A Static Type System for Message Passing

Giorgio Ghelli
Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, Italy
ghelli@dipisa.di.unipi.it

Abstract

Much research has been performed with the aim of isolating the basic notions of object-oriented languages and of describing them by basic operators which can be embedded in strongly-typed languages. In this context we define an atomic linguistic construct to capture the notion of message passing, and we define a static and strong type system, based on subtyping, for this construct. Message passing is modelled as the application of an overloaded function, whose behavior is determined only at compile-time on the basis of the class which created a distinguished argument, the object "receiving" the message. By embedding this mechanism in a strongly-typed language with subtyping and abstract data types we can obtain the full functionality of object-oriented languages. We show that this approach is strictly more expressive then the usual interpretation of message passing as selection plus application of a functional record field.

1. Introduction

Object-oriented languages are based on the notions of *objects*, *classes*, *messages/methods* and *inheritance*.

objects are entities with a record-like state, collected in *classes*. A *class* plays both the rôle of an Abstract Data Type (ADT) and of a generator of homogeneous objects; it is defined by specifying: the record type representing the structure of the state of the objects produced, the name of their abstract type, and its *methods*, i.e. the basic functions which can be applied, by *message sending*, to them.

Message sending is the object-oriented version of function call: when a message (a pair <*method name*,*args*>) is *sent* to an object *obj*, the "corresponding" method is evaluated, receiving the object *obj* (the *distinguished parameter*) and *args* as parameters. Many different methods in different classes can correspond to a given method name; the one executed depends on the class which created the *distinguished parameter* (this is known as *overloading of method names*). The method associated to a message does not depend on the compile-time type of the receiving object, but on the class which generated the object, so that this meaning can be decided only at run-time (*late binding of method names*). Besides this, the method associated at run-time to a message could even belong to a class defined after the compilation of the code issuing the message call (*dynamic extension of the meaning of method names*).

A new class *NC* can be defined by *inheritance* from an old one *OC* by specifying which fields are added to the structure of the state of the objects of *OC*, which new methods are added, and possibly how the code of some old methods changes; in this case *NC* is called a *subclass* of *OC*.

The high level of reusability of object-oriented software is due to the interplay of all of these four features: overloading, late binding, dynamic extensibility and inheritance.

This work was carried out with the partial support of Selenia, project "Very Large Knowledge Bases", of E.E.C., Esprit B.R.A. 3070 FIDE, and of Italian C.N.R., P.F. "Sistemi informatici e calcolo parallelo".

The embedding of these features into languages with well established semantics and type systems is widely studied, with three objectives: • a formal understanding of object-oriented languages, • the definition of a strong type system for these languages, • the definition of languages enjoying both a strong foundation and the high expressivity and code reusability of object-oriented languages.

Most of these researches are based on the encoding of the mechanisms of object-oriented languages in a strong type system with subtyping (see, e.g. [CarWeg85] [AlbGheOccOrs88] [CanHilOlt88] [Coo89] [CooHilCan90] [Mit90] [Red88] [Ghe91]). An object is represented by a record containing the methods of its class as functional fields; passing a message *<methName,params>* to an object *obj* is interpreted as selection and application of that functional field: *(obj.methName)(params)*. This interpretation immediately gives overloading, late binding and dynamical extensibility of methods. In short, field names are overloaded, since a unique field name can be associated with different functions in records produced by different classes. The function contained in a record field does not depend on the record type, but is decided by the function which creates the record, and can be known only at run-time (late binding). Finally, the set of the meanings of a field name can be extended, since it is always possible to define new records where a field name, already used in other records, assumes new meanings; old code accessing this new record will retrieve the new meaning. In this approach class inheritance introduces many type-level complications, since it is modelled by record concatenation, which needs complex type systems in presence of subtyping [CarMit89] [Wan89]. The result is that in these systems either no inheritance is present, like in the seminal work [Car88], either subtyping is heavily limited, like in [CooHilCan90], or both inheritance and subtyping are offered, paying the price of very complex record subtyping rules, like in [Mit90] and in [CarMit89].

The record-based approach presents another well-known technical problem: the interaction of the subtyping rules of the record and function types used to codify objects, has the undesired effect that the argument types of methods redefined in a subclass can be only generalized, rather then specialized (*specializing* means *changing into a subtype*; *generalizing* is the inverse operation). This constraint on the redefinition of method argument types is called *contravariance.*

Contravariance implies immediately that, in the record model, the object receiving the message cannot be seen as a regular argument of the method, since its type is necessarily specialized going from superclasses to subclasses; for this reason the receiving object is not seen as an argument but is accessed by its methods "by recursion" [CarWeg85]. But in turn, this way of accessing the distinguished argument entails new difficulties in the interpretation of inheritance as record concatenation, as discussed in [CooHilCan90] and [Mit90]; the solutions proposed there destroy the simplicity of the record approach.

Besides these technical problems on the distinguished argument, type contravariance is an undesired constraint also for the other method arguments, since experience shows that covariance of argument types is much more common in practice [DanTom88].

The problems above are due to the fact that the record-based approach is not a direct description of the message passing mechanism, but rather an encoding of it. To overcome these difficulties, and to study the object-oriented mechanisms at a more basic level, we introduce a completely different way of viewing objects and message passing.

In our approach inheritance and message passing are modelled, respectively, by the incremental definition and by the application of overloaded functions. The meaning of our overloaded functions can be solved only at compile-time on the basis of the class which created a distinguished argument. Conversely, objects are just entities which can remember the class which created them. This mechanism reflects the kernel of the message passing mechanism, and the record-based approach can be seen just as one "implementation" of this fundamental mechanism.

We define also a strong type system for this mechanism, i.e. a type system such that a well-typed expression cannot raise any run-time failure. Despite late binding, this type system is static: type checking is performed completely at compile-time.

This mechanism models one-half of the object-oriented paradigm, i.e. message passing with late-binding overloading, and inheritance (dynamic extensibility is modelled by the general mechanisms of the host language). The other half of the paradigm is represented by the possibility of encapsulating the state of objects and the implementation of classes. In our approach object-oriented encapsulation can be understood in terms of any general purpose encapsulation mechanism, like abstract data types, modules or existential types (see e.g. [CarWeg85] or [MitPlo85]), since our overloading mechanism can be applied to the functions defined on abstract data types too. This is not true in the record-based approach, where only record field names can be overloaded, while the functions defined on an abstract data type cannot.

To sum up, our model of message passing is more convenient than the traditional record-based one with respect to the following points:

- It allows both covariant and contravariant redefinition of methods in the context of a static and strong type system
- It allows studying simple and multiple inheritance without affecting the subtyping rules
- Non recursive methods can be defined without using recursion

But our construct allows also defining typed languages which are more expressive than traditional object-oriented languages. With respect to method definition, this is showed in the paper through the following example. Let *ColoredPoint* be a subclass of *Point*. In the record model, due to contravariance, a method *equal* comparing two *Point*'s cannot be overloaded to make it able to compare also two *ColoredPoint*'s. In an untyped object-oriented language this redefinition is allowed, but you cannot explicitly specify what happens when a *Point* and a *ColoredPoint* are compared, and this comparison may even raise a run-time error. In our model the covariant redefinition of the *equal* method is allowed, run-time failures are not possible, and you can even explicitly program how a *Point* is compared with a *ColoredPoint*. Besides this, in our model methods are first class values of the language, which is not true in the record-based model or in the untyped object-oriented languages.

The paper is structured as follows. In §2 we introduce a basic strongly-typed language, a weakly-typed "traditional" object-oriented language and its translation in the record model. In §3 we present our strong type system for late-binding overloading. In §4 we compare our approach with other ones. In §5 we discuss how dynamic extension and encapsulation can be realized. In §6 we draw some conclusions.

