Dagstuhl Seminar 97041:

High-Level Concurrent Languages

Date	January, 20–22, 1997
Organizers	Kohei Honda, University of Edinburgh Martin Odersky, Universität Karlsruhe Benjamin Pierce, Indiana University Gert Smolka, Universität des Saarlandes and DFKI Phil Wadler, Bell Labs, Lucent Technologies
Initiative and Local Organisation	Martin Müller, Universität des Saarlandes Joachim Niehren, Universität des Saarlandes

Computer systems are undergoing a revolution. Twenty years ago, they were centralized, isolated, and expensive. Today, they are parallel, distributed, networked, and inexpensive. However, advances in software construction have failed to keep pace with advances in hardware. To a large extent, this is a consequence of the fact that current programming languages were conceived for sequential and centralized programming.

Challenged by this state of affairs, a number of concurrent programming languages have been designed. These include Erlang, concurrent versions of ML or Haskell, and languages explicitly designed for concurrency such as Obliq, Oz, or Pict. The motivations behind the design of these languages are rather diverse, ranging from constraint programming, the development of graphical user interfaces, and multi-agent systems, to real-time and distributed programming.

Programming models should be simple, practical, high-level, and well-founded. This enables rigorous language specifications and opens the possibility for formal reasoning about programs. In the last decade considerable progress has been made in the development of sequential programming models, notably the functional and logic ones. In contrast, the methodology and formal machinery for designing models for concurrent programming is still underdeveloped. Since the late 1980's, however, it is rapidly evolving.

There have been three main lines of research, based on Hewitt's actor model, process calculi in the tradition of Hoare's CSP and Milner's CCS, and logic programming. The actor model captures Hewitt's early vision of concurrency as the most general form of computation and has been developed into various actor languages. Hoare and Milner suggest a model of concurrency based on *channel communication*. Different versions of *concurrent constraint programming* draw upon

ideas from concurrent and constraint logic programming, and integrate elements from process calculi.

The three traditions have developed rather independently without much communication or cooperation between groups, although there is urgent technical need for such interaction. Furthermore, the combined know how of these groups has reached a point which opens the possibility to make some fundamental progress. The seminar is intended to bring together researchers involved in the design, development, foundations, and applications of high-level concurrent programming languages and models.

Participants

Joe Armstrong, Ericsson Computer Science Laboratory
Luca Cardelli, Digital Equipment Corporation
Mads Dam, Swedish Institute of Computer Science
Denys Duchier, Universität des Saarlandes
Cédric Fournet, INRIA Rocquencourt
Robert H. Halstead Jr., Digital Equipment Corporation
Seif Haridi, Swedish Institute of Computer Science
Matthew Hennessy, Sussex University
Kohei Honda, University of Edinburg
Jean-Jacques Lévy, INRIA Rocquencourt
Ugo Montanari, University of Pisa
Martin Müller, Universität des Saarlandes
Joachim Niehren, Universität des Saarlandes
Oscar Nierstrasz, University of Berne
Martin Odersky, Universität Karlsruhe
Benjamin Pierce, Indiana University
Andreas Podelski, Max Planck Institut, Saarbrücken
Antonio Porto, DI-FCT/UNL, Lisbon
Didier Rémy, INRIA Rocquencourt
John Reppy, AT&T Bell Laboratories, New Jersey
Enno Scholz, Freie Universität Berlin
Christian Schulte, Universität des Saarlandes
Peter Sewell, Cambridge University
Gert Smolka, Universität des Saarlandes
Kazunori Ueda, Waseda University
Peter Van Roy, Université catholique de Louvain
Vasco T. Vasconcelos, University of Lisbon
Philip Wadler, Bell Labs, Lucent Technologies
Nobuko Yoshida, University of Edinburgh

Table of Abstracts

Higher Order Processes in Erlang
Joe Armstrong, Ericsson Computer Science Laboratory **7**

The Obliq Model of Distributed Computation & Mobile Ambients
Luca Cardelli, Digital Equipment Corporation, Systems Research Center **7**

Verification of Erlang Programs
Mads Dam, Swedish Institute of Computer Science **7**

A Calculus of Mobile Agents
Cédric Fournet, INRIA Rocquencourt **8**

Scheduling Issues in Parallel Programming
Robert H. Halstead, Jr., Digital Equipment Corporation, Cambridge Research Lab **8**

A Fully Abstract Denotational Model for a subset of Facile
Matthew Hennessy, Sussex University **9**

