A few of LFCS's greatest hits

Philip Wadler
University of Edinburgh
13 April 2016

Burstall, MacQueen, and Sannella: Hope (1980)

HOPE: AN EXPERIMENTAL APPLICATIVE LANGUAGE

R.M. Burstall
D.B. MacQueen[1]
D.T. Sannella

Department of Computer Science
University of Edinburgh
Edinburgh, Scotland

ABSTRACT: An applicative language called HOPE is described and discussed. The underlying goal of the design and implementation effort was to produce a very simple programming language which encourages the construction of clear and manipulable programs. HOPE does not include an assignment statement; this is felt to be an important simplification. The user may freely define his own data types, without the need to devise a complicated encoding in terms of low-level types. The language is very strongly typed, and as implemented it incorporates a typechecker which handles polymorphic types and overloaded operators. Functions are defined by a set of recursion equations; the left-hand side of each equation includes a pattern used to determine which equation to use for a given argument. The availability of arbitrary higher-order types allows functions to be defined which 'package' recursion. Lazily-evaluated lists are provided, allowing the use of infinite lists which could be used to provide interactive input/output and concurrency. HOPE also includes a simple modularisation facility which may be used to protect the implementation of an abstract data type.

2. DESIRABLE PROGRAMMING LANGUAGE FEATURES

The following points seem important in a language for writing correct and flexible programs. They are listed rather briefly but they are illustrated in the description of HOPE which follows.

No assignment: referential transparency

Applicative languages which work in terms of expressions and their values, using recursion instead of loops, seem much clearer and less error-prone. The expressions are transparent in the sense that their value depends only on their textual context and not on some notion of computational history. Each variable is given a value just once where it is declared. Assignments which alter data structures are particularly prone to cause bugs; disallowing them greatly simplifies the language although it does slow down execution. Backus [2] makes this case strongly.

Maximum use of user-defined types

The user should define his own types whenever possible; thus type 'age' rather than type 'integer'. The machine must check these types.

3.3. DEFINING FUNCTIONS

Before a function is defined, its type must be declared. For example:

```
dec reverse : list(alpha) -> list(alpha)
```

HOPE is a very strongly-typed language, and the HOPE system includes a polymorphic typechecker (a modification of the algorithm in [27]) which is able to detect all type errors at compile time. Function symbols may be overloaded. When this is done, the typechecker is able to determine which function definition belongs to each instance of the function symbol.

Functions are defined by a sequence of one or more equations, where each equation specifies the function over some subset of the possible argument values. This subset is described by a pattern (see section 3.2) on the left-hand side of the equation. For example:

```
--- reverse(nil) <= nil                           (1)
--- reverse(a::l) <= reverse(l) <> [a]            (2)
```

(the symbol <> is infix **append**). This defines the (top-level) reverse of a list; for example:

```
reverse(1::(2::nil)) = reverse(2::nil) <> [1]
                     = (reverse(nil) <> [2]) <> [1]
                     = (nil <> [2]) <> [1]
```

```
    node(tip(succ(0)),
         node(tip(succ(succ(0))),tip(0)))
```

But we would like to have trees of lists and trees of trees as well, without having to define them all separately. So we declare a <u>type variable</u>

```
    typevar alpha
```

which when used in a type expression denotes any type (including second- and higher-order types). A general definition of **tree** as a parametric type is now possible:

```
    data tree(alpha) == empty ++ tip(alpha)
                    ++ node(tree(alpha)#tree(alpha))
```

Now **tree** is not a type but a unary <u>type constructor</u> -- the type **numtree** can be dispensed with in favour of **tree(num)** .

```
module list_iterators
  pubconst *, **

  typevar alpha, beta

  dec * : (alpha->beta)#list alpha -> list beta
  dec ** : (alpha#beta->beta)#(list alpha#beta)
                                        -> beta

  infix *, ** : 6

  --- f * nil     <= nil
  --- f * (a::al) <= (f a)::(f * al)

  --- g ** (nil,b)   <= b
  --- g ** (a::al,b) <= g ** (al,g(a,b))

end


module tree_sort
  pubconst sort
  uses ordered_trees, list_iterators

  dec sort : list num -> list num

  --- sort(l) <= flatten(insert ** (l,empty))

end
```

Figure 1: A HOPE Program

Harper, Honsell, and Plotkin: A logical framework (1987, 1993)

A Framework for Defining Logics

ROBERT HARPER

Carnegie-Mellon University, Pittsburgh, Pennsylvania

FURIO HONSELL

Università di Udine, Udine, Italy

AND

GORDON PLOTKIN

Edinburgh University, Edinburgh, United Kingdom

Abstract. The Edinburgh Logical Framework (LF) provides a means to define (or present) logics. It is based on a general treatment of syntax, rules, and proofs by means of a typed λ-calculus with dependent types. Syntax is treated in a style similar to, but more general than, Martin-Löf's system of arities. The treatment of rules and proofs focuses on his notion of a *judgment*. Logics are represented in LF via a new principle, the *judgments as types* principle, whereby each judgment is identified with the type of its proofs. This allows for a smooth treatment of discharge and variable occurrence conditions and leads to a uniform treatment of rules and proofs whereby rules are viewed as proofs of higher-order judgments and proof checking is reduced to type checking. The practical benefit of our treatment of formal systems is that logic-independent tools, such as proof editors and proof checkers, can be constructed.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Formal systems, interactive theorem proving, proof checking, typed lambda calculus

---

This work was initiated in the summer of 1986, and was first reported in July 1987 at the Symposium on Logic in Computer Science in Ithaca, New York.

