2] in developing an implementation.

6.1 Refinement

Refinement, as discussed in [2], involves the following tasks.

- Designing a new (concrete) state representation, which typically is produced so that it can be straightforwardly implemented on a machine. In the case study, the new state representation might include implementing bunches of orders as lists (or arrays), and implementing functions as arrays.
- Transforming specifications based on the original state representation, to the new state representation. The *data transformation* proof techniques in [2] can be used to do this. In particular, data transformation of classes can be carried out, giving us implementations of design classes. Predicative programming data transformation techniques can be used without change for data transforming classes.
- Algorithmically refining method specifications to implementations, either by procedural or functional refinement.

Implementation of the *orders* attribute of class *System* could be through use of a list indexed by *IDs*, $newOrders : ID \rightarrow Order$. A data transformer (i.e., a ‘coupling invariant’) for such a refinement is

$$orders = newOrders(orders.id)$$

That is, each component of *orders* is in the range of *newOrders*, mapped there by its *id* attribute.

In refinement, algorithms that implement the operations of the system are designed. Abstract operations, e.g., bunch inclusion and union, function overriding, etcetera, are refined to “concrete” implementations, in terms of entities that are directly executable on a machine. For example, after applying the data transformation above, the *orders* attribute of *System* would be represented as a list, indexed by *id*. The data transformation would also be applied to the operations like *UpdateSystem*, changing bunch inclusion to a function invocation. Algorithm refinement could be used to further refine the function invocation, e.g., to list indexing.

6.2 Questions and answers

In formally expressing the requirements, and making design decisions, we answered a number of questions about the system.

- **How do we distinguish between identical orders for the same product and in the same quantity?** Orders must be uniquely identifiable; an order from one customer for product *P* in quantity *q* must be distinguishable from an order from a different customer ordering product *P* of quantity *q*. We enforced this by extending the description of orders to include identity.
- **When can orders be invoiced?** Orders can be invoiced precisely when the system has the requested product in stock. We discovered that the system (or the environment, i.e., a customer) can ask for an order to be invoiced, but if sufficient stock is not available, the invoicing will not succeed. A useful extension of the order representation would be to record “date of first invoice attempt”, so as to be able to determine how long customers have been waiting for their order to be invoiced. Then the system could organize invoices by priority, based upon wait time.
- **Is it necessary to describe when the system updates stock, or when orders are invoiced?** Our formalization of the requirements suggested no: the system’s purpose (in our interpretation) was to manage orders that it receives—it is outside of the application domain to describe the phenomena associated with when orders are created. We provided features that allowed such updates or invoicing; but they are to be invoked by the system, or the environment.
- **How many orders can the system handle?** We placed no constraints; an arbitrary number of orders can be added, an arbitrary number of products may be kept in stock (which is also unrealistic, since warehouse storage is limited) and an arbitrary amount of each product may be kept in stock. It is easy to add extra constraints, e.g., on the size of bunch *orders*, or on the domain or range of *stock*, to express such constraints if it is important to do so.

- **When can orders be cancelled?** A cancellation can be requested, but if the order has already been invoiced, it is assumed to have been sent to the customer. Therefore, invoiced orders cannot be cancelled (other mechanisms would have to be used, e.g., for returning invoiced orders).
- **What stock updates can be performed?** Addition of quantities of existing stock can be carried out, but also new stock items can be added.
- **Is the description of the system, in the way we have done it, useful?** It appears so, in particular, because we did not have to change the description of data in moving from part 1 to part 2 of the case study. This is an artifact of using predicative programming, and its style for dealing with objects.

7 Summary

The invoicing case study has been formalized in a object-oriented predicative style, based on the formal method of [2]. Formalization in predicative notation allowed answering of several questions that arose from the requirements: *how can we distinguish between identical, but separate orders?*, *when can an order be invoiced?*, *can orders be placed for products not in stock?*, *are there limits on the size of the system?*, and *when can an order be cancelled?*

Applying the predicative method refined our understanding of the problem. We understood what constituted an order—in particular, the need for some way of uniquely identifying an order. We understood that orders can only be invoiced when the stock of a product was sufficient, but also that *PENDING* orders can reside in the system, presumably indefinitely. This also suggested that orders could be placed for products not in stock. We found that cancellation of an order can occur only when an order is *PENDING*, and not after *INVOICING*. And, we found that while new quantities of stock can be added to the system, and new products could be added, products could not be removed from the system—such a feature being outside the frame of the problem description.

References

1. A. Hall, Specifying and Interpreting Class Hierarchies in Z, in *Proc. ZUM 1994*, Springer-Verlag, 1994.
2. E.C.R. Hehner, *A Practical Theory of Programming*, (Springer-Verlag, 1993).
3. Call for Papers from the International Workshop on Comparing System Specification Techniques, available at www.sciences.univ-nantes.fr/info/manifestations/invoice98.
4. M.A. Jackson, *Software Requirements and Specifications*, (Addison-Wesley, 1995).
5. C.C. Morgan, *Programming from Specifications*, Second Edition, (Prentice-Hall, 1994).