Elements of Programming Languages: Substructural Types

J. Garrett Morris
Garrett.Morris@ed.ac.uk

October 14, 2016

The point

What are types for, anyway?

- Define a subset of "well-formed" programs
- Guarantee absence of run-time errors
- Static approximation of dynamic behavior

Generally involve some level of trade-off:

$$\nvdash \text{if } \mathit{false} \text{ then } 1 \text{ else } \mathit{true} : \text{bool}$$

Substructural type systems refine traditional type systems

- Better characterizations of imperative programming, security, concurrency, &c.
- Can complicate traditional (pure) programs

Files and their openness

Successfully writing to a file:

```
val writer = new PrintWriter(new File("oops"));
writer.write("Hello, world");
writer.close();
```

Unsuccessfully writing to a file:

```
val writer = new PrintWriter(new File("oops"));
writer.close();
writer.write("Hello, world");
```

Type system doesn't track the *state* of the file.

Communication protocols

SMTP specifies not just the types of commands, but also their ordering:

```
220 beeknow.inf.ed.ac.uk ESMTP Sendmail 8...
HELO beeknow.inf.ed.ac.uk
250 beeknow.inf.ed.ac.uk Hello rockefell...
MAIL FROM: Garrett.Morris@ed.ac.uk
250 2.1.0 Garrett.Morris@ed.ac.uk... Sender ok
RCPT TO: Sam.Lindley@ed.ac.uk
250 2.1.5 Sam.Lindley@ed.ac.uk... Recipient ok
```

But:

```
220 beeknow.inf.ed.ac.uk ESMTP Sendmail 8...
HELO beeknow.inf.ed.ac.uk
250 beeknow.inf.ed.ac.uk Hello rockefell..
RCPT TO: Sam.Lindley@ed.ac.uk
503 5.0.0 Need MAIL before RCPT
```

On the having and eating of cakes

Linear type systems are about the control of resources

- "You can't have your cake and eat it too"
- Based on *linear logic*
- Key insight: role of *structural rules* in logic and type systems.

Structural rules in intuitionistic logic

Proof rules can be divided into *structural* and *logical* rules. Logical rules deal with the connectives:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

Structural rules manipulate the hypotheses:

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C} \qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C} \qquad \frac{\Gamma, A, B, \Gamma' \vdash C}{\Gamma, B, A, \Gamma' \vdash C}$$

Structural rules in intuitionistic logic

Structural rules are frequently made implicit:

$$\frac{}{\Gamma, A, \Gamma' \vdash A} \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

We can add term structure

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}$$

Removing the structural rules

- Weakening: material implication doesn't distinguish the following sentences:
  - The sum of two and two is four.
  - If the moon is made of cheese, then the sum of two and two is four.

  Relevant logics distinguish these cases, by insisting that the hypotheses of an implication be used in proving the conclusion.
- Contraction: intuitionistic logic allows arbitrary duplication of propositions. *Linear logic* restricts contraction and weakening, so as to model finite resources and state transitions.
- Exchange: removed in logics intended for natural language (Lambek calculus) and quantum mechanics.

The observations of Drs. Curry & Howard

Parallel between *types* of functional languages and *propositions* of propositional logic:

Types	Propositions
$T \to U$	$A \to B$
$T \times U$	$A \wedge B$
$T + U$	$A \vee B$

The observations of Drs. Curry & Howard

Parallel between *terms* of functional languages and *proofs* of propositional logic:

Terms	Proofs
$\dfrac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x.M : T \to U}$	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$
$\dfrac{\Gamma \vdash M : T \qquad \Gamma \vdash N : U}{\Gamma \vdash (M, N) : T \times U}$	$\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$

Linear logic: variables and implication

$$\frac{}{C \vdash C} \qquad \frac{\Gamma, A \vdash C}{\Gamma \vdash A \multimap C} \qquad \frac{\Gamma \vdash A \multimap B \qquad \Gamma' \vdash A}{\Gamma, \Gamma' \vdash B}$$

With term structure:

$$\frac{}{x : C \vdash x : C} \qquad \frac{\Gamma, x : A \vdash M : C}{\Gamma \vdash \lambda x.M : A \multimap C}$$

$$\frac{\Gamma \vdash M : A \multimap B \qquad \Gamma' \vdash N : A}{\Gamma, \Gamma' \vdash M\,N : B}$$

Communication protocols

- Communication code needs to respect protocols—ordering and timing in addition to data types.
- One approach is session types. For example, SMTP client type would include the fragment:

$$!Sender; !Rcpt; !Message; \mathsf{end}$$

  with communication functions having the types

$$\begin{aligned} &\vdash \mathsf{send} : T \times !T; S \to S \\ &\vdash \mathsf{receive} : ?T; S \to T \times S \end{aligned}$$

- But this isn't enough. What prevents "reusing" earlier points in the protocol?