2 Basic definitions

2.1 *A functional language*

This is the syntax of our basic functional language ([*a*] means *a* is optional):

Declaration:= **let** ***ide***:= Val
  | **let** [**rec**] ***ide***[:Type] := Val
  | **let** [**rec**] **type** ***typeIde*** := Type

Val:= ***constant*** | ***ide*** | Declaration; Val
  | **fun** (*ide*:Type,...,*ide*:Type). Val
  | Val(Val,...,Val)
  | **record** *label*:= *Val*,...,*label*:= *Val* **end**
  | *Val.label*
  | **rec** ***ide***[:Type]. Val | Val; Val | Val: Type

Type:= ***typeIde*** | ***basicType*** | Type×Type[1]
  | Type→Type
  | **Record** *label*: *Type*,...,*label*: *Type* **End**
  | **Rec** ***typeIde***. Type

A declaration

[1]For simplicity we use product types only to pass parameters

*"**let** ide:=Val / **let type** typeIde := typexpr"*
binds an identifier to a value or to a type in all the expressions which follow it, in a terminal session or in this paper. These are constant bindings; updatable references are also needed in the host language, but they are not defined here since they are not used in any example.

*"**fun** (ide$_1$:Type$_1$,...,ide$_n$:Type$_n$).* Val" is an expression which denotes a function: *"ide$_1$,...,ide$_n$"* are the formal parameters, *"Type$_1$,...,Type$_n$"* are the type required for the actual parameters and *"Val"* is the body of the function. *"Type$_1$×...×Type$_n$→TypeR"* is the type of this function if *"TypeR"* is the type of *"Val"*. *"Val(Val,..., Val)"* is function application.

*"**record** label := Val,...,label := Val **end**"* is a record whose type has the form *"**Record** label : Type,...,label : Type **End**"*. Record fields can be selected using the notation *"rec.field"*. The order of labels in records is irrelevant.

*"**rec** ide. expr"* returns the value of expr evaluated in an environment where *ide* is recursively bound to the value which is being built. ***let rec** ide := expr* means ***let** ide := **rec** ide. expr*, i.e. the identifier *ide* is recursively bound to the value under construction (***rec** ide. expr*) and remains bound to it after the construction (***let** ide := **rec**...*; see the example below):

*"**rec** typeide. TypeExpr"* returns the value of TypeExpr evaluated in an environment where *typeide* is recursively bound to the type which is being defined. ***let rec type** typeide := TypeExpr* means ***let type** typeide := **Rec** typeide. expr*:

```
let rec type Graph :=
        Record  value: Int
            outStar: List of Graph
        End;

let loop := rec self:Graph.
        record value:=0,
            outStar:= list of (self)
        end;

let rec loop:Graph :=
        record value:=0,
            outStar:= list of (loop)
        end;
```

"Val;Val" denotes the sequential execution of two expressions, which returns the result of the second expression.

If one of the types of *"Val"* is *"Type"* then *"Val:Type"* is the same expression, but having as types only *"Type"* and all its supertypes. This construct is used mainly to avoid ambiguity in the type of terms with too many types, like the term denoting an empty list: writing *emptylist: List of Int* we restrict the type of this empty list to the lists of integers.

The plugs *"constant"* and *"basicType"* denote a set of other values, operations and type constructors which could be useful, like lists (the constructor *List of* used above) integers, booleans, pairs etc.

A reflexive and transitive inclusion relation, *subtyping*, is defined on types [Car88]. Subtyping means that if *a:T* and *T* is a subtype of *U* (*T≤U*), then *a:U*. The typing and subtyping rules are in the Appendix A.

2.2 A weakly-typed object-oriented language

We exemplify a weakly-typed object-oriented language by adding the following ***Class*** construct, to declare classes, to our micro functional language. Notice that the resulting language is *not* our proposal; the construct below is just aimed to illustrate some points about message passing in weakly-typed object-oriented languages:

```
let ObjType(x1:T1,...,xn:Tn):=
                        Class <methods> end
```

The definition above defines a new type *ObjType* and a function ***new[ObjType]***, to generate "objects" of that type[2]. *<methods>* is a list of bindings with form

[2]In this micro language the state of the object has the same structure of the input parameters of the creation function; this is just an irrelevant simplification

ide(parameters:ParamTypes):=expr:ResultType;

each of which defines a new function

ide:ObjType →ParamTypes →ResultType,

called "a method of the class *ObjType*"; the distinguished parameter of each method is not explicitly declared, and is called *self*[3]. Classes can be defined "by inheritance" by extending other classes:

```
let ChildType(x_{n+1}:T_{n+1},...x_{n+m}:T_{n+m}):=
  Class isa ParentT_1,...,ParentT_l <methods>
  end
```

In this case *ChildType* is a subtype of $ParentT_1,...,ParentT_l$, so that any method of a $ParentT_i$ can be applied to values of type *ChildType* too (method inheritance). The creation function *new[ChildType]* requires all the input parameters of the $ParentT_i$'s plus those added in its own definition (state inheritance). If a method is specified both in a $ParentT_i$ and in *ChildType*, then it is overloaded with the new definition; we say that it is *extended*, since it acquires a new *branch* without losing the old ones (we call branches the different definitions supplied for an overloaded method). If a method is defined in more then one $ParentT_i$, but not in *ChildType*, then some *choice rule* of the language selects the inherited definition. *Single inheritance* means that there is just one parent type, so that no choice rule is needed; otherwise we have *multiple inheritance*.

When an overloaded method is applied to an object, a branch is selected depending on the run-time type of the object (the class which created it), which is in general only a subtype of the compile-time type of the expression returning it. E.g., let *Rectangle* and *Circle* be subtypes of *Picture*; in the following function the expression *pic* has compile-time type *Picture*, but if the list *lpics* contains rectangles and circles, then in each run of the *for* loop the *redraw* method of rectangles or of circles is selected:

```
let redrawAllPictures :=
    fun(lpics: List of Picture).
        for pic in lpics do redraw(pic);
```

Solving overloading on the basis of run-time (resp. compile-time) types is called *late* (resp. *early*) *binding*[4]; e.g., imperative languages use early binding for overloaded operators (like, e.g., a "+" operator overloaded on integer and floating point numbers).

The extension of a method affects also already "compiled" method calls: if a new subclass *Square* of *Picture* is added, then the already defined *redrawAllPictures* function will afterwards select the new definition of *redraw* for objects of the class *Square*. This "dynamic extension" is a kind of dynamic binding, which allows changing the meaning of a method identifier by adding new branches for new subtypes. It is related to late binding (solving overloading using the run-time type) from an implementative point of view, since both mechanisms need a run-time binding of method names, but the two notions are different (cf. §5).

The *redrawAllPictures* example suggests how, in object-oriented languages, an application, (e.g. a graphic editor), can be first implemented and packaged, and later on modified simply by adding a subclass (e.g. circles), without modifying or recompiling the existing code, exploiting critically all the four features of inheritance, overloading, late binding and dynamic extensibility.