Valid Signatures

$$\frac{}{\langle\,\rangle\ \mathrm{sig}} \qquad \text{(B-EMPTY-SIG)}$$

$$\frac{\Sigma\ \mathrm{sig} \quad \vdash_\Sigma K \quad a \notin \mathrm{dom}(\Sigma)}{\Sigma, a{:}K\ \mathrm{sig}} \qquad \text{(B-KIND-SIG)}$$

$$\frac{\Sigma\ \mathrm{sig} \quad \vdash_\Sigma A{:}\,\mathtt{Type} \quad c \notin \mathrm{dom}(\Sigma)}{\Sigma, c{:}A\ \mathrm{sig}} \qquad \text{(B-TYPE-SIG)}$$

Valid Context

$$\frac{\Sigma\ \mathrm{sig}}{\vdash_\Sigma \langle\,\rangle} \qquad \text{(B-EMPTY-CTX)}$$

$$\frac{\vdash_\Sigma \Gamma \quad \Gamma \vdash_\Sigma A{:}\,\mathtt{Type} \quad x \notin \mathrm{dom}(\Gamma)}{\vdash_\Sigma \Gamma, x{:}A} \qquad \text{(B-TYPE-CTX)}$$

Valid Kinds

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma \mathtt{Type}} \qquad \text{(B-TYPE-KIND)}$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma K}{\Gamma \vdash_\Sigma \Pi x{:}A.K} \qquad \text{(B-PI-KIND)}$$

Valid Families

$$\frac{\vdash_\Sigma \Gamma \quad c{:}K \in \Sigma}{\Gamma \vdash_\Sigma c{:}K} \qquad \text{(B-CONST-FAM)}$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma B{:}\,\mathtt{Type}}{\Gamma \vdash_\Sigma \Pi x{:}A.B{:}\,\mathtt{Type}} \qquad \text{(B-PI-FAM)}$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma B{:}K}{\Gamma \vdash_\Sigma \lambda x{:}A.B{:}\Pi x{:}A.K} \qquad \text{(B-ABS-FAM)}$$

$$\frac{\Gamma \vdash_\Sigma A{:}\Pi x{:}B.K \quad \Gamma \vdash_\Sigma M{:}B}{\Gamma \vdash_\Sigma AM{:}[M/x]K} \qquad \text{(B-APP-FAM)}$$

$$\frac{\Gamma \vdash_\Sigma A{:}K \quad \Gamma \vdash_\Sigma K' \quad \Gamma \vdash_\Sigma K = K'}{\Gamma \vdash_\sigma A{:}K'} \qquad \text{(B-CONV-FAM)}$$

Valid Objects

$$\frac{\vdash_\Sigma \Gamma \quad c{:}A \in \Sigma}{\Gamma \vdash_\Sigma c{:}A} \qquad \text{(B-CONST-OBJ)}$$

$$\frac{\vdash_\Sigma \Gamma \quad x{:}A \in \Gamma}{\Gamma \vdash_\Sigma x{:}A} \qquad \text{(B-VAR-OBJ)}$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma M{:}B}{\Gamma \vdash_\Sigma \lambda\,{:}A.M{:}\Pi x{:}A.B} \qquad \text{(B-ABS-OBJ)}$$

$$\frac{\Gamma \vdash_\Sigma M{:}\Pi x{:}A.B \quad \Gamma \vdash_\Sigma N{:}A}{\Gamma \vdash_\Sigma MN{:}[N/x]B} \qquad \text{(B-APP-OBJ)}$$

$$\frac{\Gamma \vdash_\Sigma M{:}A \quad \Gamma \vdash_\Sigma A'{:}\,\mathtt{Type} \quad \Gamma \vdash_\Sigma A = A'}{\Gamma \vdash_\Sigma M{:}A'} \qquad \text{(B-CONV-OBJ)}$$

FIG. 1. The LF type system.

$$(\text{RAA})\quad \frac{\neg\neg\varphi}{\varphi} \qquad\qquad (\text{IMP-I})\quad \frac{\begin{array}{c}(\varphi)\\ \psi\end{array}}{\varphi \supset \psi}$$

$$(\text{ALL-I}^*)\quad \frac{\varphi[x]}{\forall x.\varphi[x]} \qquad\qquad (\text{ALL-E})\quad \frac{\forall x.\varphi[x]}{\varphi[t]}$$

$$(\text{SOME-E}^\dagger)\quad \frac{\exists x.\varphi[x] \quad \begin{array}{c}(\varphi)\\ (\varphi)\psi\end{array}}{\psi}$$

*Provided that x is not free in any assumption on which φ depends.
†Provided that x is not free in ψ or any assumptions, other than φ, on which ψ depends.

