Effects for Funargs

Jeremy G. Siek$^1$  M. Vitousek$^1$  J. Turner$^1$

$^1$University of Colorado at Boulder

HOPE 2012
Chapel’s Goals: Productivity and Parallelism

- **Arrays**
  - **Data-Parallelism**
- **Value and Reference Objects**
- **StackAllocation Templates**

- **ZPL**
- **Java/C#**
- **C++**

Related to Chapel
In the context of higher-order functions and lexical scope, we can see the following code snippet:

```javascript
var my_state = read(string);
var addresses_in_my_state = new Vector(Address);

copy_if(addresses, back_inserter(addresses_in_my_state),
    fun (a:Address) {
        return a.state == my_state;
    });
```

This code demonstrates how to read a string into a variable, create a new vector to hold addresses, and then copy addresses from the current state into the new state using a higher-order function. The `copy_if` function is used to filter addresses based on a condition that checks if the address's state equals the current state.
Lexical Scope and Stack Allocation, Danger!

```
proc deriv(f: float -> float, d:float) {
    return fun(x:float) {
        return f(x + d) - f(x - d);
    };
}

proc square(x:float) { return x * x; }

var line = deriv(square, 0.01);
// Kaboom!
writeln(line(5.0));
```

---

The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. J. Moses, SIGSAM Bull., 1970.
## Representatives of Prior Work

| Research                          | Industry                                      |
|----------------------------------|-----------------------------------------------|
| Effects: Talpin & Jouvelot       | GC’d languages (H)                            |
| ML ⇒ Regions: Toft & Talpin     | Blocks in Objective C (V, H)                  |
| Cyclone: Grossman et al.         | Lambda in C++ (V, R)                         |
| RT Java: Boyapati et al.         | Inner Classes in Java (V)                    |

H: heap  
V: value  
R: reference
Design Considerations

- Efficient
- Safe
- Simple (Compiler)
- Simple (Programmer)
- Expressive

Diagram:

```
  Safe
 /    \
/     \/
Simple (Compiler)  Simple (Programmer)
        /   \        /
       /     \       /     \      
      Efficient  Expressive
```
## Capture by reference or by value

| By Reference                           | By Value                  |
|----------------------------------------|---------------------------|
| +Always Fast                           | −Sometimes slow           |
| +Downward funargs ok                   | +Downward funargs ok      |
| −No upward funargs                     | +Upward funargs ok        |
|                                        | −No mutation              |
Our Approach

1. Programmer chooses between capture by-reference or by-value. (like C++)
   - Here we show by-reference as the default.
   - The default could also be by-value or neither.

2. A type and effect system disallows escaping references. (like effect systems for regions)

3. Effects are sets of term variables. (new)
Example 1 Revisited

- No annotations
- Capture by reference is the default

```javascript
var my_state = read(string);
var addresses_in_my_state = new Vector(Address);

copy_if(addresses, back_inserter(addresses_in_my_state),
    fun (a:Address) {
        return a.state === my_state;
    });
```
Example 2 with capture by-value

- Annotate capture by-value
- Upward funarg is ok

```plaintext
proc deriv(f: float → float, d:float) {
    return fun(x:float) f, d {
        return f(x + d) − f(x − d);
    };
}

proc square(x:float) { return x * x; }

var times_two = deriv(square, 0.5);
writeln(times_two(5.0));
> 10.0
```
Example 2 with capture by-reference

- Type system detects escaping references

```plaintext
proc deriv(f: float → float, d:float) {
    return fun(x:float) {
        return f(x + d) − f(x − d);
    };
}
```

Error: f and d escape from deriv

```plaintext
proc square(x:float) {
    return x × x;
}
```

```plaintext
var times_two = deriv(square, 0.5);
writeln(times_two(5.0));
```
Inferred Effect Polymorphism

```ocaml
proc deriv(f: float → float, d:float) { 
    return fun(x:float) f,d { return f(x + d) − f(x − d); }; 
}
// deriv : ∀ε. (float → float, float) → float → float
var pi = 3.14159;

proc times_pi(x:float) { return pi * x; } // float → float
var almost_pi = deriv(times_pi, 0.5); // float → float

writeln(almost_pi(1.0));
> 3.14158999999
```

Region-based memory management in Cyclone. Grossman et al.
PLDI 2002
The Core Language

