Journal of Universal Computer Science, vol. 3, no. 2 (1997), 120-146
submitted: 11/7/96, accepted: 8/12/96, appeared: 28/2/97 © Springer Pub. Co.

Type Compatibility for Extensible Module Types, Their Reference Parameters, and Their Pointer Types

Jürgen F. H. Winkler
(Friedrich Schiller University Jena, Germany
winkler@informatik.uni-jena.de)

Abstract: Objects in object-oriented languages have often been treated as a special kind of entity different from other variables or constants. Similarly, their types, which are typically called classes, have often been treated differently from other types. This complicates the understanding of these concepts. The present paper proposes to see the classes as module types leading to a very natural integration of objects and classes into the framework of contemporary programming languages. The main part of the paper contains typing rules for module types for assignment and for value and variable parameters. It is shown that the rules for reference parameters in some existing languages lead to unexpected results and sometimes to undefined behavior. Furthermore, assignment involving dereferenced pointers to modules is studied for the first time in detail. The paper shows that the type compatibility rule for pointer assignment is not sufficient for deref assignment. The last part of the paper contains a comparison of the language definitions and of compilers for Borland Pascal with Objects, C++, Oberon, and Object CHILL.

Key Words: Object-Oriented languages, extensible module types, parameters and parameter passing, deref assignment, Borland Pascal with Objects, C++, Oberon, Object CHILL
Category: D.3 Programming Languages

1 Introduction

Objects in object-oriented languages have often been treated as a special kind of entity different from other variables or constants. Similarly, their types, which are typically called classes, have often been treated as a special kind of type. This complicates the understanding of these concepts. The present paper proposes to see the classes as module types leading to a very natural integration of objects and classes into the framework of contemporary programming languages.
The main part of the paper deals with type compatibility rules for extensible module types. We show that some languages use compatibility rules for reference parameters which lead to unexpected results and sometimes to undefined behavior. We show especially that reference parameters are not pointers. Furthermore, assignment involving dereferenced pointers to modules is studied for the first time in detail. The paper shows that the type compatibility rule for pointer assignment is not sufficient for deref assignment.
The last part of the paper contains a comparison of the language definitions and of compilers for Borland Pascal with Objects, C++, Oberon, and Object CHILL.

The formalisms used are all defined in the paper. We assume a general familiarity with the basic concepts of object-orientation. For program examples we use a suggestive syntax which should be self explanatory.
For easier reference for the reader the central definitions of the paper are collected in the Appendix.

Despite the assumed familiarity with the basic concepts of object-orientation we mention four essential concepts of OO which are relevant to the topics of this paper:

a) *module as first class value*: modules as values of variables, as parameters, and as the target value of a pointer;
b) *type derivation and substitutivity*: wherever an object OTb of a (base) type Tb is used an object OTd of a (derived) type Td can be used instead;
c) *reimplementation of operations in derived types*: an operation O defined in a type Tb may be given a new implementation (body) in a derived type Td;
d) *combination of reimplementation and substitutivity (polymorphism)*: if an operation O is applied to a variable, which may refer to objects of different (but usually related) types, then the implementation belonging to the type of the current object is used, i.e. depending on the type of the current object one of possibly several different implementations of that operation is used.

2 Why Module Types ?

In this paper we use the term *module type* for what is often called *class* or *object type* in object-oriented languages. There are several reasons for this.
First, *object* usually has a very general meaning: “something physical or mental of which a subject is cognitively aware” (Merriam Webster New Collegiate Dictionary, 1977: 791). In the field of programming languages object is also often used in a more general sense: “An object is an entity that contains (has) a value of a given type.” [Ref 83: 3-2] or “An object declaration creates one or more variables. These variables can be of any type and need not just be instances of classes.” [CW 96: 76]. This general meaning of “object” seems to be appropriate, and this is the reason not to adopt e.g. the term “object type” [CDG 92: 19]. In object-oriented languages *object* is used more narrowly for one special kind of such objects, namely for variables or constants of so called *classes*. “An *object* is a *class instance* or an array.” [GJS 96: 38]. Sometimes the term *object* is even used with several meanings: “From now on, *object* will have a precise meaning: a *record with procedure fields, accessed through a pointer*.” [RW 92: 218], “Declarations also serve to specify certain permanent properties of an object, such as whether it is a constant, a type, a variable or a procedure.” [RW 92: 284].
The second reason is that of all the different entities in contemporary programming languages the objects of object-oriented languages resemble mostly the *modules* of CHILL [Rec 93] or Modula [Wir 88b] (*package* in Ada [Ref 83], *unit* in Borland Pascal with Objects [Bor 93]). This resemblance can be characterized by the key concepts “aggregation”, “encapsulation/abstraction”, and “scoping”. The relation between module and module type is very much the same as that between a record

variable and a record type. These observations are the essential reason to use the term “module mode” in CHILL-96 [CHI 96]. (There are additionally monitor modes and task modes in CHILL-96, where, as in Algol 68, “mode” is used instead of “type”.)
The third reason is that the use of “module *type*” avoids the problematic term *class* [Win 92]. Since the classes of object oriented languages are essentially types, i.e. descriptions describing the nature of variables (and constants), it seems very natural to use the term *type* also in this case. Other more recent approaches to object-orientation also avoid the term *class* [OMG 93]. In rare cases there could be some misunderstanding because the term *module* is sometimes also used in a more general sense meaning a building block in general which could also encompass procedures. This view is especially used when speaking about linking several (object) modules into a single object program.
Another possible name for this kind of type could be *tuple type* which is even more neutral than module type. From a somewhat abstract point of view the objects of object-oriented programming are (heterogeneous) tuples i.e. aggregations of components of possibly different type. The term *tuple* is also used in Linda [CG 89] and in [Car 93]. There would be no problem to replace in this paper the term *module type* by *tuple type*. But this would not reflect the visibility properties of objects. In a tuple (e.g. a record with data and procedure components in Algol68 [KMP 69] or Modula-2 [Wir 88b]) the procedures are isolated from the data components, whereas in an object in a typical OO language the data components are automatically visible in the bodies of the procedure components.

3 Typing

In programming languages with typing entities may be *typed*. The type T of a typed entity describes the set of admissible values T_V and the set of operations T_O involving those values. A type may be *static*, i.e. T_O and especially T_V are a static property, or it may be *dynamic*. The language is strongly typed if type errors are always detected [Seb 93: 151]. There is a preference for static properties because they can be checked at compile time. But not all aspects of types are static properties. In Ada e.g. for a range type the set T_V may only be fixed at runtime but the type of the values is fixed statically:

```
        TYPE IntRange IS Integer RANGE 1..ReadAnInteger;
```

In this example the values of `IntRange` will always be integer numbers but the set T_V is only fixed at runtime (and may even be empty).
In general, a variable V may therefore have a *static type* ST(V) and also a *dynamic type* DT(V).
The application of an operation is *legal* if the static types of all operands and the result (if there is one) are correct with respect to the static semantics of the language.
The application of an operation is *safe* if all operands and the result (if there is one) have values within their types. The application of an operation is *statically safe* if

safety is a static property; it is *dynamically safe* if it is not statically safe and safety is guaranteed by dynamic checks.

Remarks:
(a) Safety is used here only with respect to typing; i.e. it should not be confused with the much broader meaning of safety in the term *software safety*.
(b) in this paper we always use simple variables. All observations and results also hold for variables which are components of larger variables as e.g. records or arrays.