FIG. 3. Some rules of first-order logic

$$
\begin{aligned}
\mathit{obj} &: \mathit{holtype} \to \mathtt{Type},\\
0 &: \mathit{obj}(\iota),\\
\mathit{succ} &: \mathit{obj}(\iota \Rightarrow \iota),\\
+ &: \mathit{obj}(\iota \Rightarrow \iota \Rightarrow \iota),\\
\times &: \mathit{obj}(\iota \Rightarrow \iota \Rightarrow \iota),\\
< &: \mathit{obj}(\iota \Rightarrow \iota \Rightarrow o),\\
\neg &: \mathit{obj}(o \Rightarrow o),\\
\wedge &: \mathit{obj}(o \Rightarrow o \Rightarrow o),\\
\vee &: \mathit{obj}(o \Rightarrow o \Rightarrow o),\\
\supset &: \mathit{obj}(o \Rightarrow o \Rightarrow o),\\
= &: \Pi s{:}\mathit{holtype}.\mathit{obj}(s \Rightarrow s \Rightarrow o),\\
\forall &: \Pi s{:}\mathit{holtype}.\mathit{obj}((s \Rightarrow o) \Rightarrow o),\\
\exists &: \Pi s{:}\mathit{holtype}.\mathit{obj}((s \Rightarrow o) \Rightarrow o),\\
\Lambda &: \Pi s{:}\mathit{holtype}.\Pi t{:}\mathit{holtype}.(\mathit{obj}(s) \to \mathit{obj}(t)) \to \mathit{obj}(s \Rightarrow t),\\
\mathit{ap} &: \Pi s{:}\mathit{holtype}.\Pi t{:}\mathit{holtype}.\mathit{obj}(s \Rightarrow t) \to \mathit{obj}(s) \to \mathit{obj}(t).
\end{aligned}
$$

The variable-occurrence condition associated with the rule ALL-I is represented using the dependent function space type of LF:

$$\text{ALL-I}:\Pi F:\iota \to o.(\Pi x:\iota.\mathit{true}(Fx)) \to \mathit{true}(\forall(\lambda x:\iota.Fx)).$$

One may say that in natural deduction ALL-I takes as premise a schematic (in x) proof of a judgment, whereas in LF it takes as premise a proof of a schematic judgment, shifting the enforcement of the variable-occurrence condition from the object logic to LF. Note the similarity between the encoding of ALL-I and IMP-I stemming from the fact that the nondependent function space is a special case of the dependent function: variable-occurrence and discharge conditions are closely related.

The existential elimination rule has both discharge and variable-occurrence conditions:

$$\begin{aligned}\text{SOME-E}:\Pi F:\iota \to o.\Pi p:o.\mathit{true}(\exists(\lambda x:\iota.Fx))\\ \to (\Pi x:\iota.\mathit{true}(Fx) \to \mathit{true}(p)) \to \mathit{true}(p).\end{aligned}$$

Note that the variable-occurrence condition on the conclusion of SOME-E rule is a matter of scoping: since p is bound outside of the scope of x, no instance of p can have x free, as required by the existential elimination rule.

Moggi, Monads (1988, 1989, 1991)

Computational lambda-calculus and monads

Eugenio Moggi[*]
LFCS
Dept. of Comp. Sci.
University of Edinburgh
EH9 3JZ Edinburgh, UK
em@lfcs.ed.ac.uk

October 1988

Abstract

The λ-calculus is considered an useful mathematical tool in the study of programming languages, since programs can be *identified* with λ-terms. However, if one goes further and uses $\beta\eta$-conversion to prove equivalence of programs, then a gross simplification[1] is introduced, that may jeopardise the applicability of theoretical results to real situations. In this paper we introduce a new calculus based on a categorical semantics for *computations*. This calculus provides a correct basis for proving equivalence of programs, independent from any specific computational model.

