Università
Ca'Foscari
Venezia

Bachelor's Degree Programme in
Informatics - Curriculum Technologies and Information Science
Ministerial Degree Code L-31 (Computer Science)

Bachelor Thesis

State of the Art of Extensible Records in Functional Languages

Supervisor
Prof. Spanò Alvise

Student
Fattori Giorgia
Identification Number 880224

Academic Year
2021-2022

Contents

0.1 Abstract

In this study we are going to explore the state of the art in scientific literature concerning functional programming languages. We will focus especially on an aspect of the ML type systems, records and the various proposals made in order to enhance their expressivity, for example extensible records, lightweight records, row types, etc.. The use of the row type on a record allows us to obtain polymorphism without the need to introduce a hierarchy based relationship between types (subtyping).Unification is used instead, so that is easier to integrate it with the ML type inference and its parametric polymorphism. Moreover, a polymorphism supporting record system allows the encoding of object-oriented programming. We will do a comparison between the most powerful record systems, the classic object-oriented programming offered by mainstream languages and other advanced object systems offered by functional languages such as OCaml, F#, Haskell and Scala.

0.2 Introduction

In order to understand this paper, only general programming knowledge is required as everything concerning the main topic will be described and explained from the beginning in quite simple terms.
For the ones who are already familiar with Record Types and their usage, it is suggested to start from the subchapter 1.3, or to give a quick read to the first part just to understand the contests.

What is a record type? Why should we use it? And why should we not?
Record types unfortunately are not quite used in mainstream software programming, indeed the first result that pops up on the screen when searching the word "record type" is related to database data structure record.
Although on a concept level the two are basically the same thing, a unit that stores different types of data within itself, the record type in programming is a type of its own, with its own set of characteristics methods and procedures, proof is that both objects and record types in programming languages are stored as a record structure when inserted into a database.
If objects and record are two different types but at their core are the same data structure, is it worth it to treat them differently? Why should we prefer one over the other? These questions will be soon answered, but first in order to make these comparisons it is necessary to take a look at one of records main features, extensibility.
Extensibility can be seen as the record-equivalent to the object inheritance, respectively they offer *Row polymorphysm* and *Subtype polymorphism*.

Extensible records are still in a developing phase both in theory and in practice, so it is only possible to explain what is known about them so far. This means that in the future to really understand their nature, it could be necessary to consult other resources beside this paper. For reference the extensible record types in practice that are treated in this paper are the ones implemented in ML-programming.

Chapter 1

Introduction to Records

1.1 What is a record type

A record type (considering the Damas–Hindley–Milner type system) is a kind of data structure that stores additional data, but it is seen as a whole unit of its own. The components of a record type are pairs made up of two elements: a variable and its associated value.
Record types are often compared to tuples, as both are collections of values. They differ in two aspects, in record types values are associated to a label (or variable). A tuple, on the other hand, collects its values without associating them to any variable, therefore if two tuples have the same values but in a different order, they are considered as two distinct entities that are not compatible with each other.
As for the latter, a clearer explanation is given by the following example: let us have

tuple1 = ('a,'b) tuple2 = ('b,'a)
'a,'b types.

If we define an method with parameter a tuple ('a,'b), the input ('b,'a) will not be valid, as they are considered as two different types by the type system.
Another benefit of records is that they can be far more descriptive compared to tuples, which makes it easier to understand their content. Labelling content also helps to get rid of ambiguity.

record::(age=12, month=10, name=Clarissa)
tuple (12,10,Clarissa)

(By watching only the tuple, we would not be able to tell if Clarissa is 12 and born in October or 10 and born in December.)

Record types, thanks to these associations, can be compared only by their content without caring about the order of its elements. A sequence of variables associated to its corresponding label is called a row.
Hence, we define a record as:

$$record ::= \{l_1 : v_1, l_2 : v_2, \ldots, l_n : v_n, \rho\}$$

where l_m is a literal label (variable) and v_m is a value with type t_m , Implicitly, on type level a record can also be defined by this formula:

$$record ::= \{l_1 : t_1, l_2 : t_2, \ldots, l_n : t_n, \rho\}$$

which represents every label with its associated type. A general t_m can be different from every other t_m in the same record. The set of admissible types t_m is τ and it's described in the table below.

$\tau ::=$
— C native Type
— $\tau \rightarrow \tau$ function type
— α generic (polymorphic) type

ρ is a row, of row type (and row kind), and it is defined as a list of label-type pairs not explicitly specified (a row can be empty as well).
Understanding ρ and its role is fundamental to understand Record types.
Row is indeed the key element that allows records to be polymorphic, since the content of ρ is not specified, it can technically contain whatever sequence of labelled values one can imagine.
A row can be substituted by a whole record, record parts, or nothing. Due to the uncertainty of its content and the possibility of replacing ρ with arbitrary values, it can be stated that ρ is polymorphic. Row polymorphism is defined as a kind of polymorphism that features the use of polymorphism on row types. Not only row types but also polymorphic variants use row polymorphism.

Record type properties and their use vary greatly from a theoretical level to a practical application level, especially when comparing their implementation in different programming languages.

1.2 In practice

The thing that differentiates the most a record type in the various programming languages are features that allow to access, modify, manipulate or handle a record in a more flexible way, unlike the traditional syntax that can be otherwise easily simulated with objects.
For example, one of the main features that distinguishes a record is the possibility to use a light-weight declaration in the code. In other words, it is not necessary to declare beforehand the record name, field-name, and field-type of each member of the record
Sometimes this kinds of records are also referred as anonymous records.

```
1. new Record C ::{a: int, b: boolean, c: char}
2. C::{a= 7, b= TRUE, c= 'c'}
(heavy-weight pseudo-code example)
```

```
1. new Record C::{a= 7, b= TRUE, c= 'c'}
(light-weight pseudo-code example)
```

Instead, it is possible to declare a new record directly by specifying the record name, its fields and its content, leaving the type inference to the type checker.
This naturally implies that lightweight (anonymous) record can be only declared in type systems that have automatic type detection. The main advantage of lightweight records is that they are more flexible and allow for dynamic modification of fields at run-time. This makes it easier to build modular and extensible systems, since new fields can be added or removed without modifying the record type definition or recompiling the code that uses the record. However, the flexibility of lightweight records can come at the cost of weaker typing guarantees, since the type of the record may not be known until run-time.

Another important feature of record types is extensibility, this feature allows to extend a record by adding new label-value pairs or row pieces using/through concatenation. A record that supports this feature is indeed called an extensible record which is the subject of this document.

```
1. new Record A ::{b:'Surname'}
2. new Record C ::{a:'Name', A} **A is row type(kind).**
3. C ={a:'Name', b:'Surname'}
(pseudo-code example of a light-weight record extension)
```

.

An extensible record is a data structure where the dynamic addition and removal of fields (also known as properties or attributes) should be allowed at run-time. This is in contrast to traditional records, which have a fixed set of fields that cannot be modified.

Extensible records are mostly useful when handling data that may have a variable number of fields in time or when the fields of a record are not defined beforehand. They can be used to represent this information and allow for new fields to be added in the future if needed.

They are also typically implemented using type systems that support row polymorphism, which allows for the creation of types that represent rows of fields with arbitrary labels and types. It is indeed thanks to row polymorphism that the extension or the removal of fields is allowed at run-time.