- **ground effects** \( \gamma \ ::= \{ x \} | \epsilon \)
- **effects** \( \varphi \ ::= \emptyset | \gamma | \varphi \cup \varphi \)
- **types** \( T \ ::= \text{int} | T \rightarrow T | \forall \epsilon. T \)
- **expressions** \( e \ ::= n | x | f | e<\varphi> \)
- **abstractions** \( f \ ::= \text{fun}(x:T) \varphi_T \overline{y} = e\{s\} | \Lambda \epsilon. f \)
- **statements** \( s \ ::= x = e(e); s | \text{ret } e(e) | \text{ret } e \)

- Function parameters \((x)\) are bound to stack locations.
- Function-local variables \((\overline{y})\) reside in the function.
Evaluation of Expressions

stack locations \( \ell \in \mathbb{N} \)
ground effects \( \gamma ::= \ldots \mid \{\ell \_T\} \)
expressions \( e ::= \ldots \mid \ell \_T \)
values \( v ::= n \mid \text{fun}(x : T)\_T y = v\{s\} \)
stack o’ values \( \sigma ::= [] \mid v :: \sigma \)

\[
[e]_\sigma = v
\]

\[
[[\ell \_T]]_\sigma = \sigma_{n - 1 - \ell} \quad \text{where} \quad n = |\sigma|
\]
\[
[[\text{fun}(x : T_1)\_T_2 y = e\{s\}]]_\sigma = \text{fun}(x : T_1)\_T_2 y = v\{s\} \quad \text{if} \quad [e]_\sigma = v
\]
\[
[[\Lambda \epsilon. \: f]]_\sigma = \Lambda \epsilon. \: f
\]
\[
[[e<\varphi>]]_\sigma = [[\epsilon := \varphi] \: e’]_\sigma \quad \text{if} \quad [e]_\sigma = \Lambda \epsilon. \: e’
\]
Abstract Machine Transitions

Stacks $\kappa ::= \[] | (x: T, s, n) :: \kappa$

States $\varsigma ::= \langle s, \kappa, \sigma, n \rangle$

\[ \varsigma \rightarrow \varsigma \]

\[ \langle (x = e_1 (e_2) ; s_1), \kappa, \sigma, n \rangle \]  \hspace{1cm} \text{(CALL)}

\[ \rightarrow \langle [y := \ell_{T_1}, \overline{z} := \overline{v}] s_2, (x: T_2, s_1, n) :: \kappa, [e_2]_\sigma :: \sigma, 1 \rangle \]

where $[e_1]_\sigma = \text{fun}(y: T_1)^{\varphi_{T_2}} \overline{z} = \overline{v} \{ s_2 \}$ and $\ell = |\sigma|$

\[ \langle \text{ret } e_1 (e_2), \kappa, \sigma, n \rangle \]  \hspace{1cm} \text{(TAILCALL)}

\[ \rightarrow \langle [y := \ell_{T_1}] s, \kappa, [e_2]_\sigma :: \text{drop}(n, \sigma), 1 \rangle \]

where $[e_1]_\sigma = \text{fun}(y: T_1)^{\varphi_{T_2}} \{ s \}$ and $\ell = |\sigma| - n$

\[ \langle \text{ret } e, (x: T, s, n_2) :: \kappa, \sigma, n_1 \rangle \]  \hspace{1cm} \text{(RETURN)}

\[ \rightarrow \langle [x := \ell_T] s, \kappa, [e]_\sigma :: \text{drop}(n_1, \sigma), n_2 + 1 \rangle \]

where $\ell = |\sigma| - n_1$
Type System, Expressions

$$\Gamma ::= \emptyset \mid \Gamma, x : T \mid \Gamma, \text{ref}(T) \mid \Gamma, \epsilon$$

$$\Gamma; \varphi \vdash e : T$$

$$x : \text{ref}(T) \in \Gamma$$
$$\{x\} \subseteq \varphi$$
$$\Gamma; \varphi \vdash x : T$$

$$x : T \in \Gamma$$
$$\Gamma; \varphi \vdash x : T$$
$$\{\ell_T\} \subseteq \varphi$$
$$\Gamma; \varphi \vdash \ell_T : T$$