2.3 The record model for object-oriented languages

The record model is a techinque to understand message passing by translating it into a functional language with rtecors and subtypes. We present the record model through one example. Consider the following declarations, expressed in the weakly-typed object-oriented toy language:

[3]In the obiect-oriented jargon, objects "contain" their own methods, and *self* is not a parameter but an auto-reference

[4]This distinction affects only languages with some form of polymorphism, like subtyping, since otherwise the run-time type of a value is just its compile time type

```
let Point(x:Int, y:Int) := Class
  get_x():= x, get_y():= y,
  equal(p:Point) := get_x(self)=get_x(p)
              & get_y(self)=get_y(p)
end;

let ColorPoint(x:Int, y:Int, color:Color) := Class isa
Point
  get_color():= color;
  equal(p:ColorPoint) := get_x(self)=get_x(p)
              & get_y(self)=get_y(p)
              & get_color( self)=get_color(p);
end;
```

In the record model they would be translated as:

```
let rec type Point :=Record get_x: Unit → Int,
              get_y: Unit → Int,
              equal: Point → Bool
         End;
let newPoint: Int×Int → Point :=
  fun (x:Int, y:Int).
  let state := record x:=x, y:=y end
  in  rec[5] self.
     record
       get_x:= fun(_:Unit). state.x,
       get_y:= fun(_:Unit). state.y,
       equal:= fun(p:Point). p.get_x()=self.get_x()
                    & p.get_y()=self.get_y()
     end;

let rec type ColorPoint :=
   Record  get_x(y): Unit → Int,
       get_color: Unit → Color
       equal: ColorPoint → Bool
   End;
let newColorPoint: Int×Int×Color → ColorPoint :=
  fun (x:Int, y:Int,color:Color).
  let state := record x:=x, y:=y, color:=color end
  in  rec self   .record
       get_x(y/color):=
                    fun(_:Unit). state.x(.y/.color),
       get_color:= fun(_:Unit). state.color,
       equal:=
              fun(p:ColorPoint).
                  p.get_x()=self.get_x()
                  & p.get_y()=self.get_y()
                  & p.get_color()=self.get_color()
 end;
```

[5]Informally, *rec ide. expr* evaluates *expr* binding *ide*, recursively, to *expr* itself (*rec self. record(self)*). *let rec ide := expr* means *let ide := rec ide. expr.*

Notice that a class definition is expanded into the definition of a record type and of a function which creates new records of that type (the *objects*). The fields of these records correspond just to the messages which can be sent to them, while the state of the object is codified in a local variable shared by these fields, called *state* above (actually this local variable is not needed to translate the toy language, while it is useful in general to model state encapsulation). To get the *x* coordinate of an object *aPoint* the expression *aPoint.get_x()* is used; the implicit argument of the *get_x* method, which would be *aPoint*, is not given explicitly, by writing *aPoint.get_x(aPoint)*, but is accessed by the *get_x* method using the variable *self* which recursively refers to the whole object, *aPoint* in this case. Notice that, intuitively, there is nothing really recursive in trivial functions like *get_x* or *equal* above; we will see that in our model this kind of recursive definitions is avoided.

The example above is a critical one, since *ColorPoint* should be a subtype of *Point*, since it is defined by inheritance from *Point* (in the weakly-typed source code), but it is not a subtype due to the covariant redefinition of the argument type of the equal method[6], and for this reason, in most strongly-typed object-oriented languages, the definition above would not be well-typed; the remaining part of this section is devoted to a better understanding of this problem.

Notice that in the above example the ability of defining classes incrementally (through inheritance), which we find in the source code, is lost in the translation: the type *ColorPoint* and the function *newColorPoint* are fully defined without exploiting the similarities with the preceding two definitions. In

[6]For subtyping, recursive types are equivalent to their infinite tree expansion; see [AmaCar90] for more details about subtyping and recursive types.

more powerful record models (see e.g. [Mit90]), inheritance is present, and is codified by using record concatenation operators, but other problems then arise.

2.4 *Inheritance and subtyping*

A subclass *SubC* inherits from *SuperC* when *SubC* is defined by describing how it differs from *SuperC*; a type *T* is a subtype of a type *U* if all the elements of *T* have also type *U*; so subtyping and inheritance are very different notions. However, in strongly-typed object-oriented languages, the type of a subclass is usually required to be a subtype of that of its superclasses, to assure that well-typed methods remain well-typed when are inherited. In fact a method in a superclass with element type *SuperObjType* is type-checked supposing *self:SuperObjType*; when the method is inherited in a subclass with element type *SubObjType*, it should be type-checked another time, under the assumption *self:SubObjType*. This second type-checking can be avoided if *SubObjType* is required to be a subtype of *SuperObjType*, since in this case *self:SubObjType* implies *self:SuperObjType*, which in turn implies the well-typing of the method.

While some important approaches to inheritance and subtyping support a weaker link between the two notions, notably [CanCooHil90] and [Mit90], this linking is a positive feature of object-oriented languages which leads to a better structuring of class definitions.

2.5 *Contravariance of method arguments in the record model*

In the record model an object type can be a subtype of another one only if the corresponding record types are in the same relation; so, when a record subtype is defined by modifying (by inheritance) a record supertype, the types of the fields which already exist in the supertype, can be only specialized in the subtype. Methods are encoded by functional record fields, and functions can be specialized only by generalizing, and not by specializing, the argument type (i.e. $T \to U \leq T' \to U$ only if $T' \leq T$). For this reason, in the record model, when a method type is redefined in a subclass, its argument type can be only generalized; this is a problem, since in many programming situation specialization would be needed. But contravariance is not just an accidental result of some combination of formal rules, which could be relaxed just by adopting different rules; it is really needed to obtain type-safety in the weakly-typed and record-based approaches, as exemplified below.

Example: due to the covariant specialization of the type of the *other* parameter of the *equal* function in the *ColorPoint* type, *ColorPoint* is not a subtype of *Point*, and if this relation holded, run-time errors could occur:

```
let Point(x,y: Int) := Class
  get_x/y() := x/y;
  equal(p:Point) := get_x(p)=get_x(self)
              & get_y(p)=get_y(self)
end

let ColorPoint(x,y: Int, color:Color)
  := Class isa Point
        get_color() := color;
        equal(other:ColorPoint) :=
            get_x(other) = get_x(self)
            & get_y(other) = get_y(self)
            & get_color(other) = get_color(self)
end

let one:=new[Point](1,1);
let oneBlue:=new[ColorPoint](1,1,Blue);
equal(oneBlue)(one);  ⇒  ????
```

If *ColorPoint* were a subtype of *Point*, the last application of *equal* would be allowed, the *ColorPoint* equality would be selected by late binding (think to *oneBlue.equal(One)* in the record model) and a *get_color* message would be sent to *one*, resulting in a run-time failure. So this example shows that the contravariance constraint cannot be relaxed in a type safe way without a reconsideration of the run-time semantics of the message passing mechanism, like the one studied in the next section.

3 The proposed type system

In this section we define a static strong type system for late-binding overloading. Late binding overloading allows message passing to be expressed, but is still more expressive; as a result, it allows, in a type safe context, both covariant and contravariant redefinition of method types.

Our overloading mechanism is defined by an operator which allows transforming a regular function in a one-branch overloaded function, an operator to extend an overloaded function by adding new branches to it, and finally by an operation of overloaded function application. Overloaded application selects a branch of the overloaded function, on the basis of the run-time type of a distinguished argument, and then applies that branch to the argument. In §3.1 we give the rules which specify how these constructs are typed, in the case of single inheritance, and which subtyping relation is defined on overloaded function types, and we discuss the compatibility of covariance and contravariance in our model. In §3.2 we generalize this discussion to the case of multiple inheritance, dealing with the conflict resolution rules at a general level. In §3.3 we exemplify our approach. In §3.4 we state some further observations.

3.1 The type rules of overloaded functions

In our model each branch of an overloaded function is associated with an "expected input type" T, which is just the input type of the branch seen as a non-overloaded function. If the run-time type of the argument is exactly one of the "expected input types", the corresponding branch is selected. Otherwise a *choice rule* is invoked, such that in any case the selected input type is a supertype of the run-time argument type; if the set of those *expected input types* which are supertypes of the run-time argument type has a minimum, that minimum is selected. We do not specify how choice rules are defined, but we define which sets of branches can be put together, with type rules which are "parametric" with respect to the way of specifying choice rules.