Games as Behavioural Types
Kohei Honda, University of Edinburg **9**

Computing with Tiles
Ugo Montanari, Dipartimento di Informatica, University of Pisa **10**

Typed Concurrent Programming with Logic Variables
Martin Müller, Universität des Saarlandes **10**

Call-By-Need versus Call-by-Value Complexity
Joachim Niehren, Universität des Saarlandes **10**

Scripting, Composition and Coordination
Oscar Nierstrasz, University of Berne **11**

Process Calculus in Direct Style
Martin Odersky, University of South Australia **11**

Pict: A Programming Language Based on the Pi-Calculus
Benjamin C. Pierce, Indiana University **11**

Temporal Properties of Concurrent Constraint Programs
Andreas Podelski, Max Planck Institut, Saarbrücken **12**

The TAO road to high-level concurrent programming
Antonio Porto, DI-FCT/UNL, Lisbon, Portugal **12**

Implicit typing a la ML for the join-calculus
Didier Rémy, INRIA Rocquencourt **13**

Issues in Concurrent Language Design
John Reppy, AT&T Labs Research **13**

The Design of Distributed Oz and its Mobility Protocol
Peter Van Roy and Seif Haridi, Université catholique de Louvain and Swedish Institute of Computer Science **14**

A Monad of Imperative Streams ... and Its Application Interactive Graphics
Enno Scholz, Freie Universität Berlin **15**

Local Channel Typing for a Distributed π-calculus
Peter Sewell, Cambridge University **15**

The Oz Programming Model
Gert Smolka, DFKI and Universität des Saarlandes, Saarbrücken **16**

Diagnosis of Concurrent Logic Programs
Kazunori Ueda, Waseda University **16**

Abstracting Communications in Mobile Processes
Vasco T. Vasconcelos, University of Lisbon **17**

Largely lambda, with a slice of pi
Philip Wadler, Bell Labs, Lucent Technologies **17**

Graph Types for Monadic Mobile Processes
Nobuko Yoshida, University of Edinburgh **20**

Higher Order Processes in Erlang

Joe Armstrong, Ericsson Computer Science Laboratory

A higher order process is a process whose behaviour is paramaterised with a lambda expression. Higher order processes are to concurrent programming what higher order functions are to sequential programming.
A small number of higher order functions (map, foldl, filter, ...) prove useful for writing sequential code since they abstract out common sequential control patterns. Similarly, a small number of higher order processes can be used to abstract out common patterns of concurrency.
Using higher order processes we can arrange to divide the solution of a problem into two parts. The first part (the higher order process) contains all the concurrent code. The second part (the paramaterising lambda expression) contains only sequential code.
By suitable choice of higher order processes we can arrange that most of the code in a large concurrent system can be written as strictly sequential code.
In this talk I presented three higher order processes called client-server, worker-supervisor and event-manager. I also talked about how these were used in a large (300,000+ line) application program written in Erlang.

The Obliq Model of Distributed Computation & Mobile Ambients

Luca Cardelli, Digital Equipment Corporation, Systems Research Center

I described the Obliq model of distributed computation, based on the free flow of network pointers. I argued that this model is great for a local-area-network or an Intranet, but does not work well over the Internet because of firewalls and widespread unreliability.
I then discussed the Ambient model of mobile computation, which is based on primitives that are better suited to Internet situations. Ambients are named, mobile collections of threads and sub-ambients. The names are used for access control and for mobility synchronization. I presented an example of pure mobility, based on passengers being transported by trains between stations.

Mobile Ambients is based on joint work with Andrew Gordon, Cambridge Computer Lab

Verification of Erlang Programs

Mads Dam, Swedish Institute of Computer Science

We consider the problem of verifying general temporal and functional properties of programs in a core fragment of the Erlang programming language. Erlang is a first-order call-by-value concurrent functional programming language. Important features which verification needs to address are dynamic spawning of processes and asynchronous buffered communication. A symbolic operational semantics of a "core Erlang" is given, along with a temporal logic based on the modal mu-calculus extended with the first-order language of equality plus a number of Erlang-specific predicates and operations. We suggest an approach to verification based on stepwise refinement of proof goals. The difficult issues are how to deal with message queues and control structures which lead to unbounded growth of state spaces: dynamic process spawning and non-tail recursion. We apply a compositional technique based on loop detection which has previously been applied to CCS and the pi-calculus. A proof system and an example proof of an RPC application is outlined.

