CPSC 311: Records **(DRAFT)**
("lec-records")

Joshua Dunfield
University of British Columbia

November 20, 2016

Updated 2016–11–20 with an explanation of `upper-bound` and another exercise.

1 Records

The first part of these notes is handwritten:
`http://www.ugrad.cs.ubc.ca/~cs311/2016W1/notes/scan-2016-11-18.pdf`

1.1 Record syntax

(See the first scanned page.)

Note the syntactic sugar `{Record {x y pos}}` for several fields with the same type.

The expression (Dot e y) evaluates e to a record, and returns the field named y.

Field names like x and y are not bindings; they don't have a scope. The only way to use a field name is in (Dot e y). If (Id y) appears in the e in (Dot e y), it must be bound in the usual way by a Let, Lam, Pair-case, etc.

We won't define substitution for this system, but if we did, the field names would never be affected by substitution. We *would* need to substitute within the *contents* of the fields.

A record with two fields is roughly the same as a pair, if we provided only Fst and Snd for accessing the parts of the pair, instead of Pair-case. (Something like Pair-case on a record would be reasonable; in Pascal, a similar feature was called `with`. Pattern matching in languages like ML works on records, too.)

Question: Can we have two fields with the same name?

Yes, in different record types. But you shouldn't make a record with two fields with the same name. My implementation doesn't check for this, and it's probably not too hard, but I don't want to commit to saying it's easy when I haven't done it.

Records can be nested inside other records, or placed into refs, or used as arguments or results to functions. We're designing records as an *orthogonal feature*: nothing about the record type, or record expressions, forces us to have any other particular feature in the language. We could have a language with records but not functions, or with records but not refs, and so on.

1.2 Width subtyping

(See the second scanned page.)

All we can do with a record is access a field using (Dot e y). It shouldn't matter if other fields are present; they can't affect the value of the field y.

Thus, if we define a function (top of the page) that expects, as its argument, a record with one field x : pos, it should be okay to pass a record with additional fields.

To do that, we need to use subtyping, so we can show that

$$(\mathsf{record}\ \ x{:}\mathsf{pos}, y{:}\mathsf{pos}) \texttt{<:}\ (\mathsf{record}\ \ x{:}\mathsf{pos})$$

The effect is kind of like subclassing in Java, at least, the part of subclassing that is about adding instance variables to the subclass that aren't present in the superclass.

This also (maybe) justifies the rather strange type $(\mathsf{record}\ )$, which is the type of (record), the record with no fields: it's a little like Java's `Object`.

1.3 Depth subtyping

Another form of subtyping that's useful for records is "depth subtyping", which says that a record with one field y, of type A, is a subtype of a record with one field y of type B, provided that A is a subtype of B. This is reminiscent of subtyping for pairs (the Sub-product rule).

1.3.1 Upper bounds

For example, depth subtyping allows us to pass a record of type $(\mathsf{record}\ \ y{:}\mathsf{pos})$ to a function that expects $(\mathsf{record}\ \ y{:}\mathsf{rat})$. According to depth subtyping, this is allowed because $\mathsf{pos} \texttt{<:}\ \mathsf{rat}$.

In an earlier version of `typeof` with subtyping, several branches called a function `upper-bound`. For example, the branch for Ite called `upper-bound` on the types of `eThen` and `eElse`. This is necessary because we might need to use Type-sub. For example, our typing rules show that $(\mathsf{Num}\ 3)$ has type pos and -5 has type int:

$$\frac{3 \in \mathbb{Z} \qquad 3 \geq 0}{\emptyset \vdash (\mathsf{Num}\ 3) : \mathsf{pos}}\text{Type-pos} \qquad\qquad \frac{-5 \in \mathbb{Z}}{\emptyset \vdash (\mathsf{Num}\ {-5}) : \mathsf{int}}\text{Type-int}$$