4 Basic Properties of Module Types

A module type MT is characterized by its set of components:

$$CS(MT) = \{C_1, ..., C_n\} \text{ with } n \geq 0.$$

Which kinds of entities can be defined as a component of a module type depends on the language: Oberon [RW 92] allows only variables and procedures, whereas Object CHILL [DW 92] and C++ [ES 90] allow constants, types, variables, and procedures.
A module type can be used to declare variables, e.g. a variable MV of type MT:

```
VAR  MV: MT;
```

There are two important subsets of a module type:

$E(MT) \subseteq CS(MT)$ is the *external interface* of MT.
If id is the identifier of a component in E(MT) MV.id is an access to this component. Outside the definition of MT such an access is legal for all elements of E(MT) but not for those in CS(MT) - E(MT). These are called the internal components. In some languages the internal components are further divided into different groups but this is of no importance to the topic of this paper.

$V(MT) \subseteq CS(MT)$ is the set of *variable components*.
A variable component of MV may assume different values as is typical for variables. In most cases the variable components are data variables but there may also be variable components of procedure or module type. It depends on the specific language and its type system which kinds of variable components are possible.

There is no fixed relationship between E(MT) and V(MT).

A type MTd may be *directly derived* from a given type MTb. The semantics of this derivation is that from a logical point of view the following relation holds: $CS(MTd) \supseteq CS(MTb)$, i.e. the derived type contains all components of its base type MTb and possibly more. Physically, MTd declares only those components which are new or modified. This form of derivation is usually called *single inheritance* because there is exactly one direct base type for a derived type MTd.
We speak of *multiple inheritance* if a derived type MTd is directly derived from several given types $MTb_1, ..., MTb_n$, where $n > 1$. In this case we say, that MTd is directly derived from each of the MTb_i. Some of the MTb_i may even be the same type, i.e. MTd may be directly derived from an MTb more than once (see e.g. [Mey

92: 167 f.]). Multiple inheritance is e.g. supported in C++ and in Eiffel [Mey 92]. Simula, Oberon, and Object CHILL support single inheritance only. With respect to the topics discussed in this paper, there is no difference between single inheritance and multiple inheritance as long as there are no problems with visibility. There are two characteristic problems with multiple inheritance in the area of visibility:

a) name clash: MTb_1, ..., MTb_n contain two or more different components with the same name;
b) repeated inheritance: one component defined in a common ancestor MTa is inherited by MTd several times via different paths from MTd to MTa and may be replicated in MTd.

There exist different solutions for these problems, e.g. to forbid the critical situation, to allow the introduction of aliases for such components in MTd, or to give priority to one of the replicas. For the following we assume that these visibility problems have been solved and especially, that there are not several replicas of a component in one MT which have the same visibility status.

The semantics of multiple inheritance is then $CS(MTd) \supseteq CS(MTb_i)$, for i = 1 .. n. For each component of MTd it is well defined from which MTb_i it has been inherited.

A type MTd is *derived* from another type MTb, $MTd \geq MTb$, if MTd is either directly derived from MTb or if it is derived from a type MT which is directly derived from MTb. The derivation relation is restricted in such a way that its graph is a set of directed acyclic graphs, because the derivation of MT from MT, either directly or indirectly, is usually not allowed. We assume the direction of the arcs from MTd to MTb. In these DAGs the nodes with no successor are minimal wrt "≥" and the nodes with no predecessor are maximal. To each element e of these DAGs belongs a nonempty set of minimal elements min(e) and a nonempty set of maximal elements max(e). We use $MT1 > MT2$ as an abbreviation for $MT1 \neq MT2 \wedge MT1 \geq MT2$.

Remark: The relation ≥ is sometimes defined the other way round. The definition here is motivated by the relation between the sets of components:

$$MTd \geq MTb \;\Rightarrow\; CS(MTd) \supseteq CS(MTb).$$

This monotonicity of derivation wrt to the component set is fundamental to the topics and results of this paper (e.g. for substitutivity).

For the topic of this paper it is important that the derivation has additionally the following monotonicity properties:

MonE) $MTd \geq MTb \Rightarrow E(MTd) \supseteq E(MTb)$ (Monotonicity of the external interface)

MonV) $MTd \geq MTb \Rightarrow V(MTd) \supseteq V(MTb)$ (Monotonicity of the variable components)

There are languages for which MonE does not always hold (e.g. for private derivation in C++).

MonV holds in most OO languages. In our framework here, MonV is a consequence of the semantics of inheritance. Because MonV is important for several properties discussed we have mentioned it particularly and given it an own name.

For pointer types of the kind “REF MT”, where MT is a module type, we transfer the ordering relation:

POrd) “REF MT1” $\geq$ “REF MT2” $:\Leftrightarrow$ MT1 $\geq$ MT2.

The ordering relation can be defined analogously for higher order pointers. For each pointer level (“REF MT”, “REF REF MT”, etc.) we obtain a derivation graph isomorphic to that for the module types. A consequence of this correspondence are propositions as e.g. min(“REF MT”) = “REF min(MT)”.

Modula-3 [CDG 92] and Java [GJS 96] see module types (actually called “object types” in Modula-3 and “class types” in Java) as a mixture of module type and “pointer to module type”: assignment e.g. is pointer assignment, dereferencing is not possible, and “variable.component-id” accesses one of the components. Therefore, not all situations, which are investigated in this paper, are relevant to these two languages.

Example

In the rest of the paper we will refer to the following example of module types:

```
TYPE BasicFigureType = MODULE
PUBLIC
   PROC Move(To: PositionType);
INTERNAL
  VAR  CurrentPosition: PositionType INIT := (0,0);
END BasicFigureType;
```

This defines a module type `BasicFigureType` which contains two components: the procedure `Move` and the variable `CurrentPosition`. The components of the external interface are written under the heading `PUBLIC`. There is one public component: the procedure `Move`. The internal components are written under the heading `INTERNAL`.

```
TYPE CircleType DERIVED_FROM BasicFigureType = MODULE
PUBLIC
   PROC SetRadius(Radius: RadiusType);
INTERNAL
   VAR CurrentRadius : RadiusType INIT := 1;
END CircleType;

TYPE SquareType DERIVED_FROM BasicFigureType = MODULE
PUBLIC
   PROC SetSideLength(Side: SideType);
INTERNAL
   VAR CurrentSideLength : SideType INIT := 1;
END SquareType;
```

For this example we have:

CS(BasicFigureType) =
{ (PROC Move(PositionType)), (VAR CurrentPosition PositionType) }

E(BasicFigureType) = { (PROC Move(PositionType)) }

V(BasicFigureType) = { (VAR CurrentPosition PositionType) }

CS(CircleType) =
{ (PROC Move(PositionType)), (VAR CurrentPosition PositionType),
(PROC SetRadius(RadiusType)), (VAR CurrentRadius RadiusType) } =

CS(BasicFigureType) $\cup$
{ (PROC SetRadius(RadiusType), (VAR CurrentRadius RadiusType) }

E(CircleType) = { (PROC Move(PositionType)), (PROC SetRadius(RadiusType)) }

V(CircleType) =
{ (VAR CurrentPosition PositionType), (VAR CurrentRadius RadiusType) }

min(SquareType) = BasicFigureType

min(“REF CircleType”) = “REF BasicFigureType”

max(BasicFigureType) = {CircleType, SquareType}

An example of multiple inheritance is:

```
TYPE ColoringType = MODULE
PUBLIC
   TYPE ColorType = (Red, Green, Blue);
   PROC SetColor(Color: ColorType);
INTERNAL
   VAR CurrentColor : ColorType INIT := Blue;
END ColoringType;

TYPE ColoredCircleType
   DERIVED_FROM CircleType, ColoringType = MODULE
PUBLIC
   PROC ColoredCircleType(Radius: RadiusType; -- constructor
                          Color: ColorType);
END ColoredCircleType;
```

It is straightforward to compute the different sets of components as in the preceding example.