A Calculus of Mobile Agents

Cédric Fournet, INRIA Rocquencourt

We propose the distributed join-calculus as a formal setting that fits the needs of distributed programming with global communication, agent-based migration, and partial failure of the system.

By adding reflexion to the chemical machine of Berry and Boudol, we first obtain a model of concurrency that is consistent with mobility and distribution. This provides a natural extension of functional programming with concurrency (fork- and join- synchronization) and with some object-oriented features. This can also be seen as a process calculus which we proved equivalent to the pi-calculus of Milner, Parrow and Walker.

We then define our calculus for mobile agents as an extension of the join-calculus, and we give its refined distributed chemical semantics. Communication, migration, failure, and failure detection are precisely defined as atomic reduction steps. However, they can still be efficiently implemented in a mostly asynchronous distributed setting. Various examples illustrate how to express remote executions, dynamic loading of remote resources and protocols with mobile agents.

We use this setting as the core of a programming language; a distributed implementation is under way, and was demonstrated during the workshop.
(Based on joint work with Georges Gonthier, Jean-Jacques Levy, Luc Maranget and Didier Remy).

Scheduling Issues in Parallel Programming

Robert H. Halstead, Jr., Digital Equipment Corporation, Cambridge Research Lab

The speed of a parallel computation is often heavily influenced by scheduling issues. Good scheduling can improve the performance of a program in two ways: by reducing overhead and by focusing resources on the most important computations. For programs expressed in terms of fine-grained threads, the "lazy task creation" implementation technique can dramatically reduce overhead by adaptively coarsening the grain of the threads actually created at run time; however, for strong models of fairness, lazy task creation is not a legal implementation technique.

Other programs benefit from speculative computing, where some computations are initiated before it is certain that their values will be needed. The "sponsors" abstraction can provide information to control speculative computations, but it is still unclear how best to present it to programmers. For both lazy task creation and sponsors, the challenge is to resolve the semantic issues in a way that helps programmers and also enables high-performance implementations.

A Fully Abstract Denotational Model for a subset of Facile

Matthew Hennessy, Sussex University

In this talk I will report on some recent work with Takis Hartonas where we study an applied typed call-by-value λ-calculus which in addition to the usual types for higher-order functions contains an extra type called proc, for processes. The constructors for terms of this type are similar to those found in standard process calculi such as CCS.

We first give an operational semantics for this language in terms of a labelled transition system which is then used to give a behavioural preorder based on contextual; the expression N dominates M if in every appropriate context if M can produce a boolean value then so can N.

Games as Behavioural Types

Kohei Honda, University of Edinburg

The talk discusses some elements of game semantics, especially the basic notions of Hyland-Ong games, from a viewpoint of behavioural types, i.e. the classification of interactive behaviours of processes. The introduction tries to be simple,

intuitive, but exact. A new presentation is used with, hopefully, added clarity. In particular we use the exact correspondence between name passing processes and strategies to clarify the structure of interaction sequences strategies are engaged in. We discuss how these ideas would be useful to gain high-level abstraction of varied process behaviours.

Computing with Tiles

Ugo Montanari, Dipartimento di Informatica, University of Pisa

In the talk we introduce a model for a wide class of computational systems, whose behaviours can be described by certain rewriting rules. We gathered our inspiration both from the worlds of term rewriting, in particular from the rewriting logic framework of Meseguer, and of concurrency theory: among the others, we considered the approaches based on structured operational semantics (SOS) (Plotkin), on contexts systems (Larsen and Xinxin) and on structured transition systems (Corradini and Montanari).
Our model recollects many properties of these sources: it provides a compositional way to describe both the states and the sequences of transitions performed by a given system, stressing their distributed nature. Forthermore, a suitable notion of typed proof allws us to take into account also those formalisms relying on the notions of synchronization and side effects to determine the actual behaviour of a system.
The model has been applied to a variety of computational paradigms. In the talk, we considered three languages: CCS, Horn clauses and π-calculus. In the CCS case, the basic tiles directly correspond to SOS rules. For logic programming (Horn clauses with SLD resolution), the tiles correspond to clauses and to pullback squares representing unification steps. Furthermore, all pullback squares can be derived from a finite number of basic squares which correspond to the steps of the unification algorithm. For π-calculus, the basic tiles are those corresponding to transitions of recursive sequential processes and those defining the behaviour of parallel composition and restriction.

