CPSC 311: Definition of Programming Languages: Subtyping: Records

23-records

DRAFT

Joshua Dunfield
University of British Columbia

November 26, 2015

1 Records

This is meant as an interim (hopefully) addition to the scanned notes

`http://www.ugrad.cs.ubc.ca/~cs311/2015W1/notes/scan-2015-11-20.pdf`

1.1 Record syntax

Given on 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 really 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 with, 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, 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

(on the second scanned page)

All we can do with a record is access a field using $(\mathsf{dot}\ e\ y)$. 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 $x : \mathsf{Pos}$, it should be okay to pass a record with additional fields.

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

$$(\mathsf{Record}\ x{:}\mathsf{Pos}, y{:}\mathsf{Pos}) <: (\mathsf{Record}\ x{:}\mathsf{Pos})$$

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 (Record), which is the type of (record), the record with no fields: it's a little like Java's `Object`.

1.3 Depth subtyping

(on the third scanned page)

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).

The scanned page shows an attempt to pass a record of type $(\mathsf{Record}\ y{:}\mathsf{Pos})$ to a function that expects a $(\mathsf{Record}\ y{:}\mathsf{Rat})$. According to depth subtyping, this is allowed because $\mathsf{Pos} <: \mathsf{Rat}$.

[TO DO: add explanation of ite, and why I had to extend the `upper-bound` function in the implementation.]

My previous implementation of `upper-bound` was only useful when one of the types was a subtype of the other. It worked fine for Pos and Rat, because Pos is a subtype of Rat, and also worked fine for Rat and Int, because Int is a subtype of Rat.

With records, the subtyping relationship is more complicated: $(\mathsf{Record}\ x{:}\mathsf{Pos}, y{:}\mathsf{Pos})$ is a subtype of $(\mathsf{Record}\ x{:}\mathsf{Pos})$. Also, $(\mathsf{Record}\ x{:}\mathsf{Pos}, z{:}\mathsf{Pos})$ is a subtype of $(\mathsf{Record}\ x{:}\mathsf{Pos})$. But $(\mathsf{Record}\ x{:}\mathsf{Pos}, y{:}\mathsf{Pos})$ is neither a subtype of $(\mathsf{Record}\ x{:}\mathsf{Pos}, z{:}\mathsf{Pos})$, nor a supertype of it.

To compute the upper bound of

$$(\mathsf{Record}\ x{:}\mathsf{Pos}, y{:}\mathsf{Pos})$$

and

$$(\mathsf{Record}\ x{:}\mathsf{Pos}, z{:}\mathsf{Pos})$$

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

$$(\text{Record } x{:}\text{Int}, y{:}\text{Pos})$$

and

$$(\text{Record } x{:}\text{Rat}, z{:}\text{Pos})$$

we get $(\text{Record } x{:}\text{Rat})$, because the upper bound of Int and Rat is Rat.

2 Downcasts

(on the fourth scanned page)

A downcast is an odd expression; certainly, its typing rule (Type-downcast) is odd. It says that if e has some type B, then $(\text{downcast } A\ e)$ has type A. No connection between A and B is required.

However, when $(\text{downcast } A\ e)$ is evaluated—I'm not bothering to write the environment or store in this rule, but they should be there in the rule in the a4 handout—we check, *during evaluation*, that the value v resulting from the expression e inside $(\text{downcast } A\ e)$ 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 example at the bottom. Suppose we had strings, with an expression $(\text{idx } eStr\ eIdx)$ that indexes into $eStr$. The index $eIdx$ must evaluate to $(\text{num } n)$, and n must be (1) an integer, (2) positive, 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 $\Gamma \vdash eIdx : \text{Pos}$. (I was too lazy to write the "$\Gamma \vdash$".)

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.

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

Using downcast, the workaround is to wrap $(\text{id } x)$ in a downcast—the scanned page is supposed to say

$$\big(\text{idx } (\text{str } \texttt{"abc"})\ (\text{downcast Pos } (\text{id } x))\big)$$

The downcast check $\emptyset \vdash v : \text{Pos}$ will always succeed, because `(< x 0)` must have evaluated to (bfalse).