

An Affine-Intuitionistic Type and Effect System

Roberto Amadio¹ Patrick Baillot² Antoine Madet¹

¹Laboratoire PPS
Université Paris 7

²LIP
ENS Lyon

LOLA workshop
07/09/2010, Edinburgh, Scotland

Motivation

intuitionistic type system

+

functional programs with

- concurrent **threads**
- **side-effects**

- ▶ **non-deterministic** programs
- ▶ **diverging** computations

Contribution

intuitionistic type system

- + **affine** typing (Linear Logic)
- + region **usages**
- + region **stratification** [Boudol 2007]

+

functional programs with

- concurrent **threads**
- **side-effects**
- + explicit duplication of data
- + **regions** abstract references, channels

- ▶ **confluence**
- ▶ **termination**

Language overview

- ▶ Values are affine (*can be used at most once*), but get intuitionistic (*duplicable*) when marked with a ‘!’
- ▶ E are call-by-value evaluation contexts
- ▶ Each rule has an affine & intuitionistic version:

$$\begin{aligned} E[(\lambda x.M)V] &\rightarrow E[M[V/x]] \\ E[\text{let } !x = !V \text{ in } M] &\rightarrow E[M[V/x]] \end{aligned}$$

$$\begin{aligned} E[\text{set}(x, V)] &\rightarrow E[*] \parallel (x \leftarrow V) \\ E[\text{pset}(x, V)] &\rightarrow E[*] \parallel (x \Leftarrow V) \end{aligned}$$

$$\begin{aligned} E[\text{get}(x)] \parallel (x \leftarrow V) &\rightarrow E[V] \\ E[\text{get}(x)] \parallel (x \Leftarrow !V) &\rightarrow E[!V] \parallel (x \Leftarrow !V) \end{aligned}$$

- ▶ Static contexts C for parallel evaluation

Typing syntax

$r, r', \dots$	(regions)
$\alpha ::= \mathbf{B} \mid A$	(types)
$A ::= \mathbf{1} \mid A \multimap \alpha \mid !A \mid \mathbf{Reg}_r A$	(value-types)
$\Gamma ::= x_1 : (u_1, A_1), \dots, x_n : (u_n, A_n)$	(variable context)
$R ::= r_1 : (U_1, A_1), \dots, r_n : (U_n, A_n)$	(region context)
$u \in \{1, \infty\}$	(variable usages)
$U \in \{[\infty, \infty]\} \cup \{[1, \infty], [0, \infty]\} \\ \cup \{[0, 0], [1, 0], [0, 1], [1, 1]\}$	(region usages)

Summing usages and contexts

$x \uplus 0$	$=$	x
$0 \uplus x$	$=$	x
$\infty \uplus \infty$	$=$	∞

$\Gamma_1 \uplus \Gamma_2$ and $R_1 \uplus R_2$ are defined such that:

- ▶ affine variables cannot be contracted
- ▶ affine reads/writes cannot be duplicated

The type system

- ▶ Region well-formedness is checked on the leaves:

$$\frac{\textcolor{red}{R} \vdash \textcolor{red}{\Gamma} \quad x : (u, A) \in \Gamma}{R; \Gamma \vdash x : A} \text{ var}$$

- ▶ Promoting data (*cf.* Linear Logic):

$$\frac{\begin{array}{c} R \uplus R' \vdash (\Gamma \uplus \Gamma') \quad \textit{saff}(R'; \Gamma') \\ R; \Gamma \vdash M : A \quad \textcolor{red}{\neg aff}(R; \Gamma) \end{array}}{R \uplus R'; \Gamma \uplus \Gamma' \vdash !M : !A} \text{ promotion}$$

- ▶ A side-effect rule:

$$\frac{\begin{array}{c} R \vdash \Gamma \quad x : (u, \text{Reg}_r A) \in \Gamma \\ r : ([v, v'], A) \in R \quad \textcolor{red}{v' \neq 0} \end{array}}{R; \Gamma \vdash \text{get}(x) : A} \text{ read}$$

Typing example

Take $M = \lambda x. \text{let } !x = x \text{ in } \text{get}(x) \parallel \text{set}(x, *)$

We need $R = r : ([1, 1], \mathbf{1})$

We can then show:

$$R; _ \vdash M : !\text{Reg}_r \mathbf{1} \multimap \mathbf{B}$$

Properties

Subject reduction

If $R; \Gamma \vdash P : \alpha$ and $P \rightarrow P'$ then $R; \Gamma \vdash P' : \alpha'$.