Typed Concurrent Programming with Logic Variables

Martin Müller, Universität des Saarlandes

We present a concurrent higher-order programming language called Plain and a concomitant static type system. Plain is based on logic variables and computes with possibly partial data structures. The data structures of Plain are procedures, cells, and records. Plain's type system features record-based subtyping, bounded existential polymorphism, and access modalities distinguishing between reading and writing.
Based on joint work with Joachim Niehren and Gert Smolka

Call-By-Need versus Call-by-Value Complexity

Joachim Niehren, Universität des Saarlandes

Up to overhead the complexity of call-by-need evaluation is smaller than the complexity of call-by-value evaluation. This is a folk theorem that I prove in my talk. The idea is to compare call-by-need evalution with call-by-value evaluation within a single calculus. This calculus has to be flexible enough for allowing both evaluation strategies. A good candidate is the call-by-let λ-calculus. By this choice, I am able to simplify a previous proof based on the π-calculus, which I presented at POPL 96.

Scripting, Composition and Coordination

Oscar Nierstrasz, University of Berne

We would like to view open systems as flexible configurations of software components glued together by general-purpose connectors. More specifically, we would like to build open, distributed applications from components glued together by connectors that realize generic coordination abstractions.
A *scripting language* allows you to write a "script" that specifies how components are glued together. A *composition language* further allows you to define new kinds of glue (i.e., connectors) that implement (for example) generic coordination abstractions. A (typed) composition language clearly needs to support objects, components, concurrency, subtyping and genericity. Other features, like higher-order abstractions and some reflective capabilities also seem to be necessary to specify general-purpose connectors, but it is not clear what set of features would constitute the minimum requirements for a composition language.
We are currently implementing an experimental framework of coordination components and connectors in Java (and Pizza) in an attempt to make these requirements more precise, and we are using Pict as an executable specification language to develop a formal model of concurrent objects, components and connectors.

Process Calculus in Direct Style

Martin Odersky, University of South Australia

We study an extension of asynchronous π-calculus where names can be returned from processes. We show that with this simple extension an extensive range of functional, state-based and control-based programming constructs can be expressed by macro expansions, similar to Church-encodings in lambda calculus. The calculus has a mapping into asynchonous π-calculus which closely corresponds to Plotkin's CPS transform for call-by-value λ-calculus.

Pict: A Programming Language Based on the Pi-Calculus

Benjamin C. Pierce, Indiana University

PICT is a programming language in the ML tradition, formed by adding a layer of convenient syntactic sugar and a static type system to a tiny core.
The core language, Milner's pi-calculus, has been used as a theoretical foundation for a broad class of concurrent computations. The goal in PICT is to identify high-level idioms that arise naturally when these primitives are used to build working programs – idioms such as basic data structures, protocols for returning results, higher-order programming, selective communication, and concurrent objects.
The type system integrates a number of features found in recent work on theoretical foundations for typed object-oriented languages: higher-order polymorphism, simple recursive types, subtyping, and a powerful partial type inference algorithm.

Temporal Properties of Concurrent Constraint Programs

Andreas Podelski, Max Planck Institut, Saarbrücken

The existing automated verification methods apply mainly to those concurrent systems where the number of concurrent processes is statically fixed and, moreover, the control flow depends essentially on only finitely many data. In this paper, we consider concurrent constraint programs with empty guards (cc* programs) for specifying systems that do not underlie these limitations. cc* programs are abstractions of cc systems with non-empty guards. We define a framework of intermittent and invariant program assertions for specifying temporal-logic properties ("liveness", "safety") of the executions of cc programs. We show that one can characterize the properties by the least and greatest fixpoints of a logical-consequence operator associated with of a cc* program, provided that the constraints have the so-called saturation property. For example, the constraints underlying the cc language Oz, equations over infinite trees, do have the saturation property. The characterization allows us to apply abstract-interpretation methods (such as set-based analysis) to verification. We obtain thus methods for *abstract debugging* and *abstract verification* of concurrent constraint programs.

The TAO road to high-level concurrent programming

Antonio Porto, DI-FCT/UNL, Lisbon, Portugal

TAO is an abstract concurrent model/language that aims to be general purpose (suitable for database systems, operating systems, the Web, etc.), high-level (abstraction and compositionality features allowing a direct representation of concepts at arbitrary conceptual levels), and promoting the separation of concerns between computation (functions, relations) and coordination (processes, interaction, change)

It is a task-oriented model: the state of an agent contains a task (expressing future activities) and a database (expressing current facts), and the operational model is one of state transitions driven by the task, whereby task reductions occur and the database may change. The basic actions are queries and commands on the database, and these are defined in very general terms, through the entailment relation in the space of situations which give semantics to the database. A query can embody an arbitrarily complex transformational computation, and commands force (nonmonotonic) database updates.

