

# Dependent Types for Standard ML

October 20, 22, 2003  
Type Refinements Seminar  
Presenter: Kaustuv Chaudhuri

1

## Reading list

---

- Index sorts and dependent types
  - Xi and Pfenning, *Dependent Types in Practical Programming*, POPL'99
- Back-end application – bounds-checks
  - Xi and Pfenning, *Eliminating array bound checking through dependent types*, PLDI'98
- Details of algorithms
  - Xi *Dependent Types in Practical Programming*, PhD thesis, 1998

(Discovery: this is excellent bedtime reading.)

## DML design goals

---

Standard obsessions:

- reduction in the space of acceptable programs
- static, decidable (bidirectional) type-checking

Novelties:

- parametric over constraint domain
  - More general than regular tree grammars
- practical considerations
  - sort comprehensions
  - existential quantification

## Constraint Domains

## Sorts

---

$\gamma ::= b \mid \mathbf{1} \mid \gamma_1 \times \gamma_2 \mid \gamma_1 \rightarrow \gamma_2 \mid \{a : \gamma \mid P\}$	sorts
$i, j ::= a \mid \langle \rangle \mid \langle i, j \rangle \mid \mathbf{f}(i)$	objects
$P ::= \top \mid \perp \mid \mathbf{p}(i) \mid i \doteq j \mid P_1 \wedge P_2 \mid P_1 \vee P_2$	propositions
$\phi ::= \cdot \mid \phi, a : \gamma \mid \phi, P$	contexts
$\Phi ::= P \mid \Phi_1 \wedge \Phi_2 \mid P \supset \Phi \mid Qa : \gamma. \Phi$ ( $Q = \forall$ or $\exists$ )	constraints
$\phi \models \Phi$	satisfaction

## Sorts — constraint domains

---

Consider a signature  $\Sigma$  of function symbols and associated index sorts. All functions in  $\Sigma$  must have sort  $\gamma \rightarrow b$ .

A constraint domain over  $\Sigma$  consists of a set of  $\Sigma$  formulas (constraint language) and a  $\Sigma$  structure (constraint realm).

## Example domain – uninterpreted functions

---

Signature contains functions  $f : b_1 \times \dots \times b_n \rightarrow b$ .

$\phi \models P$  is decidable

- Define an aux. judgement  $\phi_p \models [\phi_0](\phi)\Phi$ 
  - $\phi_0$  of the form  $a_1 : b_1, \dots, a_k : b_k$
  - $\phi_p$  of the form  $P_1, \dots, P_n$
  - $(\phi)\Phi$  is the model-theoretic function corresp. to  $\phi \models \Phi$ .

$$(\phi, a : b)\Phi = (\phi)\forall a : b. \Phi$$

$$(\phi, P)\Phi = (\phi)P \supset \Phi$$

$$(\phi, a : \gamma_1 \times \gamma_2)\Phi = (\phi)\forall a_1 : \gamma_1. \forall a_2 : \gamma_2. \Phi[a \mapsto (a_1, a_2)]$$

$$(\phi, a : \{a : \gamma \mid P\})\Phi = (\phi)\forall a : \gamma. P \supset \Phi$$

- Then,  $\phi \models P$  if  $\cdot \models [\cdot](\phi)P$

7

## Strategy $\phi_0 \models [\phi_p]P$

---

$$\frac{}{\cdot \models [\phi_0, a : \gamma]a \doteq a} \overset{\cdot}{\text{var}} \quad \frac{\text{for } m \in \{1, \dots, n\}, \cdot \models [\phi_0]i_m \doteq j_m}{\cdot \models [\phi_0]f(i_1, \dots, i_n) \doteq f(j_1, \dots, j_n)} \overset{\cdot}{\text{fn}}$$

$$\frac{\cdot \models [\phi_0, a : \gamma]P}{\cdot \models [\phi_0]\forall a : \gamma.P} \forall_R \quad \frac{P_1 \models [\phi_0]P_2}{\cdot \models [\phi_0]P_1 \supset P_2} \supset_R \quad \frac{\cdot \models [\phi_0]P_i}{\cdot \models [\phi_0]P_1 \vee P_2} \vee_{R_i}$$