5 Typing Rules for Module Types

5.1 Variables

If we declare a variable of type MT we may ask whether derivation has any influence on the type of this variable. For the declaration

```
VAR MV: MT;
```

we have the following two conditions:

MVar1) ST(MV) = MT

MVar2) ST(MV) = DT(MV) is an invariant during the lifetime of MV.

This means that MV is always of type MT as is typical for variables in typed languages. The value of MV is always a module of type MT. Therefore, derivation has no influence on the typing of variables of module type.
The main reason to use these classical rules for the typing of variables of module type is one of implementation efficiency. It would be more complicated to implement the variables if the dynamic type could vary. This polymorphism is usually reserved for variables of type “pointer to module type” for which it is much easier to implement. Pointer types are treated in sect. 6.
The preceding paragraphs hold for both statically declared variables and dynamically created anonymous variables and for variables which are components of larger variables.

5.2 Assignment

In most languages with extensible types type compatibility for assignment is usually defined in a more relaxed form than for other types. This relaxation is due to the properties of the derivation relation. For an assignment of the form:

LHS := RHS;

where ST(LHS) and ST(RHS) are module types, we have the following rules.

MAss1)	Type compatibility:	ST(LHS) $\leq$ ST(RHS)
MAss2)	Semantics:	Assignment of corresponding variable components; this is called a projection because V(ST(LHS)) $\subset$ V(ST(RHS)) is possible. This rule guarantees that MVar2 holds.

If MonV holds, the assignment is statically safe. Since assignment means assignment of a copy of the corresponding components into the subcomponents of the variable LHS, the static type of LHS is not affected at all. Therefore this assignment is NOT an example of substitutivity.
These rules are typically used in object-oriented languages with strong typing as e.g. Borland Pascal with Objects, C++, and Oberon. Object CHILL uses the stronger rule ST(LHS) = ST(RHS) because projection was not rated as very important by the first users of the language. Different systems have been developed using Object CHILL [GW 92], but none of the users missed projection. Java does not provide assignment for composite types [GJS 96: 41, 460].
For the variables

```
VAR MyFigure, YourFigure: BasicFigureType;
VAR MyCircle: CircleType;
```

the following assignments are legal and statically safe:

```
MyFigure := YourFigure; -- same type
MyFigure := MyCircle;   -- projection:
             -- MyFigure.CurrentPosition :=
             -- MyCircle.CurrentPosition
```

On the other hand the following assignment is illegal:

```
MyCircle := MyFigure;  -- which value should be
             -- assigned to MyCircle.CurrentRadius ?
```

5.3 Parameters

The properties of a formal parameter FP of a procedure P are defined by its kind. The essential point is the mechanism of parameter association, i.e. the mechanism used to associate the formal parameter FP with the actual parameter AP given in a call ″P(FP => AP);”. In the following we discuss two kinds of parameters, value parameters and reference parameters, because these two forms are mostly used in typed object-oriented languages.

5.3.1 Value Parameters

A procedure with a value parameter typically looks like:

```
PROC P(FVP: MT);
   Body
```

For value parameters parameter association is defined as follows:
the call P(FVP => AVP); is equivalent to

```
VAR FVP: MT INIT := AVP;
   Body
```

This means that “ST(FVP) = DT(FVP)” is an invariant in the scope of FVP. Often FVP is treated as a local variable (CHILL and Object CHILL, Pascal, Modula) and in other cases as a local constant (Ada). This difference is of no importance to the topic of this paper. What is important here is that parameter association is essentially the same as assignment.
This observation suggests to use the same rules for typing and semantics as for assignment. This is typically done. Borland Pascal with Objects, C++, and Oberon use ST(FVP) ≤ ST(AVP) and projection, and Object CHILL uses ST(FVP) = ST(AVP).

If we have the following procedure P:

```
PROC P(Figure: BasicFigureType);
BEGIN
   Figure.SetPosition(To => (1,4));
END P;
```

the calls

```
P(Figure => MyFigure);
P(Figure => MyCircle);
```

are both legal and statically safe if the projection rule is used.

To call this parameter a “value parameter” is somewhat unfortunate because this term does not clearly describe the role FVP plays in P (as mentioned FVP is often a *variable* in the body of P). The term is mainly implementation oriented motivated by the fact that the value of the AVP is copied into the FVP.

5.3.2 Reference Parameters

The term reference parameter (sometimes also called variable parameter) is also an implementation oriented term but does not reflect the logic of parameter association. Reference parameters are used e.g. in CHILL, C++, Modula, Oberon, and Pascal. In Ada a more abstract view of formal parameters is used.
For a reference parameter FRP in a procedure P

```
PROC P(VAR FRP: MT);
   Body
```

parameter association is defined as follows:
the call P(FRP => ARP); is equivalent to:

```
VAR FRP: MT ALIAS ARP;
   Body
```

This equivalence follows from the definition of reference parameter (see e.g. [ANS 83: 6.6.3.3]) and of the implementation strategy for such parameters (see e.g. [ST 85: 602; Har 92: 118]).
A reference parameter FRP therefore has the following important properties:

Ref1) FRP is a variable of type MT and not a reference or pointer; in this aspect it is similar to a value parameter.

Ref2) FRP is a new (additional) name for the variable identified by ARP, i.e. the connection between FRP and ARP is much more close than for a value parameter. One consequence of this property is that anywhere in the intersection of the scopes of FRP and ARP FRP can be replaced with ARP, or equivalently FRP=ARP is an invariant in this intersection.

A consequence of Ref2 is that ST(FRP) = ST(ARP) must hold. If ST(FRP) $\neq$ ST(ARP) were allowed the substitution (FRP $\rightarrow$ ARP) would give ST(ARP) $\neq$ ST(ARP), which is a clear contradiction. ST(FRP) = ST(ARP) holds for all kinds of types.
The following examples show that weaker rules for parameter association lead to situations with undefined behavior.

a) ST(FRP) $\leq$ ST(ARP)
b) ST(FRP) $\geq$ ST(ARP)

Example a) (ST(FRP) < ST(ARP)):

```
PROC  Pgreater(VAR Figure: BasicFigureType);
   VAR LocalFigure: BasicFigureType;
BEGIN
   Figure := LocalFigure;
END Pgreater;
```

The call

```
            PGreater(Figure => MyCircle);
```

is now equivalent to the following block, in which the rule Ref2 has been applied to the parameter `Figure`:

```
BLOCK
   VAR Figure: BasicFigureType ALIAS MyCircle;
   VAR LocalFigure: BasicFigureType;
BEGIN
   Figure := LocalFigure;
END;
```

and, due to the alias relation between FRP and ARP, this is equivalent to:

```
BLOCK
   VAR Figure: BasicFigureType ALIAS MyCircle;-- (1)
   VAR LocalFigure: BasicFigureType;
BEGIN
   MyCircle := LocalFigure; -- ST(LHS) > ST(RHS)     (2)
       -- which value should be assigned to
       -- MyCircle.CurrentRadius ?
END;
```

The problem which becomes manifest in the assignment (2) can already be observed in the declaration (1): if `Figure` denotes a variable of type `BasicFigureType` it cannot simultaneously denote a variable of type `CircleType`, which is different from `BasicFigureType`.