$$\Gamma; \emptyset \vdash T_1 \varphi_2 \rightarrow T_2$$
$$\Gamma, y : \overline{T}, x : \text{ref}(T_1); \varphi_2 \cup \{x\}; \{x\} \vdash s : T_2$$
$$\Gamma; \varphi_1 \vdash \text{fun}(x : T_1) \varphi_2 \overline{y} = e\{s\} : T_1 \varphi_2 \rightarrow T_2$$

$$\Gamma, \epsilon; \emptyset \vdash f : T$$
$$\Gamma; \varphi \vdash \forall \epsilon. f : \forall \epsilon. T$$

$$\Gamma; \varphi_1 \vdash e : \forall \epsilon. T$$
$$\Gamma; \emptyset \vdash \varphi_2$$
$$\Gamma; \varphi_1 \vdash e<\varphi_2> : [\epsilon := \varphi_2] T$$
Lemma (Locals not in Return Type)

If $\Gamma; \varphi_1; \varphi_2 \vdash s : T$, then $\text{FV}(T) \cap \varphi_2 = \emptyset$. 

\[
\begin{align*}
\Gamma; \varphi_1 \vdash e_1 : T & \xrightarrow{\varphi_3} T_2 & \Gamma; \varphi_1 \vdash e_2 : T_1 \\
\Gamma, x:\text{ref}(T_2); \varphi_1 \cup \{x\}; \varphi_2 \cup \{x\} \vdash s : T_3 & \quad \varphi_3 \subseteq \varphi_1 \\
\Gamma; \varphi_1; \varphi_2 \vdash x = e_1(e_2); s : T_3
\end{align*}
\]
Type System, Abstract Machine

\[ \vdash \varsigma : T \]

\[
\begin{align*}
\emptyset ; \varphi_1 ; \varphi_2 & \vdash s : T_1 \\
\emptyset ; \Sigma & \vdash \varphi_1 \\
\emptyset ; \text{take}(n, \Sigma) & \vdash \varphi_2 \\
\emptyset ; \text{drop}(n, \Sigma) & \vdash \varphi_1 - \varphi_2 \\
\vdash \sigma : \Sigma & \\
\text{drop}(n, \Sigma) & \vdash \kappa : T_1 \Rightarrow T_2 \\
\vdash \langle s, \kappa, \sigma, n \rangle & : T_2
\end{align*}
\]
\[ \vdash \sigma : \Sigma \]

\[ \vdash \epsilon : \epsilon \]

\[ \vdash \epsilon : \Sigma \]

\[ \vdash \varphi_1 \]

\[ \vdash \varphi_2 \]

\[ \vdash \varphi_1 - \varphi_2 \]

\[ \vdash s : T_2 \]

\[ \vdash \kappa : T_2 \Rightarrow T_3 \]

\[ \vdash (x : T_1, s, n) : \kappa : T_1 \Rightarrow T_3 \]
Lemma (Evaluation Safety)

If $\emptyset; \varphi \vdash e : T$, $\emptyset; \Sigma \vdash \varphi$, and $\vdash \sigma : \Sigma$, then $\emptyset; \emptyset \vdash \llbracket e \rrbracket_\sigma : T$. 
The Machine is Safe

**Theorem**

If \( \vdash \zeta : T \), then either \( \zeta \) is a final state or \( \zeta \xrightarrow{} \zeta' \) and \( \vdash \zeta' : T \).
Abstract Syntax of Featherweight Functional Chapel ($F^2C$)

variables \( x, y, z \in \text{Var} \)

integers \( n \in \mathbb{Z} \)

effect variables \( \epsilon \in \text{EVar} \)

effects \( \varphi ::= \emptyset | \{x, \ldots\} | \epsilon \)

types \( T ::= \text{int} | T \rightarrow T | T \xrightarrow{\varphi} T | \forall \epsilon. T \)

expressions \( e ::= n | x | \text{fun}<\epsilon>(x:T)\overline{y}\{s\} \)

statements \( s ::= x=e(e) ; s | \text{ret} \ epsilon | \text{ret} \ e(e) \)
Effect Variable is the default for Function Types

\[
\vdash \text{Int} \rightsquigarrow \text{Int}, \emptyset \quad \vdash T_1 \rightsquigarrow T_1'; \bar{\epsilon}_1 \\
\vdash T_2 \rightsquigarrow T_2'; \bar{\epsilon}_2 \quad \epsilon \text{ fresh} \\
\vdash T_1 \rightarrow T_2 \rightsquigarrow T_1' \overset{\epsilon}{\rightarrow} T_2'; (\epsilon \bar{\epsilon}_1 \bar{\epsilon}_2)
\]