$$\frac{P_1, P_2, \phi_p \models [\phi_0]P}{P_1 \wedge P_2, \phi_p \models [\phi_0]P} \wedge_L \quad \frac{P_1, \phi_p \models [\phi_0]P \quad P_2, \phi_p \models [\phi_0]P}{P_1 \vee P_2, \phi_p \models [\phi_0]P} \vee_L$$

$$\frac{\phi_p[a \mapsto i] \models [\phi_0]P[a \mapsto i]}{a \doteq i, \phi_p \models [\phi_0]P} \text{subst}$$

$$\frac{i_1 \doteq j_1, \dots, i_n \doteq j_n, \phi_p \models [\phi_0]P}{f(i_1, \dots, i_n) \doteq f(j_1, \dots, j_n), \phi_p \models [\phi_0]P} \text{fn}$$

8

## Algebraic constraints — example

---

```
datatype term = One | Shift of term
              | Abs of term | App of term * term
(* Sigma ==
 *   lev:sort, zero:lev, next:lev->lev *)
typeref term of lev
with One <| {l:lev} term(next(l))
     | Shift <| {i:lev} term(l) -> term(next(l))
     | Abs <| {l:lev} term(next(l)) -> term(l)
     | App <| {l:lev} term(l) * term(l) -> term(l)

(* closed terms have type term(zero) *)
```

9

## Constraint domain — linear inequalities

---

The constraint language is the set of prenex-quantified linear integer inequalities, a.k.a. Presburger Arithmetic.

- Standard decision procedures exist
- Xi and Pfenning use Fourier-Motzkin elimination
  - Very simple, but not the best in terms of efficiency, completeness, etc.
  - Based on refutation search
  - There will be a talk on this topic later

## Linear integer inequalities

---

Can be used to eliminate array bounds checks in some sample examples.

- + Uses existential types to construct subset types. Eg. `[i:int | i+1 >= 0] int(i)`. Existential types are required to admit runtime consumers (eg. `filter`), and can remove many unnecessary checks.
- Cannot represent all invariants.
  - “Elimination of these checks would require a representation of deep invariants of the algorithm that are not expressible in our type system.”  
(PLDI’98 paper, p252)

11

## What invariants *are* expressible?

---

- Array

$$\Pi s : \text{nat}. \alpha \text{ array}(s)$$

- Integers between  $l$  and  $r$

$$\Sigma n : \{n : \text{nat} \mid l \leq n \leq r\}. \text{int}(n)$$

- Array of integers between  $l$  and  $r$

$$\Pi s : \text{size}. \left( \Sigma n : \{n : \text{nat} \mid l \leq n \leq r\}. \text{int}(n) \right) \text{array}(s)$$

12

## Expressible invariants (contd.)

---

- The “unchecked” subscript function

$$\begin{aligned} \text{sub} &: \Pi s : \text{int}. \Pi i : \{i : \text{int} \mid 0 \leq i < s\}. \\ &\alpha \text{ array}(s) \rightarrow \text{int}(i) \rightarrow \alpha \end{aligned}$$

- Subscripting arrays of integers between  $l$  and  $r$

$$\begin{aligned} A &: \left( \Sigma n : \{n : \text{nat} \mid l \leq n \leq r\}. \text{int}(n) \right) \text{array}(10) \\ \text{sub}[10][3] A &: \Sigma n : \{n : \text{nat} \mid l \leq n \leq r\} \end{aligned}$$

Required showing:

$$s \doteq 10, i \doteq 3 \models 0 \leq i \leq s$$

13

## Representable invariants — lifting

---

The type of “less-than”

$$\begin{aligned} &\Pi i, j : \text{int}. \text{int}(i) \rightarrow \text{int}(j) \rightarrow \\ &\Sigma b : \{b : \text{bool} \mid (i < j \wedge b) \vee (i \geq j \wedge \neg b)\}. \\ &\text{bool}(b) \end{aligned}$$

What proposition do we know in the case arms for  
 $< [10][x] 10 x$ ?

14

# Universal Dependent Types

15

## The language $\text{ML}_0^\Pi(C)$

---

Conservative extension of  $\text{ML}_0$ .

