A constraint-based presentation of rows

(work in progress)

Outline

1. What are rows good for?
2. Row unification at a glance
3. Encoding rows as constraints
4. Conclusion

What are rows good for?

Nominal record types

The C approach:	`struct foo {`	
	`  int l; int m;`	l, m *arbitrary* field labels
	`};`	introduces the type `struct foo`
The ML approach:	`type foo = {`	
	`  int l; int m`	l, m *not* used so far
	`};`	introduces the type `foo`
ML type inference:	$\{l = 1; m = 2\}$	has type `foo`
	$\lambda r.(r.l + r.m)$	has type `foo` $\rightarrow$ `int`

Structural record types, without rows

In the structural approach, a record type lists all fields present in the record, together with their types:

$$\{l = 1; m = 1\} : \{l : \mathsf{int}; m : \mathsf{int}\}$$

Distinct record types may now have common fields. However, giving a *useful* principal type to the field access operator requires *subtyping*:

$$(.l) : \forall\alpha.\{l : \alpha\} \to \alpha$$

Record update, extension, restriction, or concatenation *cannot* be given principal type schemes.

Introducing rows

Rows are *finite* descriptions of *infinite* (in fact, cofinite), labelled families of types:

$$\tau ::= \ldots \mid l : \tau ; \tau \mid \partial \tau$$

Information about the presence or absence of fields is not encoded within the row's domain; instead, it is made explicit in the types.

$$\tau ::= \ldots \mid \mathsf{Abs} \mid \mathsf{Pre}\, \tau$$

Sorting rules prevent forming meaningless types such as $l : \mathsf{Pre}\, \mathsf{int} ; l : \mathsf{Abs} ; \alpha$ or $l : \mathsf{int} ; \alpha$.

Structural record types, with rows

The simplest built-in constant is the *empty* record:

$$\{\partial \mathsf{Abs}\}$$

Field access uses row polymorphism in a simple way:

$$(.l) : \forall \alpha\varphi.\{l : \mathsf{Pre}\,\alpha; \varphi\} \rightarrow \alpha$$

Record extension uses two occurrences of a single row variable:

$$(.l\leftarrow) : \forall \alpha\beta\varphi.\{l : \beta; \varphi\} \rightarrow \alpha \rightarrow \{l : \mathsf{Pre}\,\alpha; \varphi\}$$

Record creation is a derived form:

$$\{l = 1; m = 1\} : \{l : \mathsf{Pre}\,\mathsf{int}; m : \mathsf{Pre}\,\mathsf{int}; \partial \mathsf{Abs}\}$$

What are rows good for?

Row unification at a glance

Row unification is first-order unification...

A *multi-equation* is a multi-set of types. A *unification problem*, also called a *constraint*, is a collection of multi-equations, which may involve local names.

$$e ::= \varnothing \mid \tau = e$$

$$C ::= e \mid C \wedge C \mid \exists \alpha . C$$

Unification is presented as a set of *rewriting* rules on unification problems. Standard rules are *fusion*:

$$\frac{\alpha = e_1 \wedge \alpha = e_2}{\alpha = e_1 = e_2}$$

and *decomposition*:

$$\frac{(l : \tau_1 ; \tau_2) = (l : \tau_1' ; \tau_2') = e}{\tau_1 = \tau_1' \wedge \tau_2 = \tau_2' \wedge (l : \tau_1' ; \tau_2') = e} \qquad \frac{\partial \tau = \partial \tau' = e}{\tau = \tau' \wedge \partial \tau' = e}$$

...modulo an equational theory

One non-standard rule allows labels to *commute*:

$$\frac{(l : \tau_1 ; \tau_2) = (m : \tau_1' ; \tau_2') = e \qquad l \neq m}{\exists \alpha . [\tau_2 = (m : \tau_1' ; \alpha) \wedge \tau_2' = (l : \tau_1 ; \alpha)] \wedge (l : \tau_1 ; \tau_2) = e}$$

The symmetry in this rule can be broken by requiring $l < m$ according to some fixed, arbitrary order on labels.

Another allows extracting information out of a *uniform* row:

$$\frac{(l : \tau_1 ; \tau_2) = \partial \tau' = e}{\tau_1 = \tau' \wedge \tau_2 = \partial \tau' \wedge \partial \tau' = e}$$

Encoding rows as constraints

Towards a simpler and more efficient presentation

Premise: assume we are interested in a type system with *subtyping*. Then, the constraint solver is no longer based on unification.

Consequence: it is no longer essential to represent rows as *terms*.

Idea: by encoding rows into *constraints*, it should be possible to

- view constraint resolution as a *closure* computation, rather than as a *rewriting* process—in particular, no fresh variables will be generated;
- expose sets of labels so as to allow the use of a more efficient sorting algorithm.

A constraint language

We suppress row terms of the form $(l : \tau; \tau)$, so only row variables and uniform rows remain. Let R range over (finite or cofinite) sets of row labels.