Example b) (ST(FRP) > ST(ARP)):

```
PROC  PLess(VAR Circle: CircleType);
   VAR LocalCircle: CircleType;
BEGIN
   LocalCircle := Circle;
END PLess;
```

The call

```
             PLess(Circle => MyFigure);
```

is now equivalent to:

```
BLOCK
   VAR Circle: CircleType ALIAS MyFigure;
   VAR LocalCircle: CircleType;
BEGIN
   LocalCircle := Circle;
END;
```

and, due to the alias relation between FRP and ARP this equivalent to:

```
BLOCK
   VAR Circle: CircleType ALIAS MyFigure;
   VAR LocalCircle: CircleType;
BEGIN
   LocalCircle := MyFigure; -- ST(LHS) > ST(RHS)      (3)
      -- which value should be assigned to
      -- LocalCircle.CurrentRadius ?
END;
```

In assignments (2) and (3) ST(LHS) > ST(RHS) and V(ST(LHS)) $\supset$ V(ST(RHS)) hold, which contradict the rule Ass1 given in sect. 5.2. Assignment (3) is therefore especially dangerous because the implementation strategy for such assignments based on the rules MAss1 and MAss2 would assign an undefined value to `LocalCircle.CurrentRadius`. One solution to this problem is to check the assignment (3) dynamically. This would mean that a module assignment involving var parameters had to be implemented differently from a module assignment not involving such parameters. Apart from this, the conceptual problem of what the semantics of "`VAR Circle: CircleType ALIAS MyFigure;`" should actually be would still exist.
Borland Pascal with Objects, C++, and Oberon use the following type compatibility rule for reference parameters: ST(FRP) $\leq$ ST(ARP) which can lead to the problem presented in example a). Object CHILL uses the type compatibility rule ST(FRP) = ST(ARP) which guarantees that assignments involving reference parameters of module type are statically safe if the rules for assignment given in sect. 5.2 hold.
We discuss this problem further in sect. 7 after we have discussed the typing rules for pointer-to-module types in sect. 6.

5.4 Function result

In strongly typed procedural languages a function behaves very much in the same way as a variable, where the result type of the function corresponds to the type of the variable. Therefore MVar1 and MVar2 should also hold for function result.

6 Typing Rules for Pointer-to-Module Types

In object-oriented languages pointer-to-module types (PTMT) are typically treated differently from other pointer types. The reason is the following. The types of a derivation tree are logically related to each other in very much the same way as the variants of a record type with variants in Ada or Pascal. The variants of such a record type with variants are defined within *one* type:

```
TYPE Figure = RECORD
        CurrentPosition: PositionType;
        CASE Kind: FigureKind  OF
           Circle: CurrentRadius: RadiusType;
           Square: CurrentSideLength: SideType;
     END;
```

A variable of type Figure is polymorphic in the following way. For a variable

```
VAR  MyFigure: Figure;
```

the following assignments are legal and safe:

```
MyFigure := (Position => (3,4),
             Kind => Circle, CurrentRadius => 2);
```

```
MyFigure := (Position => (7,8),
             Kind => Square, CurrentSideLength => 4);
```

This is possible because the set of variants is fixed in the definition of the type `Figure`. We have seen in sect. 5.1 that such a polymorphism is typically not defined for variables of module type. The reason is that the variants are more isolated when they are defined via derivation. (This greater isolation has also some advantages but this is of no importance to the topic of this paper.) After the definition of the root type of a derivation hierarchy, as e.g. `BasicFigure` in the running example, the set of variants is not fixed but may be extended by the definition of additional derived types. Such derived types may even be defined in other or as other compilation units. This is the reason that implementation is more complicated than for record types with variants.
For PTMT the situation is simpler because the representation of a pointer value usually uses the same amount of storage even if the type of the value pointed to varies. This observation allows a polymorphism as used for record variables in the preceding assignment to be used for PTMT variables.

6.1 Variables

For a variable of a PTMT

```
VAR PV: REF MT;
```

typing is defined by a more relaxed rule: PV may assume pointers pointing to variables of any type MTp ≥ MT. Therefore the following is legal:

```
VAR PointerToFigure: REF BasicFigureType;
PointerToFigure := NEW BasicFigureType;
   -- PointerToFigure points to a variable of
   -- type BasicFigureType
PointerToFigure := NEW SquareType;
   -- PointerToFigure points to a variable of
   -- type SquareType
PointerToFigure := NEW CircleType;
   -- PointerToFigure points to a variable of
   -- type CircleType
```

To describe this behavior we distinguish between the static type and the dynamic type of a PTMT variable. As for other variables the static type is fixed in the declaration: ST(PV) = “REF MT” and ST(PointerToFigure) = “REF BasicFigureType”. On the other hand, the dynamic type may vary:

```
PointerToFigure := NEW BasicFigureType;
    -- PointerToFigure points to a variable of
    -- type BasicFigureType
    -- ST(PointerToFigure) = "REF BasicFigureType“  ∧
    -- DT(PointerToFigure) = "REF BasicFigureType“
PointerToFigure := NEW SquareType;
    -- PointerToFigure points to a variable of
    -- type SquareType
```

```
        -- ST(PointerToFigure) = "REF BasicFigureType“  ∧
        -- DT(PointerToFigure) = "REF SquareType“
    PointerToFigure := NEW CircleType;
        -- PointerToFigure points to a variable of
        -- type CircleType
        -- ST(PointerToFigure) = "REF BasicFigureType“  ∧
        -- DT(PointerToFigure) = "REF CircleType“
```

The set of admissible values of a PTMT “REF MT” can be characterized by:

$$\text{“REF MT”}_V = \{\text{nil}\} \cup \{p \mid p \text{ points to } V \ \wedge \ ST(V) \geq MT\}.$$

For a variable PV of a PTMT the following condition holds:

PPoly) $DT(PV) \geq ST(PV)$ is an invariant in the scope of PV.

We define

PNil) $ST(\text{nil}) \geq PT$ and $DT(\text{nil}) \geq PT$ for any PTMT PT.

The variability of the dynamic type of a PTMT variable together with the reimplementation of procedures is the technical basis for polymorphism.
If any PTMT variable PV is initialized with nil in its declaration then $DT(PV) \geq ST(PV)$ holds after the declaration of PV.
We assume that all pointer values ≠ nil are created by the operation NEW. An expression “NEW MT” has the following property:

PNew) ST(“NEW MT”) = DT(“NEW MT”) = “REF MT”.

For pointers the only operations are create, copy, and destroy. Therefore, a pointer value of type “REF MT” always points to a variable of type deref(“REF MT”) = MT. This gives us the basic property of PTMT variables:

PBasic) $PV \neq \text{nil} \Rightarrow ST(PV\uparrow) = DT(PV\uparrow) = \text{deref}(DT(PV))$

We use a notation in which the dereferencing of a pointer is indicated explicitly: if PV is a pointer variable “PV↑” is the variable pointed to by PV.
The rule PBasic is an implication of the rule MVar2 (sect. 5.1), because for any PTMT variable PV “PV↑” is an MT variable.
As a reminder we remark that despite polymorphism a PTMT variable PV of static type “REF MT” usually provides only access to the elements of E(MT). A useful consequence of this rule is that due to MonE all accesses “PV↑.id” are statically safe independently of the dynamic type of PV. Despite this constraint polymorphism is a useful mechanism. It allows the aggregation of variables of different (but related) module types into one data structure as e.g. an array or a linked list. It is furthermore useful in combination with reimplementation of procedures. Sometimes the term “redefinition” is used instead of reimplementation, and in some languages such procedures are called virtual or dynamic procedures. We do not go into further details here because we assume the reader is familiar with the main concepts of object-orientation.

6.2 Assignment

If we have PTMT variables two forms of assignment have to be distinguished:

a) LHS := RHS; where the type of LHS is a PTMT. This means that the assignment is on the level of pointer types. Therefore, this form of assignment is called *pointer assignment.*

b) LHS := RHS; where at least one of LHS and RHS has the form “expr↑” and the type of expr is a PTMT. This means that the assignment is on the level of module types and at least one of the operands is a dereferenced pointer whose type is a PTMT. Therefore, this form of assignment is called *deref assignment.* If neither LHS nor RHS has the form “expr↑” we have an assignment of the form which has been discussed already in sect. 5.2 or a pointer assignment.