1.3 The current state of the theory

When facing the record extension problem, most part of the researchers come up with a similar approach, the common objective overall is to create a new type *Record* and a new type *Row* that can be use interchangeably in certain contexts because they do share only one subtle difference.
Row is what we would consider the content of a *Record* without the {} (in case of curly brackets notation), and record, well, would be the whole record itself considered as a unit.
Keeping this in mind and reversing this logic, we could also say that *Record* is a *Row* that is state 'captured' by curly brackets.

$$\begin{aligned}
&\textit{Record}\ \mathrm{R}::\{l_1 = t_1|....|l_n = t_n\}\\
&\textit{Record}\ \mathrm{R}::\{\textit{Row}_1|....|\textit{Row}_n\}\\
&\textit{Record}\ \mathrm{R}::\{\textit{Row}\}\\
&\textit{Row}\ l_1 = t_1|....|l_n = t_n \text{ or empty}
\end{aligned}$$

The | operator is the concatenation operator that usually binds two *Row*s together giving in output a single *Row*, but in this case, since in a programming prospective the programmer does not handle *Row*s but only *Records*, the desired effect we wish to achieve is that of a *Record* | *Record* concatenation.
Row then would be an existing theoretical type that is present in the code but cannot be stored in a variable if not in its *Record* form.
Although this is the general overall objective, there are tons of variations that makes this problem more or less complicated. Let us provide a brief chronological summary of some of the most significant proposals on this matter.

1984-1989
Research in the extensible records types field begins due to the necessity of finding an alternative way to extend objects in other ways other than inheritance subtyping.
The main feature that objects are lacking to these days and led the scientific community to search for other solutions is multiple inheritance and structure-based subtyping, also known as parametric polymorphism.
One of the first proposals regarding extensible records came from Cardelli [Car84] In his paper he explains how Object is a somewhat kind of record while Record can not be fully simulated by Object, moreover he defines subtyping as a general relation between records as "a record type t is a subtype (written $\leq$) of a record type τ' if t has all the fields of τ', and possibly more, and the common fields of t and τ' are in the $\leq$ relation"[Car84]

This general definition that is suitable to describe both inheritance and stuctural subtyping is the focus of the solution proposed.
Records do not have a *Superclass*, the only things that binds them in a hierarchy is the equivalence between labels and types of fields they possess. (Objects can not support multiple inheritance because they cannot have more than one *Superclass*)

$$\text{[RECORD]} \qquad \frac{A \vdash e_1 : \tau_1 \quad \dots \quad A \vdash e_n : \tau_n}{A \vdash \{a_1 = e_1, \dots, a_n = e_n\} : \{a_i : \tau_i\}} \qquad \text{where } i \in I \subseteq 1..n$$

This model however encounters a big problem from the beginning, while wishing to have a well-typed safe type system and to support fully all kinds of polymorphism, the necessity to restrict "Inheritance typechecking to preserve soundness in presence of side effects" emerges. "Parametric polymorphism also has to be restricted in order to deal with side-effects."[Car84] This can be partially solved by "distinguish(ing) syntactically between updatable and nonupdatable record fields, and to require type equivalence (instead of type inclusion) while checking inclusion of updatable fields"[Car84]
This will be later the idea that drives Haskell to adopt a statically non-updatable variable system. In conclusion, Cardelli states that "Merging these two kinds of polymorphism does not seem to introduce new semantic problems.[...] the final goal is to achieve full integration of parametric polymorphism and multiple inheritance, merging functional programming with object-oriented programming at the semantic and typing levels."[Car84]
Another proposal was made not long after by Wand[Wan87], in his paper a similar model was discussed, and two new operations were introduced: the field selection and a single operation that either adds, remove or modify a field of the record, depending on whether the field we wish to add is already present.
Wand then introduces for the first time a new concept: Row variables and proposes to extend the ML-style polymorphic typing directly to row polymorphism making the system overall more flexible.
Thanks to the collaboration of the two a new model for record operations is then created.[CM91] This time a new set of operation is defined: extraction (r.l), restriction(r\l), and extension(r|l=v). Bounded quantification are also mentioned as one of the possible solution to the record update problem: if a record that is used in a function has to be modify and we want to remove its fields it has to keep at least the fields that are involved in that function. As for field adding that does not bring any major issue.
"However observing that bounded quantification is not by itself sufficient, Cardelli and Mitchell[CM91] used an overriding operator on types to overcome this problem."[GJ96] All of the above considerations are done under the absence of duplicate fields assumption.

Even though more than 30 years has passed, this kind of model (Caredelli-Wand) still remains as one of the main references for record systems and it is implemented similarly in most of the languages that supports Record Types.

1995-1999
From the 90s the scientific community starts to approach record from a more practical point of view, often considering already existing languages or suggesting new functionalities for language-specific contexts.

Ohori [Oho95] realised the need for another system capable of classifying types, casually speaking, "a type system for types" and defined it as *kind system*. "A kind k is either the universal kind U denoting the set of all types, a record kind of the form $\{\{l1 : \tau 1,...,tn : \tau n\}\}$ denoting the set of all record types that contain the specified fields, or a variant kind of the form$<< l1 : \tau 1, , tn : \tau n >>$ denoting the set of all variant types that contain the specified fields."[Oho95]

In 'A Polymorphic Type System for Extensible Records and Variants'[GJ96] Gaster and Jones address one of the main problems when working with many of the ML style languages: they "only allow the programmer to select the l component, r.l, from a record r if the type of r is uniquely determined at compile-time. These languages do not support polymorphic operations on records—such as a general selector function (.l) that will extract a value from any record that has an l field"[GJ96] Moreover, they spot some flows related to Ohori's new type system:"The main limitation of Ohori's type system is its lack of support for extensibility."[GJ96]
In this work they try to develop a system that supports both polymorphism and extensibility, and introduce two new operators derived from the three main operators described in Cardelli-Wand's model: update\replace and renaming. They also propose a new model for the *kind system* that differs from the Ohori's one,

$$\begin{array}{rcll} \kappa & ::= & * & \textit{the kind of all types} \\ & | & \textit{row} & \textit{the kind of all rows} \\ & | & \kappa_1 \rightarrow \kappa_2 & \textit{function kinds} \end{array}$$

provide a new set of rules for unification, and theorise a new *kind* lab for labels to use in field selection generalisation r.l (_._).
Once again, Records are suggested as a valid substitution to Objects in Object-oriented programming languages and systems, since they would not bring any major change to the already existing functions if not extending objects and making the system more flexible and prone to general programming.

Gaster and Jones efforts can be finally seen, even partially, in the TRex lan-

guage extension to Hugs (the functional programming system based on Haskell 98), however the result is quite clumsy since it was made to be totally compatible with a pre-existing system and its functionalities were a bit restricted.
Working with Haskell however brings a huge amount of hardship since, "There is no way to add or remove fields from a data structure once it has been constructed; records are not extensible. Each record type stands by itself, unrelated to any other record type; functions over records are not polymorphic over extensions of that record."[JP99] The only plausible solution then seems to be creating a new "dialect" of Haskell capable to implement record with extensions, punning, generics, polymorphism and a whole new set of functionalities not yet supported at that time.