$T: \mathcal{C} \to \mathcal{C}$ and two natural transformations $\eta: \mathrm{Id}_{\mathcal{C}} \dot{\to} T$ and $\mu: T^2 \dot{\to} T$ s.t.

$$\begin{tikzcd}
T^3A \arrow[r, "\mu_{TA}"] \arrow[d, "T\mu_A"'] & T^2A \arrow[d, "\mu_A"] \\
T^2A \arrow[r, "\mu_A"'] & TA
\end{tikzcd}
\qquad
\begin{tikzcd}
TA \arrow[r, "\eta_{TA}"] \arrow[dr, "\mathrm{id}_{TA}"] & T^2A \arrow[d, "\mu_A"] & TA \arrow[l, "T\eta_A"'] \arrow[dl, "\mathrm{id}_{TA}"] \\
& TA &
\end{tikzcd}$$

which satisfies also an extra **equalizing requirement***: $\eta_A: A \to TA$ is an equalizer of $\eta_T A$ and $T(\eta_A)$, i.e. for any $f: B \to TA$ s.t. $f; \eta_{TA} = f; T(\eta_A)$ there exists a unique $m: B \to A$ s.t. $f = m; \eta_A$*[3].

transformation s.t.

$$\begin{tikzcd}
1 \times TA \arrow[r, "\mathrm{t}_{1,A}"] \arrow[dr, "r_{TA}"] & T(1 \times A) \arrow[d, "Tr_A"] \\
 & TA
\end{tikzcd}$$

$$\begin{tikzcd}
(A \times B) \times TC \arrow[rr, "\mathrm{t}_{A\times B,C}"] \arrow[d, "\alpha_{A,B,TC}"'] & & T((A \times B) \times C) \arrow[d, "T\alpha_{A,B,C}"] \\
A \times (B \times TC) \arrow[r, "\mathrm{id}_A \times \mathrm{t}_{B,C}"] & A \times T(B \times C) \arrow[r, "\mathrm{t}_{A,B\times C}"] & T(A \times (B \times C))
\end{tikzcd}$$

satisfying the following diagrams:

$$\begin{tikzcd}
A \times B \arrow[rr, "\mathrm{id}_{A\times B}"] \arrow[d, "\mathrm{id}_A \times \eta_B"'] & & A \times B \arrow[d, "\eta_{A\times B}"] \\
A \times TB \arrow[rr, "\mathrm{t}_{A,B}"] & & T(A \times B) \\
A \times T^2B \arrow[u, "\mathrm{id}_A \times \mu_B"] \arrow[r, "\mathrm{t}_{A,TB}"] & T(A \times TB) \arrow[r, "T\mathrm{t}_{A,B}"] & T^2(A \times B) \arrow[u, "\mu_{A\times B}"']
\end{tikzcd}$$

RULE	SYNTAX		SEMANTICS
var	$\dfrac{}{x_1 \colon \tau_1, \ldots, x_n \colon \tau_n \vdash x_i \colon \tau_i}$	$=$	$\pi_i^n; \eta_{[\![\tau_i]\!]}$
let	$\Gamma \vdash e_1 \colon \tau_1$	$=$	g_1
	$\Gamma, x_1 \colon \tau_1 \vdash e_2 \colon \tau_2$	$=$	g_2
	$\Gamma \vdash (\mathrm{let}\, x_1 {=} e_1 \,\mathrm{in}\, e_2) \colon \tau_2$	$=$	$\langle \mathrm{id}_{[\![\Gamma]\!]}, g_1 \rangle; \mathrm{t}_{[\![\Gamma]\!],[\![\tau_1]\!]}; T g_2; \mu_{[\![\tau_2]\!]}$
$*$	$\dfrac{}{\Gamma \vdash * \colon 1}$	$=$	$!_{[\![\Gamma]\!]}; \eta_1$
			where $!_A$ is the only morphism from A to 1
$\langle\rangle$	$\Gamma \vdash e_1 \colon \tau_1$	$=$	g_1
	$\Gamma \vdash e_2 \colon \tau_2$	$=$	g_2
	$\Gamma \vdash \langle e_1, e_2 \rangle \colon \tau_1 \times \tau_2$	$=$	$\langle g_1, g_2 \rangle; \psi_{[\![\tau_1]\!],[\![\tau_2]\!]}$
π_{i}	$\Gamma \vdash e \colon \tau_1 \times \tau_2$	$=$	g
	$\Gamma \vdash \pi_i(e) \colon \tau_1$	$=$	$g; T(\pi_i)$

Table 3: Terms and their interpretation

- *let is the notion of reduction > defined by the following clauses:*

$$
\begin{array}{rl}
id & (\mathrm{let}\, x{=}e \,\mathrm{in}\, x) > e \\
comp & (\mathrm{let}\, x_2{=}(\mathrm{let}\, x_1{=}e_1 \,\mathrm{in}\, e_2) \,\mathrm{in}\, e) > (\mathrm{let}\, x_1{=}e_1 \,\mathrm{in}\, (\mathrm{let}\, x_2{=}e_2 \,\mathrm{in}\, e)) \\
let_v & (\mathrm{let}\, x{=}v \,\mathrm{in}\, e) > e[x{:=}\, v] \\
let.1 & nv(e) > (\mathrm{let}\, x{=}nv \,\mathrm{in}\, x(e)) \\
let.2 & v(nv) > (\mathrm{let}\, x{=}nv \,\mathrm{in}\, v(x))
\end{array}
$$

Milner, Parrow, Walker, Pi-calculus (1992)

INFORMATION AND COMPUTATION **100**, 1–40 (1992)