For the discussion of assignment it is helpful to distinguish between the state before the assignment and the state after the assignment. We use the following notation: LHS is the state of that entity before the assignment and LHS’ is the state after the assignment.

6.2.1 Pointer Assignment

The discussion in sect. 6.1 means that for pointer assignment for PTMT a more relaxed rule than for other pointer types applies. Let ST(LHS) and ST(RHS) be PTMTs. For an assignment of the form:

LHS := RHS;

we have the following rules.

PAss1) Type compatibility: ST(LHS) $\leq$ DT(RHS)

PAss2) Semantics: Assignment of a copy of the value of the RHS, which implies DT(LHS’) = DT(RHS).

The type compatibility rule PAss1 allows for ST(LHS) > ST(RHS). In this case a dynamic check is necessary to guarantee the safety of the assignment, i.e. PPoly cannot be guaranteed statically.
Some typed object-oriented languages (e.g. Borland Pascal with Objects, Eiffel, Oberon) use the type compatibility rule:

PAss3) ST(LHS) $\leq$ ST(RHS)

instead of PAss1. PAss3 is stronger than PAss1 and guarantees PPoly statically. For the moment we do not include parameters which are treated in sect. 6.3. Therefore, PPoly can only be affected by the declaration and by assignment.

PROPOSITION 1: If PAss2 and PAss3 hold and any PTMT variable PV is initialized with nil, then PPoly is a static property in programs using declaration and assignment.

PROOF: The proof is by induction on the sequence of operations manipulating one specific pointer value p. Such a sequence begins either with the declaration of

PV, with an assignment of the form “PV := NEW MT;” or with an assignment of the form “PV := nil;”. The other elements of the sequence all have the form “PV1 := PV2;” where p is the value of PV2. For any assignment “PV := RHS” we have ST(PV) = ST(PV’).

1) DT(PV) $\geq$ ST(PV) holds after the declaration (see sect. 6.1).
2) Let the assignment have the form “PV := nil;”. By PAss2 we have DT(PV’) = DT(nil). By PNil we have DT(nil) $\geq$ ST(PV) = ST(PV’). Therefore, DT(PV’) $\geq$ ST(PV’) holds after the assignment.
3) Let the assignment have the form “PV := NEW MT;”. By definition of NEW we have DT(“NEW MT”) = ST(“NEW MT”) = “REF MT”. By PAss2 we have DT(PV’) = DT(“NEW MT”) = “REF MT”. By PAss3 we have ST(“NEW MT”) $\geq$ ST(PV) = ST(PV’). It follows that “REF MT” = DT(PV’) $\geq$ ST(PV’).
4) Let the assignment have the form “PV1 := PV2;”. We assume DT(PV2) $\geq$ ST(PV2) holds before the assignment. By PAss2 and PAss3 we have DT(PV1’) = DT(PV2) $\geq$ ST(PV2) $\geq$ ST(PV1) = ST(PV1’). ☑

The rules PAss2, PAss3 and MonE ensure that pointer assignments and access to external entities of module variables pointed to by pointer variables are statically safe if only assignments are performed. The case of parameters will be discussed in sect. 6.3.
Object CHILL and Simula use a more relaxed type compatibility rule for pointer assignment:

PAss4) [ST(LHS) $\geq$ ST(RHS) $\vee$ ST(LHS) $\leq$ ST(RHS)] $\wedge$
ST(LHS) $\leq$ DT(RHS).

For PAss4 a dynamic check is necessary if ST(LHS) > ST(RHS).
Rule PAss3 is essentially the same as MAss1. But there is a big difference between these two forms of assignment:

- a variable VMT of a module type has always the same type ST(VMT), i.e. DT(VMT) = ST(VMT) is an invariant during the lifetime of VMT.
- a variable VPMT of a PTMT is polymorphic which leads to the weaker invariant DT(VPMT) $\geq$ ST(VPMT). This means that, as has been described in sect. 6.1, VPMT may point to variables of different module types. This observation is important for deref assignment.

6.2.2 Deref Assignment

A deref assignment is actually an assignment for module types, i.e. MAss1 and MAss2 must hold.
The most general form of a deref assignment is

$$\text{PVL}\uparrow := \text{PVR}\uparrow;$$

where both PVL and PVR are of type PTMT. We assume PVL and PVR are both $\neq$ nil, since otherwise the attempt to execute the statement will result in an error. As a consequence of MAss1 we obtain:

$ST(PVL\uparrow) \leq ST(PVR\uparrow) \equiv$ [PBasic]
$deref(DT(PVL)) \leq deref(DT(PVR)) \equiv$ [PDer]
$DT(PVL) \leq DT(PVR)$.

Even if we require ST(PVL) = ST(PVR) the deref assignment is not statically safe:

```
VAR  PVL, PVR: REF BasicFigureType;
PVL := NEW CircleType;
PVR := NEW BasicFigureType;
PVL↑ := PVR↑;            --  DT(PVL) > DT(PVR)!
PVR := NEW SquareType;
PVL↑ := PVR↑;            --  DT(PVL) and DT(PVR) are
                         --  not related at all!
```

The only condition making a deref assignment statically safe under PPoly is

PAss5) $max(ST(PVL)) = \{ ST(PVR) \}$

PROPOSITION 2: If PAss2 and PAss5 hold, then the deref statement
"PVL↑ := PVR↑" is statically safe under PPoly.

PROOF: Since the right hand side of PAss5 consists of a singleton set, ST(PVR) is the only leaf in the subtree whose root is ST(PVL). Therefore this tree consists just of the path from ST(PVL) to ST(PVR). If ST(PVR) is maximal PPoly implies DT(PVR) = ST(PVR). If we assume DT(PVL) > DT(PVR) this means that DT(PVL) > ST(PVR) which contradicts the fact that ST(PVR) is maximal. Therefore DT(PVL) ≤ DT(PVR) must hold and according to PBasic ST(PVL↑) ≤ ST(PVR↑), i.e. MAss1 holds. ☑

PAss5 is a very strong restriction, but if it is not used a dynamic check is necessary, i.e. PAss5 is also necessary. This can be seen at the minimal derivation tree, which has one minimal node MT1 and more than one maximal nodes MT2 and MT3: $MT1 \leq MT2 \wedge MT1 \leq MT3 \wedge \neg(MT2 \leq MT3) \wedge \neg(MT3 \leq MT2)$. If ST(PVL) = "REF MT1" ∧ ST(PVR) = "REF MT2" then DT(PVL) = "REF MT3" means that the deref assignment is not safe because MT3 may contain a component not contained in MT2.
On the other hand, a deref assignment is never safe if the two types are not on the same path. Therefore,

PAss6) ST(PVL) and ST(PVR) are on the same path

could be a minimal requirement to be checked statically. This rule is used in Object CHILL.

6.3 Parameters

6.3.1 Value Parameters

For a value parameter of a PTMT the same observations as for module types apply: the value parameter is a local variable and the association between the formal parameter FVP and the actual parameter AVP is an assignment "FVP := AVP;". Therefore, the same rules as for pointer assignment can be used.

6.3.2 Reference Parameters

In sect. 5.3.2 it is shown that ST(FRP) = ST(ARP) must hold, due to the alias relation between FRP and ARP. This rule is used in Borland Pascal with Objects, C++, Oberon, and Object CHILL.
It is easy to give examples where weaker rules lead to illegal situations. One such example is given in [CCJ 93: 151].