$\delta ::= \dots$	families
$\mu ::= \delta(i) \mid \mathbf{1} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2$	major types
$\tau ::= \mu \mid \Pi a : \gamma. \tau$	types
$p ::= \dots \mid c[\vec{a}](p)$	patterns
$e ::= \dots \mid c[\vec{i}](e) \mid \lambda a : \gamma. e \mid e[i]$	expressions
$v ::= \dots \mid c[\vec{i}](v) \mid \lambda a : \gamma. v$	values
$\theta ::= \dots \mid \theta[a \mapsto i]$	substitutions

16

## Constraint generation

---

Constraints are generated in the type-conversion rule, which in turn tries to solve type congruence  $\phi \vdash \tau_1 \equiv \tau_2$ .

$$\frac{\phi \models i \doteq j}{\phi \vdash \delta(i) \equiv \delta(j)} \quad \frac{\phi \vdash \tau_i \equiv \tau'_i}{\phi \vdash \tau_1 * \tau_2 \equiv \tau'_1 * \tau'_2} \quad * \in \{\times, \rightarrow\}$$

$$\frac{\phi, a : \gamma \vdash \tau \equiv \tau'}{\phi \vdash \Pi a : \gamma. \tau \equiv \Pi a : \gamma'. \tau'}$$

For example,  $\phi \vdash \text{intlist}(a + n + 1) \equiv \text{intlist}(m + n)$  generates the CSP  $\phi \models a + n + 1 \doteq m + n$ .

## Pattern matching

---

Pattern matching can generate new index propositions  $P$ . Given pattern  $p$  of type  $t$ , we construct the typing and index contexts, written  $p \downarrow t \triangleright (\phi; \Gamma)$ :

$$\frac{\overline{x \downarrow \tau \triangleright (\cdot; x : \tau)} \quad \overline{\langle \rangle \downarrow \mathbf{1} \triangleright (\cdot; \cdot)}}{p_i \downarrow \tau_i \triangleright (\phi_i; \Gamma_i)}$$

$$\frac{\langle p_1, p_2 \rangle \downarrow \tau_1 \times \tau_2 \triangleright (\phi_1, \phi_2; \Gamma_1, \Gamma_2)}{c : \Pi \vec{a} : \vec{\gamma}. \tau \rightarrow \delta(i) \quad p \downarrow \tau \triangleright (\phi; \Gamma)}$$

$$\frac{c[\vec{a}](p) \downarrow \delta(j) \triangleright (\vec{a} : \vec{\gamma}, i \doteq j, \phi; \Gamma)}$$

## Pattern matching — example

---

Assume  $\text{cons} : \Pi a : \text{nat}. \text{int} \times \text{intlist}(a) \rightarrow \text{intlist}(a + 1)$ .

Then,

$$\begin{aligned} & \langle \text{cons}[a](\langle x, xs \rangle), ys \rangle \downarrow \text{intlist}(m) \times \text{intlist}(n) \\ \triangleright & (a : \text{nat}, a + 1 \doteq m; x : \text{int}, xs : \text{intlist}(a), ys : \text{intlist}(n)) \end{aligned}$$

## Pattern matching — case arms

---

$$\frac{p \downarrow \tau_1 \triangleright (\phi'; \Gamma') \quad \phi, \phi'; \Gamma, \Gamma' \vdash e : \tau_2}{\phi; \Gamma \vdash p \Rightarrow e : \tau_1 \Rightarrow \tau_2}$$

$$\frac{\phi; \Gamma \vdash p \Rightarrow e : \tau_1 \Rightarrow \tau_2 \quad \phi; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\phi; \Gamma \vdash p \Rightarrow e \mid ms : \tau_1 \Rightarrow \tau_2}$$

$$\frac{\phi; \Gamma \vdash e : \tau_1 \quad \phi; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2}{\phi; \Gamma \vdash \text{case } e \text{ of } ms : \tau}$$

## Index abstraction

---

### Static semantics

$$\frac{\phi, a : \gamma; \Gamma \vdash e : \tau}{\phi; \Gamma \vdash \lambda a : \gamma. e : \Pi a : \gamma. \tau} \quad \frac{\phi; \Gamma \vdash e : \Pi a : \gamma. \tau \quad \phi \vdash i : \gamma}{\phi; \Gamma \vdash e[i] : \tau[a \mapsto i]}$$