A Calculus of Mobile Processes, I

ROBIN MILNER

University of Edinburgh, Scotland

JOACHIM PARROW

Swedish Institute of Computer Science, Kista, Sweden

AND

DAVID WALKER

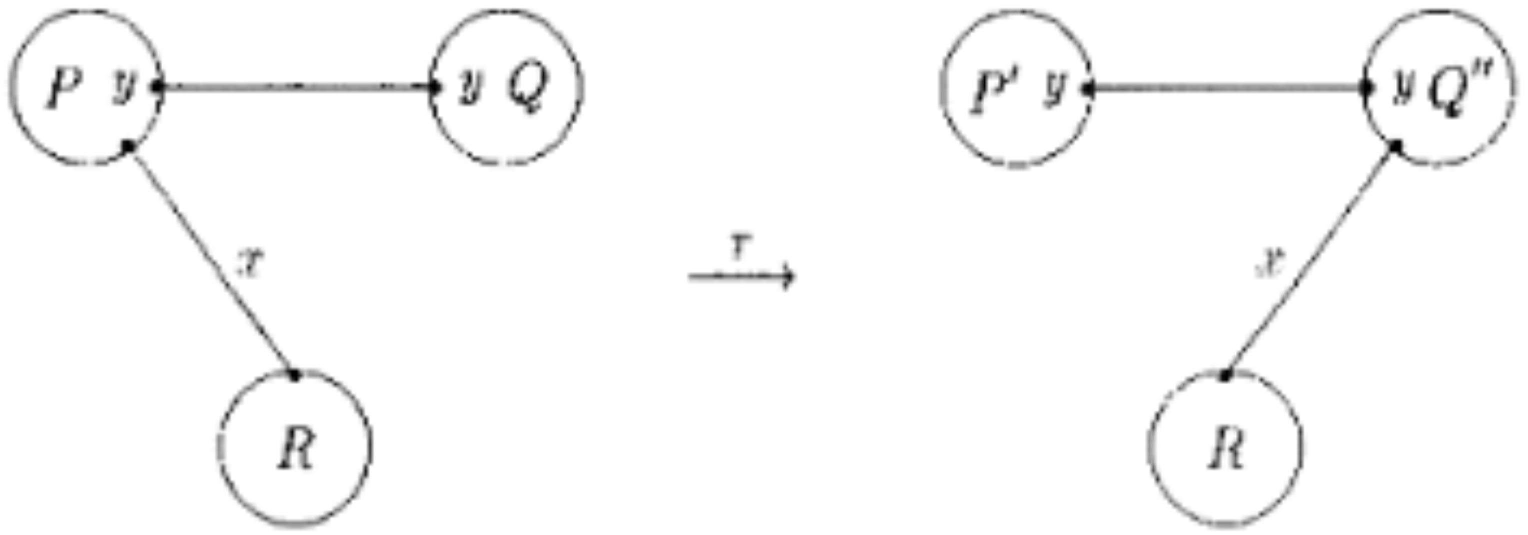
University of Warwick, England

We present the π-calculus, a calculus of communicating systems in which one can naturally express processes which have changing structure. Not only may the component agents of a system be arbitrarily linked, but a communication between neighbours may carry information which changes that linkage. The calculus is an extension of the process algebra CCS, following work by Engberg and Nielsen, who added mobility to CCS while preserving its algebraic properties. The π-calculus gains simplicity by removing all distinction between variables and constants; communication links are identified by *names*, and computation is represented purely as the communication of names across links. After an illustrated description of how the π-calculus generalises conventional process algebras in treating mobility, several examples exploiting mobility are given in some detail. The important examples are the encoding into the π-calculus of higher-order functions (the λ-calculus and combinatory algebra), the transmission of processes as values, and the representation of data structures as processes. The paper continues by presenting the algebraic theory of *strong bisimilarity* and *strong equivalence*, including a new notion of equivalence indexed by *distinctions*—i.e., assumptions of inequality among names. These theories are based upon a semantics in terms of a labeled transition system and a notion of *strong bisimulation*, both of which are expounded in detail in a companion paper. We also report briefly on work-in-progress based upon the corresponding notion of *weak* bisimulation, in which internal actions cannot be observed.

