Symbol Description

$P\|Q$	Parallel composition	$[\vec{y}]P$	Concretion
$P+Q$	summation	$x?(\vec{y}).P$	Blocking input
$!P$	replication	$x!(\vec{y}).P$	Blocking output
$(\nu\ x)P$	restriction	$x!(\!\|\vec{Q}\|\!)$	Lift
$x.P$	Sequencing	$\urcorner x\ulcorner$	Drop
$(\vec{x})P$	Abstraction	$\ulcorner P\urcorner$	Quote

Contents

List of Tables

List of Figures

Part I

Reflection, data and concurrency

Partner, Biosimilarity
505 N72nd St, Seattle, WA 98103, USA,
lgreg.meredith@biosimilarity.com

0.1 Introduction

In his Turing award acceptance paper [], Milner explicitly recognized an "equation" of the form: proposing that π-calculus was a solution for X, i.e.

$$\frac{\lambda\text{-}calculus}{sequential\ computing} = \frac{X}{concurrent\ computing}$$

that the π-calculus represented a clean, simple, compositional model of *concurrent* computation; a foundational model of concurrent computation playing the same foundational role that the λ-calculus played for sequential, functional computation.

His motivations for seeking such a model are not far from the motivations of many engaged in today's industrial and commercial computing sectors. At present we find something of a "concurrency crisis" facing the industrial and commercial software development sectors. Software development is under pressure from above and from below. From above we have already witnessed the profound change the commercialization of the Internet has had on the nature of the commercial software application. With few exceptions, software applications are now commercially uninteresting unless they are multi-user and able to interoperate over a wide range of communications protocols. From below, the end of the Moore's law "free lunch" is causing hardware vendors to "go sideways" producing multicore offerings for both consumer and enterprise markets. Taken together these two trends mean that parallelism, concurrency and distribution are the prevailing paradigms from the hardware up through the end-user's expectations of availability and responsiveness.

Unfortunately, mainstream programming models [1] have not kept up with the times, offering out-dated and out-moded abstractions for describing and implementing parallel, concurrent and distributed programs. Threads and locks, for example, are notoriously difficult abstractions with which to program because there is no syntactic support allowing the programmer – or programmatic analysis – to check by *syntactic* means important notions like "how many threads might be expected to run through this code simultane-

[1] as embodied in mainstrema programming languages like `C++` and `Java`

ously?" or "what are the actual boundaries of this critical section; what resources are guaranteed protection from simultaneous access?". Thus the commercial developer is faced with developing systems of increasing complexity with little or no direct support from the language in which she expresses her application codes.

In response, the market has seen a huge uptick of models, frameworks and solutions developed and promulgated to address the complexity of developing commercial systems in this environment. A typical commercial web application, for example, involving

- asynchronous `XML` requests through JavaScript (a.k.a. `AJAX`) []
- a webserver engine (such as `Tomcat` or `Jetty`) []
- a relational database (such as `MySQL`) or an XML database (such as `BDBXML`) (or both) []
- access to a handful of Web Services []
- and deployed over a grid []

will mix several different core concurrency models including both synchronous and asynchronous message passing as well as transaction-based models. Even when using a framework like `RubyOnRails` [] which provides end-to-end "templates" and conventions for common case applications, the developer is still implicitly keeping these different models in her head as she develops, tests and profiles for performance. While some of this (infra-)structure actually represents optimization for widely adopted usage patterns (such as consistent local storage that persists across access) much of it is accidental.

