From Functional to Object-Oriented Programming – A Smooth Transition for Beginners

Rudolf Berghammer and Frank Huch

Institute of Computer Science
University of Kiel
Olshausenstraße 40, 24098 Kiel, Germany
{rub,fhu}@informatik.uni-kiel.de

Abstract

Many Computer Science curricula at universities start programming with a functional programming language (for instance, SML, Haskell, Scheme) and later change to the imperative programming paradigm. For the latter usually the object-oriented programming language Java is used. However, this puts a burden on the students, since even the smallest Java program cannot be formulated without the notion of class and static and public method. In this paper we present an approach for changing from functional to object-oriented programming. Using (Standard) ML for the functional programming paradigm, it still prepares the decisive notions of object-orientation by specific constructs of this language. As experience at the University of Kiel has shown, this smoothes the transition and helps the students getting started with programming in the Java language.

Categories and Subject Descriptors D.1.1 [*Applicative (functional) programming*]; D.1.5 [*Object-oriented programming*]; D.3.3 [*Language constructs and features*]

General Terms Languages

Keywords SML, signature, structure, functor, Java, object, class

1. Introduction

Many Computer Science curricula at universities start with the functional programming paradigm. This is mainly due to the fact that this paradigm cannot only be used to explain many basic concepts of programming and algorithm development, but also to teach a lot of fundamental concepts of Computer Science and how these concepts evolve from each other. A further advantage of functional programming is that it uses in the for novices very important initial stage only the notion of (recursive, partial) functions, which should be known from high school. Finally, it should be mentioned that functional programs allow to demonstrate proofs of simple program properties by combining structural or well-founded induction with equational reasoning.

Although functional programming has a lot of advantages, it is also important to familiarize students with imperative and object-oriented concepts already during the first year at the university. These concepts are widely used in industry and many topics of subsequent courses use imperative or object-oriented programming languages (especially, Java and C++). At the University of Kiel we change to imperative programming after the introduction to functional programming and use, as many other universities, the object-oriented programming language Java [2, 6]. However, experience has shown that this puts a burden on the students if one starts imperative programming with Java's overhead of object-oriented notations. Even the smallest Java program cannot be formulated without the notion of class and static and public (main) methods, and neglecting these notions at the beginning proved to be unsatisfactory for teachers as well as for students[1]. Therefore, we have decided to prepare the transition to Java already on the level of SML [8, 14], the language we use for the introduction to functional programming. Fortunately, this is possible due to the very rich module system and the reference mechanism of SML. As experience has shown, our approach smoothes the transition and helps students getting started with object-orientation and programming in Java.

The remainder of the paper is organized as follows. In Section 2 we outline the concept for the two first year courses in programming at the University of Kiel and describe our approach for changing from functional SML-programming to imperative/object-oriented Java-programming without going into details. Details of the approach are presented in the next three sections. Using a running example very similar to the well-known bank account example of the textbook [1], we show how to model objects, classes, and inheritance in SML and demonstrate the great similarity of the resulting pieces of SML-code and the corresponding Java-pieces. For these sections we assume the reader to be familiar with SML, including references and the module system, and with Java. In Section 6 we show the limitations of our approach and Section 7 contains some concluding remarks.

2. The Approach

At the University of Kiel, the two first year courses in programming are divided into "Programming" (first semester) and "Algorithms and Data Structures" (second semester). Both courses combine theoretical aspects of programming and algorithm development with practical aspects of these fields. In doing so, we want to avoid two disadvantages which frequently appear if one puts too much emphasis on one of these aspects. Overemphasizing practical programming may suggest to the students that theory has little to do with practice and they can become a good programmer

[1] From a pedagogical point of view, there are many arguments against Java as first-course programming language at universities. We recommend the reader to have a look at [5].

FDPE'05 September 25, 2005, Tallinn, Estonia.

without studying theory. On the other hand, overemphasizing theory may lead to the impression that theory as "mere theory" has its right (e.g., for studying the absolute or gradual limits of what algorithmically can be solved), but it is not relevant for programming practice.