Proof hint

- ▶ typing enforces only **duplicable data** can transit through **persistent regions**

Progress

If $R; \Gamma \vdash P : \alpha$ and $P \nrightarrow$ then

$$P \equiv \nu x_1, \dots, x_m (M_1 \parallel \dots \parallel M_n \parallel S_1 \parallel \dots \parallel S_p) \quad m, n, p \geq 0$$

where M_i is a value or a stuck read.

A non-deterministic language

Some well-typed programs:

- $E[\text{get}(x)] \parallel (x \Leftarrow V_1) \parallel (x \Leftarrow V_2)$
- $E_1[\text{get}(x)] \parallel E_2[\text{get}(x)] \parallel (x \leftarrow V)$
- $E_1[\text{get}(x)] \parallel E_2[\text{get}(x)] \parallel (x \Leftarrow V)$

$$U \in \{[\infty, \infty]\} \cup \{[1, \infty], [0, \infty]\} \cup \{[0, 0], [1, 0], [0, 1], [1, 1]\}$$

A non-deterministic language

Some well-typed programs:

- $E[\text{get}(x)] \parallel (x \Leftarrow V_1) \parallel (x \Leftarrow V_2)$
- $E_1[\text{get}(x)] \parallel E_2[\text{get}(x)] \parallel (x \leftarrow V)$
- $E_1[\text{get}(x)] \parallel E_2[\text{get}(x)] \parallel (x \Leftarrow V)$

$$U \in \{\textcolor{red}{[\infty, \infty]}\} \cup \{[1, \infty], [0, \infty]\} \cup \{[0, 0], [1, 0], [0, 1], [1, 1]\}$$

A non-deterministic language

Some well-typed programs:

- $E[\text{get}(x)] \parallel (x \Leftarrow V_1) \parallel (x \Leftarrow V_2)$
- $E_1[\text{get}(x)] \parallel E_2[\text{get}(x)] \parallel (x \leftarrow V)$
- $E_1[\text{get}(x)] \parallel E_2[\text{get}(x)] \parallel (x \Leftarrow V)$

$$U \in \{[1, \infty], [0, \infty]\} \cup \{[0, 0], [1, 0], [0, 1], [1, 1]\}$$

- Single-assignment discipline [Kobayashi, Pierce, Turner 95]

A non-deterministic language

Some well-typed programs:

- $E[\text{get}(x)] \parallel (x \leftarrow V_1) \parallel (x \leftarrow V_2)$
- $E_1[\text{get}(x)] \parallel E_2[\text{get}(x)] \parallel (x \leftarrow V)$
- $E_1[\text{get}(x)] \parallel E_2[\text{get}(x)] \parallel (x \leftarrow V)$

$$U \in \{[1, \infty], [0, \infty]\} \cup \{[0, 0], [1, 0], [0, 1], [1, 1]\}$$

A non-deterministic language

Some well-typed programs:

- $E[\text{get}(x)] \parallel (x \Leftarrow V_1) \parallel (x \Leftarrow V_2)$
- $E_1[\text{get}(x)] \parallel E_2[\text{get}(x)] \parallel (x \leftarrow V)$
- $E_1[\text{get}(x)] \parallel E_2[\text{get}(x)] \parallel (x \Leftarrow V)$

$$U \in \{[0, 0], [1, 0], [0, 1], [1, 1]\}$$

- No multiple reads on volatile stores

A non-deterministic language

Some well-typed programs:

- $E[\text{get}(x)] \parallel (x \Leftarrow V_1) \parallel (x \Leftarrow V_2)$
- $E_1[\text{get}(x)] \parallel E_2[\text{get}(x)] \parallel (x \leftarrow V)$
- $E_1[\text{get}(x)] \parallel E_2[\text{get}(x)] \parallel (x \Leftarrow V)$

► Confluence

If $P \rightarrow P'$ and $P \rightarrow P''$ then either $P' \equiv P''$ or there is a Q such that $P' \rightarrow Q$ and $P'' \rightarrow Q$.

