Extensions of mini-ML: tuples

Idea:

$$(a_1, a_2, \ldots, a_n) : \tau_1 \times \tau_2 \ldots \times \tau_n$$

Extensions to mini-ML:

Expressions: $a ::= \ldots \mid (a_1, \ldots, a_n) \mid \texttt{proj}_i^n\ a$

Values: $v ::= \ldots \mid (v_1, \ldots, v_n)$

Evaluation contexts: $E ::= \ldots \mid (E, a_2, \ldots, a_n) \mid (v_1, E, \ldots, a_n) \mid \ldots \mid (v_1, v_2, \ldots, v_{n-1}, E) \mid \texttt{proj}_i^n\ E$

Reduction: $\texttt{proj}_i^n\ (v_1, \ldots, v_n) \xrightarrow{\varepsilon} v_i$

Tuples, ctd.

Types: for all integer $n \geq 0$, a type constructor $\times_n$.

Notation: we write $\tau_1 \times \ldots \times \tau_n$ for $\times_n(\tau_1, \ldots, \tau_n)$

Type rules:

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash a_n : \tau_n}{\Gamma \vdash (a_1, \ldots, a_n) : \tau_1 \times \ldots \times \tau_n} \qquad \frac{\Gamma \vdash a : \tau_1 \times \ldots \times \tau_n}{\Gamma \vdash \mathtt{proj}_i^n\ a : \tau_i}$$

When $n = 0$ the product type $\times_0$ contains only one value $()$: this corresponds to the `unit` type of OCaml.

Sums

Idea:

- a value of type $\tau_1 \times \tau_2$ is composed by a value of type τ_1 and by a value of type τ_2;
- a value of type $\tau_1 + \tau_2$ is composed by a value of type τ_1 or by a value of type τ_2;

Example: a value v of type $\texttt{int} + \texttt{string}$ is

- either an integer $\texttt{inj}_1\ 5$,
- or a string $\texttt{inj}_2$ "foo".

To deconstruct it:

$$\texttt{match}\ \ v\ \ (\texttt{fun}\ i \rightarrow \ldots)\ \ (\texttt{fun}\ s \rightarrow \ldots)$$

Sums, formally

Expressions:	$a ::= \ldots \mid \texttt{inj}_i^n\ a \mid \texttt{match}_n\ a\ a_1\ \ldots\ a_n$
Values:	$v ::= \ldots \mid \texttt{inj}_i^n\ v$
Evaluation contexts:	$E ::= \ldots \mid \texttt{inj}_i^n\ E \mid \texttt{match}_n\, E\, a_1\, \ldots\, a_n \mid \texttt{match}_n\, v\, E\, \ldots\, a_n$
	$\mid \ldots \mid \texttt{match}_n\, v\, v_1\, \ldots\, E$

Reduction: $\texttt{match}_n\,(\texttt{inj}_i^n\ v)\,v_1\,\ldots\,v_n \xrightarrow{\varepsilon} v_i\,v$

Sums, ctd.

Types: for all integer $n \geq 0$, a type constructor $+_n$.

Notation: we write $\tau_1 + \ldots + \tau_n$ for $+_n(\tau_1, \ldots, \tau_n)$

Type rules:

$$\frac{\Gamma \vdash a : \tau_i}{\Gamma \vdash \mathtt{inj}_i^n\ a : \tau_1 + \ldots + \tau_n}$$

$$\frac{\Gamma \vdash a : \tau_1 + \ldots + \tau_n \qquad \Gamma \vdash a_1 : \tau_1 \to \tau \qquad \ldots \qquad \Gamma \vdash a_n : \tau_n \to \tau}{\Gamma \vdash \mathtt{match}_n\ a\ a_1\ \ldots\ a_n : \tau}$$

Recursive types

Until here, the universe of types was *finite*. We can relax this constraint, and work with *recursive* types.