With single inheritance, for any run-time argument type, a minimum supertype always exists in the set of the expected input types, so that there is no need of specifying a choice rule. The absence of choice rules, which are also missing in approaches different from single inheritance, simplifies greatly the type rules and the notation, and we now address this. The more general setting, where the minimum supertype is not always defined, is sketched in the next section.

The type of an overloaded function with branches with expected input types A_1, A_2, and A_3, and result types, respectively, B_1, B_2 and B_3 is denoted as $\{A_1 \rightarrow B_1, A_2 \rightarrow B_2, A_3 \rightarrow B_3\}$ or (at the meta-level) as $\{A_i \rightarrow B_i\}_{i \in \{1,2,3\}}$.

The first rule below specifies which sets of types can be combined to form a well-formed overloaded type, and implicitly which sets of branches can be combined to form a well-formed overloaded function. Type rules should be read backward; for example the rule below can be read as: "(in the environment Γ) $\{T_i \rightarrow U_i\}_{i \in I}$ is a well formed type if a) for any i in I, T_i is a *choice_type* and U_i is a type, and if for any i,j in I if $T_i \leq T_j$ then $U_i \leq U_j$"; Γ is an environment collecting information about the free variables. Well-formed overloaded types are definedby the rule below simply as those types where all the component functional types are "mutually compatible". Mutual compatibility means that if the input types are in the subtyping relation, the output types are related in the same way; we will see in the next section that in case of multiple inheritance the compatibility condition is slightly stronger.

$$(\{\}\text{Form})\quad \frac{\forall^{7} i \in I.\ \Gamma \vdash T_i\ \text{choice\_type},\ \Gamma \vdash U_i\ \text{type} \qquad \forall i,j \in I\ \ \Gamma \vdash T_i \leq T_j \Rightarrow \Gamma \vdash U_i \leq U_j}{\Gamma \vdash \{T_i \rightarrow U_i\}_{i \in I}\ \text{type}}$$

The condition $\Gamma \vdash T_i$ *choice_type*, which is not formally specified here, models the fact that in real systems not every type is accepted as an expected input type; however, in our examples below we will

[7] This ∀ is a meta-level finite quantification

accept any type as a choice type. In object-oriented languages, only object types are accepted as choice types.

The result type of an overloaded function application is determined by applying the • *(choose)* operator to the set of the expected input types of the overloaded function and to the argument type: $\{T_i\}_{i\in I}\bullet A$ selects from the set of expected input types $\{T_i\}_{i\in I}$ the type corresponding to the type A. In the case of single inheritance, that type is the minimum supertype of A in $\{T_i\}_{i\in I}$, if it exists; $\{T_i\to U_i\}_{i\in I}$ returns U_i if $\{T_i\}_{i\in I}\bullet A$ returns T_i. The condition $\{T_i\}_{i\in I}$ *accepts* A is satisfied when in $\{T_i\}_{i\in I}$ there is a type which corresponds to A; in the case of single inheritance, this means simply that in $\{T_i\}_{i\in I}$ there is one supertype of A. The application of an overloaded function is denoted here as $f\bullet a$ to stress the fact that applying an overloaded function is different from applying a regular function, since it involves both a branch selection and a function application.

$$(\{\}\text{Elim})\quad \frac{\Gamma\vdash f:\{T_i\to U_i\}_{i\in I}\qquad \Gamma\vdash a:A \qquad \Gamma\vdash \{T_i\}_{i\in I}\text{ accepts }A}{\Gamma\vdash f\bullet(a):\{T_i\to U_i\}_{i\in I}\bullet A}$$

The same • notation is used at the type and at the value level just to indicate that the same rule is used to choose a type in $\{T_i\to U_i\}_{i\in I}$ at compile-time and to choose a branch in f at run-time. However, the branch of the type which is selected at compile-time could not correspond to the branch of the function which is selected at run-time. In fact we have shown in section 2.2 that the compile-time type of an expression is generally only a supertype of the run-time type of the corresponding value. The compatibility condition ($T_i \le T_j \Rightarrow \Gamma\vdash U_i \le U_j$) of the formation rule assures that in any case the type which is computed at compile-time for the overloaded application is a supertype of the actual run-time type of the result.

An overloaded function is defined by starting from a regular function and adding new type-branch pairs with the + operator; a simpler approach could be obtained by identifying regular functions with one-branch overloaded functions:

$$(\{\}\text{Intro})\quad \frac{\Gamma\vdash f:T\to U}{\Gamma\vdash \text{overload } f:\{T\to U\}}$$

$$(\{\}\text{Add})\quad \frac{\Gamma\vdash f:\{T_i\to U_i\}_{i\in I}\qquad \Gamma\vdash g:T_k\to U_k \qquad T_k\notin\{T_i\}_{i\in I}\qquad \Gamma\vdash \{T_i\to U_i\}_{i\in I\cup\{k\}}\text{ type}}{\Gamma\vdash f+g:\{T_i\to U_i\}_{i\in I\cup\{k\}}}$$

In general, $A\le B$ means that for any context $C[x]$, if any element of type B can substitute x in a type-safe way, then also any element of type A can be inserted in $C[x]$; this implies that if $T_{C[],B}$ *(resp.* $T_{C[],A}$*)* is the type of the expression obtained by putting an element of type B *(resp.* A*)* in that context, then $T_{C[],A}\le T_{C[],B}$. From this definition we have the following general rule (to be read backward, as usual):

$$(\{\}\le)\quad \frac{\forall^{8}A \text{ such that } \{V_j\}_{j\in J}\text{ accepts }A.\ \ \{T_i\}_{i\in I}\text{ accepts }A \text{ and } \Gamma\vdash \{T_i\to U_i\}_{i\in I}\bullet A\le\{V_j\to W_j\}_{j\in J}\bullet A}{\Gamma\vdash \{T_i\to U_i\}_{i\in I}\le\{V_j\to W_j\}_{j\in J}}$$

Even if this is not apparent, the usual contravariance rule for subtyping can be derived from the rules above. More formally, in Appendix B it is proved that, even in the general case:

$$\Gamma\vdash T'\le T,\ \Gamma\vdash U'\le U \ \Rightarrow\ \Gamma\vdash \{T\to U'\}\le\{T'\to U\}$$

This observation helps to clarify the fact that our system is a conservative extension of the traditional type systems for subtyping. In our system covariance and contravariance both exist with two neatly different rôles. Covariance is the compatibility constraint to be satisfied when different branches of an overloaded function are put together, used in the rule of type formation, while contravariance is the condition for subtyping, used only in subtyping rules, with no contrast between the two notions. At the end of section 3.3 we will show that the covariance condition does not prevent the contravariant specialization of method argument types.

3.2 Constraints on the set of types of an overloaded function in case of multiple inheritance

In this section we define at a general level which sets of types can be combined to form a well-formed overloaded type, in the general case of multiple inheritance, when choice rules are necessary to select one specific supertype of the run-time argument among the expected input types. This study is general in the sense that we do not commit to any specific mechanism for specifying choice rules.

For generality we suppose that each overloaded function contains its choice rule, which is also a part of its type; this is a heavy formalization which is made here to carry on this discussion at the highest level of generality; by selecting a specific way of defining this choice rule we can model, and compare, some of the known approaches to multiple inheritance. So the type of an overloaded function with a choice rule r is now denoted by $\{A\to T,\ldots,B\to U\}_r$. Instead of detailing how choice rules can be specified, we formalize some constraints on their behaviour, to preserve the property that the compile-time type of any expression is a supertype of the possible run-time types of its values. Since no operation fails if the run-time type of its argument is a subtype of the expected type, this property implies that typed expressions never fail, i.e. that our type system is *strong*.