6.3.3 PPoly as a Static Property

If for assignment to formal parameters the same rules hold as for variables PPoly is a static property.

PROPOSITION 3: If PAss2 and PAss3 hold and all PTMT variables PV are initialized with nil, then PPoly is a static property in programs using declaration, assignment, value parameters, and reference parameters.
PROOF: The proof is by induction on the sequence S of operations manipulating one specific pointer value p.

1) Value parameter: an FVP is essentially a PTMT variable and the association with the AVP is essentially an assignment. Therefore, all possible steps of S involving an FVP are already covered by the proof of Prop.1.
2) Reference parameter: an FRP is essentially a PTMT variable and the association with the ARP is essentially aliasing. Since the proof of Prop.1 does not rely on the fact that there is only one identifier for a variable all possible steps of S involving an FRP are already covered by the proof of Prop.1. ☑

6.4 Function Result

In strongly typed procedural languages a function behaves very much in the same way as a variable, where the result type of the function corresponds to the type of the variable. Therefore functions should be treated in the same way as variables.

7 Behavior in Different Languages

When discussing real languages we have to distinguish between the language definition and a compiler (or interpreter) for the language.

7.1 Languages

The tables 1 and 2 contain the compatibility rules for the languages Borland Pascal with Objects (BPO), C++, Oberon, and Object CHILL (OC) as they are given in the language definitions.

	MTAssign	PTMTAssign	DerefAssign
BPO	ST(LHS) $\leq$ ST(RHS)	ST(LHS) $\leq$ ST(RHS)	DT(LHS) $\leq$ DT(RHS)
C++	ST(LHS) $\leq$ ST(RHS)	ST(LHS) $\leq$ ST(RHS)	DT(LHS) $\leq$ DT(RHS)
Oberon	ST(LHS) $\leq$ ST(RHS)	ST(LHS) $\leq$ ST(RHS)	DT(LHS) $\leq$ DT(RHS)
OC	ST(LHS) = ST(RHS)	[ST(LHS) $\leq$ ST(RHS) $\vee$ ST(LHS) $\geq$ ST(RHS)] $\wedge$ ST(LHS) $\leq$ DT(RHS)	DT(LHS) = DT(RHS)

Table 1. *Compatibility Rules for Assignments*

	ValMTPar	RefMTPar	ValPTMTPar	RefPTMTPar
BPO	ST(FVP) $\leq$ ST(AVP)	ST(FRP) $\leq$ ST(ARP)	ST(FVP) $\leq$ ST(AVP)	ST(FRP) = ST(ARP)
C++	ST(FVP) $\leq$ ST(AVP)	ST(FRP) $\leq$ ST(ARP)	ST(FVP) $\leq$ ST(AVP)	ST(FRP) = ST(ARP)
Ober.	ST(FVP) $\leq$ ST(AVP)	ST(FRP) $\leq$ ST(ARP)	ST(FVP) $\leq$ ST(AVP)	ST(FRP) = ST(ARP)
OC	ST(FVP) = ST(AVP)	ST(FRP) = ST(ARP)	ST(FVP) $\leq$ ST(AVP)	ST(FRP) = ST(ARP)

Table 2. *Compatibility Rules for Parameters*

BPO, C++, and Oberon define the same rules for MTAssign and for PTMTAssign. OC differs in both cases; MTAssign being more restrictive and PTMTAssign being more liberal. For DerefAssign the situation is different because this form of assignment is not explicitly mentioned in the language definitions. Since a DerefAssign is essentially an assignment on the level of module values Table 1 contains the rules for MTAssign in the column for DerefAssign but taking PBasic into account. We may stress that this is our own interpretation, the language definitions just do not answer this question explicitly.
Table 2 shows furthermore that BPO, C++, and Oberon use two different compatibility rules for reference parameters:

a) if the type of the formal paramter is a MT the rule ST(FRP) $\leq$ ST(ARP) is used;
b) if the type of the formal parameter is a PTMT the rule ST(FRP) = ST(ARP) is used (which is also used for all other types).

As has already been mentioned in sect. 5.3.2 the compatibility rule for RefPTMTPar is not fully compatible with the alias relation between FRP and ARP. The reason for this erroneous rule seems to be that reference parameters are mistaken for pointers

("Reference parameters (in Pascal and Modula called VAR-parameters) are considered as local pointers to which a reference to the actual parameter is assigned initially. It follows that the same relaxation holds for both value and reference parameters." [Wir 88a: 209]). The alias relation is in fact implemented very similar to a pointer but on language level it has a different meaning.

7.2 Compilers

The behavior of compilers for these four languages is given in Table 3. It shows that the behavior of some of these compilers differs from the corresponding language definitions.

BPO: the compiler used is Borland Pascal with Objects V7.0 running under Windows 3.1 (case 33 could only be executed under MS DOS 6.0). All checks that looked as if they could be useful were activated.
When using a deref assignment several assignments, which are illegal if the assignment is done directly, are executed (cases 11, 12, 15, 16). Other assignments, which are legally executed if the assignment is done directly, are flagged at compile time (cases 17, 18).
In the case of a RefMTPar case 35 shows the problems discussed in sect. 5.3.2. The test program contains the assignments "LocalVariable := Parameter;" and "Parameter := LocalVariable;". They are both executed without any warnings.

			Oberon		BPO		C++		OC	
			La	Co	La	Co	La	Co	La	Co
MTAss										
1	L = R		+	+	+	+	+	+	+	+
2	L < R		+	+	+	+	+	+	-	cte
3	L > R		-	cte	-	cte	-	cte	-	cte
4	L s R		-	cte	-	cte	-	cte	-	cte
PTMTAss										
5	L = R	R = r	+	+	+	+	+	+	+	+
5a		R < r	+	+	+	+	+	+	+	+
6	L < R	R = r	+	+	+	+	+	+	+	+
7	L > R	R = r	-	cte	-	cte	-	cte	-	rte
7a		L = r	-	cte	-	cte	-	cte	+	+
8	L s R	R = r	-	cte	-	cte	-	cte	-	cte

Table 3. *Behavior of Compilers*

Continued on next page

Deref Assignment

9	L = R	l↑ = r↑	+	+	+	+	+	+	+	+
10		l↑ < r↑	+	+	+	+	+	+	-	rte
11		l↑ > r↑	-	rte	-	+	-	+	-	rte
12		l↑ s r↑	-	rte	-	+	-	+	-	rte
13	L < R	l↑ = r↑	+	rte	+	+	+	+	+	+
14		l↑ < r↑	+	+	+	+	+	+	-	rte
15		l↑ > r↑	-	rte	-	+	-	+	-	rte
16		l↑ s r↑	-	rte	-	+	-	+	-	rte
17	L > R	l↑ = r↑	+	cte	+	cte	+	cte	+	+
18		l↑ < r↑	+	cte	+	cte	+	cte	-	rte
19		l↑ > r↑	-	cte	-	cte	-	cte	-	rte
20		l↑ s r↑	-	cte	-	cte	-	cte	-	rte
21	L s R	l↑ = r↑	-	cte	-	cte	-	cte	-	cte
22		l↑ < r↑	-	cte	-	cte	-	cte	-	cte
23		l↑ > r↑	-	cte	-	cte	-	cte	-	cte
24		l↑ s r↑	-	cte	-	cte	-	cte	-	cte

ValMTPar

25	F = A		+	+	+	+	+	+	+	+
26	F > A		-	cte	-	cte	-	cte	-	cte
27	F < A		+	+	+	+	+	+	-	cte
28	F s A		-	cte	-	cte	-	cte	-	cte

