Lecture:
Records and Variants

15-312: Foundations of Programming Languages
Jason Reed (jcreed+@cs.cmu.edu)

November 2, 2004

1 Records

One disadvantage of using tuples to aggregate together many pieces of data is that it requires the programmer to use a possibly long — and certainly opaque — sequence of `fst`s and `snd`s to extract the desired component. Even in a realistic language with features like pattern-matching, the programmer using a tuple would need to at least remember in which order all of the elements come. It may be especially hard to remember the meaning of the individual elements if many are of the same type.

A solution to this problem is to introduce *labelled records*. We assume that there is an infinite supply of *labels* ℓ (which we may imagine as, for instance, strings) and extend our language as follows:

$$\begin{array}{rrcll} \textit{Types} & \tau & ::= & \cdots & \mid \{\bar{\ell} : \bar{\tau}\} \\ \textit{Values} & v & ::= & \cdots & \mid \{\bar{\ell} = \bar{v}\} \\ \textit{Expressions} & e & ::= & \cdots & \mid \{\bar{\ell} = \bar{e}\} \\ & & & & \mid \#\ell(e) \end{array}$$

Where $\bar{\ell} : \bar{\tau}$ is shorthand for $\ell_1 : \tau_1, \ldots \ell_n : \tau_n$ and similarly for $\bar{\ell} = \bar{e}$ and $\bar{\ell} = \bar{v}$.

The expression $\{\bar{\ell} = \bar{e}\}$ constructs a record where, for each i the ℓ_i field of the record is assigned the expression e_i. The deconstructor is the record projection $\#\ell(e)$, which is like the tuple projections `fst` and `snd`, except that it requests the named field ℓ from the record expression e. A record expression is a value just in case all of its fields are assigned expressions which happen to be values.

A record is well-typed if all of its components are, and a labelled projection is well-typed if its argument is a record that contains that label. Formally, the typing rules for the new constructs are as follows.

$$\frac{\Gamma \vdash e_i : \tau_i \qquad \text{(for all } i\text{)}}{\Gamma \vdash \{\bar{\ell} = \bar{e}\} : \{\bar{\ell} : \bar{\tau}\}} \qquad \frac{\Gamma \vdash e : \{\bar{\ell} : \bar{\tau}\}}{\Gamma \vdash \#\ell_i(e) : \tau_i}$$

To evaluate a record, we evaluate all of its components in the order they are given. To evaluate a projection, we evaluate the expression it projects. When a projection applies to a record value, we return the value with the label of the projection. We give the formal evaluation rules in small-step style.

$$\frac{e_i \mapsto e_i'}{\{\bar{\ell} = \bar{v}, \ell_i = e_i, \bar{\ell}'' = \bar{e}''\} \mapsto \{\bar{\ell} = \bar{v}, \ell_i = e_i', \bar{\ell}'' = \bar{e}''\}} \qquad \frac{e \mapsto e'}{\#\ell(e) \mapsto \#\ell(e')}$$

$$\frac{}{\#\ell_i(\{\bar{\ell} = \bar{v}\}) \mapsto v_i}$$

It is a natural question to ask how records can be subtyped. In fact, though they are in a sense 'merely' a generalization of tuples, they admit richer subtyping properties. The subtyping that we expect to be able to do from our experience with tuples is still present, and we refer to it as *depth subtyping* of records. The idea is that if we have two record types with the same labels, but of different types, if the types of one record are individually all subtypes of the other's, then the one record type as a whole is a subtype of the other.

$$\frac{\tau_i \leq \tau_i' \qquad \text{(for all } i\text{)}}{\{\bar{\ell} : \bar{\tau}\} \leq \{\bar{\ell} : \bar{\tau}'\}}$$

If we fix a particular record type $\{\bar{\ell} : \bar{\tau}\}$ and a list of coercions $f_1 : \tau_1 \leq \tau_1', \ldots, f_n : \tau_n \leq \tau_n'$, then the coercion that witnesses $\{\bar{\ell} : \bar{\tau}\} \leq \{\bar{\ell} : \bar{\tau}'\}$ can be written as

```
fn record : {lab1 : tau1, ..., labn : taun} =>
 {lab1 : f1(#lab1(record)),
  lab2 : f2(#lab2(record)),
  ...
  labn : fn(#labn(record))}
```

However, remember that $\tau \leq \tau'$ in general can be interpreted as *an expression of type τ can safely be substituted in a place that expects an expression of type τ'*. Since all the information we get out of records is by projecting their fields, it cannot hurt us if the record we are handed has 'too many' fields. As long as it has the fields that we want, more fields besides those are acceptable. Thus a record type that includes more fields than another, without changing any types of the original, is a subtype of it. This is called *width subtyping*, since a record with more fields is seen as 'wider.'