2004-2005
Even though Record field addressing ambiguity was discussed by many it was always avoided under the assumption of not having duplicate labels.
Leijen then introduces a new way to deal with duplicates: "A predicate (r\l) restricts r to rows that do not contain a label l."[Lei04] His model, that was implemented in the experimental Morrow compiler, is based off the Gaster-Jones[GJ96] proposal implementing the three basic operations on records plus the rename and the update operation.
To speed up the run-time access to record labels, Leijen reuses the block of memory model used by Ohori[Oho95] and Gaster-Jones[GJ96], this time using a lexicographical order in order to apply binary search.
Leijen then highlights the necessity to make Labels become first-class values that can be passed as arguments to functions. In this way general field selection function would be allowed:

$$\text{select l r} = \text{r.l}$$

and also other polymorphic in the label functions in general. To do this Leijen also introduces a family of label constructor written as @l [Lei04] :

$$@l :: \text{Lab } @l$$

However, "The explicit annotation of label constants quickly becomes a burden as most labels are constant in practice."[Lei04] as later we will see also in chapter 2.1.1 CTRex.
To use in a polymophic way labels in functions, other few constructs are introduced: *any* a polymophic label that matches all the labels existing, *overloading* done though pattern matching to label l otherwise *any* and *type cases* where the function itself deduces witch label is replaceable to the *any* label in order to satisfy a boolean condition. A new version of the *kind system* is once again defined:

$$
\begin{array}{lcll}
\kappa & ::= & * & \textit{the kind of value types} \\
 & | & \mathsf{row} & \textit{the kind of row types} \\
 & | & \mathsf{lab} & \textit{the kind of label types} \\
 & | & \kappa \to \kappa' & \textit{the kind of type constructors}
\end{array}
$$

this time adding the *lab* kind for Labels.

Not long after Leijen comes up with a new implementable system, this time with a more straightforward new-user friendly approach[Lei05](on Morrow) that features anonymous record construction.
So this time the model proposed is more limited, it avoids artificial constructs and "all operations are checked and the type system statically prevents access to labels that are absent"[Lei05] in order to favour a more practical usage. "Furthermore, extension is polymorphic and not limited to records with a fixed type, but also applies to previously defined records, or records passed as an argument"[Lei05]
He also introduces the concept of safe functions that operates on records only if the labels used are present on that record, otherwise no action is performed. In this way, functions that could be suitable for some records can still be declared as a general function and give an output only when the parameters passed to the function are accepted.
Two semantics are then introduced, when extending a record in a duplicate-label-free environment with a new field that has a label already existing, the "old" field with that label is updated with the content of the "new" field, while in an environment that accepts duplicate labels the new field would be simply added to the record. In order to deal with duplicate labels in a more intuitive way, Leijen also introduces *scoped labels*.
Scoped Labels are non-other than *types* used as specifications after a duplicate label name. [Lei04] :

({x = 4; x = "abc" }− x).x **:: String**
(::String is the scoped label)

In this system, once again, for practical purpose the kind system is restricted only to κ,*,$\kappa -> \kappa$. This system compared to the others previously presented is maybe the most safe, easy to check and intuitive.

2006-today
In recent years, lots of people researched on the matter, but mostly looking at the problem from a practical prospective. For example, in 'Compiling Structural Types on the JVM'[DO09] Scala records, or in this case structural types are compared based on their performances, or in 'Extending Scala with Records'[KH18]

where a model similar to the one described by Leijen in 'Extensible records with scoped labels'[Lei05] is proposed with some adaptation to suit the Scala language.

As for the ML style programming languages, Daan Leijen's model proposed on 'Extensible records with scoped labels'[Lei05] seems to be the most expressive and practical system to keep as a reference when dealing with lightweight extensible records.

Chapter 2

Record Type in programming languages

Record types are not supported in every programming language, but can be still simulated through other type structures (structs, classes, new type def..). However, when they are implemented they usually carry some unique features along with them. The main features desirable for systems that supports record types are: pattern matching compatibility, type variables support, easy field access, type-safety, equality operator for records, extensibility , lightweight declarations and polymorphism.
However, we have to deal with the limitations given by the already existing systems, and find alternative ways to support this features using other constructs.
Let us see how this programming languages are supporting Record Types and what features they offer.

2.1 Haskell

Haskell is a statically-typed purely functional language. It natively supports records as algebraic data types (ADTs), which are defined using the *data* keyword. From Haskell 1.3 and subsequently, records are defined following Jones model suggestion as "is simply syntactic sugar for equivalent operation on regular algebraic data types." where "Neither record-polymorphic operations nor subtyping are supported." [Hud+07]
Here's an example:

```
data Chrt = Chrt{ name:: String, age::Int} deriving Show
g :: Chrt
g = Chrt "Gandalf" 1650
```

Natively Haskell supports only record that are declared as a $data < name >$, this operation can be seen as a new structural type definition where the final usable type has name $< name >$.
A new record instance can be created using the sum type syntax, where the type of the record is actually a sugar-syntax form of its own type constructor.
Haskell has a much shorter-to-declare copy-update method for records compared to other programming languages, whenever a copy of one existing record is needed, a simple

$$newrecord\_name = oldrecord\_name$$

notation can be used.
It is also possible to create a copy of a pre-existing record and partially or totally modify its fields by adding a $\{label_1 = new\_content_1 \ ... \ label_n = new\_content_n\}$ after the *oldrecord_name*, it is not necessary to write every single field of the record, only the ones that we wish to modify.

```
gw :: Chrt
gw = g{name= "Gandalf the white"}
```

To access a field content, unlikely from most of the other programming languages, we have to use the

$$field\_label \ record\_name$$

notation, where $field\_label$ is a function that given a record that contains this label returns the content associated to said label.

```
samename :: Eq a => a -> a -> String
samename s1 s2 =  if s1 == s2 then "same" else "different"

a :: String
a = samename (name gw) (name g)
```

In order to confront two records, a function similar to *samename* is needed: records cannot be confronted using the == operator, the only thing possible is to confront its content, passing to a generic confronting function each one of the fields.
Pseudo-coded as:

rec_comp f_a f_b = if f_a == f_b −−for each field

On a practical level f_a and f_b are the instances of the same-labelled field in the record a and b respectively.
Adopting the same approach to compare two records however fails even when trying to compare two records that are built with the same data type defined earlier. This because "In the non-extensible record system in Haskell currently, the records are typed nominally, which means that we see if two record types are the same by checking the names of the records. For example, to check if the type HRec is the same as HRec2, we check if their names (HRec and HRec2) are equal.[Has13]

```
samerec :: Eq a => a -> a -> String
samerec r1 r2 = if r1 == r2 then "same rec" else "different rec"

s :: String
s = samerec g g
```

samerec in this case throws an error message:
"No instance for (Eq Chrt) arising from a use of 'samerec'",
meaning that Eq (==) cannot operate with custom defined data types.

Since the standard GHC package does not offer much to work with, perhaps we could search for other extensions or ways to implement lightweight extensible records.