Add

$$\tau ::= \ldots \mid \mu\alpha.\tau$$

to the syntax of types and consider types up-to

$$\mu\alpha.\tau \approx \tau[\alpha \leftarrow \mu\alpha.\tau]$$

In the type inference algorithm, the equation $\alpha \stackrel{?}{=} \alpha \to \alpha$ now has a solution: the substitution that associates the regular tree $\mu t.t \to t$ to α.

More on recursive types

```
$ ocaml -rectypes
        Objective Caml version 3.08.1

# fun x -> x x;;
- : ('a -> 'b as 'a) -> 'b = <fun>
```

Too many programs now pass the type checker (for instance, all the terms of the untyped lambda-calculus).

But recursive types might be useful:

$$\text{IntList} = \mathtt{unit} + \mathtt{int} \times \text{IntList}$$

How to reconcile the type inference philosophy and recursive types?

Algebraic types: examples

A concrete type to talk about integers and floats:

```
type num = Integer of int | Real of float
```

The type of points in the space:

```
type point = { x : float; y : float; z : float }
```

The type of arithmetic expressions:

```
type expr = Constant of int
          | Variable of string
          | Add of expr * expr
          | Diff of expr * expr
          | Prod of expr * expr
          | Quotient of expr * expr
```

More examples

We can parametrize an algebraic type:

```
type 'a option = None of unit | Some of 'a
type 'a list = Nil of unit | Cons of 'a * 'a list
type ('a, 'b) pair = { fst : 'a; snd : 'b }
```

- `option` and `list` are not types, but *type constructors* of arity 1, `pair` is a type constructor of arity 2.
- `int list` and `(int, float) pair` are types.

Concrete types

The general form of a concrete type declaration is

$$\texttt{type } (\alpha_1, \ldots, \alpha_p)\ t = C_1 \texttt{ of } \tau_1 \mid \ldots \mid C_n \texttt{ of } \tau_n$$

If $p = 0$ we write type $t = C_1$ of $\tau_1 \mid \ldots \mid C_n$ of τ_n.

We require that for all i, it holds $\mathcal{L}(\tau_i) \subseteq \{\alpha_1, \ldots, \alpha_p\}$.

Concrete type, ctd.

Expressions: $a ::= \ldots \mid C_i\, a \mid \texttt{match}\, a\ C_1{:}a_1\ \ldots\ C_n{:}a_n$

Values: $v ::= \ldots \mid C_i(v)$

Evaluation contexts: $E ::= \ldots \mid C_i(E) \mid \texttt{match}\, E\ C_1{:}a_1 \ldots C_n{:}a_n$
$\mid \texttt{match}\, v\ C_1{:}E \ldots C_n{:}a_n \mid \ldots$

Types: $\tau ::= \ldots \mid (\tau_1, \ldots, \tau_p)\ t$

Reduction:

$$\texttt{match}\,(C_i\, v)\ C_1{:}v_1\ \ldots\ C_n{:}v_n \quad \xrightarrow{\varepsilon} \quad v_i\ v$$

Concrete types, ctd. [2]

Type rules:

$$\frac{\Gamma \vdash a : \varphi(\tau_i) \qquad \mathrm{dom}(\varphi) = \{\alpha_1, \ldots, \alpha_p\}}{\Gamma \vdash C_i\, a : \varphi((\alpha_1, \ldots, \alpha_p)\ t)}$$

$$\frac{\begin{array}{c}\Gamma \vdash a : \varphi((\alpha_1, \ldots, \alpha_p)\ t) \qquad \Gamma \vdash a_1 : \varphi(\tau_1 \to \tau) \qquad \ldots \qquad \Gamma \vdash a_n : \varphi(\tau_n \to \tau) \\ \mathrm{dom}(\varphi) = \{\alpha_1, \ldots, \alpha_p\}\end{array}}{\Gamma \vdash \mathtt{match}\, a\ C_1{:}a_1\ \ldots\ C_n{:}a_n : \varphi(\tau)}$$

where the substitution φ highlights the fact that the type rule is valid for all the instantiations of the parameters $(\alpha_1, \ldots, \alpha_p)$.