Tasks can be composed in parallel but also with generic sequential, synchronous, choice and atomization operators, which together promote a more high-level description of certain processes. For procedural abstraction there are named tasks and recursive task decomposition rules. Lexically scoped logical variables in tasks further add to high-level expressivity. The way they work puts no constraints on the data syntax, therefore higher-order features are available.

There is a recursive structure of agents for dynamic forms of locality. An agent can have named subagents besides its task and database, and a task can be delegated to a named agent. This works as a remote procedure call that in combination with sequentiality provides a high-level mechanism, as low-level handshaking protocols are hidden in the implementation. The agent structure, with suitable forms of visibility control, allows for a useful implicit use of contextual inheritance of rules and databases.

Implicit typing a la ML for the join-calculus

Didier Rémy, INRIA Rocquencourt

We adapt the Damas-Milner typing discipline to the join-calculus. The main result is a new generalization criterion that extends the polymorphism of ML to join-definitions. We prove the correctness of our typing rules with regards to a chemical semantics. We also relate typed extensions of the core join-calculus to functional languages.

Based on joint work with Cédric Fournet, Cosimo Laneve, and Luc Maranget

Issues in Concurrent Language Design

John Reppy, AT&T Labs Research

Writing correct and robust concurrent programs is hard, and the most important tool the programmer has for this task is the concurrent language she is using. I have been thinking about the design of concurrent programming languages from this perspective for the past ten years, or so, and have developed some strong opinions (as well as some languages). This talk presents a biased view of many of the issues facing a concurrent language designer.

A good concurrent language should support modular programming, which means that there need to be mechanisms for abstracting and composing concurrent behaviors. Another important feature is the expressiveness of the primitives; if they are too low-level, the programmer must expend significant effort implementing higher-level concurrency mechanisms. On the other hand, higher level mechanisms are inflexible, because the hardwire in complex patterns of interaction (e.g., Ada's rendezvous mechanism). The ideal situation is for the language to provide a reasonable set of low-level primitives and mechanisms for composing higher-level operations in a uniform way. Another important issue are synchronization guarantees provided by the primitives. While stronger guarantees may carry some implementation cost and run-time overhead, they provide a more robust programming model.

The language Concurrent ML (CML) reflects this design philosophy. Its most important feature is support for first-class synchronous abstractions, which allows the construction of libraries of higher-level and/or application-specific communication and synchronization abstractions. I give an example of the distributed implementation of Linda-style tuple spaces as a CML library. The resulting communication operations can be used in exactly the same contexts as those operations that are builtin to CML.

The slides for this talk are available on the CML homepage at:
http://www.research.att.com/ jhr/sml/cml.

The Design of Distributed Oz and its Mobility Protocol

Peter Van Roy and Seif Haridi, Université catholique de Louvain and Swedish Institute of Computer Science

We argue that *mobility between sites* should be part of the basic language semantics in a language for distributed programming. We present a language, Distributed Oz, in which all language entities are extended with a distributed behavior. This behavior is carefully designed to respect a simple and expressive language semantics when sites are disregarded. This allows transparent programming, i.e., a computation behaves correctly independently of how it is partitioned

between sites. We argue that extending the language semantics with *mobility control* enables *efficient* distributed programming by giving the programmer a simple and effective control over network communication patterns. Mobility control is the ability for objects to migrate between sites or to remain stationary at one site. In this way, the syntax and semantics of objects are the same regardless of whether they implement stationary servers or mobile agents. We show how to use Distributed Oz to program arbitrary migratory behavior. We give the language semantics and its distributed refinement. We formally specify the mobility protocol and prove that it implements the language semantics. Distributed Oz has been implemented as an extension to the publicly-available Oz 2.0 system.

A Monad of Imperative Streams ... and Its Application Interactive Graphics

Enno Scholz, Freie Universität Berlin

A calculus is presented which is suitable for performing concurrent I/O in a functional programming language. It is defined as a monad on top of the functional language Haskell. The monad is a conservative extension of Haskell's IO monad. Whereas an object of type IO a represents an imperative program which, at the end of its execution, produces a value to type a, an object of the new St monad represents an imperative program which, at arbitrary times, may produce values of type a. Thus, imperative programs may be interleaved in a nonpreemptive way. Moreover, functional (= static) relationships may be established between imperative, stateful (= dynamic) objects.
It is demonstrated how the St monad is used in the PIDGETS framework to program interactive graphics.