### Dynamic semantics

$$\frac{e \Downarrow v}{\lambda a : \gamma. e \Downarrow \lambda a : \gamma. v} \quad \frac{e \Downarrow \lambda a : \gamma. v}{e[i] \Downarrow v[a \mapsto i]}$$

## Universal dependent types — external language

---

Standard procedure — define a language  $\text{DML}_0(C)$  that isn't explicitly typed, and provide an elaboration to  $\text{ML}_0^\Pi(C)$  that preserves static semantics.

As you have been trained to expect, this elaboration is bidirectional.

Linking elaboration to constraint solving is complicated, but not complex.

## Executive summary of elaboration

---

- Define two elaborations, one natural (1) and one constraint-generating (2):

$$\phi; \Gamma \vdash e \Downarrow \tau \Rightarrow e^* \quad (1)$$

$$\phi; \Gamma \vdash e \Downarrow \tau \Rightarrow [\psi]\Phi \quad (2)$$

The second one is intended for implementation.

- Prove: if (2) and  $\phi \triangleright \theta : \psi$  and  $\phi[\theta] \models \Phi[\theta]$ , then (1) post- $\theta$ .
  - In other words, the constraint-generating elaboration produces a principal solution.
- More detail coming up in subsequent talks.

## Existential Dependent Types

## Summary of motivations

---

1. In order to give a type for `filter`

```
fun filter p nil = nil
  | filter p (h :: t) =
    if p x then h :: filter t else filter t
```

$$\prod_{m : \text{nat. intlist}(n)} \rightarrow \Sigma n : \text{nat. intlist}(n)$$

2. In order to give a type for existing ML code. Turn the type of any existing function  $f : \text{intlist} \rightarrow \text{intlist}$  into:

$$(\Sigma n : \text{nat. intlist}(n)) \rightarrow \Sigma n : \text{nat. intlist}(n)$$

3. Other pragmatic motivations – eg. removing array bounds checks.

25

## The language $\text{ML}_0^{\Pi, \Sigma}(C)$

---

$\tau ::= \dots \mid \Sigma a : \gamma. \tau$	types
$e ::= \dots \mid \langle i, e \rangle \mid \underline{\text{let}} \langle a, x \rangle = e \underline{\text{in}} e' \underline{\text{end}}$	expressions
$v ::= \dots \mid \langle i, v \rangle$	values

Development of the explicitly typed  $\text{ML}_0^{\Pi, \Sigma}(C)$  is straightforward.

Note, the following is prohibited.

$i ::= \dots \mid \underline{\text{let}} \langle a, x \rangle = e \underline{\text{in}} i \underline{\text{end}}$	index objects
---	---------------

In other words, we have  $\pi_2$  but not  $\pi_1$  for  $\Sigma$ .

## Elaboration

---

Elaboration for existential types turns out to be pretty complex, compared to the case for universal types.

Consider

$$\text{rev} : \Pi n : \text{nat. intlist } (n) \rightarrow \text{intlist}(n)$$

Suppose  $\text{rev}(l)$  occurs somewhere in the code, and the argument  $l$  synthesizes at  $\Sigma m : \text{int. intlist } (m)$ .

There is no immediately obvious way to instantiate  $\text{intlist}(N)$  ( $N$  an evar corresponding to the  $\Pi$ ) with  $\Sigma m : \text{int. intlist}(m)$

27

## The DML approach

---

Perform a variant of A-translation.

Turns  $\text{rev}(l)$  into let  $\langle a, x \rangle = l$  in  $\text{rev}[a](x)$  end.