Although Record wildcards and Record puns offer an alternative way to write records and feature some handy operations, they still do not fully support extension operations or lightweight declarations.
In order to make this kind of operations, at the moment, we have to relay on

third-party experimental libraries: for example CTRex, a library that was born with the intent of implementing some of the features proposed in Daan Leijen's "Extensible records with scoped labels".[Lei05]

2.1.1 CTREx

CTRex aims to offer an extensible record system where duplicate labels, record extensions and row polymorphism is allowed. Record Types in CTRex are defined as:

$$\text{rec\_name} = \text{labelname} := \text{value} \ .| \ (...) \ .| \ \text{empty}$$

where Rec_name is the name of the record and it has type Rec, labelname is the label of the field and has type Label, .| is a syntactic sugar for the extension operation that concatenate a RecOp type to a row of Rec type and gives in output a Rec type (RecOp .| Rec) , value is simply the field value and empty is an empty record with also type Rec.
"In an extensible record system the record type are structural: two records have the same type if they carry the same fields with the same types, i.e. if they have the same row. This also means we do not have to declare the type of the record before using it."[Has13]

```
{nb = Label :: Label "nb"
ab = Label :: Label "ab"
kb = Label :: Label "kb"
berlin = nb :="Berlin" .| ab := 3721459 .| kb := 891.12 .| empty

ng = Label :: Label "ng"
ag = Label :: Label "ag"
kg = Label :: Label "kg"
cg = Label :: Label "cg"
fg = Label :: Label "fg"


ger = ng :="Germany" .| ag := 84404137 .| kg := 357582 .| berlin}
```

Labels are definable as shown in the code above and unfortunately there is not a more compact way to declare them. It is possible to extend a record with a pre-existing record by simply replacing the *empty* keyword with the name or the record

we want to add. Record can be placed only in the last position so an operation like :

```
-- r1, r2 are already defined records.
r3 = label_1 := value_1 .| r1 .| r2
```

is not allowed. This is mainly because the := operator is a syntactic sugar for the value assignation operation, that given a label with type symbol and a value returns a RecOp type (l := v).

(:=) Symbol− >a− >RecOp
(.|) RecOp − >Rec− >Rec

Speaking from a theoretical point of view RecOp is also a Rec, but in this case it is not. Considering the fact that the .| operator that construct records need a RecOp and a Rec as input it is easy to see why such operation (r1 .| r2) is not allowed, r1 type is not RecOp.
However there is another operation called merge that has .++ as its sugar syntax operator, which given two Rec as input returns a single Rec.

(.++) Rec− >Rec− >Rec

Merge syntax can be used instead of the .| one when linking multiple records, but it can be used only if the two record we want to merge are disjoint.

rec = l := v .| r1 .++ r2
−−allowed

The redundant label declaration problem is partially solvable thanks to the fact that CTRex supports duplicate labels: a label can be used multiple times in different records, moreover these Record Types also support the usage of type variables in their declaration.
All of these features gives us a pretty flexible record system to work with, let us see some other examples and limitations.

```
rx :: Label "rx"
rx = Label :: Label "rx"
ry :: Label "ry"
ry = Label :: Label "ry"

rec2 t = rx := t .| empty
rec1 = rx := 23 .| empty
```

The Haskell Type System does not bind in a permanent way a generics to a certain type after the first usage, so it allows us to use many times the same function with different values and types in all of the code as shown below. *rec*4 and *rec*5 are extensions of *rec*3 but they do not have the same field types. Their types can be seen as:

rec3 'a 'b = rx :: 'a , ry :: 'b
rec4 = rx :: String , rx_r :: Tuple (Int,Int) , ry_r :: String
rec5 = rx :: String , rx_r :: String , ry_r :: Int
$--rx_r$, ry_r notation to distinguish the fields that belong to rec3

```
rec3 v w = rx := v .| ry := w .| empty
rec4 = rx := "Roxy Music" .| rec3 (28,38) "England"
rec5 = rx := "J.R.R.Tolkien" .| rec3 "The Hobbit" 304
```

Since the Haskell programming language does not allow value reassignment whenever a record extension is made it needs to be stored in a new variable, but this means that we also created a new record.
There is a way however that allow us to declare extensible records "on the go", and that is to declare a general record as a constructor function.

x = Label :: Label "x"
rec v r = x := v .| r

So a record construction can be done through passing the *rec* function as a lambda instead of the *r* generics recursively.

rec v r = x := v .| r
another_f r = some_output
res = another_f rec v_1 (rec v_2 (rec ...(rec v_n empty)..))

Using this method we can pass to a function an extended record of *rec* without saving its multiple extensions in other variables unless is needed.

```
add_member m r = rx := m .| r
add_genre g r = rx := g .| r
member_of_rm = add_genre "Ambient" (add_member "Brian Eno" rec4)
```

Record themselves can not be reassigned, however their fields content can be updated with the :< − operator as shown in the code below.

```
up_rec r x = rx :<- x .| r

recup = up_rec rec1 "Demo"
--updated content of label rx
recup1 = up_rec ger "Demo"
--no effect (ger does not have label rx)
recup2 = up_rec member_of_rm "Demo"
--up_rec does not know wich rx label to update
```

When passing to the *up_rec* function, that simply updates the rx labelled value of a record r to x, a record as input that does not contain an rx label the function has no effect on the record. It is possible to pass any record to this function, regardless the presence or absence of the target label and the number of fields of the record.

```
recA = rx := "Ping" .| ry := "Pong" .| empty
recB = ry := "Pong" .| rx := "Ping" .| empty

res_i = equal recA recB
-- return Yes, because position does not matter
```

CTRex records can be compared using the '==' operator unlike object and Haskell native record declaration done through ADTs.
However there are some cases where a $False$ output is thrown even in a $True$ condition: in the code above the *equal* function has in input two records with the same labels and content, but since there is a label ambiguity the compiler has to relay on the fields order instead of their label only.

```
n_rec x r = rx := x .| r
res2 = equal r1 r2

--record type
r1 = rx::Int (4) | rx::Int (3)
r2 = rx::Int (3) | rx::Int (4)
```

If record comparison is needed it is recommended to use either records without label duplicates or to rename labels with the Rename $l' ::< -|l$ operator before the use.

```
equal a b = if a == b then "Yes" else "No"
res = equal rec1 (rec2 23)
--return Yes
res2 = equal (n_rec 4 (n_rec 3 empty)) (n_rec 3 (n_rec 4 empty))
--output No, because both record have multipe rx labels

res3 = equal (rec2 3) (rec3 3 4)
--does not work because number of elements is different
```

This duplicate labels problem does not only affect functions such as comparison but also the simplest field selection operations.
In CTRex the field selector operation can be done using the *dot notation* differently from native Haskell. To solve ambiguity once again the GHC operates in a ordered manner from left to right: when more than one field has the same label the leftmost field is selected. There is still one other operator that can be used in this cases if the programmer knows the label position in the record, the restriction operator .−.
By using the restriction operator it is possible to go right by one position when using the selector(Code above), this because the $. - l$ operator temporary deletes the leftmost field labelled as l. This operation has no side-effect on the record whatsoever so if the record where the restriction was been used upon is called or used in another part of the code it will still has all of its original fields.

```
a_rec =recd.rx
--print last rx inserted label : 22
b_rec = (recd.-rx).rx
--print: 23
```

CTRex does not allow to use Label types as type parameter in functions if not for the general field selection function

$$\text{select r l} = \text{r.l}$$

that is type-implemented as a $safe\ function$, meaning that it operates only when l is a valid label for r.

Even if CTRex seems to be one of the most complete language extensions for extensible record it still presents some flows:

- Records cannot pattern-match due to the fact that Label is not a first-class type.
- A record that uses the extension operator does not have Rec as type:

code −− rec t = rx := t .| empty
type −− rec :: a − > Rec ('Records.R '["rx" 'Records. : − >a])

- For a more practical use a nicer syntax for Label construction is needed, the one that is currently available is too verbose.