Having an effective model of concurrent and distributed computation that could successfully encompass the different styles of concurrent execution found in today's commercial applications – much in the way that the λ-calculus was able to embrace different sequential programming paradigms leading to epithets like "lambda the ultimate declarative" or "lambda the ultimate imperative" or, eventually, just "lambda the ultimate" [] – would serve many purposes. First and foremost, it would provide some handle or leverage on application complexity, much in the way that λ-abstraction provided sorely needed abstraction over various program structure details. The appeal of the λ-calculus has ever been that its simplicity and abstraction enabled a view into the essential aspects of program structure. Certainly, Milner and others working in the area of concurrency theory have held hopes that one or more of the calculi developed there would provide a similar vantage point for concurrent applications. Such a vantage point, as we have seen with the theoretical development of the λ-calculus, is also the grounds on which we can hope to approach desirable goals, like correctness, or automated program transformation; and, these goals have very practical implications for robustness, availability and responsiveness. We cannot resist also pointing out that

automated program transformation has seen a resurgence of commercial interest as witnessed in support for so-called "refactoring" in industrial interactive development environments.

Even as we applaud the virtues of the λ-calculus, however, we need to recognize that there was a considerable effort over some 50 years to get from Church and co's original formulations to a family of theories that can be seen to underlie and inform useful, performant systems such as `OCaml`, or `Haskell` or `Scala`, or ... . Today, we have systems like Sewell's `Caml Light` [], effectively *mechanically formalizing* the theory that encompasses the lion's share of the operational semantics of the `OCaml` language, but the language design communities have not always seen formalisms like the λ-calculi as much more than approximate descriptions – toy models – of real programming languages and execution engines. There was a two-way dialogue between theory and practice, taking place over several decades, to develop a sufficiently detailed formal account to even contemplate something like `OCaml Light` – though many dreamed of it. Thus, even if Milner's π-calculus, or some family relative, does embody a λ-like, foundational model of concurrent computation, getting from there to a theory that can fruitfully underpin the efforts of the commercial web application developer is likely to be a similar effort as the passage from λ-calculus to high-performance functional programming languages and execution engines.

Following Milner's lead, then, this chapter explores desiderata for solutions to the following "equation":

$$\frac{\lambda\text{-}calculus}{\{\texttt{Haskell}, \texttt{OCaml}, \texttt{Scala}\}} = \frac{\pi\text{-}calculus}{X}$$

0.1.1 20-20 hindsight

We seek to take advantage of our hindsight perspective on the development from the λ-calculus to high-performance functional systems. We focus on the introduction of rich data structuring techniques, noting that the original formulations of both calculi lack such techniques. In the case of the lambda calculus, as we argue below, on the one hand, it was necessary to incorporate a rich account of data to obtain a level of utility where it could be seen as informing the design and semantics of something like `Haskell` or `OCaml`, while on the other, it was a non-trivial task to reconcile λ's wholly operational view of data with an account that could connect a wide range of developers with a wide range of application tasks.

The suggestion is that it is not egregiously revisionist to view the history of the development of functional theory and practice in terms of a sequence

of transformations of λ-calculus. Each transformation yields a calculus with a better, more fine-grained account of programming practice, simultaneously presenting proposals for refinements of programming practice. Further, the idea is to focus primarily on those transformations that have had to do with providing richer accounts of data. Data and control are tightly interwoven, in general, but more so in languages derived from an underlying algebraic design, like the λ and π-calculi. Thus, such a focus is not really limiting, in our view, but primarily a mechanism to scope and frame the investigation.

In terms of our "equational" characterization, we are suggesting that the calculi underlying languages like `Haskell` or `OCaml` look like $T_0(T_1(\ldots T_n(\lambda)\ldots))$. We are positing that with a focus on those transformations associated with providing better accounts of data we find candidates for transformations to apply to the π-calculus to get better and better accounts of a viable concurrent programming practice.

$$\frac{T_0(T_1(\ldots T_n(\lambda\text{-}calculus)\ldots))}{\{\texttt{Haskell}, \texttt{OCaml}, \texttt{Scala}\}} = \frac{T_0(T_1(\ldots T_n(\pi\text{-}calculus)\ldots))}{X}$$

At a qualitative level, we are putting forward two mutually supporting positions regarding desiderata in language design that bear on the introduction of data and control structuring techniques.