Since more than one decade we use the functional paradigm in the first semester and the imperative paradigm in the second one. Due to certain reasons (e.g., its very clean implementation of most of the important object-oriented concepts and also to meet demands of industry), some years ago Java has been selected for the course on algorithms and data structures and it has been decided that the transition from the functional language to Java appears at the end of the first course. Giving the course on programming the first time and thereby starting with Java from the scratch, we noticed that such an approach puts a great burden on the students and the results did not meet our expectations. As a consequence, we decided to prepare the transition already on the level of SML. This was enabled by the following (rough) structure of the previous course on programming.

(1) Mathematical preliminaries (sets, logical notation, induction, terms, term replacement etc.).
(2) Introduction to functional programming (first-order recursive functions [10] over primitive types, parameter passing, unfold-fold technique etc.).
(3) Advanced concepts of functional programming (datatype declarations, recursive data types, pattern matching, higher-order functions, polymorphism, advanced programming techniques etc.).
(4) Data abstraction and modularization (information hiding, abstract types, signatures, structures, functors etc.).
(5) Introduction to Java (basic concepts of object-orientation, while-programs over primitive types, loop invariants, assertion technique for development etc.).

In the refined course, Part (4) is extended by a section which introduces and motivates object-orientation as a specific approach of modularization. Then descriptions of the fundamental notions *object*, *class*, and *inheritance* are given (*polymorphism* is discussed already in (3)). The main focus of this is to convey these notions in general and not as particular constructs of a programming language. To reach this aim, we proceed as follows:

- First objects are specified as "things" that have attributes and can perform actions.
- Then classes are introduced as descriptions of all objects (the *instances*) of a particular kind, together with the creation of instances.
- Finally, inheritance is explained as a mechanism that allows to derive new classes from given ones to deal with specific cases in adequate manner.

Having introduced these three notions, it is shown how they can be modeled within the known language SML by means of a suited example. Due to the great similarity of the resulting SML-code and a later formulation of the example in Java, this new approach smoothes the transition from SML to Java and avoids, as experience has shown, many teething troubles with the new language and paradigm. Of course, the extension of Part (4) required to shorten the other parts of the course a little bit and to move some topics into the second semester to stay within the time scheduled for one semester. This, however, caused no serious problems. A German version of the actual lecture notes on programming (winter semester 2004/05) is available via the Web [3].

3. Modeling Objects

Object-oriented programming is based on objects. As already mentioned in the last section, abstractly these are "things" that have attributes (also called fields) associated with it and that can perform certain actions. Attributes are ascertainable through their values. In object-oriented programming languages like Java they are specified by variables. Based on this, actions are performed by calls of methods which, in the most simple case, either compute values or change values of variables.

As an example, bank accounts can be considered as objects. In a very simple case (see [1]) the attributes of an account are given by the current account balance and the overdraft agreement. Two possible actions that compute values are the computation of the available money and the computation of the account balance. Attributes are changed, for example, by defining the new overdraft, by depositing and withdrawing money, and by deducing charges.

To model objects in SML, we use references instead of variables and functions instead of methods. Then an object corresponds to an SML-structure which consists of declarations of references for the attributes and of functions for the actions. Usually attributes (and auxiliary methods) of objects are declared to be private. Also this information hiding can be modeled in SML. We only have to define a SML-signature that exactly contains the names and types of the non-private (i.e., public) functions and after that to restrict structures through this signature.