\[
\vdash T_1 \overset{\varnothing}{\rightarrow} T_2 \rightsquigarrow T_1' \overset{\varnothing}{\rightarrow} T_2'; \bar{\epsilon}_1 \bar{\epsilon}_2 \
\vdash T \rightsquigarrow T'; \bar{\epsilon}' \\
\vdash \forall \bar{\epsilon}. \ T \rightsquigarrow \forall \bar{\epsilon}. \ T'; \bar{\epsilon}'
\]
Type System and Elaboration, Expressions

$$\Gamma \vdash e \leadsto e : T ! \varphi$$

$$\Gamma \vdash n \leadsto n : T ! \emptyset$$

$$\Gamma \vdash x : \text{ref}(T) \in \Gamma$$

$$\Gamma \vdash x \leadsto x : T ! \{x\}$$

$$x : T \in \Gamma$$

$$\Gamma \vdash x \leadsto x : T ! \emptyset$$

$$\vdash T_1 \leadsto T'_1; \bar{e}_2$$

$$\bar{e}_1 \bar{e}_2, x : \text{ref}(T'_1), \bar{y} : \overline{T}; x \vdash s \leadsto s' : T_2 ! \varphi_1 \quad \varphi_2 = \varphi_1 - \{x\}$$

$$\Gamma \vdash \text{fun}<\bar{e}_1>(x : T_1) \bar{y}\{s\}$$

$$\leadsto \Lambda \bar{e}_1 \bar{e}_2. \text{fun}(x : T'_1)^{\varphi_3}_{T_2} \bar{y} = \bar{y}\{s'\}$$

$$: \left(\forall \bar{e}_1 \bar{e}_2. \ T'_1 \xrightarrow{\varphi_3} T_2\right) ! \varphi_3$$
Type System and Elaboration, Statements

\[ \Gamma; z \vdash s \rightsquigarrow s' : T!\varphi \]

\[ \Gamma \vdash e_1 \rightsquigarrow e'_1 : \forall \vec{e}. T_1 \xrightarrow{\varphi_4} T_2 !\varphi_1 \quad \Gamma \vdash e_2 \rightsquigarrow e'_2 : T_3 !\varphi_2 \]

\[ \text{match}(T_3, [\vec{e} := \bar{u}] T_1) = \{\bar{u} \mapsto \bar{\varphi}\} \]

\[ \Gamma, x:\text{ref}(T_2); x \bar{z} \vdash s \rightsquigarrow s' : T_3 !\varphi_3 \]

\[ \Gamma; \bar{z} \vdash x = e_1(e_2); s \rightsquigarrow x = e'_1<\bar{\varphi}>(e'_2); s' : T_3 ! [\vec{e} := \bar{\varphi}] \varphi_4 \cup \bigcup_{i=1}^{3} \varphi_i \]

\[ \Gamma \vdash e \rightsquigarrow e' : T!\varphi \quad \text{FV}(T) \cap \bar{z} = \emptyset \]

\[ \Gamma; \bar{z} \vdash \text{ret } e \rightsquigarrow \text{ret } e' : T!\varphi \]

\[ \Gamma \vdash e_1 \rightsquigarrow e'_1 : \forall \vec{e}. T_1 \xrightarrow{\varphi_3} T_2 !\varphi_1 \quad \Gamma \vdash e_2 \rightsquigarrow e'_2 : T_3 !\varphi_2 \]

\[ \text{match}(T_3, [\vec{e} := \bar{u}] T_1) = \{\bar{u} \mapsto \bar{\varphi}\} \quad \text{FV}(T_2) \cap \bar{z} = \emptyset \]

\[ \Gamma; \bar{z} \vdash \text{ret } e_1(e_2) \rightsquigarrow \text{ret } e'_1<\bar{\varphi}>(e'_2) : T_2 ! [\vec{e} := \bar{\varphi}] \varphi_3 \cup \varphi_1 \cup \varphi_2 \]
Effect Argument Inference, Types