The well-formedness conditions for an overloaded type are the following ones:

Notation:

$\{T_i\}_{i\in I,s}\bullet A$ is the type selected for A by the choice rule s among $\{T_i\}_{i\in I}$

$\{T_i\}_{i\in I,s}$ *accepts* A
means "$\{T_i\}_{i\in I,s}\bullet A$ is well defined"

$\{T_i\to T_i\}_{i\in I,s}\bullet A=U_j$
is the same as $\{T_i\}_{i\in I,s}\bullet A=T_j$

i (internal choice) $\forall T.\ \{T_i\}_{i\in I,s}\bullet A\in\{T_i\}_{i\in I}$

ii (nearest choice) $\forall j\in I\ A\le T_j\le\{T_i\}_{i\in I,s}\bullet A \Rightarrow T_j=\{T_i\}_{i\in I,s}\bullet A$

iii (supertype choice) $\forall A.\ \Gamma\vdash A\le\{T_i\}_{i\in I,s}\bullet A$

iv (downward closure)
$\forall A,B.\ \Gamma\vdash A\le B,\ \{T_i\}_{i\in I,s}$ accepts $B \Rightarrow \{T_i\}_{i\in I,s}$ accepts B

v_s[8] (choice covariance)
$\forall A,B.\ \Gamma\vdash A\le B \Rightarrow \Gamma\vdash\{T_i\to T_i\}_{i\in I,s}\bullet A\le\{T_i\to T_i\}_{i\in I,s}\bullet B$

Conditions i-iii do not need any explanation. Conditions ii specifies that no type in $\{T_i\}_{i\in I}$ can be "nearer" to A than the selected type; it implies that, if a minimum supertype of T exists among $\{T_i\}_{i\in I}$, that minimum is selected. To understand the last two constraints, suppose that A<B (i.e. A≤B and A≠B), that T is a type unrelated to U and that $f{:}\{A\to T, B\to U\}_r$. Suppose that f is applied as an overloaded function to the value of an expression b whose compile-time type is B, and whose run-time type is A (e.g. b is a formal parameter of type B which is bound to an actual parameter created with type A). Since the compile-time type of b is B, then the compile-time type of $f\bullet(b)$ is U. But when f is applied to a value b with run-time type A, the late binding mechanism selects, among the f branches, the piece of code with type $A\to T$, so that the application returns a value with run-time type T, unrelated with its compile-time type U. The rule v_s prevents this kind of problem by forcing the type T to be a subtype of the output type U computed by the compiler, while rule *iv* enforces the condition that for any possible run-time type A corresponding to a compile-time type B the corresponding branch selection is well defined.

To express condition v_s in a more useful way, we define the following preorder on a set τ of expected input types with respect to a choice rule s ($T\sqsubseteq_{s,\tau}U$: read T can be chosen for U by s in τ):

[8]The s subscript stands for "strong covariance" since we have also a weaker form of this rule, v_w.

Def.: (subtype choice preorder)

$$T \sqsubseteq_{s,\tau} U \quad \Leftrightarrow_{def} \quad \exists A \leq B: \tau \bullet_s A = T,\ \tau \bullet_s B = U$$

Lemma: $\sqsubseteq_{s,\tau}$ extends $\leq$ on τ; $\sqsubseteq_{s,\tau}$ is reflexive and transitive

Condition v_s can be now expressed as:

v_s (choice covariance for $\{T_i \to U_i\}_{i \in I,s}$):

$$\forall i,j \in I. \quad \Gamma \vdash T_i \sqsubseteq_{s,\{T_i\}_{i\in I}} T_j \Rightarrow \Gamma \vdash U_i \leq U_j$$

In systems without multiple inheritance, i-iv are satisfied by the choice rule selecting the minimum expected input supertype of the argument type, and $v_{s(trong)}$ is equivalent to the following requirement, which is weaker than the one needed in the general case:

v_w (type covariance for $s, \{T_i \to U_i\}_{i \in I,s}$):

$$\forall i,j \in I. \quad \Gamma \vdash T_i \leq T_j \qquad \Rightarrow \Gamma \vdash U_i \leq U_j$$

The five conditions above can be exploited to discuss, with some generality, how choice rules can be specified in the case of multiple inheritance. We do not enter in this discussion, but simply list some of the possibilities:

- When a new choice-type with multiple ancestors is defined, all the overloaded functions defined for a non-empty set of supertypes of this type without a minimum have to be explicitly overloaded for this type. With this constraint condition *ii* specifies completely how the choice is performed, so that no choice rule has to be specified.
- When a new choice-type with multiple ancestors is defined, a linear order is defined on its immediate ancestors. $\tau \bullet A$ always chooses the minimum supertype of A with respect to this order.
- A global linear order is defined on all the choice types; $\tau \bullet A$ always chooses the minimum supertype of A with respect to this order.
- Single inheritance.

By specializing our constraints to the four cases above we obtain the rules specifying what must be checked whenever a new choice type is defined. In this way we can compare in a unique setting different approaches to multiple inheritance, which is more difficult in the record model.

3.3 An example

We will now exemplify the proposed approach, using just single inheritance for simplicity. For simplicity we suppose that every type can be used as an expected input type, while restricting these types, for example to products of named data types or of abstract data types, would be more usual (named data types are used in [AlbGheOrs91]). We consider again the ColorPoint example of §2.5:

```
let type P := Record x: Int, y: Int End;
let eq1: {P×P→Bool} :=
  overload (fun (p,q:P). p.x=q.x & p.y=q.y;

let type CP
        := Record x: Int, y: Int, color: Color End;

let eq1: {P×P→Bool, CP×CP→Bool} :=
   eq1
   + (fun (p,q:CP). eq1•(p,q) & p.color=q.color);

let eq2: {P×P→Bool, CP×CP→Bool,
          P×CP→Bool, CP×P→Bool} :=
   eq1 + (fun (p:P,q:CP).false)
   + (fun (p:CP,q:P).false);

let OneRed(OneBlue) :=
   record x:=1,y:=1,color:=Red(Blue) end;
let One := record x:=1,y:=1 end;

eq1(OneBlue,OneRed)  → false
  % OneBlue and OneRed are compared as colored
eq1(OneBlue,One)   → true
  % We can safely compare Points and ColorPoints
eq1(One,OneBlue)   → true
  % eq1 compares them as Points (One=OneBlue)
eq2(One,OneBlue)   → false
  % eq2 does not equate colored and uncol. points
```

eq1 is a safe version of the old *equal* method; it compares a point with a colored point as if they were both points, and never fails, while in §2.5 we have seen that *eq1(OneBlue,One)* would fail, if allowed, in the traditional model. *eq2* shows that we can decide how a point is compared with a colored point (consider them always different (*eq2*), or compare them as uncolored (*eq1*), or consider an uncolored point as a transparent one...). So our general-purpose overloading is both safer and more expressive than the weakly-typed object-oriented overloading, and more expressive than the record model, where *eq1* was ill-typed and *eq2* not expressible.