In the case of our bank account example, the declaration of an object `A1` is described by the following SML-code. The two references `st` and `cr` in the structure declaration are used for the two attributes "account balance" and "overdraft"; the five functions realize the actions mentioned above. Hiding of `st` and `cr` is obtained by restricting the declared structure through an appropriate signature.

```
signature Account =
  sig
    val Available : unit -> int;
    val AccountBalance : unit -> int;
    val SetOverdraft : int -> unit;
    val Deposit : int -> unit;
    val Withdraw : int -> unit;
    val Charge : int -> unit;
  end;

structure A1 : Account =
  struct
    val st : int ref = ref 0;
    val cr : int ref = ref 0;

    fun Available () : int =
      !st + !cr;
    fun AccountBalance () : int =
      !st;
    fun SetOverdraft (n : int) : unit =
      cr := n;
    fun Deposit (n : int) : unit =
      st := !st + n;
    fun Withdraw (n : int) : unit =
      if n <= Available()
        then st := !st - n
        else ();
    fun Charge (n : int) : unit =
      st := !st - n;
  end;
```

As this example shows, we prefer to type SML-functions completely although the SML language possesses a sophisticated type

inference mechanism. By adhering to this style, we hope to focus the student's attention on the importance of being aware of the arguments and results of each function one introduces in the course of a program. In the context of modeling object-orientation, furthermore, the result type `unit` indicates that a function/method changes values of attributes. It directly corresponds to the specification `void` in Java.

4. Modeling Classes

Abstractly, a class is a description of all objects of a particular kind, i.e., objects with the same attributes and actions. These objects are called instances. For each class there exists a mechanism for creating its instances.

Having a look at the bank account example of Section 3, one observes that the description of the account/object `A1` essentially is given by the right-hand side of the structure declaration, i.e., the code from `struct` to `end`. Creating a new account in the course of a program, say `A2`, therefore can be obtained by repeating the structure declaration by `A2` in lieu of `A1`. However, this is a laborious way of creating new objects. There exists a much more elegant way. It uses the functor mechanism of SML: an SML-functor operates on structures to produce other structures. Typically, the resulting structure is defined in the usual way, i.e., its constituents are parenthesized by the keywords `struct` and `end`, where in the declarations the constituents of the parameter structure may be used. From this point of view, functors are "parameterized" structures.

We have decided to model classes by functors, since functors frequently generalize structures in a way very similar to the generalization of objects to classes[2]. This approach enables us to model the generation of instances by simple functor calls which is possible due to the fact that SML-functors are not referentially transparent. If a functor is called with an argument twice, then the results of the calls are different structures.

In our bank account example, a parameterless functor modeling the class of accounts immediately arises from the above structure as follows:

```
functor new_Account () : Account =
  struct
    val st : int ref = ref 0;
    val cr : int ref = ref 0;

    fun Available () : int =
      !st + !cr;
    fun AccountBalance () : int =
      !st;
    fun SetOverdraft (n : int) : unit =
      cr := n;
    fun Deposit (n : int) : unit =
      st := !st + n;
    fun Withdraw (n : int) : unit =
      if n <= Available()
        then st := !st - n
        else ();
    fun Charge (n : int) : unit =
      st := !st - n;
  end;
```

Structurally, this SML-code is very similar to the code in Java, as the following Java-class `Account` for bank accounts shows:

```
class Account {
  private int st;
  private int cr;

  public int Available () {
    return st + cr; }
  public int AccountBalance () {
    return st; }
  public void SetOverdraft (int n) {
    cr = n; }
  public void Deposit (int n) {
    st = st + n; }
  public void Withdraw (int n) {
    if (n <= st + cr) st = st - n; }
  public void Charge (int n) {
    st = st - n;}
}
```

Using the above SML-functor `new_Account`, it is possible to generate accounts by structure declarations with simple functor calls as right-hand sides. A manipulation of instances then is possible by calls of the functions of the declared structures in statement lists, as the following simple example shows:

```
structure A1 : Account = new_Account();
A1.SetOverdraft(100);
A1.Deposit(100);
structure A2 : Account = new_Account();
A2.SetOverdraft(500);
A1.Withdraw(50);
```

We generate an account `A1`, put its overdraft to 100 units of money, and deposit 100 units of money. Then we generate a new account `A2` and put the overdraft of `A2` to 500 units of money. Finally, we withdraw 50 units of money from the first account `A1`.

