The Cream Type System

Thierry Martinez

Acknowledgments to Julien Martin

Contraintes Project–Team
INRIA Paris–Rocquencourt Research Centre

Wednesday 18th November 2009

1. Introduction
2. The Cream Type System
3. A Hindley-Milner based type inference algorithm
4. Conclusion
The Cream Modelling Language

- A rule-based language for modelling satisfaction and optimisation problems. [RAC’09]
- The language enjoys directives for the declarative specification of search-heuristics. [CPAIOR’09]
- Very few data structures: fd variables, integer constants, lists and records.
- Aggregators over lists. No recursion.


The Cream Modelling Language

Object definitions

```haskell
queen(I) = { row = I, column = _ }.
board(N) = map(I, [1 .. N], queen(I)).
```

Rule declarations

```haskell
no_attack(Q0, Q1) -->
    Q0:column # Q1:column
    and Q0:row - Q0:column # Q0:row - Q1:column
    and Q0:row + Q0:column # Q0:row + Q1:column.
no_attack(L) -->
    forall(Q0 in L,
        forall(Q1 in L,
            Q0:row < Q1:row => no_attack(Q0, Q1))).
```

Query

```haskell
? let(N = 10,
    Board = board(N),
    domain(Board, 1, N) and no_attack(Board)
    and labelling(Board)).
```
Cream Type Constructors

- **int** the type of integer constants and finite domain variables
  - 1, 10, I, N :: int

- **bool** the type of constraints and rules (truth values)
  - Q0:column # Q1:column :: bool
  - Q0:row # Q1:row => no_attack(Q0, Q1) :: bool
  - 1 :: bool

- **[ \( \tau \)]** the type of lists with elements of type \( \tau \) (homogeneous lists)
  - [1 .. N] :: [int]

- **\{ f_1: \( \tau_1 \), ..., f_n: \( \tau_n \)\}** the type of records with
  a field \( f_1 \) carrying a value of type \( \tau_1 \), ..., 
a field \( f_n \) carrying a value of type \( \tau_n \)
  - queen(I) = \{ row = I, column = _ \}.
    queen(\( \alpha \)) :: \{ row: \( \alpha \), column: \( \beta \}\}
  - board(N) = map(I, [1 .. N], queen(I))
    board(int) :: [{ row: int, column: \( \alpha \)}]
What Kind Of Guarantees Could Be Expected?

- Type consistency in a call:
  
  \[
  \text{board}(\text{int}) :: \{ \text{row: int, column: } \alpha \}\]
  
  \text{board}([1 .. 10]) should fail to type.

- Projection validity: \text{queen}(\alpha) :: \{ \text{row: } \alpha, \text{column: } \beta \}
  
  queen(N):column should fail to type.

- Object construction validity:
  
  \text{no_attack}(\{ \text{row: int, column: int } \}) :: \text{bool}
  
  no_attack(\{\text{line = 4, column = 2}\}) should fail.

- But: a well-typed Cream program can go wrong (with respect to the rewriting system \(\rightarrow\))
  
  \text{nth}(1, []) is well-typed but \text{nth}(1, []) \not\rightarrow."
What is a type judgement?

The type system associates a type to every (well-typed) expression (e.g. $1 + 1 :: \text{int}$).

Expressions may depend on a context of bound variables (arguments of a definition, let-binding, iterators). In the general case, a type judgment is a relation between:

- a type environment $\Gamma$: a mapping between bound variables and their type. (e.g. $I :: \text{int}$, $L :: [\text{int}]$)
- an expression $e$
- the type $t$ of $e$ under $\Gamma$

A type judgement is denoted:

$$\Gamma \vdash e :: t$$

e.g. $I :: \text{int} \vdash [I] :: [\text{int}]$
Cream Typing Rules

Basis:

- Integer constants
  \[ \Gamma \vdash n :: \text{int} \geq 2 \quad n \in \mathbb{N} \]
  \[ \Gamma \vdash 0 :: \text{bool} \]
  \[ \Gamma \vdash 1 :: \text{bool} \]

- Bound variables, FD variables, empty lists.
  \[ X :: \tau, \Gamma \vdash X :: \tau \]
  \[ \Gamma \vdash X :: \text{int} \]
  \[ X \notin \Gamma \]
  \[ \Gamma \vdash [] :: [\tau] \text{ type} \]