Local Channel Typing for a Distributed ß-calculus

Peter Sewell, Cambridge University

In the distributed setting there are essential differences (in performance and failure behaviour) between local and global communication. Nonetheless, *as far as possible* a programming language for distributed migratory computation should allow the two to be accessed uniformly.
I gave a distributed π-calculus, which might be used as a partial basis for a distributed Pict-like language, and its structural congruence/reduction semantics. It integrates location and migration primitives, based on those of the Distributed Join Calculus, with asynchronous π.
I went on to give a type system in which the input and output capabilities for channels may be either global or constrained to be local. This allows optimization,

e.g. for communication on channels whose receivers are constrained to be at the same location as the channel. Subtyping allows communications to be accessed uniformly.

The Oz Programming Model

Gert Smolka, DFKI and Universität des Saarlandes, Saarbrücken

The Oz Programming Model (OPM) is a concurrent programming model based on logic variables. It can express eager and lazy functional programming and concurrent object-oriented abstractions. The primitive data structures of OPM are first-class procedures, cells, names, and abstract values. Control is based on sequential composition and thread creation. Basic synchronization is automatic since statements block until their input variables are bound to (partial) data structures.
The talk intoduced OPM and illustrated its expressivity with examples including higher-order functions, locks, channels, records, and abstract types.
The talk did not cover the constraint extension of OPM, which provides for constraint programming by adding a full constraint store, propagators, and first-class spaces. Extended OPM serves as the basis of Oz, a programming system that was demonstrated at night in Dagstuhl's wine cellar.
Volume 1000 of Springer's LNCS series contains an introduction to a previous version of OPM not having sequential composition and abstract values.

Diagnosis of Concurrent Logic Programs

Kazunori Ueda, Waseda University

Strong moding and constraint-based mode analysis are expected to play fundamental roles in debugging concurrent logic/constraint programs as well as in establishing the consistency of communication protocols and in optimization. Mode analysis of Moded Flat GHC is a constraint satisfaction problem with many simple mode constraints, and can be solved efficiently by unification over feature graphs. In practice, however, it is important to be able to analyze non-well-moded programs (programs whose mode constraints are inconsistent) and present plausible "reasons" of inconsistency to the programmers in the absence of mode declarations.
I discussed the application of strong moding to systematic and efficient static program debugging. The basic idea, which turned out to work well at least for small programs, is to find a minimal inconsistent subset from an inconsistent set of mode constraints and indicate the symbols (or symbol occurrences) in the program text that imposed those constraints. A bug can be pinpointed better by

finding more than one overlapping minimal subset. These ideas can be readily extended to finding multiple bugs at once. For large programs, stratification of predicates narrows search space and produces more intuitive explanations. Stratification plays a fundamental role in introducing mode polymorphism as well.

1. *K. Ueda, and M. Morita, Moded Flat GHC and Its Message-Oriented Implementation Technique. New Generation Computing, Vol.13, No.1 (1994), pp.3-43.*

2. *K. Ueda, Experiences with Strong Moding in Concurrent Logic/Constraint Programming. In Proc. Int. Workshop on Parallel Symbolic Languages and Systems, LNCS 1068, Springer, 1996, pp.134-153.*

3. *K. Cho and K. Ueda, Diagnosing Non-Well-Moded Concurrent Logic Programs. In Proc. 1996 Joint International Conference and Symposium on Logic Programming (JICSLP'96), M. Maher (ed.), The MIT Press, 1996, pp.215-229.*

Abstracting Communications in Mobile Processes

Vasco T. Vasconcelos, University of Lisbon

Witnessing the increase of complexity on the objects that names may carry in process algebras — from CCS, through the (monadic and then the polyadic) pi-calculus, to the calculus of objects (where names carry a label together with a tuple of names) — we propose a framework where communications are taken from an abstract universe.

The calculus, parametric on what processes may exchange, is based on the asynchronous pi-calculus, the novelty being the usage of "located guarded-processes" as receptors. Such a process fires the body of a guard that matches the incoming message.

Types emerge from the actual syntax of communications by means of a choice and a "carries" constructor. A type assignment system ensures that well-typed programs will not go wrong.