Comparing this code with the following corresponding code in Java, the similarity becomes even more evident than in the case of classes; only the generation of the two instances syntactically differ.

```
Account A1 = new Account();
A1.SetOverdraft(100);
A1.Deposit(100);
Account A2 = new Account();
A2.SetOverdraft(500);
A1.Withdraw(50);
```

So far, we only considered parameterless functors. If we change to functors with parameters, then we are even able to model parameterized classes. This can be explained by our bank account example as well.

In the previous modeling of bank accounts the generation of a new account comes along with an overdraft agreement of 0 units of money. In practice, however, usually the opening of a new bank account is combined with a specific overdraft agreement. Taking the initialization of the overdraft as parameter, this can be modeled by a parameterized class. Then, instantiation means not only to create a new account but also to designate a particular value to its initial overdraft.

In SML a parameterization of a class can be modeled by a functor containing structures as parameters. In the case of the account example, name and type of the initial overdraft are specified by the signature of the parameter structure, for example, as follows:

```
signature Param =
  sig
    val o : int;
  end;
```

[2] In [14] SML-functors are even considered as a generalization of the idea of templates in C++, i.e., parameterized classes. We will descend to this at the end of Section 4.

The modeling functor itself is obtained from the previous functor new_Account by adding a parameter of signature Param, say P, and changing the initialization of cr from 0 to P.o. Hence, we have:

```
functor new_Account1 (P : Param) : Account =
  struct
    val st : int ref = ref 0;
    val cr : int ref = ref P.o;

    fun Available () : int =
      !st + !cr;
    fun AccountBalance () : int =
      !st;
    fun SetOverdraft (n : int) : unit =
      cr := n;
    fun Deposit (n : int) : unit =
      st := !st + n;
    fun Withdraw (n : int) : unit =
      if n <= Available() then st := !st - n
                          else ();
    fun Charge (n : int) : unit =
      st := !st - n;
  end;
```

If we call this functor, for example, with the anonymous structure

```
struct
  val o : int = 500;
end
```

as argument, then an instance is created with 500 as intital value of cr. In words this means that a new bank account is opened and this is combined with an overdraft agreement of 500 units of money.

It is obvious, how to change the signature Param and the functor new_Account1 to obtain besides an initial overdraft agreement also an initial deposit when opening a new bank account.

5. Modeling Inheritance

Having demonstrated how to model objects and classes in the language SML, it remains to show that this approach is appropriate for modeling inheritance as well. Inheritance is the third central concept of object-orientation and describes the deduction of new classes (the subclasses) from given ones (the superclasses) to deal with specific cases in adequate manner. We believe that inheritance is best explained to beginners by concrete examples which frequently occur in practice. Most of these examples are based on *specialization by inheritance*, that is the adaption of an existing general framework to a particular situation , since this is the central technique of object-orientation (see [11] for more details).

In the context of our running bank account example, a first kind of specialization is given by student accounts which are free of charge. For our SML model this means that we have to override the function Charge of the functor new_Account in such a way that a call of the new version does not collect charges, i.e., is without any effect. The result is the following functor:

```
functor new_StudAccount () : Account =
  struct
    structure A : Account = new_Account();
    open A;

    fun Charge (n : int) : unit =
      ();
  end;
```

In the structure specification forming the body of the functor new_StudAccount, we use a structure declaration in combination with the opening of the declared structure A via open A as a comfortable way to include the constituents of new_Account into the structure of new_StudAccount. Then the definition of Change from the structure A is overriden by a redefinition of the function. As the functor declaration shows, student accounts possess the same interface/signature as general bank accounts.

Another example of specialization by inheritance, which usually involves a modification of the signature, is to add specific functionality to a class. In this case it is sufficient just to add the new functionality. we will explain this again by means of our running examnple.