Alternative approach: constructors and destructors

For the type `num`, we might define:

$$\begin{aligned}
\texttt{Integer} &: \texttt{int} \to \texttt{num} \\
\texttt{Real} &: \texttt{float} \to \texttt{num} \\
\texttt{match}_{\texttt{num}} &: \forall\beta.\ \texttt{num} \to (\texttt{int} \to \beta) \to (\texttt{float} \to \beta) \to \beta
\end{aligned}$$

For the type α `list`, we might define:

$$\begin{aligned}
\texttt{Nil} &: \forall\alpha.\ \texttt{unit} \to \alpha\ \texttt{list} \\
\texttt{Cons} &: \forall\alpha.\ (\alpha \times \alpha\ \texttt{list}) \to \alpha\ \texttt{list} \\
\texttt{match}_{\texttt{list}} &: \forall\alpha, \beta.\ \alpha\ \texttt{list} \to (\texttt{unit} \to \beta) \to (\alpha \times \alpha\ \texttt{list} \to \beta) \to \beta
\end{aligned}$$

Records

The general form of a concrete type declaration is

$$\texttt{type}\ (\alpha_1, \ldots, \alpha_p)\ t = \{e_1 : \tau_1; \ldots; e_n : \tau_n\}$$

$$\begin{array}{lrl}
\text{Expressions:} & a ::= & \ldots \mid \{e_1 = a_1; \ldots; e_n = a_n\} \mid a.e_i \\
\text{Values:} & v ::= & \ldots \mid \{e_1 = v_1; \ldots; e_n = v_n\} \\
\text{Evaluation contexts:} & E ::= & \ldots \mid \{e_1 = E; \ldots; e_n = a_n\} \mid \ldots \\
 & & \mid \{e_1 = v_1; \ldots; e_n = E\} \mid E.e \\
\text{Types:} & \tau ::= & \ldots \mid (\tau_1, \ldots, \tau_p)\ t
\end{array}$$

Reduction:

$$\{e_1 = v_1; \ldots; e_n = v_n\}.e_i \ \xrightarrow{\varepsilon}\ v_i$$

Records, ctd.

Type rules:

$$\frac{\Gamma \vdash a_1 : \varphi(\tau_1) \qquad \ldots \qquad \Gamma \vdash a_n : \varphi(\tau_n) \qquad \mathrm{dom}(\varphi) = \{\alpha_1, \ldots, \alpha_p\}}{\Gamma \vdash \{e_1 = a_1; \ldots; e_n = a_n\} : \varphi((\alpha_1, \ldots, \alpha_n)\ t)}$$

$$\frac{\Gamma \vdash a : \varphi((\alpha_1, \ldots, \alpha_n)\ t) \qquad \mathrm{dom}(\varphi) = \{\alpha_1, \ldots, \alpha_p\}}{\Gamma \vdash a.e_i : \varphi(\tau_i)}$$

Again, the substitution φ highlights the fact that the type rule is valid for all the instantiations of the parameters $(\alpha_1, \ldots, \alpha_p)$.

Digression: generalised algebraic data types

An interpreter for a simple language of arithmetic expressions:

```
type term = Num of int | Inc of term | IsZ of term | If of term * term * term

type value = VInt of int | VBool of bool

let rec eval = fun a -> match a with
  | Num x -> VInt x
  | Inc t -> ( match (eval t) with VInt n -> VInt (n+1) )
  | IsZ t -> ( match (eval t) with VInt n -> VBool (n=0) )
  | If (c,t1,t2) -> ( match (eval c) with
     | VBool true -> eval t1
     | VBool false -> eval t2 )
```

Unsatisfactory: nonsensical terms like `Inc (IfZ (Num 0))`, lots of fruitless tagging and un-tagging.