$$\overline{\{\bar{\ell} : \bar{\tau}, \bar{\ell}' : \bar{\tau}'\} \leq \{\bar{\ell} : \bar{\tau}\}}$$

The coercion function for this subtyping can be written as

```
fn record :
  {lab1  : tau1,  ..., labn  : taun,
   lab1' : tau1', ..., labm' : taum'} =>
 {lab1 : #lab1(record),
  lab2 : #lab2(record),
  ...
  labn : #labn(record)}
```

Finally, we would like to be able to say that two record types are isomorphic if they only differ in the order of their fields.

$$\frac{\pi \text{ a permutation of } 1 \ldots n \qquad \ell_{\pi i} = \ell'_i \qquad \tau_{\pi i} = \tau'_i \qquad (\text{for all } i)}{\{\bar{\ell} : \bar{\tau}\} \leq \{\bar{\ell}' : \bar{\tau}'\}}$$

Since if π is a permutation, so too is its inverse π^{-1}, and we have that if one record type is a permuation of the other, then both are subtypes of each other, i.e. isomorphic.

The coercion function (assuming `lab1'`... `labn'` are a permutation of the original labels `lab1`... `labn`) is

```
fn record : {lab1 : tau1,  ..., labn : taun} =>
 {lab1' : #lab1'(record),
  lab2' : #lab2'(record),
  ...
  labn' : #labn'(record)}
```

However, if we look back to the development of the idea of subtyping, we recall that it is important that we don't need to add a separate rule for transitivity. This is because having a transitivity rule would make it unclear how to write an algorithm for checking whether subtyping holds, since we don't know how to guess the 'middle' type that appears in both premises but not in the conclusion.

The rules we have given so far for record subtyping do not admit transitivity as an admissible rule. That is, they themselves don't allow us to derive as many facts as we would be able to if we also had a transitivity rule. For instance, if we had transitivity, we could conclude that $\{x : int, y : int, z : int\} \leq \{y : float, x : int\}$ by using depth subtyping to change y's type from int to float, width subtyping to drop the field z, and permutation subtyping to interchange the positions of x and y. We cannot actually show this with just the three rules above.

To fix this, we introduce a single record subtyping rule that encompasses all three forms of record subtyping at once:

$$\frac{\pi \text{ a permutation of } 1 \ldots n \qquad \ell_{\pi i} = \ell''_i \qquad \tau_{\pi i} \leq \tau''_i \qquad (\text{for all } i)}{\{\bar{\ell} : \bar{\tau}, \bar{\ell'} : \bar{\tau'}\} \leq \{\bar{\ell''} : \bar{\tau''}\}}$$

If we fix a particular record type $\{\bar{\ell} : \bar{\tau}\}$ and a list of coercions $f_1 : \tau_{\pi 1} \leq \tau''_1, \ldots, f_n : \tau_{\pi n} \leq \tau''_n$, and assume that `lab1'`...`labn'` are a permutation of the original labels `lab1`...`labn` via π, then then the coercion that witnesses the resulting fact can be written as

```
fn record : {lab1  : tau1,  ..., labn  : taun,
             lab1' : tau1', ..., labn' : taum'} =>
 {lab1' : f1(#lab1'(record)),
  lab2' : f2(#lab2'(record)),
  ...
  labn' : fn(#labn'(record))}
```

2 Variants

Just as binary sum types express the alternation between two possibilities in a way dual to the combination of two pieces of data of binary product types, we can take records — which combine an arbitrary number of

smaller pieces of data — and describe their dual, the concept of *named variants* which allows for the alternation among arbitrary number of possibilities. Named variants are familiar, if not under that name, to any ML programmer. In ML, a `datatype` declaration that doesn't use polymorphism or recursive types is just a declaration of a named variant. We extend our language as follows:

$$\begin{array}{rrcll} \textit{Types} & \tau & ::= & \cdots & \mid \langle \bar{\ell} : \bar{\tau} \rangle \\ \textit{Values} & v & ::= & \cdots & \mid \langle \ell = v \rangle_{\langle \bar{\ell} : \bar{\tau} \rangle} \\ \textit{Expressions} & e & ::= & \cdots & \mid \langle \ell = e \rangle_{\langle \bar{\ell} : \bar{\tau} \rangle} \\ & & & & \mid \mathtt{case}(e, \bar{x}.\bar{e}) \end{array}$$

where again we use shorthand like $\bar{\ell} : \bar{\tau}$ for $\ell_1 : \tau_1, \ldots \ell_n : \tau_n$ and $\bar{x}.\bar{e}$ for $x_1.e_1, x_2.e_2, \ldots, x_n.e_n$. Like records, variant types are specified by giving a list of labels and types. Here the list means a set of possibilities rather than a set of fields. Consequently, the expressions and values of a variant type only contain one label and one expression. In order for these expressions to have unique types, we add type annotations, just as we did for sum types. Also familiar from sum types is the destructor, a case construct. The difference is that the case has arbitrarily many branches, instead of just two.