For bank accounts an additional functionality may be the possibility of online banking. In the most simple case this means to have an additional hidden Boolean attribute and an additional visible method with a truth value as argument such that a call of the method enables or disables online banking by changing the value of the hidden Boolean attribute accordingly. For our SML model this involves an extension of the signature Account by the name and functionality of the new method and an extension of the functor new_Account by its implementation, which uses a hidden reference declaration for implementing the new attribute. In the following SML-code the include-operation is used as a comfortable way to include the constituents of a signature in another one, and for the inclusion of constituents of a structure in an another structure again the open-operation is applied.

```
signature ExtAccount =
  sig
    include Account;
    val SetOnlinebanking : bool -> unit;
  end;

functor new_ExtAccount () : ExtAccount =
  struct
    structure A : Account = new_Account();
    open A;
    val ob : bool ref = ref false;

    fun SetOnlinebanking (b : bool) : unit =
      ob := b;
  end;
```

Because of the initialization of the reference ob by false, online banking is not possible for a newly opened bank account.

At this place it also should be remarked that a functor with a parameter of signature Account can be applied to a structure of the more general signature ExtAccount without previous conversion. We use this property of the ML module system to introduce the notion of *subtyping* to the students.

Here is the extension of the former Java-class Account to a subclass ExtAccount which models bank accounts with the possibility of online banking.

```
class ExtAccount extends Account {
  private bool ob;

  public void SetOnlinebanking (bool b) {
    ob = b;
  }
}
```

Since in Java the initial value of a Boolean variable is false, again online banking is not possible for a newly opened bank account.

Of course, from a purely syntactic point of view the Java-class ExtAccount is much more simple than the corresponding signature ExtAccount and functor new_ExtAccount in SML. However, structurally there is again a great similarity between the SML- and

the Java-code for the extended of bank accounts, and exactly this is what we want to demonstrate.

6. Limitations of the Approach

In the preceeding sections we presented a way for changing from functional SML-programming to object-oriented Java-programming. It is especially tailored for beginners to make the transition as smooth as possible. Due to this aim, of course, the approach has its limitations and deficiencies if considered from a "mere object-oriented" point of view. These limitations will be discussed in the following.

The first limitation regards binding which we explain by an example. Consider the following situation. From the Java-class `Account` of Section 4 it is obvious to derive the following subclass `StudAccount` for student accounts which redefines the method `Charge` for deducing charges according to the former SML-model:

```
class StudAccount extends Account {
  public void Charge (int n) {
    ;}
}
```

Furthermore, let an instance `A` be generated. In Java then the class name used in the `new`-operator determines binding, i.e., which definition of the method name has to be used. A generation of `A` via

```
StudAccount A = new StudAccount();
```

implies that the method `A.Charge` is bound to the `Charge` of the subclass `StudAccount` if it is applied to an integer, whereas the assignment

```
StudAccount A = new Account();
```

implies that `A.Charge` is bound to the `Charge` of the superclass `Account`. In SML it is not easy to model this principle of *dynamic binding* which is the basis of the object-orientated variant of polymorphism. One needs structures as constituents of signatures and has carefully to follow the paths in the structures hierarchy.

We are also aware of the fact that our approach in some technical details is inaccurate if considered under theoretical aspects. For example, in object-oriented programming languages objects are values and not first-order only entities as signatures, structures, and functors in SML are.

Finally, we also know that we only introduce the core ideas of object-orientation and only show the students how the central concepts objects, classes, and inheritance can be modeled in SML. This also includes some small example programs but, of course, not a fundamental introduction into the development of programs in an object-oriented manner.

We do not regard these limitations and deficiencies as a disadvantage. Our approach never has been intended to be a detailed introduction in object-oriented analysis, design, and programming and further central concepts like frameworks or the well-known design patterns work orginating from [7]. This would be far beyond the possibilities of an introductionary course in programming. It is only supposed to smooth the transition from functional to object-oriented programming to help students of a first university course getting started with object-orientation and programming in Java. The second semester course on algorithms and data structures and subsequent specific lectures on software-engineering, object-orientation etc. together with their accompanying practical exercises then have to undertake the task of a detailed treatment of imperative and object-oriented programming in Java.