Inductive steps: hypotheses are on top of the line, conclusions on bottom

\[ \Gamma \vdash X_1 :: \tau \cdots \Gamma \vdash X_n :: \tau \]
\[ \Gamma \vdash [X_1, \ldots, X_n] :: [\tau] \]

\[ \Gamma \vdash \{f_1 = X_1, \ldots, f_n = X_n\} :: \{f_1 :: \tau_1, \ldots, f_n :: \tau_n, \text{uid} :: \text{int}\} \]

\[ \Gamma \vdash e_1 :: \text{int} \quad \Gamma \vdash e_2 :: \text{int} \]
\[ \Gamma \vdash [e_1 \ldots e_2] :: [\text{int}] \]

\[ \Gamma \vdash e :: \{f_1 :: \tau_1, \ldots, f_n :: \tau_n\} \]
\[ \Gamma \vdash e :: f_i :: \tau_i \]
Type Coercions

We make coercions explicit with a new syntactic construction: $\mu$

- Reification: `bool` is a subtype of `int`

\[
\Gamma \vdash e :: bool \\
\Gamma \vdash \mu_{bool \rightarrow int}(e) :: int
\]

- Projection: \{f: $\tau$\} is a subtype of \{f: $\tau$, g: $\tau'$\}

\[
\Gamma \vdash e :: \{f_1 : \tau_1, ..., f_n : \tau_n\} \\
\Gamma \vdash \mu_\pi(e) :: \{f_{\pi_1} : \tau_{\pi_1}, ..., f_{\pi_k} : \tau_{\pi_k}\}
\]
Typing Definitions and Calls

Typing a call should be equivalent to type the body of the definition given the arguments as environment:

\[
\Gamma \vdash e_1 :: \tau_1 \cdots \Gamma \vdash e_n :: \tau_n \quad (f(X_1, \ldots, X_n) = e) \in P \quad X_1: \tau_1, \ldots, X_n: \tau_n \vdash e :: \tau\]

\[
\Gamma \vdash f(e_1, \ldots, e_n) :: \tau
\]

Goal: associate to the definition \(f(X_1, \ldots, X_n) = e\) a principal type, that is to say a type valid for this definition that is more general than any other valid type. With subtyping, the principal type between \texttt{bool} and \texttt{int} is \texttt{int}. But there are definitions which can take either \texttt{int}, or \texttt{[int]}, or...: \(\texttt{id(X)} = X\).

Principality comes from the fact that if Cream Typing Rules are such that if a definition can be called with two unrelated type (e.g. \texttt{int}, or \texttt{[int]}), it can be called with any type \(\tau\).

Let \(\alpha, \beta, \ldots\) be a countable set of type variable. A type schema is of the form:

\[
\forall \alpha \beta \ldots (f(\tau_1, \ldots, \tau_n) :: \tau)
\]

Since Cream definition are top-level, all type schema are closed.

Identity definition has type: \(\forall \alpha, (\texttt{id}(\alpha) :: \alpha)\)
Rule declarations and object definition

- **Rule declarations** (and queries) define constraints, the type system enforces that they return bool.

- **Object definition** define data structure, the type system enforces that they don’t return bool.

Consequence: \( f = 1 \) has type \( f :: \text{int} \) (by coercion) whereas \( p \rightarrow 1 \) has type \( p :: \text{bool} \). \( p \) is usable as a predicate, \( f \) isn’t.
Type Unification

Goal: Guess the type of the arguments of a definition.

Use type variable as (yet) unknown type.

\[
X_1 : \alpha_1, \ldots, X_n : \alpha_n \vdash e :: \alpha
\]

\[
\Gamma \vdash f(X_1, \ldots, X_n) = e :: f(\alpha_1, \ldots, \alpha_n) : \alpha
\]

Typing rules enforce equality between types.

Row variables: for (yet) unknown fields of a record.

\[
\Gamma \vdash e :: \{ f : \tau, \rho \}
\]

\[
\Gamma \vdash e : f :: \tau
\]

Generalization to type schema for definitions.
bool → int Coercion in a Hindley-Milner framework

Type constructors value(bool) and value(int).

Generalization of value(bool) to value(α) in type schema.

μbool→int only introduced on predicate arguments and value-let. let (X = (1 = 1), X and X = 1) is ill-typed: the first usage of X has type value(bool) whereas the second has type value(int).
Conclusion

- Early detection of errors
- Coding discipline:
  - Homogeneous data structures
  - Enforces separation between rule declarations and object definitions
- Type inference could be less restrictive on coercions with a Cardelli/Mitchell algorithm.