The possibility of selecting a branch on the basis of more then one parameter is the origin of the expressive power of our approach; the outermost $\rightarrow$ in each element of the overloaded type specifies the examined parameters. For example, consider the two following overloaded types:

$$\{(A\times A)\rightarrow T, (B\times B)\rightarrow T\}_r$$
$$\{A\rightarrow(A\rightarrow T), B\rightarrow(B\rightarrow T)\}_r$$

The first case is the type of an overloaded function which selects the branch on the basis of both arguments, in the second case only the first argument is considered for branch selection. Notice that if $A\leq B$ and B not $\leq$ A, only the first type is well formed; so in the second case, where choice is restricted to just one argument, our approach has the same contravariant behaviour of the traditional one. On the other hand the following types are both well formed:

$$\{(A\times A)\rightarrow T, (B\times B)\rightarrow T\}_r$$
$$\{(A\times B)\rightarrow T, (B\times A)\rightarrow T\}_r$$

The first one is the type of a two-parameters functio, where the second parameter is specialized in a covariant way when the first parameter is specialized to type B. The second one is the type of a function where the second parameter is generalized in a contravariant way to type A when the first parameter is specialized to type B. Both types are well-formed, showing that in our approach both covariant and contravariant argument type specialization are allowed.

3.4 Final considerations

Our approach to overloading is very general, but many instances of that approach have unpleasant properties. In object-oriented languages only object types, i.e. named user defined abstract data types, can be used as choice type. But user defined types can acquire new subtypes, when new types are defined, so that well-formed types can lose well-formedness. For example, suppose that two types *Worker* and *Student* are defined, and that an overloaded function *code* of type $\{Worker\rightarrow Int, Student\rightarrow String\}_r$ is defined too. The type of *code* is well formed until *Worker* and *Student* have no common subtype. As soon as such a common subtype *TeachingFellow* is defined, the well-formedness of the type of the *code* function would depend on the behaviour of its *r* choice method. We can distinguish three alternative ways to deal with this problem:

- avoiding user defined types: this is an elegant approach, but cannot be used to understand object-oriented languages
- fixing the set of user defined types: also in this case we avoid the problem of evolving type hierarchies, but this approach is effective only to study the type-system of object-oriented languages which are implementated in a compilative way, where the program is read once to collect the type hierarchy and a second time to type-check the methods, and is not useful to study languages which are suitable for an interactive use
- allowing evolving hierarchies: this is the general and most interesting case.

Our general approach encopasses all the three cases above, but is expecially useful to deal with the last case, which is the most interesting and important one.

4 Related researches

4.1 The Canning Cook Mitchell Hill Olthoff approach

Method covariance has been obtained by W. Cook, J. Mitchell and others in the important case when all the arguments have the same type (like in the comparison examples above) [CooHilCan90]

[Mit90]. They use the record model, but they allow the definition of functions which operate not only on all the subtypes of a type, but on a wider set, exploiting their notion of F-bounded quantification [CanCooHilMitOlt89]. Then a subclass can inherit methods even if its type is not a subtype of the superclass, at least in the limited case cited in the second line; so the covariance-contravariance dilemma is faced by breaking the subtyping-inheritance link.

The Cook-Canning-Hill and Mitchell approaches (which are different but share these basic ideas) are much more complex than traditional approaches, since they rely on F-bounded quantification. Moreover, inheritance, which is realized in our model by the overloading "+" operator, is realized in these approaches by using record concatenation, which is not compatible with the standard record subtyping rules. To allow record concatenation, in the Cook-Canning-Hill approach values of a subtype cannot be substituted for values of supertypes everywhere, but only as parameters of functions where this substitution is explicitly allowed; this is a very limited use of the subtyping mechanism. On the other hand the concatenation problem is solved in the Mitchell approach by retaining the usual notion of subtyping, but by exploiting a more complex type system, where record types contain not only some of the labels which *must* be found in the record but also some of the labels which *cannot* be there.

Besides this, these approaches solve just a specific, even if important, case of covariance, and in this case they do not have full subtyping but only inheritance plus that weaker notion of subtyping which is expressed by F-bounded quantification. But this comparison is not fair, since solving a special case of the covariance-contravariance conflict is not the basic aim of the Cook-Canning-Hill and Mitchell approaches, whose fundamental contribution is a clean definition of the operation of inheritance in the record-based model and of its relation with subtyping and with record concatenation.

4.2 Overloading and conjunctive types

A type can be read as a predicate satisfied by its values, e.g., "$f{:}A \rightarrow B$" means: f is a function which, receiving a parameter satisfying A, does not raise run-time type failures, and its result satisfies B. In this setting, subtyping is implication: $A \leq B$ means that every value satisfying A satisfies also B. Conjunctive types are types which represent the conjunction of the predicates associated to their component types; they have been introduced by Coppo, and have been studied in the context of programming languages by Reynolds (see e.g. [Cop81] [Rey77]). Record types are a kind of conjunctive type: "*r:Record l:A End*" means "*r.l* returns a value satisfying A", and "*r:Record* $l_1{:}A_1,\ldots,l_n{:}A_n$ *end*" is a conjunctive type meaning "*r:Record* $l_1{:}A_1$ *end* $\wedge\ldots\wedge$ *r:Record* $l_n{:}A_n$ *end*". Conjunctive types are strictly related to non-trivial subtypes: $(P \wedge Q) \Rightarrow P$ becomes, in the language of types, $\{A \wedge B\} \leq \{A\}$. Our overloaded functions can be read as another special case of conjunctive types, since $f{:}\{A \rightarrow B,\ C \rightarrow D\}$ means "$f{:}A \rightarrow B \wedge f{:}C \rightarrow D$". This link is important mainly from a semantic point of view, since conjunctive type possess a clear semantics based on intersection (or on indexed products, see [BruLon90]), which can be transferred to our overloaded function types.

6 Type safe dynamic extension and encapsulation

Our model supports directly inheritance, overloading and late binding, but does not support dynamic extensibility and encapsulation, which are the other two features of the object-oriented approach. In this section we give a hint about the encoding of these two features.

Dynamic extensibility must be simulated in a strongly-typed language with static binding by exploiting updatable references; a method is modelled as an updatable reference to an overloaded function, and dynamic extension is modelled by updating that reference. This is possible, in a strongly-typed context, since dynamic extension always extends an overloaded function with type $\{T_i \rightarrow U_i\}_{i \in I}$ to an overloaded function with type $\{T_i \rightarrow U_i\}_{i \in J}$, such that

$$\{T_i \rightarrow U_i\}_{i \in J} \leq \{T_i \rightarrow U_i\}_{i \in I}.$$

Encapsulation, in our approach, is truly orthogonal to message passing; this is one important result, since breaking down the features of object-oriented languages into orthogonal atomic notions is among the aims of this research. In our approach, encapsulation can be obtained by using any of the well known mechanisms of ADT's, existential types or modules (see [MitPlo85], [CarWeg85]).

Referring to our example, we can encapsulate the type *Point* in a module exporting an abstract version *AbsPoint* of it, together with a creation function and a selector *equal: AbsPoint→Int*. In another module we can do the same with the type *CPoint*, and finally we can collect the two *equal* functions into one overloaded function. This is not possible in the record-based approach (refer to the translation in §2.3), since in that case, once *equal* has been transformed, by a general purpose encapsulation mechanism, from a record field name to a function name, then it is no more possible to overload *equal*. For this reason, in the record-based model, encapsulation cannot be considered orthogonal to message passing.

This is just a very brief discussion about encapsulation; to be more complete we should distinguish encapsulation at least into state encapsulation and method implementation encapsulation, which are actually supported in the record model, and ADT-like encapsulation, i.e. the possibility of specifying that an object type is different from any other, which, in the record model, suffers from the problem highlighted above.

7 Conclusions and directions for future work

We have defined a static and strong type system for late-binding overloaded functions. It can be used to give a strongly-typed model for object-oriented languages which is strictly more expressive than the classical record-based model, and even of the traditional untyped model. We have discussed the origins of the well-known conflict contravariance-covariance and have formally shown that the two notions are not mutually exclusive.