The syntax of agents may be summarized as follows:

$$
\begin{aligned}
P ::= {} & \mathbf{0} \\
& \mid P_1 + P_2 \\
& \mid \bar{y}x.P \\
& \mid y(x).P \\
& \mid \tau.P \\
& \mid P_1 \mid P_2 \\
& \mid (x)P \\
& \mid [x = y]P \\
& \mid A(y_1, \ldots, y_n)
\end{aligned}
$$

The situation is not much different when the link y between P and Q is private. In this case the proper flow graphs are



The privacy of the y-link is represented by a restriction, so the transition is now

$$(y)(\bar{y}x.P' \mid y(z).Q') \mid R \xrightarrow{\tau} (y)(P' \mid Q'\{x/z\}) \mid R. \tag{2}$$

We translate each λ-term M into a map $[\![M]\!]$ from names to agents. To understand the agent $[\![M]\!]u$, where u is any name ($\in \mathcal{N} - \mathcal{V}$), we may think of u as *pointing* to the argument sequence appropriate for a particular occurrence of M. More precisely, if M eventually reduces to a λ-abstraction $\lambda x M'$, then the corresponding derivative of $[\![M]\!]u$ will receive along the link u two names: a pointer to M's first argument, and a pointer to the rest of its argument sequence. Thus u represents a list, just as lists are represented in Example 7. Here is the full definition of the encoding function $[\![\ \ ]\!]$:

$$[\![\lambda x M]\!]u \overset{\text{def}}{=} u(x)(v).[\![M]\!]v \tag{33}$$

$$[\![x]\!]u \overset{\text{def}}{=} \bar{x}u \tag{34}$$

$$[\![MN]\!]u \overset{\text{def}}{=} (v)([\![M]\!]v \mid (x)\bar{v}xu.x(w).[\![N]\!]w) \qquad (x \text{ not free in } N) \tag{35}$$

Note that the variable x occurs free in the translation of the λ-term x; hence in Eq. (33) x will normally occur free in $[\![M]\!]v$.

Abramsky, Bellin, Scott, Proofs as Processes (1994)

Theoretical Computer Science 135 (1994) 5–9
Elsevier

Proofs as processes

Samson Abramsky

Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK

The main purpose of this short paper is to serve as an introduction to the following paper, "On the π-calculus and linear logic", by Gianluigi Bellin and Philip Scott. The circumstances from which it arises are as follows. In June 1991 I gave a lecture on "Proofs as processes" at the symposium held at Tel-Aviv University to celebrate Boris Trakhtenbrot's 70th birthday [6]. Material from this lecture also appeared in lectures subsequently given at the International Category Theory Meeting in Montreal (June 1991) and the London Mathematical Society Symposium on Category Theory in Computer Science in Durham (July 1991). The material was also presented in my tutorial lecture on linear logic given at the International Logic Programming Symposium in San Diego in October 1991. The lecture notes for these talks, particularly the tutorial lecture, were quite widely circulated; however, I did not write up a paper for publication, for reasons shortly to be explained.

My point of departure in this work was the "propositions as types" paradigm (encompassing such notions as "Curry–Howard isomorphism", realizability, functional interpretation and BHK semantics) familiar from proof theory and typed functional programming (see e.g. [9]). In this paradigm, we have the correspondences

Formulas	Types
Proofs	(Functional) Programs
Normalisation of proofs	Computation

Thus a familiar proof rule such as implication elimination is equated to the type inference rule for function application:

$$\frac{\Gamma \vdash t : A \Rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B}.$$