2.2 OCaml

Ocaml is a multi-paradigm language that supports object-oriented, functional and imperative programming. It supports record natively but unfortunately with tons of limitations. *type* is the keyword used to define a new record and the fields declaration is pretty much intuitive.

```
 type character = {
  fullname: string;
  mutable age : int;
  race : string;
};;
```

It is possible to make a field mutable by writing *mutable* before the field label name. Fields information access is done though *dot notation* r.l where r is the record name and l is the label name.
To modify values ← symbol is used. (old value ← new value)

```
let g : character = {fullname = "Gandalf"; age = 1560 ; race = "wizardkind"};;

g.age <- g.age+5 ;;

utop # print_int g.age;;
1565- : unit = ()
```

Ocaml has a strong type inference, (also known as let polymorphism) so there is no need to specify the type of a parameter if its type is deductible from the formula. However if two records have same-label-field, the type deducted is the latest declared record.

```
 type character = {
  fullname: string;
  mutable age : int;
  race : string;
};;

type wildBeast = {
  fullname: string;
  mutable age : int;
  food : string;
};;
let samename c1 c2 =
  if c1.fullname = c2.fullname then true else false;;
```

```
utop # samename;;
-:wildBeast -> wildBeast -> bool = <fun>
-( 00:19:55 )-<command 12>-----------------------------
utop # samename g g ;;
Error: This expression has type character
         but an expression was expected of type wildBeast
```

There is no way to declare records in a light-weight anonymous manner in Ocaml. Record extensibility is not natively supported, (there are no external libraries either) if a new record that extends an old one is needed there is no way to declare it but to create a new record and insert all the fields manually. However, there is still one useful tool that OCaml offers to make a "customizable" polymorphic record: Type variables.

2.2.1 Type variables

Type variables (also known as Generics) is an undefined type that is used as a bookmark, it is defined as *'ident* where usually *ident* is a single character. Whenever the compiler can deduct the type variable type, it will change its type in all the occurrences of *'ident* in the code. So technically we could define a record as:

type 'a 'b 'c *rec* { a = 'a ; b ='b ; c ='c}
(*pseudo-code implementation*)

The downside of this practice is that even though I can create instances of record using the same type definition I cannot change the number of elements of record, they are fixed.

```
 type ('a,'b,'c)free_rec = {
  a : 'a;
  b : 'b;
  c : 'c;
}

let newrec1 ={a=195;b=("Thorin","Oakenshield");c="The Hobbit"};;

let newrec2 ={a="Riverside Building";b=20;c=[1,2,3,4]};;

let sum n1 n2 = n1+n2

let newrec3 ={a=(5,33); b=17; c=sum};;
```

In the code above as we can see a new record with three generics, $free\_rec$, is defined. While the fields of an instance can be filled with whatever value ones likes, the number of fields can not be modified and will be always the same as the record type defined.

To summarise the above, Ocaml records offer a good alternative to objects, they support type variables, method as fields, and mutable field content, but they do not offer what so ever kind of extensibility while object do thanks to inheritance.

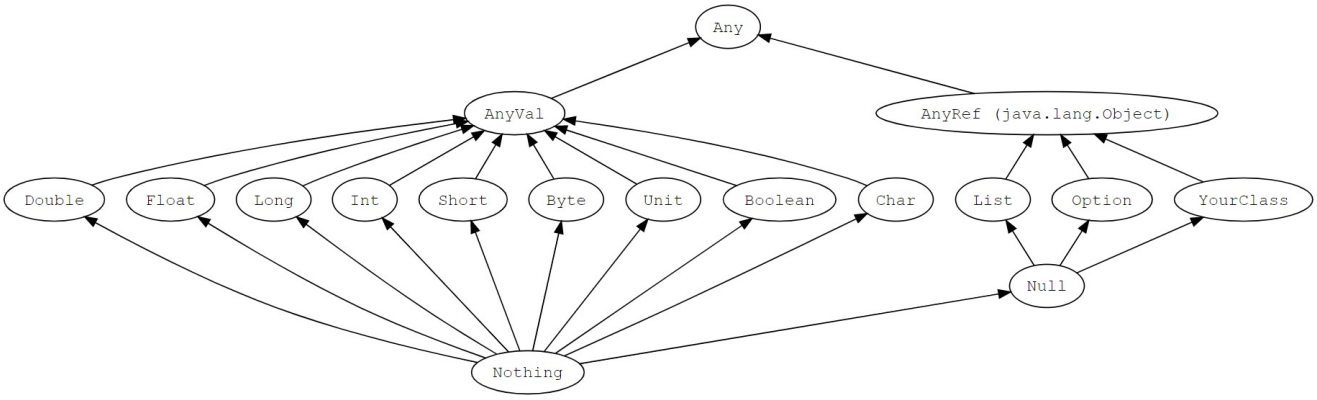


Figure 2.1: Scala Type Hierarchy

2.3 Scala

In Scala (excluding external libraries) there is no "Record Type", but it's possible to simulate a record behaviour and structure using structural typed classes. (Structural typed anonymous classes in light-weight record case)

```
 val person1 = new {
    val fullname = "John Doe";
    val age = 40
  }
  println(person1.fullname)
  //(anonymous struct example)

case class Person(fullname: String, age: Int)

val person2 = Person("Frodo Baggins", 26)
println(person2.fullname)

//(heavy-weight class example)
```

One of the advantages of structural type is that its fields are accessible through dot notation,(object.propriety) which is widely and frequently used not only in functional programming but also in other programming languages(e.g., C++, Java, etc..).

It's important to remember that Scala, even though it fully supports functional

programming, at its core is a pure OO language, this means that every single value and class can be casted to object. (Class is a subtype of object as shown in Figure 2.1)

```
val b = new { val fullname = "John Doe"; val age = 40 }

  if(a==b)
    println("yes")
  else {
    println("no")
  }
  // return is : no
```

As a matter of fact if two records declared like a or b and then are compared with the == operator, the result is 'False'. It is natural to expect this result since when comparing two separate instances of the same class, even if the content of each field is identical, the answer is 'False' unless the second instance is actually pointing to the same address of the first one in the memory.

In Scala, parametric polymorphism can be achieved through bounded quantification. With bounded quantification, we can create a general purpose method that accepts one or more class instances as input only if those classes have the same label names of their fields and are "bounded" to be subtypes of a given type.

```
def youngest[R <: {val age: Int}](a: R, b: R): R =
                                              if (a.age <= b.age) a else b
  val p1 = new {val fullname="John Doe"; val age=40}
  val p2 = new {val fullname="Frodo Baggins"; val age=26}

  println(youngest(p1,p2).fullname)

  //use of achievable parametric polymorphism
```

This means that all records accepted by this new general-purpose method are either superclasses or subclasses of each other.
Bounded quantification can also be viewed, in a sense, as a parametric polymorphic technique constrained by hierarchy-based subtyping.

```
def youngest_1[R <: {val age: Int}](a: R, b: R): R = if (a.age <= b.age) a else b
val pe1 = new {val fullname="John Doe"; val age=40}
val pe2 = new {val fullname="Frodo Baggins";
```

```
                val age=26; val city ="Hobbiville"}

println(youngest_1(pe1,pe2).fullname)
//Bounded quantification respected

val per1 = new {val fullname="John Doe";
                val age=40; val activity ="Serial Killer"}
val per2 = new {val fullname="Frodo Baggins";
                val age=26; val city ="Hobbiville"}
//println(youngest_1(per1,per2).fullname)

//error: inferred type arguments [Object] do not conform to method youngest_2's
// type parameter bounds [R <: AnyRef{val age: Int}]
```

As for this case Least Upper Bound is used.
"Since the JVM does not support structural typing natively, Scala realizes this feature by using reflection and polymorphic inline caches"[DO09] In order to pass objects to a structural reference type erasure is needed, the compiler deletea the original type and substitutes it to type Object.
"When a method is called on the object, Scala's type system knows that the runtime class implements the method and that it can be safely called, but to convince the JVM of this fact a reflective call is needed."
[Kar17]
"Scala supports both nominal typing for object-oriented constructs such as classes, traits, and singleton objects, as well as structural typing in the form of refinement types. The structural typing is reserved for instances of already-existing classes, however (breaking separate compilation), and there exists no language-provided constructor for record literals, nor support for typesafe polymorphic operations such as record extension. Furthermore, the runtime performance of structural member access is debated, as it utilizes reflection on the JVM."[KH18]

As for the extensibility of such data types, unfortunately Scala does not have a built-in support for extenible data structure.
In other words to represent extensible record types and their functionality we have to relay once again on third-party libraries and language extensions.
One popular library for working with record-like data structures in Scala is Shapeless. Shapeless is a library for generic programming in Scala, which includes support for creating "HList" data structures that can be used to represent extensible records.