We have left many details to be verified. We should show that, e.g., a notion of state, the "super" operator of object-oriented languages, and mutual recursion among methods, could be all added without problems in our approach. The extension of our mechanism to languages offering parametric polymorphism and existential types, like Fun [CarWeg85], raises many interesting issues.

We should prove decidability of type checking for our type system. The fact that subtyping in this system is not irreflexive, i.e. there exists $U \neq V$ such that $U \leq V$ and $V \leq U$, could constitute a problem [CurGhe91]. It can be solved by introducing a calculus at the type level equating all and only the pairs of types which are mutually subtype-related.

We should define formally operational and denotational semantics for our operators, and then we should give a formal proof of the property of strong typing for our system. The kernel of this proof is the property run-time type≤compile-time type, discussed in the paper, while the remaining part of the work should be routine.

Finally we could design a little object-oriented language around our type system to prove its usability.

References

[AmaCar90]: Amadio R., and L. Cardelli, "*Subtyping Recursive Types*", DEC-SRC technical report, Palo Alto DEC System Research Center, Ca., USA, 1990.

[AlbGheOccOrs88]: Albano A., G. Ghelli, E. Occhiuto and R. Orsini, "*Galileo Reference Manual Version 2.0*", Servizio Editoriale Università di Pisa, Italy, 1988.

[AlbGheOrs91]: Albano A., G. Ghelli and R. Orsini, "*Object and Classes for a Database Programming Language*", Tech. Rep. PFSICP-CNR 5/24, Rome, 1991.

[BruLon90]: Bruce K.B. and G. Longo, "*A Modest Model of Records, Inheritance and Bounded Quantification*", *Information & Computation*, 87 (1/2), 1990.

[CanHilOlt88]: Canning P.S., W.L. Hill and W. Olthoff, *"A kernel language for object-oriented programming"*, TR. STL-88-21, HP Labs, 1988.

[CanCooHilMitOlt89]: Canning P.S., W.R. Cook, W.L. Hill, J.C. Mitchell and W. Olthoff, "F-bounded polymorphism for object-oriented programming", *Proc. of Conf. on Functional Progr. Languages and Comp. Arch.*, 1989.

[Car88]: Cardelli L., "A Semantics of Multiple Inheritance", *Information & Computation*, 76 (2/3), 1988.

[CarMit89]: Cardelli L. and J.C. Mitchell, "Operations on Records", *Proc. Fifth Intl. Conf. on Mathematical Foundation of Programming Semantics, Tulane Univ.*, New Orleans, 1989.

[CarWeg85]: Cardelli L. and P. Wegner, "On understanding types, data abstraction and polymorphism", *ACM Computing Surveys*, 17 (4), 1985.

[CooHilCan90]: Cook W.R., W.L. Hill and P.S. Canning, "Inheritance is not subtyping", *Proc. of POPL '90*, 1990.

[Cop81]: Coppo M., M. Dezani-Ciancaglini and B. Venneri, "Functional Character of Solvable Terms", *Z. Math. Logik Grundlag Math.*, 27, 1981.

[CurGhe91]: Curien P.L. and G. Ghelli, "Coherence of Subsumption", *Mathematical Foundations of Computer Science*, to appear.

[DanTom88]: Danforth, S. and C. Tomlinson, "Type Theories and Object-Oriented Programming", *ACM Surveys*, 20 (1), 1988.

[Ghe91]: Ghelli G., "Modelling features of object-oriented languages in second order functional languages with subtypes", in *Foundations of Object-Oriented Languages* (G. Rozenberg ed.), Springer-Verlag, Berlin, 1991.

[Mit90]: Mitchell, J., "Toward a typed foundation for method specialization and inheritance", *Proc. of POPL '90*, 1990.

[MitPlo85]: Mitchell J.C. and G.D. Plotkin, "Abstract Types Have Existential Types", *Proc. of POPL '85*, 1985.

[Red88]: Reddy, U.S., "Objects as Closures: Abstract semantics of object-oriented languages", *Proc. of ACM Conf. on Lisp and Functional Programming '88*, 1988.

[Rey77]: Reynolds J.C., "Conjunctive Types and Algol-like Languages", *Proc. of LICS 87*, 1987.

[Wan89]: Wand M., "Type inference for record concatenation and multiple inheritance", *Proc of LICS '89*, pp. 92-97, Asilomar, CA, 1989.

Appendix A: The type rules

Syntax

$$\begin{array}{l} A ::= t \mid A \to A \mid A \times A' \\ \quad \mid \textbf{Rcd}\ l_1{:}A_1,\ldots,l_n{:}A_n\ \textbf{End} \mid \{A,\ldots,A\} \end{array}$$

$$\begin{array}{l} a ::= x \mid \textbf{fun}(x_1{:}A_1,\ldots,x_n{:}A_n)\textbf{.}a \mid a(a_1,\ldots,a_n) \\ \quad \mid \textbf{rcd}\ l_1{:=}a_1,\ldots,l_n{:=}a_n\ \textbf{end} \mid a.l \\ \quad \mid \textbf{overload}\ a \mid a + a \mid a\bullet(a_1,\ldots,a_n) \end{array}$$

Environments

(∅env) $\quad ()\ \mathrm{env}$

(≤env) $\dfrac{\Gamma\ \mathrm{env} \quad \Gamma \vdash A\ \mathrm{type}}{\Gamma,\ t{\le}A \quad \mathrm{env}}$

(: env) $\dfrac{\Gamma\ \mathrm{env} \quad \Gamma \vdash A\ \mathrm{type}}{\Gamma,\ x{:}A \quad \mathrm{env}}$

Types

(RcdForm) $\dfrac{\forall i.\ \Gamma \vdash A_i \quad \mathrm{type}}{\Gamma \vdash \mathrm{Rcd}\ l_1{:}A_1,\ldots,l_n{:}A_n\ \mathrm{End}\ \ \mathrm{type}}$

(→/×Form) $\dfrac{\Gamma \vdash A\ \ \mathrm{type}\ \Gamma \vdash B\ \ \mathrm{type}}{\Gamma \vdash A \to/\!\times B \quad \mathrm{type}}$

(ch.Form)

$$\begin{array}{ll} & \forall i \in I \qquad\qquad \Gamma \vdash T_i \ \ \mathrm{choice\_type} \\ \mathrm{i} & \forall A \in \mathrm{acc.}\{T_i\}_{i\in I,s}.\{T_i\}_{i\in I,s}\bullet A \in \{T_i\}_{i\in I} \\ \mathrm{ii} & \forall A \in \mathrm{acc.}\{T_i\}_{i\in I,s},\ \forall j \in I. \\ & \quad A{\le}T_j{\le}\{T_i\}_{i\in I,s}\bullet A. \Rightarrow\ T_j{=}\{T_i\}_{i\in I,s}\bullet A \\ \mathrm{iii} & \forall A \in \mathrm{acc.}\{T_i\}_{i\in I,s}.\ \Gamma \vdash A \le \{T_i\}_{i\in I,s}\bullet A \\ \mathrm{iv} & \forall B \in \mathrm{acc.}\{T_i\}_{i\in I,s}.\ \forall A \text{ such that } \Gamma \vdash A{\le}B. \\ & \qquad\qquad\qquad A \in \mathrm{acc.}\{T_i\}_{i\in I,s} \\ \hline & \Gamma \vdash s\ \mathrm{choice\_for}\ \{T_i\}_{i\in I} \end{array}$$

