A constraint-based presentation and generalization of rows

Outline

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

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` $\to$ `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. The field access operator may (must!) be given the following type scheme:

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

However, unless *subtyping* is present, this is essentially useless. Furthermore, record update, extension, restriction, or concatenation *cannot* be given principal type schemes.

Introducing rows (Wand, Rémy)

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 $l : \mathsf{int}; \alpha$. Subtyping is optional.

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:

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

Record extension uses two occurrences of a single row variable:

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

Record creation is a derived form:

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

Row unification is first-order unification...

A *multi-equation* e is a multi-set of types, written $\tau_1 = \ldots = \tau_n$. *Unification problems* are built out of multi-equations, conjunction, and existential quantification. Unification may be viewed as *rewriting* of unification problems. Standard rules are *fusion*:

$$\alpha = e_1 \wedge \alpha = e_2 \quad \rightarrow \quad \alpha = e_1 = e_2$$

and *decomposition*:

$$(l : \tau_1; \tau_2) = (l : \tau_1'; \tau_2') = e \quad \rightarrow \quad \tau_1 = \tau_1' \wedge \tau_2 = \tau_2' \wedge (l : \tau_1'; \tau_2') = e$$
$$\partial\tau = \partial\tau' = e \quad \rightarrow \quad \tau = \tau' \wedge \partial\tau' = e$$

...modulo an equational theory

Two nonstandard rules reflect our interpretation of rows as families of types. One allows row labels to *commute*:

$$\begin{array}{lll} & (l : \tau_1; \tau_2) = (m : \tau_1'; \tau_2') = e & \\ \rightarrow & \exists\alpha.[\tau_2 = (m : \tau_1'; \alpha) \wedge \tau_2' = (l : \tau_1; \alpha)] \wedge (l : \tau_1; \tau_2) = e & (l \neq m) \end{array}$$

The other allows extracting information out of a *uniform* row:

$$\begin{array}{ll} & (l : \tau_1; \tau_2) = \partial\tau' = e \\ \rightarrow & \tau_1 = \tau' \wedge \tau_2 = \partial\tau' \wedge \partial\tau' = e \end{array}$$

Towards a simpler presentation

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

Consequence: union/find no longer plays a central role. It is no longer essential to represent rows as *terms*.

Idea: how about encoding rows as part of *constraints*, and viewing constraint resolution as a *closure* process, rather than as a *rewriting* process? Because a closure computation does not allocate fresh variables, the complexity analysis would be simplified.

A constraint language

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

We introduce two basic constraint forms:

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

If τ_1 and τ_2 are rows, the interpretation of $L : \tau_1 \leq \tau_2$ is: *for every* $l \in L$, the types found in τ_1 and τ_2 *at index* l must be in the subtype relation. Thus, a row constraint encodes a (finite or cofinite) conjunction of type constraints.

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:

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

Record creation is a derived form:

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

Solving row constraints

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

$$\tau_1 \leq \alpha \wedge \alpha \leq \tau_2 \quad \rightarrow \quad \tau_1 \leq \tau_2$$
$$L_1 : \tau_1 \leq \alpha \wedge L_2 : \alpha \leq \tau_2 \quad \rightarrow \quad L_1 \cap L_2 : \tau_1 \leq \tau_2$$

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

$$\{\tau_1\} \leq \{\tau_2\} \quad \rightarrow \quad \mathcal{L} : \tau_1 \leq \tau_2$$
$$L : \partial\tau_1 \leq \partial\tau_2 \quad \rightarrow \quad \tau_1 \leq \tau_2 \qquad \text{if } L \neq \varnothing$$

No new variables or terms are generated. For efficiency, one may add

$$L : \tau_1 \leq \tau_2 \wedge L' : \tau_1 \leq \tau_2 \quad \rightarrow \quad L \cup L' : \tau_1 \leq \tau_2$$

Constraint satisfiability may 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 that appear in it.

Towards generalizations of rows?

The algorithm only requires filters L 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.

Contributions

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