GADT

Remember that we can see constructors as functions:

```
Num : int -> term
If : term * term * term -> term   (etc...)
```

Idea: generalise this into:

```
type 'a term =
  Num : int -> int term
  Inc : int term -> int term
  IsZ : int term -> bool term
  If  : bool term * 'a term * 'a term -> 'a term
```

This rules out nonsensical terms like `(Inc (IfZ (Num 0)))`, because `(IfZ (Num 0))` has type `bool term`, which is incompatible with the type of `Inc`.

GADT, ctd.

Also, the evaluator becomes stunningly direct:

```
let rec eval = fun a -> match a with
  | Num i -> i
  | Inc t -> (eval t) + 1
  | IsZ t -> (eval t) = 0
  | If (c,t1,t2) -> if (eval c) then (eval t1) else (eval t2)
```

where `eval : a term -> a` .

See:

S. Peyton Jones, G. Washburn, S. Weirich, *Wobbly types: type inference for generalised algebraic data types*, 2004.

V. Simonet, F. Pottier, *Constraint-based type inference with guarded algebraic data types*, INRIA TR, 2003.

Imperative programming: references

A *reference* is a cell of memory whose content can be updated.

allocation : `ref` a creates a new memory cell, initialises it with a, and returns its address;

access : if a is a reference, $!a$ returns its content;

update : if a_1 is a reference, $a_1 := a_2$ change its content into a_2, and returns $()$ of type `unit`.

Notation:

$$a_1; a_2 \qquad \text{means} \qquad \texttt{let } x = a_1 \texttt{ in } a_2$$

References: reduction semantics

Expressions: $a ::= \ldots \mid \ell$ memory address

Values: $v ::= \ldots \mid \ell$ memory address

$$
\begin{array}{rcll}
(\texttt{fun}\ x \to a)\ v/s & \xrightarrow{\varepsilon} & a\{x \leftarrow v\}/s & & (\beta) \\
(\texttt{let}\ x = v\ \texttt{in}\ a)/s & \xrightarrow{\varepsilon} & a\{x \leftarrow v\}/s & & (\mathit{let}) \\
\texttt{fst}\ (v_1, v_2)/s & \xrightarrow{\varepsilon} & v_1/s & & (\mathit{fst}) \\
\texttt{snd}\ (v_1, v_2)/s & \xrightarrow{\varepsilon} & v_2/s & & (\mathit{snd}) \\
\texttt{ref}\ v/s & \xrightarrow{\varepsilon} & \ell/s\{\ell \mapsto v\} & \text{si } \ell \notin \mathrm{Dom}(s) & (\delta_{\mathsf{ref}}) \\
!\ell/s & \xrightarrow{\varepsilon} & s(\ell)/s & & (\delta_{\mathsf{deref}}) \\
:= (\ell, v)/s & \xrightarrow{\varepsilon} & (\,)/s\{\ell \mapsto v\} & & (\delta_{\mathsf{assign}})
\end{array}
$$

$$
\frac{a_1/s_1 \xrightarrow{\varepsilon} a_2/s_2}{E[a_1]/s_1 \to E[a_2]/s_2}\ (\text{context})
$$

Example

$$\texttt{let } r = \texttt{ref } 3 \texttt{ in } r := !r + 1; !r/\emptyset$$

$$\begin{aligned}
&\longrightarrow \quad \texttt{let } r = \ell \texttt{ in } r := !r + 1; !r/\{\ell \mapsto 3\} \\
&\longrightarrow \quad \ell := !\ell + 1; !\ell/\{\ell \mapsto 3\} \\
&\longrightarrow \quad \ell := 3 + 1; !\ell/\{\ell \mapsto 3\} \\
&\longrightarrow \quad \ell := 4; !\ell/\{\ell \mapsto 3\} \\
&\longrightarrow \quad (\,); !\ell/\{\ell \mapsto 4\} \\
&\longrightarrow \quad !\ell/\{\ell \mapsto 4\} \\
&\longrightarrow \quad 4
\end{aligned}$$

References: types

Types: $\tau ::= \ldots \mid \tau\ \mathtt{ref}$ type of references whose content type is τ.

