Supplementary Notes on Records

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 18
October 29, 2002

A common generalization of the notion of a product is a *record*. A record is like a tuple, except that the components are named explicitly by a *record label*. All labels in a record must be distinct. Records can also be used as the foundation for object-oriented programming idioms in a functional language. In the case, an object would be represented as a record, and a record label would be either a field name or a method name.

We begin by studying records in themselves; later we consider how to model some features of objects. We extend the type system by *record types* that we denote by ρ; we use l to denote record labels.[1]

$$\begin{array}{rrcl} \text{Types} & \tau & ::= & \ldots \mid \{\rho\} \\ \text{Record Types} & \rho & ::= & \cdot \mid l{:}\tau, \rho \end{array}$$

We extend expressions to allow the formation of records, denoted by r, and also the selection of a field from a record, written as $e.l$ for a record label l.

$$\begin{array}{rrcl} \text{Expressions} & e & ::= & \ldots \mid \{r\} \mid e.l \\ \text{Records} & r & ::= & \cdot \mid l{=}e, r \end{array}$$

We sometimes use parentheses to enclose record types or record so the scope of the ',' is more clearly visible. Such parentheses are not properly part of the syntax of the language. We have a new typing judgment $r : \rho$, used in the following rules.

[1]Not to be confused with memory locations that we use to study mutable references.

$$\frac{\Gamma \vdash r : \rho}{\Gamma \vdash \{r\} : \{\rho\}} \qquad \frac{\Gamma \vdash e : \{\rho\} \quad \rho = \rho_1, l{:}\tau, \rho_2}{\Gamma \vdash e.l : \tau}$$

$$\frac{}{\Gamma \vdash (\cdot) : (\cdot)} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash r : \rho}{\Gamma \vdash (l{=}e, r) : (l{:}\tau, \rho)}$$

Note that the field selection operation $e.l$ will always yield a unique answer on well-typed records. This is because labels in a record must be unique. The order of the fields in a record is significant, although we discuss below how this can be relaxed using *exchange subtyping* for records.

In this notation, the empty record expression corresponds to the unit type. Note that there is a minor ambiguity in that the empty record and its type are both denoted by '$\cdot$', that is, $\vdash \{\cdot\} : \{\cdot\}$. As usual, we omit a leading '$\cdot$' in a record.

A pair $\texttt{pair}(e_1, e_2)$ can represented by the record $\{1{=}e_1, 2{=}e_2\}$. Then the first and second projection are defined by $\texttt{fst}(e) = e.1$ and $\texttt{snd}(e) = e.2$, respectively. This is the approach taking in Standard ML, using the notation $\#l(e)$ instead of $e.l$.

Records in this form may make code more readable, because instead of writing $\texttt{fst}(e)$, we write $e.l$, where l is presumably a meaningful label. However, a much greater advantage can be derived from records if we add rules of subtyping. Before we describe this, we give the operational semantics for records. There are two new judgments, $r\ \mathsf{value}$ and $r \mapsto r'$. A record is a value if all fields are values, and we evaluate the components of a record from left to right.

$$\frac{r \mapsto r'}{\{r\} \mapsto \{r'\}}$$

$$\frac{}{(\cdot)\ \mathsf{value}} \qquad \frac{v\ \mathsf{value} \quad r\ \mathsf{value}}{(l{=}v, r)\ \mathsf{value}}$$

$$\frac{e \mapsto e'}{(l{=}e, r) \mapsto (l{=}e', r)} \qquad \frac{r \mapsto r'}{(l{=}v, r) \mapsto (l{=}v, r')}$$

$$\frac{e \mapsto e'}{e.l \mapsto e'.l} \qquad \frac{r\ \mathsf{value} \quad r = (r_1, l{=}v, r_2)}{\{r\}.l \mapsto v}$$

The progress and type preservation theorems now also have to account for the new judgment, but this is entirely straightforward and omitted here.

There are three forms of subtyping that can be considered together or in isolation: *depth subtyping*, *width subtyping* and *exchange subtyping*.

The general rule passes from ordinary types to record types and is common to all forms of subtyping.

$$\frac{\rho \leq \rho'}{\{\rho\} \leq \{\rho'\}} \leq\text{record}$$

Depth subtyping. This is the idea we used for product subtyping, applied to records. Subtyping is co-variant in all fields of a record.

$$\frac{\tau \leq \tau' \quad \rho \leq \rho'}{(l{:}\tau, \rho) \leq (l{:}\tau', \rho')} \leq_d\textit{field} \qquad \frac{}{(\cdot) \leq (\cdot)} \leq_d\textit{empty}$$

We do not show formally how to construct coercions, but consider the following sample coercion.

$$\lambda r.\{x{=}\mathsf{itof}(r.x), y{=}\mathsf{itof}(r.y)\} : \{x{:}\mathsf{int}, y{:}\mathsf{int}\} \leq \{x{:}\mathsf{float}, y{:}\mathsf{float}\}$$

Width subtyping. The idea of width subtyping is that we can always coerce from a record with more fields to a record with fewer by dropping some extra fields. We separate out the idea of exchange and allow fields to be dropped only at the end of a record.

$$\frac{\rho \leq \rho'}{(l{:}\tau, \rho) \leq (l{:}\tau, \rho')} \leq_w\textit{field} \qquad \frac{}{\rho \leq (\cdot)} \leq_w\textit{empty}$$

Note that the $\leq_w$*field* rule does not allow any subtyping on the field itself—this would require the combination of width and depth subtyping.