0.1.1.0.1 Data-control coherence Firstly, one of the key facets of a successful language design is the degree to which data structuring techniques and control structuring techniques *cohere*. On the one hand, the enthusiastic exploration of object-oriented language design proposals can be understood as driven by an underlying desire to find mediating structures between data and control, to find ways to make them cohere, regardless of the ultimate success or failure of the techniques embraced by the programme. On the other, the proposals painstakingly driven by the interaction of practice and theory – a much slower, more rigorous and demanding community process – resulting in the modern functional language designs, like `ML` or `Haskell`, are noted for their coherence – which can be measured, to some degree in their simplicity and opportunity for abstraction. Examples include

- the happy alignment of type constructs like summation and tupling with pattern-matching
- parametric polymorphism (a.k.a. generics), resulting in dual abstraction across data-structuring and algorithms exerting control flow through these patterns
- monads, by abstracting the notion of composition, provide generalized mechanisms for contructing and destructuring containers, hence

Wadler's observations of monads as providing generalized notions of comprehension [], which have been widely adopted in industry, even beyond mainstream functional languages.

0.1.1.0.2 Reflection Secondly, we argue that an alternative measure of this coherence is found in meta-programming, and more specifically reflective capabilities. The ease by which program structure, including control constructs, may be treated programmatically as data, and conversely, the ease by which such programmatically generated data may be deployed and executed, provides for a practical test of this coherence property. This measure is akin to the first practical test of a compiler: bootstrapping []. Similarly, the experiences of users of an earlier generation of functional languages, such as `Lisp` or `Scheme` on the ease of converting back and forth between `S`-expressions and executable code gave them a sense of that kind of coherence and seeded the development of numerous applications of meta-programming []. Ultimately, a variety of practical requirements of large-scale development have driven support for reflection-based meta programming into mainstream programming languages such as `Java` or Microsoft's `.net` languages. We stress that meta-programming techniques are essential in large-scale development because – even when developers are enabled by good abstractions – the programs are often too large to be written by hand; developers must employ programs to write programs [2], and in many cases it is more effective to do this dynamically [3], i.e. to generate programs during runtime.

Interestingly, reflection-based meta-programing is yet to appear in statically typed mainstream functional languages. There are noteworthy developments, such as `MetaOCaml` [], and `Template Haskell` [] can be seen as a recognition of some of the basic requirements of meta-programming [], but as yet the natural combination of rich, but strongly and statically typed data constructs and monadic structuring techniques, on the one hand with reflective capabilities, on the other has not come about in any of the mature functional platforms. This may be due, in part, from the historical accident that functional language development bifurcated along so-called "static" and "dynamic" lines, with the former line proceeding almost directly from `ML` and the latter proceeding more or less directly from `Lisp`. It has taken the various communities some time to understand the value of the feature sets explored in "isolation" and will take still more time to integrate these notions. It may also be due to a lack of theoretical treatment of reflection. Again, there are a handful of efforts [], but none aimed at industrial scale on the one hand, or foundations on the other.

Either way, we are arguing that the lessons learned in developing rich data structuring techniques, and reflective meta-programming techniques for func-

[2] a good example is the commonly accepted practice of generating the map from in-memory representations of data structures to their backing store – a.k.a. the object-relational mapping

[3] cf. the popular Spring framework

tional programming should be applied to the development of concurrent programming. Much of what has gone on in the passage from the λ-calculus to modern functional programming has been about development of feature proposals that can be seen in terms of a succession of natural, logical transformations from the core theory (λ-calculus) to a theoretical account informing and describing the practical features. The development of an industrial strength concurrent language could do worse than to follow this model. A good deal of evidence has been gathered since Milner's Turing award acceptance paper that the π-calculus is an excellent candidate as an analog of the λ-calculus in this development model.

0.1.2 Summary of contributions and overview

In some sense, then, one could argue that – given our long years of experience with adding data structures to the λ-calculus – it merely remains to turn the handle to add common data structures and reflection-based techniques to Milner's theory. In the case of data structures, proposals like Fournet's applied π [] or Brown, Laneve and Meredith's PiDuce [], can be seen in this light. Here we offer an alternative. We are proposing a novel integration of data structuring techniques and reflection-based meta-programming. It turns out that substitution-based models of computation, which include rewrite systems [], and especially nominal rewrite systems [] which include λ- and π-calculi, are amenable to the application of a functor which generates a reflective theory. This theory is then subject to a second functor that generates data (de-)structuring techniques from the reflective apparatus.