2.3.1 Shapeless

Shapeless uses HLists (short for "heterogeneous list") to represent records.In this example, the :: operator is used to create a list-like data structure where each element of the HList is a key-value pair, where the key is a label of type Symbol and the value can be of any type. The − >> operator is used to create a key-value pair by associating a label with a value.

```
{ val r =("l_1" ->> "v_1":: ...::("l_n"->>"v_n")::HNil }
```

Where HNil is an empty HList. The :: operator can be also used to link not only single elements but also HLists together.

```
val h = ("name" ->> "Hercule Poirot") :: ("age" ->> 46) :: HNil
val hs = h :: ("Greetings"->"Salut!")::HNil
// hs::(Hercule Poirot,46,Salut!)
```

Shapeless supports equality checks since the HLists are saved in the memory with their elements in order.

```
val c = ("name" ->> "John Doe") :: HNil == ("name" ->> "John Doe") :: HNil
//true
```

Field access can be done through the ¡record name¿.get(¡name of label¿) construct.

```
val n =h.get("name")
//Hercule Poirot
```

"The HLists are fundamentally ordered, and so permutation subtyping is not provided."[Kar17] While Shapeless does provide some support for working with extensible records in Scala, it is not a first-class language feature and can sometimes be cumbersome to use.

2.3.2 Compossible

The Compossible language extension was born from the need of finding a way to make transformations on structured data in a type safe concise way. Records in Compossible does not need reflection like the Scala ones. Record declaration is done using the following scheme:

$$\text{val r} = \text{Record}(l\_1{=}v_1, \ldots l\_n{=}v_n\ )$$

The operator ++ is used for record concatenation:

$$\text{val r2} = \text{r} \mathbin{++} \text{Record}(l\_1{=}v_1, \ldots l\_n{=}v_n\ )$$

This operation also accepts Scala case classes declaration as one of its members.

```
{
case class Person(fullname: String, age: Int)

val person = Person("Frodo Baggins", 26)
val r2 = person ++ Record(town = "Hobbiville")
}
```

Field access is done through *dot notation*, and it also feature the copy method as native Scala does.

```
{
val r = Record(name="James", surname ="Bond")
val r1 = r.copy(surname ="Brown")

r1::( James , Brown )
}
```

Compossible does not support equality checks nor bounded quantification. (side-note: Eclipse has a quite decent IDE support for Compossible while Intellij has no support at all.)
Overall it is a fun language extension, useful for simple data management projects because it has a pretty clean and intuitive syntax.

2.4 F#

In F# records have some limitations, in the Microsoft F# documentation they are defined as "simple aggregates of named values, optionally with members. They can either be structs or reference types. They are reference types by default." [Var21c]

```
[ attributes ]
type [accessibility-modifier] typename =
    { [ mutable ] label1 : type1;
      [ mutable ] label2 : type2;
      ... }
    [ member-list ]
```

It is possible to declare a record using the syntax above where *typename* is the name of the record type, *label*1 and *label*2 are names of values and *type*1 and *type*2 are the types of these values respectively. $member - list$ is an optional list of members for the type. " You can use the [$< Struct >$] attribute to create a struct record rather than a record which is a reference type."[Var21c]
The declaration of a record (heavy-weight) is done as the following:

```
type Person = { FN: String; A: int }

let personA = { FN= "Lupin"
                A = 35}

//classic record declaration
```

The light-weight implementation of a record can be done through anonymous-records. Similarly defined as "simple aggregates of named values that don't need to be declared before use. You can declare them as either structs or reference types. They're reference types by default."[Var21a]

```
let person0 =
    let n = "John Doe"
    let a = 40

    {| fullname = n; age = a |}
    //lightweight record
```

"Records are immutable by default; however, you can easily create modified records by using a copy and update expression. You can also explicitly specify a mutable field." [Var21c]

```
let person_1 = {| person1 with City = "Hobbiville" |}
    //let introduces a new record
```

As for the majorities of the programming languages also F# uses the *dot notation* to access a property of the record.

```
let person1 =
    let n = "Frodo Baggins";
    let a = 26;

    {| fullname = n; age = a |}

let fname = person1.fullname
    //fname : Frodo , no errors
```

F# records do not support any concatenation operation, therefore the only way to combine two or more records is to make a new record that has all the fields of the other records I want to combine. Technically it is not a proper extension but, a new member declaration.
The *with* keyword can be used when creating a new record that has all the same fields of an other record plus some new fields of its own. Instead of creating new fields it is also possible to update an existing field just by writing its label and the updated value after the keyword *with*.

A record can be also used in pattern matching by associating each field to a specific value ad returning its correspondent output case. The _ symbol is used when the field at its left can be matched to any value.

```
let evaluateName (person: Person, s: String) =
    match person with
    | {FN=s; A=_} -> printfn "Same name"
    | {FN=_; A=_} -> printfn "Not the same name"

evaluateName(personA,"Lupin")
// true , no errors
    //pattern matching
```

Declaring a general purpose method that accepts record as input can be tricky. F# requires the programmer to specify each field of the record that we intend to use with a $[name - rec] : \{|l_1 : t_1; ...l_n : t_n|\}$ notation.

```
let samename (a0 : {| fullname : String ; age :int |},
              a1: {| fullname : String ; age :int |})=
    if a0.fullname = a1.fullname then true else false

    //the function input is too verbose,
    //I'd like to declare my records requisites only once

let res = samename(person0,person1)
    //output: false, no errors
```

Which is a pretty verbose notation to use every single time. (Imagine having to deal with a record that has more than ten fields)
The only way that allow us to avoid this notation is to declare explicitly the record by defining a new record type, and then use this type in our newly declared method.

```
let samename1 (a1 :Person, a2: Person)=
    if a2.FN= a1.FN then true else false
```

This way might seem more convenient but it comes at its costs, for each record type that is intended to be used in a method a new record type must be created, even if we might use it only once in the entire program.

```
let person0 =
    let n = "John Doe";
    let a = 40;

    {| fullname = n; age = a |}

let person_0 =
    let n = "John Doe"
    let a = 40
    let j = "Serial Killer"

    {| fullname = n; age = a ; job =j|}

let samename (a0 : {| fullname : String ; age :int |},
              a1: {| fullname : String ; age :int |})=
    if a0.fullname = a1.fullname then true else false

    //the function input is too verbose,
    //I'd like to declare my records requisites only once
```

One of the most logical solutions to this problem that one might think is to specify only the necessary fields that a record must have in order to be accepted as input by the method.
However, since F# does not support record extensibility this solution does not work. Record types do not support any type of inheritance or row polymorphism therefore the type checker has no way to check for compatibility between them. Only specified fields are accepted and no others.