We can give width subtyping both a subset and a coercion interpretation. The subset interpretation would say that a value of type $\{\rho\}$ can be any extension of $\{\rho, \rho'\}$: the additional fields ρ' are simply ignored. This is common in object-oriented languages. It requires a so-called boxed representation where every object is simply a pointer to the actual object, so that for the purpose of argument passing, every record has the same size.

The coercion interpretation would explicitly shorten the object, which is not usually practical. Nonetheless, under this interpretation we might have a coercion such as

$$\lambda r.\{x{=}r.x, y{=}r.y\} : \{x{:}\mathsf{float}, y{:}\mathsf{float}, c{:}\mathsf{int}\} \leq \{x{:}\mathsf{float}, y{:}\mathsf{float}\}$$

Exchange subtyping. This means we can reorder the fields. We formalize this by allowing corresponding fields to be picked out from anywhere in the middle of the record.

$$\frac{(\rho_1, \rho_2) \leq (\rho_1', \rho_2')}{(\rho_1, l{:}\tau, \rho_2) \leq (\rho_1', l{:}\tau, \rho_2')} \leq_x\textit{field} \qquad \frac{}{(\cdot) \leq (\cdot)} \leq_x\textit{empty}$$

An implementation that allows exchange subtyping would typically sort the fields alphabetically by field name.

It is straightforward to construct type systems for record that combine depth, width, and exchange subtyping. We will see what is needed in order to model some object-oriented features, following Chapter 18 of Benjamin C. Pierce: *Types and Programming Languages*, MIT Press, 2002. We only sketch the rationale and implementation below; for more detail see the above reference.

Objects. As a very first approximation we think of an object as a record with some internal state. This internal state is *encapsulated* in that it can only be accessed through the visible fields of the method. We use a simple counter as an example.

```
Counter = {get:1 -> int, inc:1 -> 1};
c : Counter =
    let x = ref 1
     in
        {get = λ_:1. !x,
         inc = λ_:1. x := !x+1}
    end;
```

In the terminology of object-oriented languages, `x` is a private field, accessible only to the methods `get` and `inc`.

We can increment and then read the counter by sending messages to `c`. This is accomplished by calling the functions in the fields of `c`.

```
(c.inc(); c.inc(); c.get()); ↦* 3
```

Object Generators. We can package up the capability of creating a new counter object.

```
newCounter : 1 -> Counter =
  λ_:1.
    let x = ref 1 in
        {get = λ_:1. !x,
         inc = λ_:1. x := !x+1}
    end;
```

Subtyping. We can easily create an object with more methods. Width subtyping allows us to use the object with more methods in any place the the object with fewer methods is required.

```
ResetCounter =
  {get:1 -> int, inc:1 -> 1, reset:1 -> 1};
newResetCounter : 1 -> ResetCounter =
  λ_:1
    let x = ref 1 in
      {get = λ_:1. !x,
       inc = λ_:1. x := !x+1,
       reset = λ_:1. x := 1}
    end;
```

Grouping Instance Variables. The instance variable x was just a single variables; it is more consistent with the approach to group them into a record. The modification is completely straightforward.

```
Counter = {get:1 -> int, inc:1 -> 1};
newCounter : 1 -> Counter =
  λ_:1.
    let r = {x = ref 1} in
        {get = λ_:1. !(r.x),
         inc = λ_:1. r.x := !(r.x)+1}
    end;
```

Simple Classes. We can extract the instance variables and make them a parameter of the instance creation mechanism. This will allow us to give a simple model of subclassing and inheritence.

```
CounterRep = {x : int ref};
Counter = {get:1 -> int, inc:1 -> 1};
CounterClass : CounterRep -> Counter =
  λr:CounterRep.
   {get = λ_:1. !(r.x),
    inc = λ_:1. r.x := !(r.x)+1};
newCounter : 1 -> Counter =
  λ_:1. let r = x=ref 1 in counterClass r end;
```

The possibility of a shared representation allows us to create an instance of the `ResetCounter` subclass by first constructing a `Counter`.

```
ResetCounterClass : CounterRep -> ResetCounter =
  λr:CounterRep.
    let super = counterClass r in
     {get = super.get,
      inc = super.inc,
      reset = λ_:1. r.x := 1}
    end;
newResetCounter : 1 -> ResetCounter
  λ_:1. let r = {x=ref 1} in resetCounterClass r end;
```

Adding Instance Variables. So far, subtyping only allows us to use instances of a subclass where instances of a superclass are required. When we add instance variables, we need it in another place, namely where the representation of the instance of the superclass is created.

```
BackupCounter =
  {get:1 -> int, int:1 -> 1, reset:1 -> 1,
   backup:1 -> 1};
BackupCounterRef =
  {x:int ref, b:int ref};
backupCounterClass =
  λr:BackupCounterRep.
    let super = resetCounterClass r in    % subtyping here
      {get = super.get,
       inc = super.inc,
       reset = λ_:1. r.x := !(r.b),
       backup = λ_:1. r.b := !(r.x)}
    end;
```

As we can see, references to instances of the superclass are easy. But references to the methods of the class itself within the method are somewhat tricker, but an essential technique in object-oriented languages. We discuss this in the next lecture.