We note that applied π makes no commitment to a specific set of data types, rather provides mechanism to enrich the ambient theory with additional equational theories characterizing given data types. The `PiDuce` proposal on the other hand follows industry trends to make a commitment to a set of types that aligns with `XML`. In the context of a discussion of coherence of data and control both of these extremes raises the interesting question of whether there are *intrinsic* criteria for selecting a core set of data types. This is where the reflective proposal differentiates itself for in the reflective account the (data) structures necessary to express processes become the basis of the core set of data types. This automatically ensures a certain kind of coherence between data and control calling to mind a refinement of the way `S`-expressions reflect the means of expressing `Lisp`-programs.

It should be stressed that we are not proposing that the calculus immediately resulting from the application of the two functors described in the previous paragraphs immediately scales to an industrial strength concurrent language. In our view, this would only be verified by concerted application to industrial scale problems and the proposal is still too new for results of that kind to be reported. Rather, by describing a mechanistic means of deriving the computational model from Milner's theory, we are providing a means to systematically explore the design space which – when coupled with our

experience from functional programming – seems a pragmatic approach to the problem. Moreover, there are some promising features of the proposal. One of these is a functorial mechanism for deriving both a logic and a type system for the proposed calculus. These come directly from Caires' work on behavioral and spatial logics and types []. Apart from providing much needed support for mechanical assistance in checking programm correctness and adherence to spec, we take this as supporting evidence that the proposed means of exploring the design space is not without merit.

We have used the word functor above because there is indeed a category theoretic presentation of the results. However, we cannot simultaneously present the derived calculus with sketch of its applications and the category theoretic account without doubling the length of this chapter. We settle on a more traditional syntactic and operational account of the calculus (and how it is derived from Milner's theory) and leave the category theoretic presentation for a subsequent document.

0.1.2.1 Overview of chapter

Apart from a brief digression into the history of the development of semantics of functional languages the chapter follows the following outline. First we review the core presentation of Milner's π-calculus with careful attention to certain design choices that impact the development of our derived calculus. Next we review the derivation of a reflective version of that calculus. Then we derive the data (de-)structuring techniques that turn the reflective machinery into a data. Then we present some examples of coding in this calculus. Penultimately, we illustrate the derivation of a logic and it's application to one of the examples. Finally, we conclude with some discussion of future directions for this work.

Now, in the interest of developing a little narrative tension, before we dive into the presentation, we take a brief and highly idiosyncratic digression through some of the history of the semantics of functional languages.

0.1.3 A function of history

As mentioned above, there is a notable commonality between the two starting points, λ and π-calculi: the total absence of rich data types. Both calculi, in their original presentations, lack any extrinsic notion of data. Put the other way around, both calculi take a strongly operational view of data: *data is as data does*, to coin a phrase. In neither calculus do we find "builtin" notions of common data types such as the natural numbers or linked lists. Rather, we find encodings of these types as behaviors. For the λ-calculus, the paradigmatic example, of course, is the Church numerals [], and more generally, Church encodings [], which represent data types as certain kinds of lambda terms. In a similar manner, in Milner's Polyadic π-calculus tutorial [], one finds encodings of data structures from the natural numbers to linked

lists as processes.

To sharpen the point, neither the λ-calculus nor the π-calculus actually enjoy the "program-as-data-data-as-program" property of even early functional programming languages like `Scheme`. There are no `S`-expressions in the λ-calculus. There are λ-terms that can be formally recognized as behaving the way `S`-expressions might be expected to behave. Much in the same way that in biology all "software" has to be realized as hardware, all data types in these calculi have to be realized as classes of behaviors. Standing *outside* of these calculi we can recognize certain behaviors as corresponding to what we know as certain types of data, but internally, these classes of behaviors are not separated from the others.