"It is not currently possible to define an anonymous record with mutable data. There is an open language suggestion to allow mutable data."[Var21a] It still possible however, to declare a record using Generics(Type Variables), and obtain a similar effect to mutable field records.

2.4.1 Type Variables

Similarly to OCaml also F# supports the use of type variables in a record, therefore it is possible to declare a record with n fields and m generics and initiate it as ones likes.

```
 type 'a Plant = { N: String; Poison: Boolean; Edible: Boolean; Extra: 'a }

let plant = { N= "Orchid"
              Poison= false
              Edible= true
              Extra = ("petals number",3)}

type ('a,'b,'c) general_rec = { a: 'a ; b : 'b ; c : 'c }

let rec1 = { a= "Agatha"
             b= "Christie"
             c= 1890 }
```

Also in this case it is only possible to declare a record with n fields since no kind of record extension is supported.
"Records have the advantage of being simpler than classes, but records are not appropriate when the demands of a type exceed what can be accomplished with their simplicity."[Var21b]

Chapter 3

Comparison between theory and programming languages

3.0.1 Theory vs practice

For this chapter we will mainly consider two candidates as representatives of the best models available in practice and in theory: Haskell extension CTRex and Daan Leijen model described in "Extensible records with scoped labels"[Lei05] Obviously a practical implementation will always lack something when compared to a theoretical model, but let us do a comparison between the two anyway.

Syntax
CTRex Syntax is pretty similar to the mathematical model of records,

CTRex: <record name> $= l_1 := v_1$.| . . ..| $l_n := v_n$.|empty
Theory: <record name> $= \{l_1 = v_1$,...$l_n = v_1\}$

however, when declaring a record CTRex uses the extension construct .| to build the record.

Extension
Similarly to the standard record declaration in order to extend a record the Extension operator is used in both cases almost identically.

CTRex: <record name> $= l_1 := v_1$.| . . ..| $l_n := v_n$.| < other record >
Theory: < record name > $=$ { $l_1 = v_1$ | . . .| $l_n = v_n$ | < other record > }

Selection and restriction
Selection and restriction are both implemented in an almost identical way.

CTRex: Restriction : (r.− l) Selection: (r.l)
Theory: Restriction : (r − l) Selection: (r.l)

Renaming
In theory renaming is done through field restriction followed by a new field extension, while in CTRex we can use a more practical sugar syntax for rename (::< −|).

CTRex: (oldL::< −|newL)
Theory: { m< −l} = {l=r.m | r− m} −− rename m to l

Updating
It is possible in both system to update a value of a field with label l.

CTRex: (l:< −new value)
Theory: { l:=new value} = {l=new value | r− l}

Duplicate labels
Both of the systems do support duplicate labels, when extending a record with an already existing label, they just simply add the label-value pair to the record. Not surprisingly they also both adopt the same approach when accessing to duplicate label fields in the record: since the record is analysed from left to right, we need to use the restriction operation as many times as we need to point to our desired label to access a specific label. (one restriction corresponds to one right shift)

Main differences

- While in theory there is no function mentioned to create a record that extends two or more records, CTRex introduces a *merge* operator .++, a new record can be created as the following : newrec = oldrec1. + +oldrec2.
- To solve ambiguity Leijen introduces *scoped labels* a type that can be used as a tag after a label to let the system know on which duplicate labels it needs to operate.

  (rec).l::Type

- Pattern matching is not supported in CTRex while it is not mentioned in the theory, however, since Leijen system supports a mechanism capable of matching duplicate labels to types (*scoped labels*) it could also probably support pattern matching.

- The extension operation in Leijen system returns a Record of Row type while CTRex does not, it returns a construct type similar to $Rec(Records.R[\ "l"\ Records.R : -> a])$

To sum everything up we said in this chapter: records in practice are actually not too far from the theory presented, however, they still carry some major issues that limit their expressiveness. Labels are not first-class types so we cannot use them as parameters in functions and in pattern matching.
In Haskell variables cannot be updated after binding for type-safety reasons, so this brings us to a huge amount of variable declarations if we wish to store all of our records.
If we wish to have a more flexible and generic programming oriented system, type-safety must be partially sacrificed.

Chapter 4

Use of Record Type vs Traditional Object System

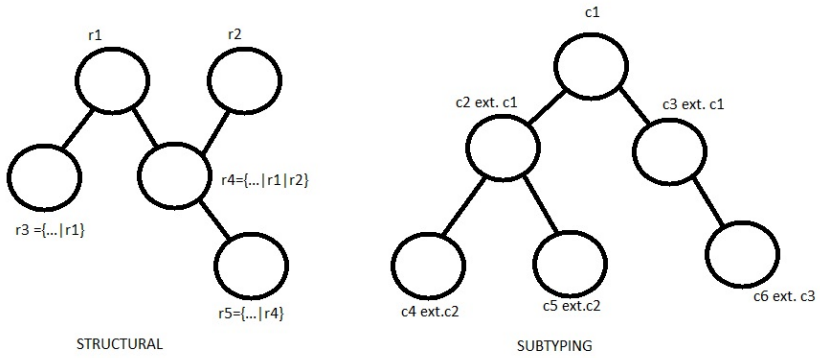
Objects and Records are both collections of labelled values stored on the memory as records. Records may resemble object-like logical data structures yet they're very different from objects. While records are essentially a row of field data allocated on the stack by default, objects are made up of their field definitions and use the heap instead of the stack, and this is why when an instance of an object is created, more memory is used compared to a record instance.
Objects are in some way a more restrictive kind of record, where every field must be pre-declared in a class. Class fields and methods can have access modifiers (Java) that allow to hide information or prevent its modification. Records do not have keywords that allow to modify information visibility but some do have the *mutable* keyword that marks which fields can be updated. As far as self-referencing is concerned, objects can "call upon themselves" using a keyword -usually this- that allows operations that need writing or reading on the object instance itself. This is not something we can do with records and we would have to explicitly manipulate them.
Another aspect that we would need to look into is inheritance: objects have a more strict hierarchical system set upon definition of the object class. As for records, their hierarchy is purely based on sharing the same label-type couples: a record a that has at least all the fields of another record b is considered to be an extension of b.
This general definition of extension is quite resembling Cardelli's[Car84] one where a record type t is subtype of another record t' if it has all fields of t' or more.
Following Cardelli's proposal we can conclude that inheritance-based subtyping implies structural subtyping.
For example, in order to use a polymorphic function fun(r1 with (a:'a,b:'b),r2 with (a:'a,b:'b)) that takes two objects with fields, a of type 'a and b of type 'b

of two general classes r1 and r2 and gives an output in an OO system, it would be necessary to introduce a new class or interface c that has a:'a and b:'b, find all of the existing classes that already have a:'a and b:'b and make them extend or implement c and then finally substitute c int the variables of the fun function like: $fun\ (c1 : c, c2 : c)$.
In a record system, the same function could be implemented with something along the lines of: fun[R <: {val a: 'a, val b: 'b}](r1: R, r2: R).
Let us see some other examples:

```
fun(r with a:'a,b:'b) -> something
fun2(r: A)-> something_else
fun3(r: B)-> something_else

class A {
 a : 'a
}
class B {
b : 'b
}

class ABC extend A {
 a : 'a
 b : 'b
 c : 'c
}

class BDA extend B {
 b : 'b
 d : 'd
 a : 'a
}

class AB extends ??? {
 a : 'a
 b : 'b
}
```