7. Conclusion

In this paper we have presented a new approach for a smooth transition from functional programming in SML to object-oriented programming in Java. It is especially designed for beginners in Computer Science at universities and has been tested at the University of Kiel in the winter semester 2004/05. We are highly pleased with the result. The preparation of object-orientation still in SML (and additional material in the lecture notes on the web) led to the fact that at the end of the semester most of the students had understood the basic principles of object-orientation and had been able to write (of course, not too large) Java programs. The proposed approach avoided many difficulties which appeared during an earlier course, where we started with Java from scratch. As a by-product, our approach leads to a deeper understanding of the module system of SML.

There exists an object-oriented ML-dialect, called OCaml [12]. Hence, the question arises: Why not start with OCaml instead of SML? Reasons for our use of SML are that it is the most popular member of the ML family, used in all well-established textbooks, and excellent public domain implementations exist. Furthermore, we want to place emphasis on modularization and encapsulation. This is adequately supported by the module system of SML. On the basis of the restricted time available for its treatment, we believe that our approach leads to a deeper understanding than a separation of modularization, encapsulation, and object-orientation using OCaml.

Our approach is based on special features of the SML programming language, especially its module system. Hence, the question arises whether it can be translated to other functional programming languages, especially Haskell [9, 4] and Scheme [1, 13]) which frequently are used for the introduction to functional programming at universities. We have not investigated this in great detail, but believe that in both cases the answer is "in principle 'yes', but ...". Concerning Haskell, we can use I/O-monads and `IORefs` to model attributes/variables and functions to change their values. Based on this, it seems to be possible to model objects and classes by functions, too. But we assume that the resulting model is rather complex and far beyond the possibilities of beginners. In contrast to Haskell, it is very easy to model attributes/variables and the change of their values in Scheme using the assignment-operation `!set` and the two special procedures `set-car!` and `set-cdr!`. But, to our best knowledge, in the standard of the language (as e.g., described in the language report [13]) there are no constructs for data abstraction and modularization. Hence, we believe that also Scheme is not appropriate for our approach.

Acknowledgement

We thank the referees for valuable remarks.

References

[1] Abelson H., Sussman G.J., Sussmann J.: Structure and interpretation of computer programs. MIT Press, 1999.

[2] Arnold K., Gosling J.: The Java programming language. Addison-Wesley, 1996.

[3] Berghammer R.: Informatik I (Programmierung). Lecture notes, University of Kiel, Inst. of Computer Science, available via URL www.informatik.uni-kiel.de/inf/Berghammer/teaching, 2005.

[4] Bird R.S.: Introduction to functional programming using Haskell. 2nd edition, Prentice Hall, 1998.

[5] Böszörmenye L.: Why Java is not my favorite first-course language. Software – Concepts & Tools 19, 141-145, 1998.

[6] Deitel H.M., Deitel P.J.: Java: How to program. 6th edition, Pearson Education International, 2005.

[7] Gamma E., Helm R., Johnson R., Vlissides J.: Design patterns: Elements of reusable object-oriented software. Addison Wesley, 1995.

[8] Harper R., Milner R., Tofte M.: The definition of Standard SML. MIT Press, 1991.

[9] Hudak P., Preyton-Jones S.L., Wadler P. (eds.): Report on the programming language Haskell. ACM SIGPLAN Notices 27(5), 1992.

[10] Loeckx J., Sieber K.: The foundations of program verification. Wiley, 1984.

[11] Poetzsch-Heffter A.: Konzepte objektorientierter Programmierung. Springer, 2000.

[12] Remy D., Vouillon J.: Objective ML: An effective object-oriented extension to ML. Theory and Practice of Object Systems 4 (1), 27-50 1998.

[13] Rees J., Clinger W. (eds.): The revised report on the algorithmic language Scheme. Lisp Pointers 4(3), 1991.

[14] Ullmann J.D.: Elements of ML programming, SML97 edition. Prentice Hall, 1998.