We introduce two constraint forms:

$$\begin{array}{rcll} C & ::= & \tau \leq \tau & \text{where both sides are types} \\ & \mid & R : \tau \leq \tau & \text{where both sides are rows} \end{array}$$

Typing records with row constraints

The type of the *empty* record is unchanged:

$$\{\partial \mathsf{Abs}\}$$

Field access uses a row constraint to extract the desired field:

$$(.l) : \forall \alpha\varphi[l : \varphi \leq \partial(\mathsf{Pre}\,\alpha)].\{\varphi\} \to \alpha$$

Record extension uses two row constraints:

$$(.l\leftarrow) : \forall \alpha\varphi_1\varphi_2[l : \partial(\mathsf{Pre}\,\alpha) \leq \varphi_2 \wedge \bar{l} : \varphi_1 \leq \varphi_2].\{\varphi_1\} \to \alpha \to \{\varphi_2\}$$

Record creation is a derived form:

$$\{l = 1; m = 1\} : \forall \varphi[l : \partial(\mathsf{Pre\,int}) \leq \varphi \wedge m : \partial(\mathsf{Pre\,int}) \leq \varphi \wedge \overline{lm} : \partial\mathsf{Abs} \leq \varphi].\{\varphi\}$$

Solving row constraints

The usual transitivity rule is supplemented with a new one for row constraints:

$$\begin{array}{llcl}
(\textsc{Trans}) & \tau_1 \leq \alpha \wedge \alpha \leq \tau_2 & \rightarrow & \tau_1 \leq \tau_2 \\
(\textsc{Trans-Row}) & R_1 : \tau_1 \leq \alpha \wedge R_2 : \alpha \leq \tau_2 & \rightarrow & R_1 \cap R_2 : \tau_1 \leq \tau_2
\end{array}$$

Constraints on types can produce constraints on rows, and vice-versa:

$$\begin{array}{lrcl}
(\textsc{Prop-Type-Row}) & \{\tau_1\} \leq \{\tau_2\} & \rightarrow & \mathcal{R} : \tau_1 \leq \tau_2 \\
(\textsc{Prop-Row-Type}) & R : \partial\tau_1 \leq \partial\tau_2 & \rightarrow & \tau_1 \leq \tau_2 \\
 & & & \text{if } R \neq \varnothing
\end{array}$$

Note that *no* new variables or terms are generated.

Complexity

A conjunction $R : \tau_1 \leq \tau_2 \wedge R' : \tau_1 \leq \tau_2$ can be eagerly rewritten $R \cup R' : \tau_1 \leq \tau_2$.

Sets of row labels R can be represented as balanced trees, so union and intersection operations can be performed reasonably efficiently.

Constraint satisfiabillity can be decided in time $O(n^3 m \log m)$, where n is the size of the constraint and m the number of row labels which appear in it.

Algorithm: data structures

An *edge* e is a pair of terms $\tau_1 \leq \tau_2$.

The *current constraint* C is a mapping from *Type* edges to weights of the form $0 \mid 1$ and from *Row* edges to weights of the form R.

The *waiting queue* Q is a list of constraints whose consequences remain to be discovered.

Invariant: the constraints in Q are in C already; constraints in Q have non-null weights; constraints in Q do *not* overlap with one another or with C.

Algorithm $\textsc{Insert}(c)$

if c is e **and** $C(e) = 0$ Cost: $O(1)$

$$\textbf{then} \begin{cases} C(e) \leftarrow 1 \\ Q \leftarrow Q, c \end{cases}$$

else if c is $R : e$ Cost: $O(1 + |R| \log m)$

$$\textbf{then} \begin{cases} \textbf{let } R_1 = R \setminus C(e) \\ \textbf{if } R_1 \neq \varnothing \\ \textbf{then } \begin{cases} C(e) \leftarrow R \cup C(e) \\ Q \leftarrow Q, R_1 : e \end{cases} \end{cases}$$

else if c is **false then fail**

Algorithm $\textsc{Process}(c)$ (excerpt)

$$
\begin{array}{l}
\textbf{if } c \text{ is } R_1 : \tau_1 \leq \alpha \\
\textbf{then for each } \tau_2 \\
\quad \textbf{do } \begin{cases} \textbf{let } R_2 = C(\alpha \leq \tau_2) \\ \textsc{Insert}(R_1 \cap R_2 : \tau_1 \leq \tau_2) \end{cases}
\end{array}
$$

The cost of the inner **do** statement is $O(1 + |R_1| \log m)$ when R_1 is finite, and $O(m \log m)$ otherwise.

The cumulative cost when R_1 varies remains $O(m \log m)$.

The cumulative cost when τ_1, τ_2 and α vary is $O(n^3 m \log m)$.

Towards generalizations of rows?

The algorithm only requires filters R to support union, intersection, difference, and a test for emptiness. So,

- filters need not be sets of atoms (row labels);
- even if they are, they need not be given *in extenso*.

This suggests generalizations along two directions:

- by changing the *nature* of filters: e.g. sets of fixed-length words, sets of arbitrary-length words, sets of trees, etc.;
- by changing the *language* of filters: e.g. assuming a user-extensible ordering on atoms, provide syntax for the set of all atoms below a given atom.

Conclusion

Conclusion

- A new constraint-based presentation of rows, which mixes more easily with algorithms for solving subtyping constraints, and has perhaps more potential for interesting extensions.
- A complexity analysis of the combination of subtyping, rows and (in the paper) conditional constraints.