ValPTMTPar

29	F = A	A = a	+	+	+	+	+	+	+	+
29a		A < a	+	+	+	+	+	+	+	+
30	F > A	A = a	-	cte	-	cte	-	cte	-	rte
30a		F = a	-	cte	-	cte	-	cte	+	+
30b		F < a	-	cte	-	cte	-	cte	+	+
31	F < A	A = a	+	+	+	+	+	+	+	+
32	F s A	A = a	-	cte	-	cte	-	cte	-	cte

RefMTPar

33	F = A		+	+	+	+	+	+	+	+
34	F > A		-	cte	-	cte	-	cte	-	cte
35	F < A		+	+ (rte)	+	+	+	+	-	cte
36	F s A		-	cte	-	cte	-	cte	-	cte

Table 3. *Behavior of Compilers*

Continued on next page

RefPTMTPar

37	F = A	A = a	+	+	+	+	+	+	+	+
37a		A < a	+	+	+	+	+	+	+	+
38	F > A	A = a	-	cte	-	cte	-	cte	-	cte
39	F < A	A = a	-	cte	-	cte	-	+	-	cte
40	F s A	A = a	-	cte	-	cte	-	cte	-	cte

Abbreviations used in Table 3:

La: Language definition; Co: Compiler behavior

L: ST(LHS); R: ST(RHS); l: DT(LHS); r: DT(RHS); s: sibling

A: ST(actual parameter); F: ST(formal parameter); a: DT(actual parameter)

+ : legal / successfully executed; - : illegal;
cte: compile time error; rte: runtime error

Table 3 (cont'd.). *Behavior of Compilers*

C++: the compiler used is Borland C++ V4.0 and V3.1 running under Windows 3.1 (the executable cases where executed under V3.1). All checks that looked as if they could be useful were activated.

When using a deref assignment several assignments, which are illegal if the assignment is done directly, are executed (cases 11, 12, 15, 16). Other assignments, which are legally executed if the assignment is done directly, are flagged at compile time (cases 17, 18).

In the case of a RefMTPar case 35 shows the problems discussed in sect. 5.3.2. The test program contains the assignments "LocalVariable := Parameter;" and "Parameter := LocalVariable;". They are both executed without any warnings.

In the case of a RefPTMTPar case 39, which is illegal in the language definition and which is illegal if the assignment is done directly, is executed without any warnings

Oberon: the compiler used is Oberon (TM) System 3 V1.2 (Compiler NW 1.8.91 / ARD 5.93) running under MS DOS 6.0.

For deref assignment illegal cases are caught either at compile time or at runtime. Case 13, which is legal, results in program abortion (TRAP 19). Other assignments, which are legally executed if the assignment is done directly, are flagged at compile time (cases 17, 18).

In the case of a RefMTPar case 35 shows the problems discussed in sect. 5.3.2. The test program contains the assignments "LocalVariable := Parameter;" and "Parameter := LocalVariable;". Parameter association is executed without warning. The program is aborted when attempting to execute "Parameter := LocalVariable;" (TRAP 19).

OC: the compiler used is Object-CHILL V3.01 running under OS/2 2.1. The generation of runtime checks for module types and PTMT was activated. The compiler and the generated programs comply with the language definition.

8 Comparison with Other Work

Work on typing for object-oriented languages (OOL) has often been based on typing for functional programming (FP) [Bru 93; CCH 89; CL 94; CL 95; CP 93] or on constructive logic [Car 93]. The type systems studied in these approaches usually deal with the types of simple and higher order functions. Practical OOL do mostly not support higher order functions in general but contain other elements which are not covered by the approaches based on FP, as e.g. assignment and pointers. Therefore, the approach of this paper and the approaches based on FP overlap only partially. Similar differences are also observed in [CCH 89a]. The very comprehensive paper [Car 93] contains also a number of those concepts used in practical OOL. The paper [LW 93] is based on a "proof-theoretic" approach instead of a "model-theoretic", on which the majority of work an typing is based. [LW 93] has a focus on assertions and does neither treat reference parameters nor pointers. Similarly, [PS 94] deals neither with reference parameters nor with pointers (in the sense that pointer level and the level of dereferenced pointers are distinguished).

In this paper we deal with four aspects:

a) typing of extensible module types (EMT)
b) typing of assignment for EMTs
c) typing of value and reference parameters whose types are EMTs
d) typing of pointer types whose base types are EMTs

a) Typing of EMT is usually well covered by work on typing. In this paper we repeat the definitions to make the paper self contained. In comparison to [Car 93] EMT are very similar to a combination of record types and interfaces. This is typical for current programming languages as e.g. C++ or Object CHILL. Java, on the other hand, distinguishes between interface types and module types (actually called "class" types) [GJS 96]. Another difference is that the types of the components are not taken into account when defining MT1 ≤ MT2 whereas they are in [Car 93]. This reflects the definition of MT1 ≤ MT2 as of MT2 being an *extension* of MT1 [Wir 88a]. If MT2 is obtained by adding components to the component set of MT1 then the common components of MT1 and MT2 are necessarily the same. This again reflects current practice. The approach of [Car 93] describes therefore a richer set of typing facilities whereas this paper is more focused on the facilities of languages currently in use.

b) Assignment is typically avoided in FP. [Car 93] also covers assignment, but it seems in a more limited form than in OOL, because the typing of the assignment is given by the rule "var(A) := A". In OOL assignment often includes projection when the type of the RHS is an extension of the type of the LHS.

c) Reference parameters are usually not needed in approaches based on FP because there are no variables. In OOL reference parameters are local variables which may be read and updated. [Car 93] discusses IN and OUT parameters. Since an IN parameter may have a VAR component the question of reference parameters arises indirectly and is solved in the same way as in this paper: $(A<:B) \wedge (B<:A) \Rightarrow var(A) <: var(B)$.

d) Pointer and pointer types are not used in FP and they are also not treated in [Car 93]. Even in the area of practical OOL this question has not been discussed very thoroughly. The combination of polymorphism of pointers and the dereferencing of such pointers leads to problems as shown in this paper.

9 Discussion

9.1 Reference Parameters

The survey of the type compatibility rules in the four languages shows that three of them (BPO, C++, Oberon) have a rule for reference parameters of module type which may lead to situations with undefined behavior. The technical properties and consequences of this rule are:

a) the rule interferes with the homogeneity of the language. In those three languages assignments between module variables are statically safe. The rule for reference parameters of module type makes assignments involving such parameters statically unsafe. Such parameters are conceptually conceived as variables but differ in their properties from other variables.

b) more important is the fact that the statically declared type of a reference parameter of module type is no longer a static property; due to the aliasing the actual type may be different for different calls.

In C++ a special kind of values is used to realize the effect of reference parameters; these values are called references. When used as a formal parameter a reference to a module type is converted to “a reference to the base class sub-object of the derived class object” [ES 90: 38, 49]. No precise definition of the sub-object is given. The compiler does obviously not generate any code for this conversion. Therefore, this sub-object is a sort of hybrid: the external interface is that of the basic type and the behavior is that of the derived type of the ARP. It is therefore possible to manipulate data components defined only in a derived type by calling a procedure which is re-implemented in the derived type and which manipulates such components. This has been validated by a corresponding experiment with the Borland C++ compiler. In order to produce a consistent sub-object the compiler should generate code to alter the pointer to the type descriptor such that it points temporarily to the descriptor of the base type.

9.2 Deref Assignment

It seems that up to now not all ramifications of the deref assignment have been seen. Table II suggests that BPO, C++, and Oberon use statically the same rule as for pointer assignment: ST(LHS) ≤ ST(RHS). Oberon, which does runtime checks, traps the erroneous cases (and one legal one (# 13)) at runtime. BPO and C++ (Borland) do not recognize the erroneous cases which are possible if only ST(LHS) ≤ ST(RHS) is checked at compile time.