In this example, the approach of creating a new class with the fields required by the function and then make all the records having those fields extend that class seems quite disastrous. In the hypothesis of having two classes ABC and BDA extending respectively class A and class B and the need of create a new class AB with both fields form A and B we could state that making AB extend either A, B

or nothing at all would bring some major issues.
Case1: Case1: AB extends A, than ABC now extends AB, so if we try to pass an AB or an ABC instance to $fun3$ it would be an invalid input.
Case3: AB extends B, than BDA now extends AB, so if we try to pass an AB or a BDA instance to $fun2$ it would be an invalid input.
Case3: AB does not extend anything and now BDA and ABC extend AB, fun now would finally work but, both AB, ABC and BDA would not be valid input for either $fun2$ or $fun3$.
Record on the other hand could easily solve this problem by simply declaring as parameter of the function a record r that possess at least a:'a and b:'b.
Subtype polymorphism certainly has its upsides but its many limitations make classes and objects quite unsuitable for general programming. From this considerations we can conclude that subtype polymorphism is a much weaker form of polymorphism compared to structural one, indeed we can also state that subtype polymorphism is in a sense a kind of structural polymorphism that is strictly limitated by a tree-like hierarchy, while the hierarchy in row polymorphism could be rather compared to a graph.



In Cardelli's model for multiple inheritance[Car84] it is also stated that merging the two kinds of polymorphism does not introduce any semantic problem, proving that structural polymorphism can emulate subtyping but not the other way around.
Record can also emulate both *overriding* and the *Super* construct.

```
public class Berries{
 String name = Berry;

 public void Poisonous(){
 System.out.println("maybe");
 }
 }
```

```
 public class RedBerries extends Berries{
 String name = RedBerry;
 String colour = "Red";

 public void berrycolour(){
 System.out.println("this"+ super.name + "is" + colour);
 }
 }

public class Strawberries extends RedBerries{
 String Type = "Strawberry";

 @override
 public void Poisonus(){
 System.out.println("Not poisonous");
 }
}
```

In this example the @*override* keyword is used as a tag to highlight the fact that a method imported from a superclass has been updated. Super is instead used before a field name to indicate that the referred value is not the one implemented in the current class but in his superclass.

```
nameb = Label :: Label "nameb"
poisonous = Label :: Label "poisonous"
typeb = Label :: Label "typeb"
colorb = Label :: Label "colorb"
berrycol= Label :: Label "berrycol"

berries= nameb := "Berry" .| poisonous := putStrLn "maybe" .| empty
redberries= colorb := "red".| nameb:<-"RedBerry".|
            berrycol := putStrLn ("this " ++ berries.nameb ++" is "
            ++ redberries.colorb).| berries
strawberry = typeb:="strawberry".| poisonous:<-putStrLn "Not poisonous"
            .|redberries
```

Record simply uses the update function to override functions and refers to its "superclass" through *dot notation*. Although the label declaration is quite redundant, if we look only at records and class syntax, records are overall a little bit more simple and clean.

From a pure programming point of view, both Objects and Records are valid options, it really depends on what the target is. Records are more handy in programs that make abundant use of general functions or where long data collections are handled. Objects on the other hand are really good for programs where memory management and self-referencing is needed.

Chapter 5

Final considerations

Extensible records came a long way to this days and still they have to be developed to their fullest potential.Through the history of records we have seen how much records have evolved and how still some problems are persistent to this days, one and maybe the most critical being the restriction of parametric polymorphism for the sake of preserving type safety and vice versa.
Their structure makes them a valid alternative to other structured types such as Structs, Classes, Tuple..etc. while keeping a clear, not too wordy, user-friendly syntax. Even if partially developed extensible lightewight records are one of the most powerful tools to manage data in a flexible and generic way, however, there is not a single language that implements all the features we listed so far: the only thing we can do is choose our target language based on the features we need to implement on our program.

Bibliography

[Car84] Luca Cardelli. "A semantics of multiple inheritance". In: *Semantics of Data Types*. Ed. by Gilles Kahn, David B. MacQueen, and Gordon Plotkin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 51–67. ISBN: 978-3-540-38891-3.

[Wan87] Mitchell Wand. "Complete Type Inference for Simple Objects". In: *Logic in Computer Science*. 1987.

[CM91] Luca Cardelli and John C. Mitchell. "Operations on records". In: *Mathematical Structures in Computer Science* 1.1 (1991), pp. 3–48. DOI: 10.1017/S0960129500000049.

[Oho95] Atsushi Ohori. "A Polymorphic Record Calculus and Its Compilation". In: *ACM Trans. Program. Lang. Syst.* 17.6 (Nov. 1995), pp. 844–895. ISSN: 0164-0925. DOI: 10.1145/218570.218572. URL: https://doi.org/10.1145/218570.218572.

[GJ96] Benedict R. Gaster and Mark P. Jones. "A Polymorphic Type System for Extensible Records and Variants". In: 1996.

[JP99] Mark P Jones and Simon Peyton Jones. "Lightweight Extensible Records for Haskell". In: *Haskell Workshop*. Paris. ACM. ACM, Oct. 1999. URL: https://www.microsoft.com/en-us/research/publication/lightweight-extensible-records-for-haskell/.

[Lei04] Daan Leijen. *First-class labels for extensible rows*. Tech. rep. UU-CS-2004-51. UTCS Technical Report. Dec. 2004. URL: https://www.microsoft.com/en-us/research/publication/first-class-labels-for-extensible-rows/.

[Lei05] Daan Leijen. "Extensible records with scoped labels". In: *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*. Ed. by Marko C. J. D. van Eekelen. Vol. 6. Trends in Functional Programming. Intellect, 2005, pp. 179–194.

[Hud+07] Paul Hudak et al. “A History of Haskell: Being Lazy with Class”. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, 2007, 12–1–12–55. ISBN: 9781595937667. DOI: 10.1145/1238844.1238856. URL: https://doi.org/10.1145/1238844.1238856.

[DO09] Gilles Dubochet and Martin Odersky. “Compiling Structural Types on the JVM: A Comparison of Reflective and Generative Techniques from Scala’s Perspective”. In: *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ICOOOLPS ’09. Genova, Italy: Association for Computing Machinery, 2009, pp. 34–41. ISBN: 9781605585413. DOI: 10.1145/1565824.1565829. URL: https://doi.org/10.1145/1565824.1565829.

[Has13] HaskellWiki. *CTRex — HaskellWiki*. [Online; accessed 8-March-2023]. 2013. URL: https://wiki.haskell.org/index.php?title=CTRex&oldid=57294.

[Kar17] Olof Karlsson. “Record Types in Scala: Design and Evaluation”. PhD thesis. 2017. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-211048.

[KH18] Olof Karlsson and Philipp Haller. “Extending Scala with Records: Design, Implementation, and Evaluation”. In: *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*. Scala 2018. St. Louis, MO, USA: Association for Computing Machinery, 2018, pp. 72–82. ISBN: 9781450358361. DOI: 10.1145/3241653.3241661. URL: https://doi.org/10.1145/3241653.3241661.

[Var21a] Various. *Microsoft documentation, Anonymous Records*. Nov. 2021. URL: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/anonymous-records.

[Var21b] Various. *Microsoft documentation, Classes(F#)*. May 2021. URL: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/classes.

[Var21c] Various. *Microsoft documentation, Records (F#)*. Dec. 2021. URL: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/records.