But if we try to use the above derivations as the second and third premises of Type-ite, we get stuck:

$$\frac{\emptyset \vdash (\mathsf{Bfalse}) \qquad \dfrac{3 \in \mathbb{Z} \quad 3 \geq 0}{\emptyset \vdash (\mathsf{Num}\ 3) : \mathsf{pos}}\text{Type-pos} \qquad \dfrac{-5 \in \mathbb{Z}}{\emptyset \vdash (\mathsf{Num}\ {-5}) : \mathsf{int}}\text{Type-int}}{\emptyset \vdash (\mathsf{Ite}\ (\mathsf{Bfalse})\ \underbrace{(\mathsf{Num}\ 3)}_{eThen}\ \underbrace{(\mathsf{Num}\ {-5})}_{eElse}) : ???}\text{Type-ite}$$

Type-ite requires the *same* type in each branch. On paper (given enough time to think about it), we can fix this by using Type-sub to "forget" that—in addition to being an integer—$(\mathsf{Num}\ 3)$ is a positive integer. That gives us the same type, int, for both $eThen$ and $eElse$, which allows us to apply Type-ite.

$$\frac{\emptyset \vdash (\mathsf{Bfalse}) \qquad \dfrac{\dfrac{3 \in \mathbb{Z} \quad 3 \geq 0}{\emptyset \vdash (\mathsf{Num}\ 3) : \mathsf{pos}}\text{Type-pos} \qquad \dfrac{}{\mathsf{pos} \texttt{<:}\ \mathsf{int}}\text{Sub-pos-int}}{\emptyset \vdash (\mathsf{Num}\ 3) : \mathsf{int}}\text{Type-sub} \qquad \dfrac{-5 \in \mathbb{Z}}{\emptyset \vdash (\mathsf{Num}\ {-5}) : \mathsf{int}}\text{Type-int}}{\emptyset \vdash (\mathsf{Ite}\ (\mathsf{Bfalse})\ \underbrace{(\mathsf{Num}\ 3)}_{eThen}\ \underbrace{(\mathsf{Num}\ {-5})}_{eElse}) : \mathsf{int}}\text{Type-ite}$$

In the code for `typeof`, we don't have the luxury of thinking about where to use Type-sub. Instead, we always use the *upper bound* of the types of $eThen$ and $eElse$. In effect, `typeof` is implementing a rule that looks like this:

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash eThen : AThen \qquad \Gamma \vdash eElse : AElse}{\Gamma \vdash (\mathsf{Ite}\ e\ eThen\ eElse) : \textit{upper-bound}(AThen, AElse)}\ \text{Type-ite-upperbound}$$

The idea of *upper-bound*$(A1, A2)$ is that it returns a type that is a supertype of both $A1$ and $A2$, that is, if *upper-bound*$(A1, A2) = B$ then $A1$ <: B and $A2$ <: B.

We actually want it to be the *least upper bound*: for example, rat is *an* upper bound of pos and int, because pos <: rat and int <: rat, but it is not the *least* upper bound because int is also a supertype of pos and int.

The earlier version of `upper-bound` just checked whether one of the types ($A1$, $A2$) was a subtype of the other. It worked for pos and rat, because pos is a subtype of int, and int is a subtype of rat. for rat and int, because int is a subtype of rat.

1.3.2 Upper bounds of record types

With records, the subtyping relationship is more complicated: (record x:pos, y:pos) is a subtype of (record x:pos). Also, (record x:pos, z:pos) is a subtype of (record x:pos). But (record x:pos, y:pos) is neither a subtype of (record x:pos, z:pos), nor a supertype of it.

To compute the upper bound of

(record x:pos, y:pos)

and

(record x:pos, z:pos)

we need to take all the fields in common, that is, $\{x, y\} \cap \{x, z\} = \{x\}$; for each of those fields, make a recursive call to find the upper bound of the types. In this example, there is one field in common, x, and it has the same type pos, so we take the upper bound of pos and pos, which is pos.

If we compute the upper bound of

(record x:int, y:pos)

and

(record x:rat, z:pos)

we get (record x:rat), because the upper bound of int and rat is rat.