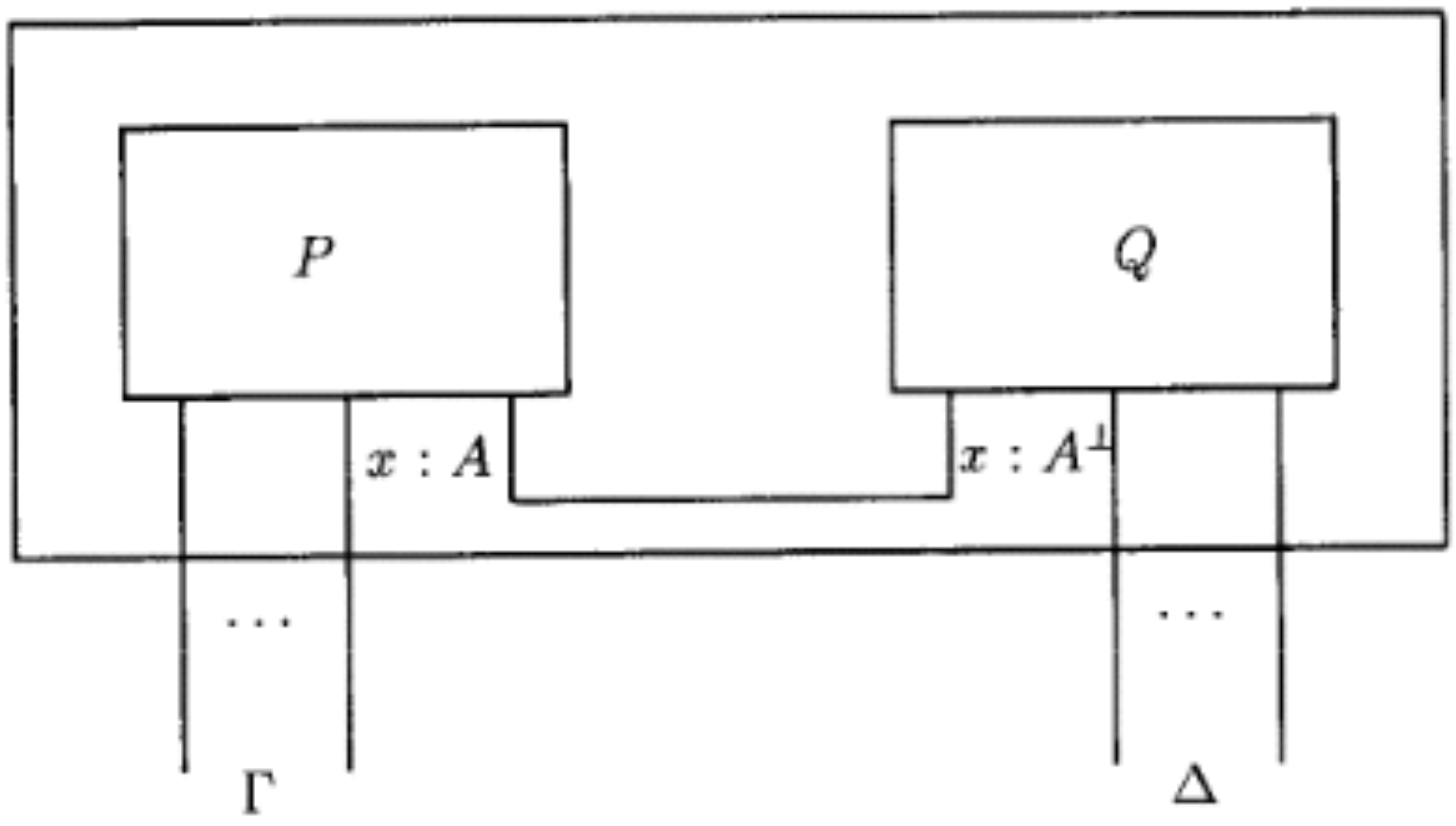


Fig. 1.

$$\frac{\vdash \Gamma, A \quad \vdash \Delta, A^{\perp}}{\vdash \Gamma, \Delta}$$

Theoretical Computer Science 135 (1994) 11–65
Elsevier

On the π-calculus and linear logic

G. Bellin*

Equipe de Logique, Université de Paris VII, 2 Place Jussieu, F-75251 Paris Cedex 05, France

P.J. Scott**

Department of Mathematics, University of Ottawa, 585 King Edward, Ottawa, Ont., Canada K1N 6N5

Abstract

Bellin, G. and P.J. Scott, On the π-calculus and linear logic, Theoretical Computer Science 135 (1994) 11–65.

We detail Abramsky's "proofs-as-processes" paradigm for interpreting classical linear logic (CLL) (Girard, 1987) into a "synchronous" version of the π-calculus recently proposed by Milner (1992, 1993). The translation is given at the abstract level of proof structures. We give a detailed treatment of information flow in proof-nets and show how to mirror various evaluation strategies for proof normalization. We also give soundness and completeness results for the process–calculus translations of various fragments of CLL. The paper also gives a self-contained introduction to some of the deeper proof-theory of CLL, and its process interpretation.

3.1. *The Abramsky translation: the multiplicatives*

Logical rule	*π-translation*
$\vdash x:A, y:A^{\perp}$	$Ixy = x(a)\bar{y}\langle a\rangle$
$\dfrac{\overset{\vdots F}{\vdash \vec{w}:\Gamma, x:A} \quad \overset{\vdots G}{\vdash \vec{v}:\Delta, y:B}}{\vdash \vec{w}:\Gamma, \vec{v}:\Delta, z:A\otimes B}\otimes$	$\bigotimes_{z}^{x,y}(F,G)\vec{w}\vec{v}z = \nu xy(\bar{z}\langle xy\rangle(F\vec{w}x \parallel G\vec{v}y))$
$\dfrac{\overset{\vdots F}{\vdash \vec{w}:\Gamma, x:A, y:B}}{\vdash \vec{w}:\Gamma, z:A \parr B}\parr$	$(\parr_{z}^{x,y})(F)\vec{w}z = z(xy)F\vec{w}xy$
$\dfrac{\overset{\vdots F}{\vdash \vec{u}:\Gamma, x:C} \quad \overset{\vdots G}{\vdash \vec{v}:\Delta, y:C^{\perp}}}{\vdash \vec{u}:\Gamma, \vec{v}:\Delta}\mathit{Cut}$	$\mathit{Cut}^{z}(F,G)\vec{u}\vec{v} = \nu z(F\vec{u}\,[z/x] \parallel G\vec{v}\,[z/y])$

Honda, Kubo, Vasconcelos, Session Types (1994,1998)

LANGUAGE PRIMITIVES AND TYPE DISCIPLINE FOR STRUCTURED COMMUNICATION-BASED PROGRAMMING

KOHEI HONDA*, VASCO T. VASCONCELOS†, AND MAKOTO KUBO‡

ABSTRACT. We introduce basic language constructs and a type discipline as a foundation of structured communication-based concurrent programming. The constructs, which are easily translatable into the summation-less asynchronous π-calculus, allow programmers to organise programs as a combination of multiple flows of (possibly unbounded) reciprocal interactions in a simple and elegant way, subsuming the preceding communication primitives such as method invocation and rendez-vous. The resulting syntactic structure is exploited by a type discipline à la ML, which offers a high-level type abstraction of interactive behaviours of programs as well as guaranteeing the compatibility of interaction patterns between processes in a well-typed program. After presenting the formal semantics, the use of language constructs is illustrated through examples, and the basic syntactic results of the type discipline are established. Implementation concerns are also addressed.