Largely lambda, with a slice of pi

Philip Wadler, Bell Labs, Lucent Technologies

Lambda calculus remains a source of inspiration for work on concurrent calculi, and so the first part of the talk summarised some recent developments in lambda calculus. The second part of the talk reversed the usual idea of embedding lambda calculus within pi calculus, and compared two sets of primitives for embedding pi calculus within lambda calculus.

PART I: Some recent developments in lambda calculus. * Moggi's computational calculus, lambda-c, was first proposed in his 1988 technical report on monads. It has the following grammar and laws.

$$\begin{array}{ll} \text{terms} & L, M, N ::= V \mid P \\ \text{values} & V ::= x \mid \lambda x.N \\ \text{non-values} & P ::= LM \mid \text{ let } x = M \text{ in } N \end{array}$$

$$\begin{array}{ll} \text{(beta.v)} & (\lambda x.N)V \longrightarrow N[x := V] \\ \text{(beta.let)} & \text{ let } x = V \text{ in } N \longrightarrow N[x := V] \\ \text{(eta.v)} & \lambda x.(Vx) \longrightarrow V \\ \text{(eta.l)} & \text{ let } x = M \text{ in } x \longrightarrow M \\ \text{(assoc)} & \text{ let } y = (\text{ let } x = L \text{ in } M) \text{ in } N \longrightarrow \text{ let } x = L \text{ in } (\text{ let } y = M \text{ in } N) \\ \text{(let.1)} & PM \longrightarrow \text{ let } x = P \text{ in } xM \\ \text{(let.2)} & VP \longrightarrow \text{ let } y = P \text{ in } Vy \end{array}$$

This calculus contains Plotkin's call-by-value calculus, lambda-v, which consists onlyof (beta.v) and (eta.v). It is just large enough to have as equalities all equations that must hold between any two terms with side effects (where the general notion of side effect may be modeled using a monad).

* In particular, lambda-c allows us to strengthen Plotkin's classic CPS results. We view CPS as a translation from lambda-c to lambda-cps (the smallest subset of lambda-v that contains in the image of the CPS translation and is closed under reduction). Write M* for the CPS translation of a lambda-c term M, and N♯ for the inverse CPS translation of a lambda-cps term N. Then we have

$$\text{lambda-c } \vdash M \longrightarrow\!\!\!\!\!\rightarrow N\sharp \text{ iff lambda-cps } \vdash M* \longrightarrow\!\!\!\!\!\rightarrow N.$$

This is an instance of a Galois connection (or an adjoint).

* A variant of lambda-c provides a better model of space as well as time. Replace (beta.v) and (beta.let) by the following.

$$\begin{array}{ll} (I) & (\lambda x.N)M \longrightarrow \text{ let } x = M \text{ in } N \\ (V) & \text{ let } x = V \text{ in } C[x] \longrightarrow \text{ let } x = V \text{ in } C[V] \\ (G.v) & \text{ let } x = V \text{ in } N \longrightarrow N, \qquad \text{if x not free in N} \end{array}$$

* A further variant provides a calculus that models call-by-need rather than call-by-value. Replace (G.v) by the following.

$$(G) \quad \text{let } x = M \text{ in } N \longrightarrow n, \qquad \text{if x not free in N}$$

The let terms preserve sharing, while the switch from (G.v) to (G) means that an unneeded term may be discarded without first being evaluated.

Further discussion of some of these points can be found in the papers:

1. *A reflection on call-by-value. Amr Sabry and Philip Wadler. International Conference on Functional Programming, ACM Press, Philadelphia, May 1996.*

2. *A call-by-need lambda calculus. Zena Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. 22'nd Symposium on Principles of Programming Languages, ACM Press, San Francisco, California, January 1995.*

3. *Lazy vs. strict. Philip Wadler. ACM Computing Surveys, June 1996.*

PART II: Embedding pi in lambda * One way to embed pi calculus in lambda calculus is to augment the lambda calculus with one constant for each construct of the pi calculus, modeling binding with higher-order functions in the way first suggested by Church. SML notation is used for the lambda calculus side, where Ch is the type of channels, and Pr the type of processes.

$$\begin{array}{rcll} P, Q & ::= & \nu x.P & new : (Ch \to Pr) \to Pr \\ & & !x[\overline{y}].P & send : Ch * Chlist * Pr \to Pr \\ & & ?x[\overline{z}].Q & recv : Ch * (Chlist \to Pr) \to Pr \\ & & P|Q & op|| : Pr * Pr \to Pr \\ & & 0 & none : Pr \end{array}$$

For example, here is a pi calculus term and its encoding.