More generally, subexpressions are explicitly wrapped in let-expressions, producing an explicitly staged translation.

$$\ulcorner \langle e_1, e_2 \rangle \urcorner = \underline{\text{let}} \ x = \ulcorner e_1 \urcorner \ \underline{\text{in}} \ \underline{\text{let}} \ y = \ulcorner e_2 \urcorner \ \underline{\text{in}} \ \langle x, y \rangle \ \underline{\text{end}} \ \underline{\text{end}}$$

$$\ulcorner e_1(e_2) \urcorner = \underline{\text{let}} \ x = \ulcorner e_1 \urcorner \ \underline{\text{in}} \ \underline{\text{let}} \ y = \ulcorner e_2 \urcorner \ \underline{\text{in}} \ x(y) \ \underline{\text{end}} \ \underline{\text{end}}$$

etc.

28

## Why this approach doesn't work

---

In

let  $x = e_1$  in let  $y = e_2$  in  $x(y)$  end end

both  $e_1$  and  $e_2$  must synthesize.

Yet, in a bidirectional setting, applications  $e_1(e_2)$  require that  $e_2$  merely *check* against a type.

Thus, A-translation disallows more programs than  $\text{ML}_0^{\Pi, \Sigma}(C)$  can handle!

## The DML approach

---

Don't worry about it.

## The DML approach — actually

---

$$\lceil e_1(e_2) \rceil = \begin{cases} \underline{\text{let}}\ x_1 = \lceil e_2 \rceil\ \underline{\text{in}}\ x_1(e_2)\ \underline{\text{end}} & \text{if } e_2 \text{ is valuable} \\ \underline{\text{let}}\ x = \lceil e_1 \rceil\ \underline{\text{in}} \\ \quad \underline{\text{let}}\ y = \lceil e_2 \rceil\ \underline{\text{in}}\ x(y)\ \underline{\text{end}} & \text{otherwise} \\ \underline{\text{end}} \end{cases}$$

Works for essentially any program you'd ever think of writing in SML.

31

## Recovering bidirectionality in DML

---

Have *two* different let expressions:

let<sup>↓</sup> : synthesizes a type for the bound var.

let<sup>↑</sup> : checks the type of the bound expression

What's a canonical reference for this approach?

32

## The Dunfield-Pfenning approach

---

The problem is basically that single types aren't enough for type ascriptions, which rises from a tension between the "reach" and the lexical scope of index variables.

The proposed solution is to introduce typing ascriptions, etc.

You've seen it already.

## Future of dependent ML

---

The Dunfield-Pfenning approach makes DML obsolete.

However, the constraint-generating elaboration in DML might have some use for program-analysis.

## Another point

---

I said earlier:

There is no immediately obvious way to instantiate  $\text{intlist}(N)$  with  $\Sigma m : \text{int}. \text{intlist}(m)$

But, that's only because we can't take the first projection of sigma-types.

## The reverse example

---

If  $l$  synthesizes at  $\Sigma n : \text{nat}. \text{intlist}(n)$ , then:

$$\text{rev}[\pi_1 \ l] \ l : \text{intlist}(\pi_1 \ l)$$

## The Dunfield-Pfenning counterexample

---

Start with a strange “identity” function

$$“\lambda x. (\lambda z. x)()” \uparrow \Pi n : \text{nat}. \text{list}(n) \rightarrow \text{list}(n)$$

Since  $\lambda z. x$  is in the function-position of an application, it must synthesize a type. However, it is a  $\lambda$ -abstraction, so it’s the programmer who supplies the type.

It is clear that:

$$n : \text{nat} ; x : \text{list}(n) \vdash \lambda z. x \uparrow \text{unit} \rightarrow \text{list}(n)$$

but the programmer cannot write down “ $\text{list}(n)$ ” because  $n$  is nowhere in the term!

37

## The case of DML

---

Since DML exposes the “index” lambdas, we can write:

$$\lambda n : \text{nat}. \underline{\text{lam}} x. (\underline{\text{lam}} z. x \downarrow 1 \rightarrow \text{list}[n])()$$

Dunfield-Pfenning frown at index lambdas.

- Possibly because such a requirement makes it no longer conservative over the ML term language.

38

Conjecture:

---

$$\lambda x. (\lambda z. x \downarrow \mathbf{1} \rightarrow \mathbf{list}(\pi_1 x))() \uparrow$$
$$\prod n : \mathbf{nat}. (\Sigma m : \{m : \mathbf{nat} \mid m = n\}. \mathbf{list}(m)) \rightarrow \mathbf{list}(n)$$