Operators:

$$\begin{aligned} \mathtt{ref} &: \forall\alpha.\ \alpha \to \alpha\ \mathtt{ref} \\ ! &: \forall\alpha.\ \alpha\ \mathtt{ref} \to \alpha \\ := &: \forall\alpha.\ \alpha\ \mathtt{ref} \times \alpha \to \mathtt{unit} \end{aligned}$$

Is this enough? Is the resulting language *safe*?

The polymorphic references problem

Consider

```
let r = ref (fun x → x) in
r := (fun x → x+1);
(!r) true
```

- `r` receives the polymorphic type $\forall\alpha.\ (\alpha \to \alpha)$ `ref`;
- the update $\texttt{r} := (\texttt{fun x} \to \texttt{x} + 1)$ is well-typed (use r at type $(\texttt{int} \to \texttt{int})$ `ref`);
- the application $(!\texttt{r})$ `true` is also well-typed (use `r` at type $(\texttt{bool} \to \texttt{bool})$ `ref`);
- the expression is well-typed, but...
- ...its reduction blocks on `true+1`.

Analysis of the problem

Memory addresses are like identifiers: the typing environment associates $types/type\text{-}schemas$ to memory addresses.

If Γ associates type-schemas σ to addresses ℓ, we have

$$\frac{\Gamma(\ell) \leq \tau}{\Gamma \vdash \ell : \tau} \text{ (loc-inst)}$$

This is not safe because if $\ell : \forall\alpha.\tau$ with α free in τ, then we can write a value of type $\tau[\alpha \leftarrow \texttt{int}]$, and read at a different type $\tau[\alpha \leftarrow \texttt{bool}]$ (see previous example).

Analysis of the problem, ctd.

If Γ associates types τ to addresses ℓ, we have

$$\Gamma \vdash \ell : \Gamma(\ell) \quad \text{(loc)}$$

and the operations ! and := are safe again. But the well-typed expression

$$\emptyset \vdash \texttt{let}\ r = \texttt{ref}\,(\texttt{fun}\ x \to x)\ \texttt{in}\ (!r)\,1; (!r)\,\texttt{true} : \texttt{bool}$$

reduces to (reduce the $\texttt{ref}\,(\texttt{fun}\ x \to x)$ subterm):

$$\texttt{let}\ r = \ell\ \texttt{in}\ (!r)\,1; (!r)\,\texttt{true}\ /\ \{\ell \mapsto \texttt{fun}\ x \to x\}$$

which cannot be typed anymore! It should hold

$$\ell : (\alpha \to \alpha)\ \texttt{ref} \vdash \texttt{let}\ r = \ell\ \texttt{in}\ (!r)\,1; (!r)\,\texttt{true} : \texttt{bool}$$

but α is now free in the environment and cannot be generalised.

Conclusion

We must:

1. associate types to addresses in the environment;

2. restrict the type system so that it satisfies the property:

   When we type `let` $x = a$ `in` b, we should not generalise the variables in the type of a that might appear in the type of a reference allocated during the evaluation of a.

A solution

Generalise only non-expansive expressions:

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad a_1 \text{ non-expansive} \qquad \Gamma; x : Gen(\tau_1, \Gamma) \vdash a_2 : \tau_2}{\Gamma \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2}$$

In the other cases:

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \Gamma; x : \tau_1 \vdash a_2 : \tau_2}{\Gamma \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2}$$

Non-expansive expressions

Idea: the syntactic structure of the non-expansive expressions ensures that their evaluation does not create references.