$$\begin{array}{l} \nu x.\nu y.new(fnx \Rightarrow new(fny \Rightarrow \\ !x[y].Psend(x,[y],P) \\ |?x[z].Q||recv(x, fn[z] \Rightarrow Q) \end{array}$$

* Here is a second set of constants that may be used to represent concurrency, akin to the monadic style used in Concurrent Haskell. It is also surprisingly close to the style used in Concurrent ML. The type Ev corresponds to events in Concurrent ML, or the monad in Concurrent Haskell. The type unit has one value, written ().

$$\begin{array}{l} op >>=' aEv * ('a \to' bEv) \to' bEv \\ op >>: unitEv *' bEv \to' bEv \\ return :' a \to' aEv \\ new2 : ChEv \\ send2 : Ch * Chlist \to unitEv \\ recv2 : Ch \to (Chlist)Ev \\ fork2 : unitEv \to unitEv \end{array}$$

For example, here is the same term as above, encoded in the new style.

$$\begin{array}{l} new2 >>= (fnx \Rightarrow new2 >>= (fny \Rightarrow \\ fork2(send2(x,[y]) >> P) >> \\ recv2(x) >>= (fn[z] \Rightarrow Q))) \end{array}$$

* Here is how constants in the second style may be defined in terms of those in the first style. Those familiar with monads will recognize the monadic encoding

of CPS.

$$\begin{array}{rcl} type'aEv & = & ('a \to Pr) \to Pr \\ e >>= f & = & fnc \Rightarrow e(fnx \Rightarrow fxc) \\ return(x) & = & fnc \Rightarrow cx \\ send2(x, \overline{y}) & = & fnc \Rightarrow send(x, \overline{y}, c()) \\ recv2(x) & = & fnc \Rightarrow recv(x, fn\overline{z} \Rightarrow c\overline{z}) \\ fork2(e) & = & fnc \Rightarrow e(fn() \Rightarrow none)||c() \\ e >> f & = & e >>= (fn() \Rightarrow f) \end{array}$$

* The use of a monadic style in Concurrent ML is at first blush rather a surprise. Monads are typically used to encode a call-by-value style of evaluation with side effects into a pure functional language without side effects, where the side-effecting function of type 'a → 'b corresponds to a side-effect free function of type 'a → 'b Ev. (Replace Ev by whatever monad captures the side effects you are interested in.) Under this encoding, the functions

$$\begin{array}{l} new2 : ChEv \\ send2 : Ch * Chlist \to unitEv \\ recv2 : Ch \to (Chlist)Ev \end{array}$$

might be rewritten as

$$\begin{array}{l} new3 : unit \to Ch \\ send3 : Ch * Chlist \to unit \\ recv3 : Ch \to Chlist \end{array}$$

and so why do we need the monad Ev? The answer becomes clearer when we look at the next combinator,

$$fork2 : unitEv \to unitEv$$

which would be encoded as

$$fork3 : (unit \to unit) \to unit$$

where the argument is made into a function, the usual call-by-value trick for turning computations into values. This is workable, but cumbersome, and the attractions of making the monad explicit become clearer.

John Reppy points out there is a further impetus for representing processes explicitly ith the type ('a Ev) rather than implicitly with a side-effecting function (unit → 'a). In Concurrent ML there is a choice operator, which is similar to fork except the arguments should all denote guarded processes, rather than arbitrary processes. (Choice also differs from fork in taking a list of arguments rather than two, but that is an orthogonal issue.) It is easy enough to contrive that the type ('a Ev) only denotes processes that are guarded, thereby providing an additional impetus for using the explicit ('a Ev) in preference to the implicit (unit → 'a).

Graph Types for Monadic Mobile Processes

Nobuko Yoshida, University of Edinburgh

While types for name passing calculi have been studied extensively in the context of sorting of polyadic π-calculus the type abstraction on the corresponding level is not possible in the monadic setting, which was left as an open issue by Milner [Milner 92]. We solve this problem with an extension of sorting which captures dynamic aspects of process behaviour in a simple way. Equationally this results in the full abstraction of the standard encoding of polyadic π-calculus into the monadic one: the sorted polyadic π-terms are equated by the basic behavioural equality in the polyadic calculus if and only if their encodings are equated in the basic typed behavioural equality in the monadic calculus. This is the first result of this kind we know of in the context of the encoding of polyadic name passing, which is a typical example of translation of high-level communication structures into π-calculus. The construction is general enough to be extendable to encodings of calculi with more complex operational structures.