({}Form)

$$\begin{array}{ll} & \forall i,j \in I. \quad \Gamma \vdash T_i \ne T_j \\ & \forall i \in I. \quad \Gamma \vdash U_i\ \mathrm{type} \quad \Gamma \vdash s\ \mathrm{choice\_for}\ \{T_i\}_{i\in I} \\ \mathrm{v} & \forall i,j \in I \text{ s.t. } \Gamma \vdash T_i \subsetneq_{s,\{T_i\}_{i\in I}} T_j.\ \Gamma \vdash U_i \le U_j \\ \hline & \Gamma \vdash \{T_i \to U_i\}_{i\in I,s}\ \ \mathrm{type} \end{array}$$

Subtypes

(Id≤) $\dfrac{\Gamma \vdash A\ \mathrm{type}}{\Gamma \vdash A{\le}A}$

(Trans≤) $\dfrac{\Gamma \vdash A{\le}B \quad \Gamma \vdash B{\le}C}{\Gamma \vdash A{\le}C}$

(×≤) $\dfrac{\Gamma \vdash A \le A' \ \Gamma \vdash B \le B'}{\Gamma \vdash A{\times}B \le A'{\times}B'}$

(Abs≤) $\dfrac{\Gamma, t{\le}A, \Gamma'\ \mathrm{env} \quad t \text{ not in } \Gamma'^{9}}{\Gamma \vdash t{\le}A}$

(→≤) $\dfrac{\Gamma \vdash A{\le}A' \ \ \Gamma \vdash B{\le}B'}{\Gamma \vdash A' \to B \le A \to B'}$

(Record ≤)

$$\begin{array}{l} \forall i{:}1,\ldots,n\ \Gamma \vdash A_i{\le}B_i \\ \forall i{:}n{+}1,\ldots,n{+}m\ \Gamma \vdash A_i\ \mathrm{type} \\ \hline \Gamma \vdash \mathrm{Rcd}\ l_1{:}A_1,\ldots,l_n{:}A_n,m_1{:}A_{n+1},\ldots,l_m{:}A_{n+m}\ \mathrm{End} \\ \qquad\qquad\qquad \le \mathrm{Rcd}\ l_i{:}B_1,\ldots,l_n{:}B_n\ \mathrm{End} \end{array}$$

({}≤) $\forall A \in \mathrm{accepted}\{V_j\}_{j\in J,r}$ then

$$\begin{array}{l} \quad A \in \mathrm{acc.}\{T_i\}_{i\in I,s} \ \text{ and} \\ \Gamma \vdash \{T_i \to U_i\}_{i\in I,s}\bullet A \le \{V_j \to W_j\}_{j\in J,r}\bullet A \\ \hline \Gamma \vdash \{T_i \to U_i\}_{i\in I,s} \le \{V_j \to W_j\}_{j\in J,r} \end{array}$$

Expressions:

(Var:) $\dfrac{\Gamma, x{:}A,\ \Gamma'\ \mathrm{env} \quad x \text{ is not in } \Gamma'}{\Gamma, x{:}A,\ \Gamma' \vdash x{:}\ A}$

(Subsump) $\Gamma\ \vdash a{:}\ A \quad \Gamma \vdash A \le B$

[9]A variable t (x) is in Γ if it is present in a left hand side of an element t ≤ A (x : A) of the environment.

$\Gamma \vdash a: B$

(→ Intro) $$\frac{\Gamma, x_1{:}A_1,\ldots,x_n{:}A_n \vdash b: B}{\Gamma \vdash \mathrm{fun}(x_1{:}A_1,\ldots,x_n{:}A_n).a: A_1\times\ldots\times A_n \rightarrow B}$$

(→ Elim) $$\frac{\Gamma \vdash f: A_1\times\ldots\times A_n \rightarrow B \quad \Gamma \vdash a_i: A_i}{\Gamma \vdash f(a_1,\ldots,a_n): B}$$

(RcdIntro) $$\frac{\forall i{:}1,\ldots,n \quad \Gamma \vdash a_i{:}A_i}{\Gamma \vdash \mathrm{rcd}\ l_1{:=}a_1,\ldots,l_n{:=}a_n\ \mathrm{end}: \mathrm{Rcd}\ l_1{:}A_1,\ldots,l_n{:}A_n\ \mathrm{End}}$$

(RcdElim) $$\frac{\Gamma \vdash r: \mathrm{Rcd}\ l_1{:}A_1,\ldots,l_n{:}A_n\ \mathrm{End}}{\Gamma \vdash r.l_i : A_i}$$

({}Intro) $$\frac{\Gamma \vdash f{:}T \rightarrow U \quad \Gamma \vdash s\ \mathrm{choice\_for}\ \{T\}}{\Gamma \vdash \mathrm{overload}_s\ f: \{T \rightarrow U\}_s}$$

({}Add) $$\frac{\Gamma \vdash f: \{T_i \rightarrow U_i\}_{i\in I,r} \quad \Gamma \vdash g: T_k \rightarrow U_k \quad \Gamma \vdash \{T_i \rightarrow U_i\}_{i\in I\cup\{k\},s}\ \mathrm{type}}{\Gamma \vdash f +_s g{:}\{T_i \rightarrow U_i\}_{i\in I\cup\{k\},s}}$$

({}Elim) $$\frac{\Gamma \vdash f{:}\{T_i \rightarrow U_i\}_{i\in I,s} \quad \Gamma \vdash a{:}A \quad \Gamma \vdash \{T_i\}_{i\in I,s}\ \mathrm{accepts}\ A}{\Gamma \vdash f\bullet_s(a){:}\{T_i \rightarrow U_i\}_{i\in I,s}\bullet A}$$

Appendix B: Compatibility of covariance and contravariance

Suppose that:

$\Gamma \vdash T' \leq T \quad \Gamma \vdash U' \leq U$

$\Gamma \vdash \{T' \rightarrow U\}_r\ \mathrm{type}\ \Gamma \vdash \{T \rightarrow U'\}_s\ \mathrm{type}$

We want to prove that :

$\Gamma \vdash \{T \rightarrow U'\}_s \leq \{T' \rightarrow U\}_r$

i.e. that:

$\forall A \in \mathrm{accepted}\{T'\}_r$

$A \in \mathrm{accepted}\{T\}_s,\ \{T \rightarrow U'\}_s\bullet A \leq \{T' \rightarrow U\}_r\bullet A$

Proof:

a)	Hyp.:	$A \in \mathrm{accepted}\{T'\}_r$
b)	$\Rightarrow_{i\ (\text{only } T' \in \{T'\})}$	$\{T'\}_r\bullet A = T'$
c)	$\Rightarrow$	$\{T' \rightarrow U\}_r\bullet A = U$
	$\Rightarrow_{iii,b)}$	$A \leq T'$
	$\Rightarrow_{T' \leq T}$	$A \leq T$
	$\Rightarrow_{T \geq A}$	$\min\{U \geq A \mid U \in \{T\}\} = \min\{T\} = T$
d)	$\Rightarrow_{ii}$	$\{T\}_s\bullet A = T$
e)	$\Rightarrow$	$\{T \rightarrow U'\}_s\bullet A = U'$
f)	$\Rightarrow_{d)}$	$A \in \mathrm{accepted}\{T\}_s,$
g)	$\Rightarrow_{e),\ U' \leq U}$	$\Gamma \vdash \{T \rightarrow U'\}_s\bullet A \leq \{T' \rightarrow U\}_r\bullet A$

a) ⇒ f,g):

$$\frac{\forall A \in \mathrm{accepted}\{T'\}_r \quad A \in \mathrm{acc.}\{T\}_s \text{ and } \Gamma \vdash \{T \rightarrow U'\}_s\bullet A \leq \{T' \rightarrow U\}_r\bullet A}{\Gamma \vdash \{T \rightarrow U'\}_s \leq \{T' \rightarrow U\}_r}$$