Non-expansive expressions:

$a_{ne} ::=$	x	identifiers
	$\mid c$	constants
	$\mid op$	operators
	$\mid \texttt{fun}\ x \to a$	functions
	$\mid (a'_{ne}, a''_{ne})$	pairs of non-expansive expressions
	$\mid \texttt{fst}\ a_{ne}$	projections of non-expansive expressions
	$\mid \texttt{snd}\ a_{ne}$	
	$\mid op(a_{ne})$	if $op \neq \texttt{ref}$
	$\mid \texttt{let}\ x = a'_{ne}\ \texttt{in}\ a''_{ne}$	`let` binding

Examples

Not well-typed anymore:

```
let r = ref (fun x → x) in
r := (fun x → x+1);
(!r) true
```

- `ref (fun x → x)` is expansive,
- `r` receives a type $(\tau \rightarrow \tau)$ `ref`,
- the second line requires $\tau =$ `int`,
- the third $\tau =$ `bool`.

Well-typed terms:

```
let id = fun x → x in (id 1, id true)
let id = fst((fun x → x), 1) in (id 1, id true)
```

Examples, ctd.

Surprise! Not well-typed:

```
let k = fun x → fun y → x in
let f = k 1 in
(f 2, f true)
```

because `k 1` is expansive, and `f` receives a type $\tau \rightarrow$ `int`.

But η-expansion saves us. This expression is now well-typed:

```
let k = fun x → fun y → x in
let f = fun x -> k 1 x in
(f 2, f true)
```

Why isn't application non-expansive?

Reference creation can be hidden inside function application:

```
let f x = ref(x) in
let r = f(fun x → x) in ...
```

Wait, the type of `r` is $(\alpha \rightarrow \alpha)$`ref` and it mentions explicitely `ref`: maybe we can use this information...

A more subtle example

```
let functional_ref =
  fun x →
    let r = ref x in ((fun newx → r := newx), (fun () → !r)) in
let p = functional_ref(fun x → x) in
let write = fst p in
let read = snd p in
write(fun x → x+1);
(read()) true
```

Observe that the type of `functional_ref` is $\forall\alpha.\, \alpha \to (\alpha \to \texttt{unit}) \times (\texttt{unit} \to \alpha)$, and does not mention `ref`, but the result of `functional_ref` is functionally equivalent to a value of type α `ref`.

Safety with references, begin

Remark: all the previous results about the typing relation $\Gamma \vdash a : \tau$ still hold (including the Substitution Lemma).

Definition: a memory state s is well-typed in Γ, denoted $\Gamma \vdash s$, iff $\mathrm{Dom}(s) = \mathrm{Dom}(\Gamma)$ and for all adress $\ell \in \mathrm{Dom}(s)$, there exists τ such that $\Gamma(\ell) = \tau\ \texttt{ref}$ and $\Gamma \vdash s(\ell) : \tau$.

Definition: we say that an environment Γ *extends* Γ_1 if Γ extends Γ_1 when considered as partial functions.

The less-typable-than relation revisited

Definition: a_1/s_1 is less typable than a_2/s_2, denoted $a_1/s_1 \sqsubseteq a_2/s_2$, if for all environment Γ and type τ,

- if a_1 is non-expansive: a_2 is non-expansive, and $\Gamma \vdash a_1 : \tau$ and $\Gamma \vdash s_1$ imply $\Gamma \vdash a_2 : \tau$ and $\Gamma \vdash s_2$.

- if a_1 is expansive: $\Gamma \vdash a_1 : \tau$ and $\Gamma \vdash s_1$ imply that there exists Γ' extending Γ such that $\Gamma' \vdash a_2 : \tau$ and $\Gamma' \vdash s_2$.

Reduction preserves typing

Proposition 12. *If* $a_1/s_1 \stackrel{\varepsilon}{\rightarrow} a_2/s_2$*, then* $a_1/s_1 \sqsubseteq a_2/s_2$*.*

Proof: Case analysis on the reduction rule applied. □

Proposition 13. [Monotonicity of $\sqsubseteq$] *For all evaluation context* E*,* $a_1/s_1 \sqsubseteq a_2/s_2$ *implies* $E[a_1]/s_1 \sqsubseteq E[a_2]/s_2$*.*

Proof: See next slide. □

Proposition 14. [Reduction preserves typing] *If* $a_1/s_1 \rightarrow a_2/s_2$*, then* $a_1/s_1 \sqsubseteq a_2/s_2$*.*