\[
\text{match}(\text{Int}, \text{Int}, \Sigma) = \Sigma \\
\text{match}(T_1 \xrightarrow{\varphi_1} T_2, T_3 \xrightarrow{\varphi_2} T_4, \Sigma) = \Sigma_3 \\
\text{where } \Sigma_1 = \text{match}(T_3, T_1, \Sigma) \\
\Sigma_2 = \text{match}(\varphi_1, \varphi_2, \Sigma_1) \\
\Sigma_3 = \text{match}(T_2, T_4, \Sigma_2) \\
\text{match}(\forall \epsilon. T_1, \forall \epsilon. T_2, \Sigma) = \text{match}(T_1, T_2, \Sigma)
\]
Effect Argument Inference, Effects

\[ \text{match}(\varphi, u, \Sigma) = \Sigma[u := \varphi] \]
  if \( u \notin \text{dom}(\Sigma) \)

\[ \text{match}(u, \varphi, \Sigma) = \Sigma[u := \varphi] \]
  if \( u \notin \text{dom}(\Sigma) \)

\[ \text{match}(\varphi_1, \varphi_2, \Sigma) = \Sigma \]
  if \( \vdash \Sigma(\varphi_1) \subseteq \Sigma(\varphi_2) \)
Conclusion

- Effects are sets of variables.
- Type system prevents escaping references to the stack.
- Programmer controls capture by reference or by value.

Questions?
Effect Subset

\[ \vdash \varphi \subseteq \varphi \]

\[ \vdash \emptyset \subseteq \varphi \]
\[ \vdash \gamma \subseteq \gamma \]

\[ \vdash \varphi_1 \subseteq \varphi_3 \quad \vdash \varphi_2 \subseteq \varphi_3 \]
\[ \vdash \varphi_1 \cup \varphi_2 \subseteq \varphi_3 \]

\[ \vdash \gamma \subseteq \varphi_1 \text{ or } \vdash \gamma \subseteq \varphi_2 \]
\[ \vdash \gamma \subseteq \varphi_1 \cup \varphi_2 \]
Effect Union

\[ \varnothing \uplus \varphi = \varphi \]
\[ \varphi \uplus \emptyset = \varphi \]
\[ (\varphi_1 \cup \varphi_2) \uplus \varphi_3 = (\varphi_1 \uplus \varphi_3) \uplus (\varphi_2 \uplus \varphi_3) \]
\[ \gamma \uplus (\varphi_1 \cup \varphi_2) = (\gamma \uplus \varphi_1) \uplus (\gamma \uplus \varphi_2) \]
\[ \gamma_1 \uplus \gamma_2 = \begin{cases} 
\gamma_1 & \text{if } \gamma_1 = \gamma_2 \\
\gamma_1 \cup \gamma_2 & \text{otherwise}
\end{cases} \]
\[ [\epsilon := \varphi_3](\varphi_1 \cup \varphi_2) = [\epsilon := \varphi_3] \varphi_1 \uplus [\epsilon := \varphi_3] \varphi_2 \]
Effect Intersection

\[ \phi \cap \phi \]

\[
(\phi_1 \cup \phi_2) \cap \phi_3 = (\phi_1 \cap \phi_3) \cup (\phi_2 \cap \phi_3)
\]

\[
\gamma \cap (\phi_2 \cup \phi_3) = (\gamma \cap \phi_2) \cup (\gamma \cap \phi_3)
\]

\[
\gamma_1 \cap \gamma_2 = \begin{cases} 
\gamma_1 & \text{if } \gamma_1 = \gamma_2 \\
\emptyset & \text{if } \gamma_1 \neq \gamma_2 
\end{cases}
\]

\[
\emptyset \cap \phi = \emptyset
\]

\[
\phi \cap \emptyset = \emptyset
\]
Effect Difference

\[ \varphi - \varphi \]

\[
(\varphi_1 \cup \varphi_2) - \varphi_3 = (\varphi_1 - \varphi_3) \cup (\varphi_2 - \varphi_3)
\]

\[
\gamma - (\varphi_2 \cup \varphi_3) = (\gamma - \varphi_2) - \varphi_3
\]

\[
\gamma_1 - \gamma_2 = \begin{cases} 
\emptyset & \text{if } \gamma_1 = \gamma_2 \\
\gamma_1 & \text{otherwise}
\end{cases}
\]

\[
\varphi - \emptyset = \varphi
\]

\[
\emptyset - \varphi = \emptyset
\]