Linear typing in practice

Simplest example: receive two integers on channel c, send their sum back along c:

$$\begin{aligned}\lambda c.\mathsf{let}\ (x, d) &= \mathsf{receive}\, c\ \mathsf{in}\\ \quad \mathsf{let}\ (y, e) &= \mathsf{receive}\, d\ \mathsf{in}\\ \quad \mathsf{let}\ f &= \mathsf{send}\,(x + y, e)\ \mathsf{in}\ f\end{aligned}$$

Linear typing in practice

Type system rules out misuse of channels; the following are ill-typed

$$
\begin{aligned}
\lambda c.\mathrm{let}\ (x, d) &= \mathrm{receive}\ c\ \mathrm{in}\\
\mathrm{let}\ (y, e) &= \mathrm{receive}\ c\ \mathrm{in}\\
\mathrm{let}\ f &= \mathrm{send}\ (x + y, e)\ \mathrm{in}\ f
\end{aligned}
$$

$$
\begin{aligned}
\lambda c.\mathrm{let}\ (x, d) &= \mathrm{receive}\ c\ \mathrm{in}\\
\mathrm{let}\ (y, e) &= \mathrm{receive}\ d\ \mathrm{in}\\
\mathrm{let}\ f &= \mathrm{send}\ (x + y, e)\ \mathrm{in}\ ()
\end{aligned}
$$

Two views of products

In functional languages, we can use pairs in either of two ways:

- Pattern matching (let $(x, y) = M$ in N), accesses both components of the pair
- Selector functions (*fst* M, *snd* M) access one component of the pair

These are different from a linear perspective:

- Pattern matching uses the resources in constructing both elements of the pair
- Selector functions only use the resources used in constructing one element

Products in linear logic

Two view of products correspond to different product types in linear logic:

$$\frac{\Gamma \vdash A \qquad \Gamma' \vdash B}{\Gamma, \Gamma' \vdash A \otimes B} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \mathbin{\&} B}$$

$$\frac{\Gamma \vdash A \otimes B \qquad \Gamma', A, B \vdash C}{\Gamma, \Gamma' \vdash C} \qquad \frac{\Gamma \vdash A \mathbin{\&} B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \mathbin{\&} B}{\Gamma \vdash B}$$

Products in linear logic

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma, \Gamma' \vdash (M, N) : A \otimes B} \qquad \frac{\Gamma \vdash M : A \otimes B \qquad \Gamma', x : A, y : B \vdash N : C}{\Gamma, \Gamma' \vdash \mathsf{let}\ (x, y) = M\ \mathsf{in}\ N : C}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash [M, N] : A \mathbin{\&} B} \qquad \frac{\Gamma \vdash M : A \mathbin{\&} B}{\Gamma \vdash \mathit{first}(M) : A} \qquad \frac{\Gamma \vdash M : A \mathbin{\&} B}{\Gamma \vdash \mathit{second}(M) : B}$$

Sums in linear logic

Sums in linear logic look mostly as they do in standard functional languages:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{left}(M) : A \oplus B} \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathsf{right}(M) : A \oplus B}$$

$$\frac{\Gamma \vdash L : A \oplus B \qquad \Gamma', x : A \vdash M : C \qquad \Gamma', y : B \vdash N : C}{\Gamma, \Gamma' \vdash \mathsf{case}\ L\ \{\mathsf{left}(x) \Rightarrow M; \mathsf{right}(y) \Rightarrow N\} : C}$$

Why are there two forms of product but only one form of sum?

Integrating traditional and substructural programming

Linear logic includes *modalities* for the structural rules

$$\frac{!\Gamma \vdash A}{!\Gamma \vdash !A} \quad \frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B} \quad \frac{\Gamma \vdash B}{\Gamma, !A \vdash B} \quad \frac{\Gamma \vdash !A \qquad \Gamma', A \vdash B}{\Gamma, \Gamma' \vdash B}$$

($!\Gamma$ means that every proposition in Γ is of the form $!A$)
Can we use these to program with non-linear values?

$$\frac{!\Gamma \vdash M : A}{!\Gamma \vdash !M : !A} \quad \frac{\Gamma \vdash M : !A \qquad \Gamma', x : A \vdash N : B}{\Gamma, \Gamma' \vdash \mathsf{let}\ !x = M\ \mathsf{in}\ N : B}$$

Makes the common case (non-linear values) hard!

Integrating traditional and substructural programming

Can we infer when values are linear?

- Integers and Booleans are non-linear, channels and capabilities are linear
- Pairs are non-linear if their components are non-linear, linear otherwise
- &c.

Doesn't work for functions:

- When is a function from A to B linear? When its closure includes linear values
- Type of the closure not included in the type $A \to B$.

Other substructural logics

Linear logic is only one basis for substructural typing. Other options include:

- *Uniqueness types*: a linearity-like mechanism used in Clean, a Haskell-like language.
- *Separation logic*: an alternative approach to restricting contraction. Many successful in software verification, but not typing. Captures partitions of variables among parts of programs.