Proof: Consequence of Lemmas 12 and 13. □

Proof of monotonicity of $\sqsubseteq$

Proof: Induction on the structure of the evaluation contexts. The interesting case is when the context is $\texttt{let}\ x = E\ \texttt{in}\ a$. (We could not prove this case without the restriction of generalisation to non-expansive expressions). Let Γ and τ such that $\Gamma \vdash \texttt{let}\ x = E[a_1]\ \texttt{in}\ a : \tau$ and $\Gamma \vdash s_1$. The typing derivation is of the form below:

$$\frac{\Gamma \vdash E[a_1] : \tau_1 \qquad E[a_1] \text{ non-expansive} \qquad \Gamma; x : Gen(\tau_1, \Gamma) \vdash a : \tau}{\Gamma \vdash \texttt{let}\ x = E[a_1]\ \texttt{in}\ a : \tau}$$

Applying the induction hypothesis to $E[a_1]$, we obtain $E[a_1]/s_1 \sqsubseteq E[a_2]/s_2$. Then, since $E[a_1]$ is non-expansive, we obtain $\Gamma \vdash E[a_2] : \tau_1$ and $\Gamma \vdash s_2$ and $E[a_2]$ is non-expansive. Thus, we can build the derivation below:

$$\frac{\Gamma \vdash E[a_2] : \tau_1 \qquad E[a_2] \text{ non-expansive} \qquad \Gamma; x : Gen(\tau_1, \Gamma) \vdash a : \tau}{\Gamma \vdash \texttt{let}\ x = E[a_2]\ \texttt{in}\ a : \tau}$$

and the expected result follows. $\square$

Shape of values

Proposition 15. [Shape of values acccording to their type] *Let Γ be an environment that binds only adresses ℓ. Let $\Gamma \vdash v : \tau$ and $\Gamma \vdash s$.*

1. *If $\tau = \tau_1 \rightarrow \tau_2$, then either v is of the form $\texttt{fun}\ x \rightarrow a$, or v is an operator op;*
2. *if $\tau = \tau_1 \times \tau_2$, then v is a pair (v_1, v_2);*
3. *if τ is a base type T, then v is a constant c.*
4. *if $\tau = \tau_1\ \texttt{ref}$, then v is a memory address $\ell \in \mathrm{Dom}(s)$.*

Proof: by inspection of the typing rules. $\square$

Safety, end

Proposition 16. [Progression Lemma] *Let Γ be an environment that binds only addresses ℓ. Suppose $\Gamma \vdash a : \tau$ and $\Gamma \vdash s$. Then, either a is a value, or there exists a' and s' such that $a/s \rightarrow a'/s'$.*

Proof: analogous to that of the Progression Lemma for mini-ML. $\square$

Theorem 5. [Safety] *If $\emptyset \vdash a : \tau$ and $a/\emptyset \rightarrow^\star a'/s'$ and a'/s' is a normal form with respect to $\rightarrow$, then a' is a value.*

The approach of SML'90

Idea: distinguish *a*pplicative type variables from *imperative type variables*, and generalise only the first ones.

Types: $\tau ::= \alpha_a \mid \alpha_i \mid T \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 \; \texttt{ref}$
Imperative types: $\bar{\tau} ::= \alpha_i \mid T \mid \bar{\tau}_1 \to \bar{\tau}_2 \mid \bar{\tau}_1 \times \bar{\tau}_2 \mid \bar{\tau}_1 \; \texttt{ref}$

Substitutions: $[\alpha_a \leftarrow \tau, \alpha_i \leftarrow \bar{\tau}]$.

Operators:

$$
\begin{array}{rcl}
! & : & \forall \alpha_a.\ \alpha_a \; \texttt{ref} \to \alpha_a \\
:= & : & \forall \alpha_a.\ \alpha_a \; \texttt{ref} \times \alpha_a \to \texttt{unit} \\
\texttt{ref} & : & \forall \alpha_i.\ \alpha_i \to \alpha_i \; \texttt{ref}
\end{array}
$$

SML'90, ctd.