Exercise 1. Suppose we decided to add a language feature

(Setfield e x $e2$)

that **updates** the field x in record e with $e2$. Setting aside the question of how to define evaluation for Setfield, would width subtyping and depth subtyping still make sense? If not, what kind of subtyping for records would we need instead?

Exercise 2. Does the implementation of `upper-bound` in `a5.rkt` really do what we want? Try to find an example of types $A1$ and $A2$ such that

`(upper-bound A1 A2)`

returns `#false` even though, according to the subtyping rules on a5, there is a type B such that $A1$ <: B and $A2$ <: B.

If you find an example that involves record types, try to find another example that does not.

2 Downcasts

A Downcast is an odd expression; certainly, its typing rule (Type-downcast) is odd. It says that if e has some type B, then $(\mathsf{Downcast}\ A\ e)$ has type A. It checks that A is a subtype of B. But that's the *opposite* of Type-sub!

$$\frac{A <: B \qquad \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{Downcast}\ A\ e) : A}\ \text{Type-downcast} \qquad \frac{env; S1 \vdash e \Downarrow v; S2 \qquad \emptyset \vdash v : A}{env; S1 \vdash (\mathsf{Downcast}\ A\ e) \Downarrow v; S2}\ \text{SEnv-downcast}$$

When $(\mathsf{Downcast}\ A\ e)$ is evaluated, we check, *during evaluation*, that the value v resulting from the expression e inside $(\mathsf{Downcast}\ A\ e)$ actually does have type A. If v doesn't have type A, then no evaluation rule applies, and the interpreter raises an error.

This is motivated by the following example. Suppose we had strings, with an expression $(\mathsf{Idx}\ eStr\ eIdx)$ that indexes into $eStr$. The index $eIdx$ must evaluate to $(\mathsf{Num}\ n)$, and n must be (1) an integer, (2) positive ($n \geq 0$), and (3) less than the length of the string that $eStr$ evaluates to. Since we have a type specifically for positive integers, pos, conditions (1) and (2) can be enforced in the typing rule for idx via a premise $\Gamma \vdash eIdx : \mathsf{pos}$.

$$\frac{\Gamma \vdash eStr : \mathsf{string} \qquad \Gamma \vdash eIdx : \mathsf{pos}}{\Gamma \vdash (\mathsf{Idx}\ eStr\ eIdx) : \mathsf{string}}\ \text{Type-idx}$$

But suppose we have, in our Fun program, an identifer x of type int, and we want to use x to index into a string. We can use Ite to check whether x is positive (the "else" branch in the expression shown), so checks (1) and (2) in the interpreter will always succeed.

$$\begin{aligned} x : \mathsf{int}, s : \mathsf{string} \vdash \Big(\mathsf{Ite}\ &\big(\mathsf{Binop} < (\mathsf{Id}\ x)\ (\mathsf{Num}\ 0)\big) \\ &(\mathsf{Str}\ \texttt{"bad"}) \\ &\big(\mathsf{Downcast}\ \mathsf{pos}\ (\mathsf{Idx}\ (\mathsf{Id}\ s)\ (\mathsf{Id}\ x))\big)\Big) : \mathsf{string} \end{aligned}$$

Unfortunately, the typing rule with premise $\Gamma \vdash eIdx : \mathsf{pos}$ doesn't let us use $(\mathsf{Id}\ x)$ as that index, because all we know is that x has type int, not that x has type pos.

We can use Downcast to make this work:

$$\big(\mathsf{Idx}\ (\mathsf{Id}\ s)\ (\mathsf{Downcast}\ \mathsf{pos}\ (\mathsf{Id}\ x))\big)$$

The downcast check $\emptyset \vdash v : \mathsf{pos}$ will always succeed, because $\big(\mathsf{Binop} < (\mathsf{Id}\ x)\ (\mathsf{Num}\ 0)\big)$ must have evaluated to (Bfalse).