Dependent Types in Practical Programming

Hongwei Xi, Frank Pfenning
(POPL 99)
Presented by : Bomshik Kim

4/27/2000

□ Contents

□ Introduction

- Type-checking in realistic languages
  - theoretically decidable
  - practically feasible without lots of type annotations

- In this paper
  - designing a type system for practical programming
  - with a restricted form of dependent types
  - without destorying properties of ML

□ Example - append

```
datatype 'a list = nil | cons of 'a * 'a list

typeref 'a list of nat with
  nil <| 'a list (0)
| cons <| {n:nat}'a * 'a list(n) -> 'a list(n+1)


fun('a)
  append(nil,ys) = ys
| append(cons(x,xs),ys) = cons(x,append(xs,ys))
where append <| {m':nat}{n:nat}
      'a list(m) * 'a list(n) -> 'a list(n+m)
```

□ syntax for $\mathbf{ML}_0$

mini-ML with Pattern Matching

base types β	$::=$	bool $\mid$ int
		$\mid$ other user defined datatypes
types σ, τ	$::=$	$\beta \mid 1 \mid \tau * \sigma \mid \sigma \to \tau$
patterns p	$::=$	$x \mid c(p) \mid <> \mid < p_1, p_2 >$
matches ms	$::=$	$(p \Rightarrow e) \mid (p \Rightarrow e \mid ms)$
expressions e	$::=$	$x \mid <> \mid < e_1, e_2 > \mid c(e)$
		$\mid$(case e of ms)$\mid$(lam $x : \tau.e$)
		$\mid e_1(e_2) \mid$let $x = e_1$in e_2 end
		$\mid$(fix $f : \tau.v$)
values v	$::=$	$x \mid c(v) \mid <> \mid < v_1, v_2 >$
		$\mid$(lam $x : \tau.e$)
contexts Γ	$::=$	$\cdot \mid \Gamma, x : \tau$

□ Constraint Domains

index sorts

$\gamma \ ::= \ b|1|\gamma_1 * \gamma_2|\{a : \gamma|P\}$

index propositions

$P \ ::= \ \top|\bot|p(i)|P_1 \vee P_2|P_1 \wedge P_2$

index objects

$i, j \ ::= \ a|()|(i, j)|f(i)$

index contexts

$\phi \ ::= \ \cdot|\phi, a : \gamma|\phi, P$

index constraints

$\Phi \ ::= \ P|\Phi_1 \wedge \Phi_2|P \supset \Phi|\forall a : \gamma.\Phi|\exists a : \gamma.\Phi$

satisfaction relation

$\phi \models \Phi$

Φ is satisfied in the constraint domain under index context ϕ

□ syntax of $\mathbf{ML}_0^{\Pi}$

families δ	$::=$	family of built-in or user-declared refined types
constructor sig S	$::=$	$\cdot \mid S, c : \Pi a_1 : \gamma_1 .. \Pi a_n : \gamma_m . \tau \to \delta(i)$
major types μ	$::=$	$\delta(i) \mid 1 \mid (\tau_1 * \tau_2) \mid (\tau_1 \to \tau_2)$
types τ	$::=$	$\mu \mid (\Pi a : \gamma . \tau)$
patterns p	$::=$	$... \mid c[a_1]...[a_n](p)$
expressions e	$::=$	$... \mid c[i_1]...[i_n](e) \mid (\lambda : \gamma . e) \mid e[i]$
values v	$::=$	$... \mid c[i_1]...[i_n](v) \mid (\lambda : \gamma . v)$
substitutions θ	$::=$	$... \mid \theta[a \mapsto i]$

□ Example

fix $append$: $\Pi m : nat.\Pi n : nat.$
$intlist(m) * intlist(n) \to intlist(m + n).$
$\lambda m : nat.\lambda n : nat.$lam $l : intlist(m) * intlist(n).$
case l of
$< nil, ys >\Rightarrow ys$
$< cons[a](< x, xs >), ys >\Rightarrow$
$cons[a + n](< x, append[a][n](< xs, ys >) >)$

```
fun append[0][n](nil, ys) = ys
 |  append[a+1][n](cons[a](x,xs),ys) =
         cons[a+n](x,append[a][n](xs,ys))
where append <| {m:nat}{n:nat}
      intlist(m) * intlist(n) -> intlist(m+n)
```

□ Type Conversion

1. $$\frac{\phi \models i \doteq i'}{\phi \vdash \delta(i) \equiv \delta(i')}$$

2. $$\frac{\phi \vdash \tau_1 \equiv \tau_1', \phi \vdash \tau_2 \equiv \tau_2'}{\phi \vdash \tau_1 * \tau_2 \equiv \tau_1' * \tau_2'}$$

3. $$\frac{\phi \vdash \tau_1' \equiv \tau_1', \phi \vdash \tau_2 \equiv \tau_2'}{\phi \vdash \tau_1 \rightarrow \tau_2 \equiv \tau_1' \rightarrow \tau_2'}$$