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad \Gamma; x : \mathit{GenAppl}(\tau_1, \Gamma) \vdash a_2 : \tau_2}{\Gamma \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2}$$

$$\mathit{GenAppl}(\tau, \Gamma) = \forall \alpha_{a,1} \ldots \alpha_{a,n}.\ \tau$$

where $\{\alpha_{a,1}, \ldots, \alpha_{a,n}\} = \mathcal{L}_a(\tau) \setminus \mathcal{L}_a(\Gamma)$ are the applicative variables free in τ but not in Γ.

$$\frac{\Gamma \vdash a_1 : \tau_1 \qquad a_1 \text{ non expansive} \qquad \Gamma; x : \mathit{Gen}(\tau_1, \Gamma) \vdash a_2 : \tau_2}{\Gamma \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2}$$

Examples

`let id = fun x → x in`	id : $\forall\alpha_a.\ \alpha_a \rightarrow \alpha_a$
`let f = id id in`	f : $\forall\alpha_a.\ \alpha_a \rightarrow \alpha_a$
`(f 1, f true)`	ok
`let r = ref(fun x → x) in`	r : $(\alpha_i \rightarrow \alpha_i)\ ref$
`r := fun x → x+1;`	α_i is now int
`(!r) true`	error
`let f = fun x → ref(x) in`	f : $\forall\alpha_i.\ \alpha_i \rightarrow \alpha_i$
`let r = f(fun x → x) in`	r : $(\alpha_i \rightarrow \alpha_i)\ ref$
`r := fun x → x+1;`	α_i is now int
`(!r) true`	error

Effects and regions

The type and effect discipline, Jean-Pierre Talpin and Pierre Jouvelot, *Information and Computation* 111(2), 1994.

Typed Memory Management in a Calculus of Capabilities, Karl Crary, David Walker, Greg Morrisett, *Conference Record of POPL'99, San Antonio, Texas*.

Exceptions

Idea: have a mechanism to signal an error. The signal propagates across the calling functions, unless it is catched and treated.

Example:

```
try 1 + (raise "Hello") with x → x
```

reduces to

```
"Hello"
```

Exceptions, formally

Expressions: $a ::= \ldots \mid \texttt{try } a_1 \texttt{ with } x \rightarrow a_2$

Operators: $op ::= \ldots \mid \texttt{raise}$

$$
\begin{array}{rcll}
\texttt{try } v \texttt{ with } x \rightarrow a & \xrightarrow{\varepsilon} & v & \\
\texttt{try raise } v \texttt{ with } x \rightarrow a & \xrightarrow{\varepsilon} & a[x \leftarrow v] & \\
\Delta[\texttt{raise } v] & \rightarrow & \texttt{raise } v & \text{if } \Delta \text{ is not } [\,]
\end{array}
$$

Evaluation contexts:

$$E ::= \ldots \mid \texttt{try } E \texttt{ with } x \rightarrow a$$

Exception contexts:

$$\Delta ::= [\,] \mid \Delta\ a \mid v\ \Delta \mid \texttt{let } x = \Delta \texttt{ in } a \mid (\Delta, a) \mid (v, \Delta) \mid \texttt{fst } \Delta \mid \texttt{snd } \Delta$$

Answers:

$$r ::= v \mid \texttt{raise } v$$

The type of exceptions:

$$\tau ::= \ldots \mid \texttt{exn}$$

Type rules:

$$\texttt{raise} : \forall\alpha.\ \texttt{exn} \rightarrow \alpha$$

$$\frac{\Gamma \vdash a_1 : \tau \qquad \Gamma; x : \texttt{exn} \vdash a_2 : \tau}{\Gamma \vdash \texttt{try}\ a_1\ \texttt{with}\ x \rightarrow a_2 : \tau}$$