Divergence

Take $M = \text{let } !y = \text{get}(x) \text{ in } y^*$

Diverging computation with **circular region**:

$$\begin{array}{lll} & M & \parallel (x \Leftarrow !(\lambda y.M)) \\ \rightarrow & \text{let } !y = !(\lambda y.M) \text{ in } y^* & \parallel (x \Leftarrow !(\lambda y.M)) \\ \rightarrow & (\lambda y.M)^* & \parallel (x \Leftarrow !(\lambda y.M)) \\ \rightarrow & M & \parallel (x \Leftarrow !(\lambda y.M)) \\ \rightarrow & \dots & \end{array}$$

Region stratification [G. Boudol, CONCUR 2007]

- ▶ An **effect** system to record interaction with the store:
 - ▶ $e, e', \dots$ are a finite sets of regions
 - ▶ Type annotation: $A \xrightarrow{e} \alpha$
 - ▶ Type & effect judgement : $R; \Gamma \vdash P : (\alpha, e)$
- ▶ Stratification intuitively:
 - ▶ Give **rank** to regions and see R as an **ordered list**
 - ▶ Values stored in a region can only have latent effects on lower regions, *i.e.*

$$\frac{R \vdash A \quad r \notin \text{dom}(R)}{R, r : (U, A) \vdash}$$

- ▶ Diverging program's region context is not stratified:

$$r : (U, !(1 \xrightarrow{\{r\}} 1)) \not\vdash$$

Termination

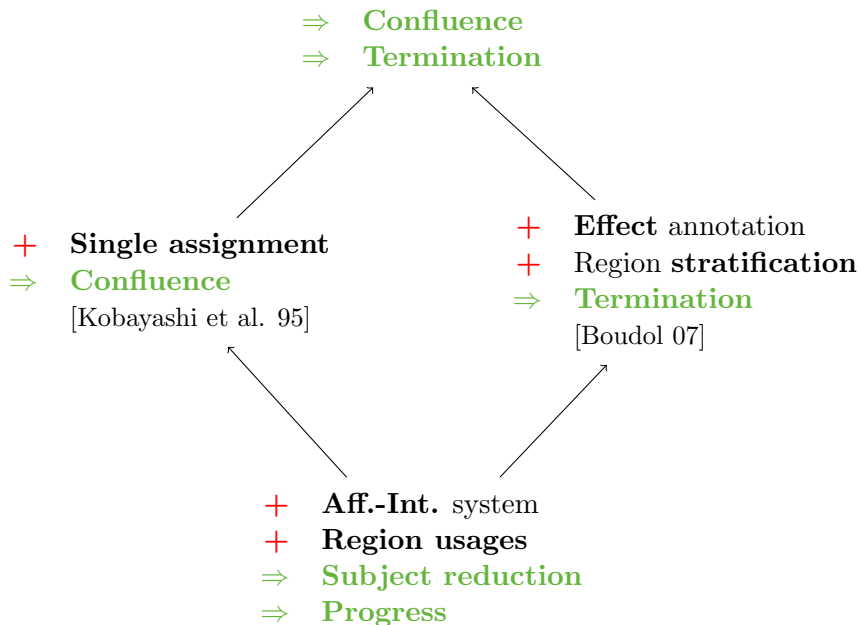
- From aff.-intuitionistic. system to intuitionistic system:

$$\begin{array}{rcl} \underline{!A} & = & \underline{A} \\ \underline{A \multimap B} & = & \underline{A} \rightarrow \underline{B} \\ \underline{!M} & = & \underline{M} \\ \underline{\text{let } !x = M \text{ in } N} & = & (\lambda x. \underline{N}) \underline{M} \\ \underline{(x \leftarrow V)} & = & (x \leftarrow \underline{V}) \end{array}$$

Termination

By simulation in the intuitionistic system we get termination of our affine-intuitionistic system.

Gluing things together



Conclusion

Extension of an affine-intuitionistic type system
to **multi-threaded** programs with **effects**

- ▶ Combination of logical methods:
 - ▶ region usages
 - ▶ region stratification
- ▶ Side-effects deal with data duplication explicitly

Future work

- ▶ Add **inductive data types**, **region polymorphism**, ...
- ▶ Refine further towards **light linear logics** to get a bound on the **computational complexity** of programs

Thank you!