A variant expression is well-typed if it contains an expression which is well-typed at the branch of the variant type corresponding to the given label. A case statement is well-typed if the expression being cased over is of variant type, and all the branches, when given a variable of the type of one of the branches of the variant, share the same result type. Formally, the typing rules are:

$$\frac{\Gamma \vdash e : \tau_i}{\Gamma \vdash \langle \ell_i = e \rangle_{\langle \bar{\ell} : \bar{\tau} \rangle} : \langle \bar{\ell} : \bar{\tau} \rangle} \qquad \frac{\Gamma \vdash e : \langle \bar{\ell} : \bar{\tau} \rangle \qquad \Gamma, x_i : \tau_i \vdash e_i : \sigma \quad \text{(for all } i\text{)}}{\Gamma \vdash \mathtt{case}(e, \bar{x} : \bar{e}) : \sigma}$$

The evaluation rules work very much like those for sum types. In both the constructor and the first argument of the deconstructor, as usual, we evaluate expressions until they become values. A case statement applied to a variant value v then takes the appropriate branch, according to the label of v, and the expression of v is substituted for the branch's variable.

$$\frac{e \mapsto e'}{\langle \ell_i = e \rangle_{\langle \bar{\ell} : \bar{\tau} \rangle} \mapsto \langle \ell_i = e' \rangle_{\langle \bar{\ell} : \bar{\tau} \rangle}} \qquad \frac{e \mapsto e'}{\mathtt{case}(e, \bar{x}.\bar{e}) \mapsto \mathtt{case}(e', \bar{x}.\bar{e})}$$

$$\frac{}{\mathtt{case}(\langle \ell_i = v \rangle, \bar{x}.\bar{e}) \mapsto \{v/x\}e_i}$$

A noticeable difference between variants and records is that variants, as presented above, require a type annotation on every variant expression. However, if we introduce subtyping (especially if we are working in a system of bidirectional typechecking) we can relax the notion that every expression must have a unique type, and drop the type annotation. In this case the syntax for expressions and values is simply

$$\begin{array}{rrcll} \textit{Values} & v & ::= & \cdots & \mid \langle \ell = v \rangle \\ \textit{Expressions} & e & ::= & \cdots & \mid \langle \ell = e \rangle \\ & & & & \mid \mathtt{case}(e, \bar{x}.\bar{e}) \end{array}$$

and the typing rules and evaluation rules are modified by dropping type annotations wherever they appear.

Now the subtyping principles that hold for variants are dual to those that hold for records. We have again depth subtyping for variants:

$$\frac{\tau_i \leq \tau_i' \qquad \text{(for all } i\text{)}}{\langle \bar{\ell} : \bar{\tau} \rangle \leq \langle \bar{\ell} : \bar{\tau}' \rangle}$$

and also subtyping by permutation:

$$\frac{\pi \text{ a permutation of } 1 \ldots n \qquad \ell_{\pi i} = \ell_i' \qquad \tau_{\pi i} = \tau_i' \qquad \text{(for all } i\text{)}}{\langle \bar{\ell} : \bar{\tau} \rangle \leq \langle \bar{\ell}' : \bar{\tau}' \rangle}$$

The rule for width subtyping, however, is reversed. While a record with more fields contains more information, (and hence is a subtype of a record with a subset of its fields) a variant with more branches conveys less information: with more possibilities, we are less certain what branch is present. So a variant with more branches is a supertype of a variant type with fewer.

$$\frac{}{\langle \bar{\ell} : \bar{\tau} \rangle \leq \langle \bar{\ell} : \bar{\tau}, \bar{\ell}' : \bar{\tau}' \rangle}$$

By similar reasoning as before, we actually want to combine all three rules into one so that transitivity is admissible instead of necessary as a separate rule. The single rule for variant subtyping is as follows.

$$\frac{\pi \text{ a permutation of } 1 \ldots n \qquad \ell_{\pi i} = \ell_i'' \qquad \tau_{\pi i} \leq \tau_i'' \qquad \text{(for all } i\text{)}}{\langle \bar{\ell} : \bar{\tau} \rangle \leq \langle \bar{\ell}'' : \bar{\tau}'', \bar{\ell}' : \bar{\tau}' \rangle}$$