10 References

ANS 83 ANSI / IEEE 770 X3.97-1983: American National Standard Pascal Computer Programming Language. IEEE, New York, 1983

Bor 93 Borland GmbH: Borland Pascal mit Objekten 7.0 - Programmierhandbuch. Langen, 1993

Bru 93 Bruce, Kim B.: Safe Type Checking in a Statically-Typed Object-Oriented Programming Language. 20th POPL 1993, pp. 285..298

Car 93 Cardelli, Luca: Typeful Programming. SRC Research Report 45, Jan 1, 1993, DEC SRC Palo Alto. (Earlier Version in: Neuhold, E.J.; Paul, M. (eds): Formal Description of Programming Concepts. Springer, 1991)

CCH 89 Cook, William; Hill, Walt; Canning, Peter: Inheritance is not Subtyping. Report STL-89-17, Hewlett-Packard Laboratories, Palo Alto

CCH 89a Canning, Peter S.; Cook, William, R.; Hill, Walter L.; Olthoff, Walter G.: Interfaces for Strongly-Typed Object-Oriented Programming. OOPSLA’89, SIGPLAN Notices 24,10 (1989) 457..467

CCJ 93 Lee, Joon-Kyung; Jo, Chang-Hyun; Lee, Dong-Gill; Choi, Wan; Choi, Go-Bong; Lee, Chung-Kun: An Efficient Implementation of Type-Test and Type-Guard for an Object-Oriented Switching System. Infocom’93, Bombay 25-27 Nov. 1993, 148-155

CDG 92 Cardelli, Luca; Donahue, James; Glassman, Lucille; Jordan, Mick; Kalsow, Bill; Nelson, Greg: Modula-3 Language Definition. SIGPLAN Not. 27,8 (1992) 15 .. 42

CG 89 Carriero, Nicholas; Gelernter, David: Linda in Context. CACM 32,4 (1989) 444..458

CHI 96 CHILL Homepage: http://www1.informatik.uni-jena.de/languages/chill/chill.htm

CL 94 Chambers, Craig; Leavens, Gary T.: Typechecking and Modules for Multi-Methods. OOPSLA’94, SIGPLAN Not. 29.10(1994) 1..15

CL 95 Castagna, Guiseppe; Leavens, Gary T.: Foundations of Object-Oriented Languages - 2nd Workshop Report - SIGPLAN Notices 30, 2 (1995) 5..11

CP 93 Caseau, Yves; Perron, Laurent: Attaching Second-Order Types to Methods in an Object-Oriented Language. ECOOP’93, Springer 1993, LNCS 707, pp. 142..160

CW 96 Werther, Ben; Conway, Damian: A Modest Proposal: C++ Resyntaxed. SIGPLAN Notices 31, 11 (1996) 74..82

DW 92 Winkler, Jürgen F.H.; Dießl, Georg: Object CHILL - An Object-Oriented Language for Systems Implementation. ACM Computer Science Conference’92, 139-147

ES 90	Ellis, Margaret A.; Straustrup, Bjarne: The Annotated C Reference Manual. Addison-Wesley, 1990. 0-201-51459-1
GJS 96	Gosling, James; Joy, Bill; Steele, Guy: The Java™ Language Specification. Addison-Wesley, 1996. 0-201-63451-1
GW 92	Günther, W., Wackerbarth, G.: Designing ISDN Call Processing Software by Using Object-Oriented Techniques. Int. Switching Symp. 1992, Yokohama, Vol.1, p.174-178
Har 92	Harbinson, Samuel P.: Modula-3. Prentice Hall, 1992. 0-13-596404-0
KMP 69	Wijngaarden, A. van (Ed.); Mailloux, B. J.; Peck, J. E. L.; Koster, C. H. A.: Report on the Algorithmic Language ALGOL 68. Num. Math. 14 (1969) 79-218
LW 93	Liskov, Barbara; Wing, Jeanette M.: A New Definition of the Subtype Relation. ECOOP'93, Springer 1993, LNCS 707, pp. 118..141
Mey 92	Meyer, Bertrand: Eiffel: the Language. Prentice Hall, 1992. 0-13-247925-7
OMG 93	OMG: The Common Object Request Broker: Architecture and Specification. Revision 1.2, Draft 29 December 1993
PS 94	Palsberg, Jens; Schwartzbach, Michael I.: Static typing for object-oriented programming. Science of Computer Programming 23 (1994) 19..53
Rec 93	Recommendation Z.200 (11/93) - CCITT High Level Language (CHILL). ITU, Geneva, 1993. See also: ISO/IEC 9496:1995 CCITT high level language (CHILL)
Ref 83	Reference Manual for the Ada Programming Language. ANSI / MIL-STD 1815 A, 1983. Springer, LNCS 155, 1983.
RW 92	Reiser, Martin; Wirth, Niklaus: Programming in Oberon. ACM Press and Addison-Wesley, 1992. 0-201-56543-9
Seb 93	Sebesta, Robert W.: Concepts of Programming Languages. The Benjamin / Cummings Publ. Comp., Redwood City, 2nd ed. 1993. 0-8053-7130-3
ST 85	Tremblay, Jean-Paul; Sorenson, Paul G.: The Theory and Practice of Compiler Writing. McGraw-Hill, 1985. 0-07-065161-2
Win 92	Winkler, Jürgen F.H.: Objectivism: "CLASS" considered harmful. CACM 35,8 (1992) 128-130
Wir 88a	Wirth, N.: Type Extensions. ACM TOPLAS 10, 2 (1988) 204-214
Wir 88b	Wirth, Niklaus: Programming in Modula-2. Springer, Berlin etc., 1988. 3-540-15078-1

Acknowledgments

I am very grateful to S. Kauer, D. Schmelz, H. Toparkus, J. Waldmann, and the referees for very helpful comments on earlier drafts of this paper. I also thank T. Brandl for doing the experiments with the Object CHILL compiler.

Appendix: Collected Definitions

MonE) $\text{MTd} \geq \text{MTb} \Rightarrow \text{E(MTd)} \supseteq \text{E(MTb)}$

MonV) $\text{MTd} \geq \text{MTb} \Rightarrow \text{V(MTd)} \supseteq \text{V(MTb)}$

POrd) “REF MT1” $\geq$ “REF MT2” $:\Leftrightarrow$ MT1 $\geq$ MT2.

MVar1) ST(MV) = MT for `VAR MV: MT;`

MVar2) ST(MV) = DT(MV) is an invariant during the lifetime of MV.

MAss1) Type compatibility for MT assignment: ST(LHS) $\leq$ ST(RHS)

MAss2) Semantics: Assignment of corresponding variable components; this is called a projection because V(ST(LHS)) $\subset$ V(ST(RHS)) is possible. This rule guarantees that MVar2 holds.

Ref1) FRP is a variable of type MT and not a reference or pointer; in this aspect it is similar to a value parameter.

Ref2) FRP is a new (additional) name for the variable identified by ARP, i.e. the connection between FRP and ARP is much more close than for a value parameter. One consequence of this property is that anywhere in the intersection of the scopes of FRP and ARP FRP can be replaced with ARP, or equivalently FRP=ARP is an invariant in this intersection.

PPoly) DT(PV) $\geq$ ST(PV) is an invariant in the scope of PV.

PNil) ST(nil) $\geq$ PT and DT(nil) $\geq$ PT for any PTMT PT.

PNew) ST(“NEW MT”) = DT(“NEW MT”) = “REF MT”.

PBasic) PV $\neq$ nil $\Rightarrow$ ST(PV$\uparrow$) = DT(PV$\uparrow$) = deref(DT(PV))