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

Example)
$\phi \models (a + n) + 1 \doteq m + n$
is generated in order to derive
$\phi \vdash intlist((a + n) + 1) \equiv intlist(m + n)$
where ϕ is $m : nat, n : nat, a : nat, a + 1 \doteq m$

□ Typing Rules for patterns

1. $\overline{x \downarrow \tau \triangleright (\cdot; x : \tau)}$

2. $\overline{<>\downarrow 1 \triangleright (\cdot; \cdot)}$

3. $\dfrac{p_1 \downarrow \tau_1 \triangleright (\phi_1; \Gamma_1) \quad p_2 \downarrow \tau_2 \triangleright (\phi_2; \Gamma_2)}{< p_1, p_2 >\downarrow \tau_1 * \tau_2 \triangleright (\phi_1, \phi_2; \Gamma_1, \Gamma 2)}$

4. $\dfrac{S(c) = \Pi \vec{a} : \vec{\gamma}.(\tau \rightarrow \delta(i)) \quad p \downarrow \tau \triangleright (\phi_1; \Gamma)}{c[\vec{a}](p) \downarrow \delta(j) \triangleright (\vec{a} : \vec{\gamma}, i \doteq j, \phi_1; \Gamma)}$

Assume

$cons$: $\Pi a : nat.int * intlist(a) \rightarrow intlist(a + 1)$

$cons[a](< x, xs >), ys) \downarrow intlist(m) * intlist(n)$
$\triangleright\ a : nat, a + 1 = m;$
$x : int, xs : intlist(a), ys : intlist(n)$

□ Erasure Function

mapping an expression in $\mathsf{ML}_0^{\Pi}(C)$ into one in ML_0

$$\|\delta(i_1, ..., i_n)\| = \delta$$
$$\|\tau_1 * \tau_2\| = \|\tau_1\| * \|\tau_2\|$$
$$\|\tau_1 \to \tau_2\| = \|\tau_1\| \to \tau_2\|$$
$$\|\Pi a : \gamma.\tau\| = \|\tau\|$$
$$\|c[i_1]...c[i_n](e)\| = c(\|e\|)$$
$$\|(\mathsf{lam}\ x : \tau.e)\| = (\mathsf{lam}\ x : \|\tau\|.\|e\|)$$
$$\|(\lambda a : \gamma.e)\| = \|e\|$$
$$\|e[i]\| = \|e\|$$
$$\|\mathsf{fix}\ x : \tau.v\| = \mathsf{fix}\ x : \|\tau\|.\|v\|$$
$$\|\Gamma, x : \tau\| = \|\Gamma\|, x : \|\tau\|$$

□ Properties

Theorem 1 (Type preservation)
Given e, v in $ML_0^{\sqcap}(C)$ such that e $\rightarrow_d$ v is derivable. If $\phi; \gamma \vdash e : \tau$ is derivable, then $\phi; \Gamma \vdash v : \tau$ is derivable

Theorem 2
If $\phi; \Gamma \vdash e : \tau$ is derivable in $ML_0^{\sqcap}(C)$, then $\|\Gamma\| \vdash \|e\| : \|\tau\|$ is derivable in ML_0.

Theorem 3 (Soundness)
If e $\rightarrow_d$ v derivable in $ML_0^{\sqcap}(C)$, then $\|e\| \rightarrow_0 \|v\|$ is derivable.

Theorem 4 (Completeness)
Given $\phi; \Gamma \vdash e : \tau$ derivable in $ML_0^{\sqcap}(C)$. If $\|e\| \rightarrow_0 v_0$ is derivable for some v_0 in ML_0, then there exists v in $ML_0^{\sqcap}(C)$ such that $e \rightarrow_d v$ and $\|v\| = v_0$.

Theorem 5
$ML_0^{\sqcap}(C)$ is a conservative extension of ML_0

□ Elaboration

Elaboration : mapping from External Lang to Internal Lang
It's hard for programmer to annotate all types explicitly.

$\mathsf{DML}_0(C) \Longrightarrow \mathsf{ML}_0\Pi$

Synthesizing types($\uparrow$) : W algorithm
Checking types($\downarrow$) : M algorithm

Why are these two typing methods useful?
Why do they use these methods?

$\Rightarrow$ π.π;;)

□ Existential Dependent Types

There are many properties of indices which can't be expressed.

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

We can't know the exact size of result list.
We know there exists some $m \leq n$.
(m : size of output , n : size of input)

So, We need existential types which mediate between dependent and ordinary ML types to handle unknown things.

□ Application

- Program Error Detection

  Check whether given properties is preserved or not

- Dead Code Elimination

  Make more compact and more efficient codes

□ Conclusion

1. allow the programmer to express more program properties through types

2. catch more errors at compile time