*Dept. of Computer Science, University of Edinburgh, UK. †Dept. of Computer Science, University of Lisbon, Portugal. ‡Dept. of Computer Science, Chiba University of Commerce, Japan.

Wadler, Propositions as Sessions (2012, 2014)

JFP **24** (2-3): 384–418, 2014. © Cambridge University Press 2014
doi:10.1017/S095679681400001X First published online 31 January 2014

*Propositions as sessions**

PHILIP WADLER
University of Edinburgh, South Bridge, Edinburgh EH8 9YL, UK
(*e-mail:* `wadler@inf.ed.ac.uk`)

Abstract

Continuing a line of work by Abramsky (1994), Bellin and Scott (1994), and Caires and Pfenning (2010), among others, this paper presents CP, a calculus, in which propositions of classical linear logic correspond to session types. Continuing a line of work by Honda (1993), Honda *et al*. (1998), and Gay & Vasconcelos (2010), among others, this paper presents GV, a linear functional language with session types, and a translation from GV into CP. The translation formalises for the first time a connection between a standard presentation of session types and linear logic, and shows how a modification to the standard presentation yields a language free from races and deadlock, where race and deadlock freedom follows from the correspondence to linear logic.

'The new connectives of linear logic have obvious meanings in terms of parallel computation. [. . .] Linear logic is the first attempt to solve the problem of parallelism at the logical level, *i.e., by making the success of the communication process only dependent of the fact that the programs can be viewed as* proofs *of something, and are therefore sound'*.

—Girard 1987 (emphasis as in the original)

1 Introduction

Functional programmers know where they stand: upon the foundation of λ-calculus. Its canonicality is confirmed by its double discovery, once as natural deduction by Gentzen and again as λ-calculus by Church. These two formulations are related by the Curry–Howard correspondence, which takes

propositions *as* types,
proofs *as* programs, and
normalisation of proofs *as* evaluation of programs.

The correspondence arises repeatedly: Hindley's type inference corresponds to Milner's Algorithm W; Girard's System F, corresponds to Reynold's polymorphic λ-calculus; Peirce's law in classical logic corresponds to Landin's J operator (better known as call/cc).

In Abramsky (1994) and Bellin & Scott (1994), the following two rules interpret the linear connectives $\otimes$ and $\parr$.

$$\frac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, z : B}{\nu y, z. x\langle y, z\rangle.(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B} \otimes \qquad \frac{R \vdash \Theta, y : A, z : B}{x(y, z).R \vdash \Theta, x : A \parr B} \parr$$

Under their interpretation, $A \otimes B$ is the type of a channel which outputs a pair of an A and a B, while $A \parr B$ is the type of a channel which inputs a pair of an A and a B. In the rule for $\otimes$, process $\nu y,z.x\langle y,z\rangle.(P \mid Q)$ allocates fresh channels y and z, transmits the pair of channels y and z along x, and then concurrently executes P and Q. In the rule for $\parr$, process $x(y,z).R$ communicates along channel x obeying protocol $A \parr B$; it receives from x the pair of names y and z, and then executes R.

This work puts a twist on the above interpretation. Here we use the following two rules to interpret the linear connectives $\otimes$ and $\parr$.

$$\frac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, x : B}{\nu y. x\langle y\rangle.(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B} \otimes \qquad \frac{R \vdash \Theta, y : A, x : B}{x(y).R \vdash \Theta, x : A \parr B} \parr$$

Under the new interpretation, $A \otimes B$ is the type of a channel which outputs an A and then behaves as a B, while $A \parr B$ is the type of a channel which inputs an A and then behaves as a B. In the rule for $\otimes$, process $\nu y.x\langle y\rangle.(P \mid Q)$ allocates fresh variable y, transmits y along x, and then concurrently executes P and Q. In the rule for $\parr$, process $x(y).R$ receives name y along x, and then executes R.

$$\frac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, z : B}{\nu y, z. x\langle y, z\rangle.(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B} \otimes \qquad \frac{R \vdash \Theta, y : A, z : B}{x(y, z).R \vdash \Theta, x : A \parr B} \parr$$

$$\frac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, x : B}{\nu y. x\langle y\rangle.(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B} \otimes \qquad \frac{R \vdash \Theta, y : A, x : B}{x(y).R \vdash \Theta, x : A \parr B} \parr$$

Happy Birthday, LFCS!
Many happy returns!