Given the recent work on genericity, such as Oliveira's reconciliation of two generic programming proposals [] – which relies heavily on a deep comparison of Church and Parigot-style encodings – as well as Barry Jay's pattern-matching calculi [], it is probably inadvisable to take a strong stance that a boldly (or even exclusively) operational view of data is unrealistic or impractical. Rather, the dividing line between program and data has always been source of fruitful debate, and as we want to argue in the sequel provides an excellent instrument in the current investigation.

In fact, we want to observe that the passage from Church's λ-calculus to something as simple as PCF (λ-calculus with builtin naturals and booleans) [] constituted a signficant challenge to the theory at the time. A fully abstract semantics for PCF was something like 25 years from the statement of the problem to a solution []. We stress, relatively satisfactory *encodings* of data structures as behaviors followed fairly closely on the heels of the development the λ-calculus (respectively, the π-calculus). In the case of the λ-calculus the obvious *embeddings* of the booleans and natural numbers took 25 years to give satisfactory semantic accounts of. Moreover, the solution involved no less than two *revolutionary* ideas: linear logic and games semantics [].

We submit that this was not an accident, but rather strong evidence that the programme of embedding a notion of data in a decidedly operational theory is deceptively *radical*, requiring dual – but fully reconcilable – views of

- data as an "entity" upon which to operate
- data as behavior.

The synergetic connection of linear logic and games semantics provided, we submit, just the right mix of logical and operational perspectives into a finer-grained, more intensional account of program structure that enabled the construction of reconcilable versions of these two views of data [].

Additionally, we note that to be of utility to a broad class of developers the operations of the "entity" view have to have a clear and natural mapping to some traditional view of the data type while also observing the operational semantics of the ambient calculus in which they have been embedded. In

the case of λ-calculus and the natural numbers, to use the standard example, a Peano-like presentation provides the mediating view. More generally, a generators-and-relations-style presentation of data structures – which just happens to align with an algebraic data type `ADT`-like presentation – provides an acceptable mediating view. Fortuitously, the majority of commercial developers have much more exposure to treatments of data types that more closely resemble something like `ADT`s. Even the transition between object-oriented views of rich data types and `ADT`s is not difficult for a competent commercial developer to make.

0.1.4 Data processing

In the π-calculus the situation is slightly different. The calculus does come equipped with a single, impoverished data type: channel (a.k.a. port or name). In fact, as we will see in the following sections, the π-calculus is actually parametric in the notion of channel: when provided with a notion of channel it provides a notion of processes that synchronize on those channels and pass messages consisting of (tuples) of those channels. Further, the π-calculus is crucially dependent on an effective equality of channels being supplied. Neither substitution nor synchronization nor alpha-equivalence can be effected or decided without it.

Chapter 1

Re-introducing the polyadic π-calculus

1.1 Concurrent process calculi

In the last thirty years the process calculi have matured into a remarkably powerful analytic tool for reasoning about concurrent and distributed systems. Process-calculus-based algebraic specification of processes began with Milner's Calculus for Communicating Systems (CCS) [26] and Hoare's Communicating Sequential Processes (CSP) [15] [6] [17] [16], and continues through the development of the so-called mobile process calculi, e.g. Milner, Parrow and Walker's π-calculus [31], Cardelli and Caires's spatial logic [9] [8] [7], or Meredith and Radestock's reflective calculi [23] [24]. The process-calculus-based algebraical specification of processes has expanded its scope of applicability to include the specification, analysis, simulation and execution of processes in domains such as:

- telecommunications, networking, security and application level protocols [1] [2] [19] [21];
- programming language semantics and design [19] [14] [13] [43];
- webservices [19] [21] [22];
- and biological systems [10] [11] [37] [36].

Among the many reasons for the continued success of this approach are two central points. First, the process algebras provide a compositional approach to the specification, analysis and execution of concurrent and distributed systems. Owing to Milner's original insights into computation as interaction [29], the process calculi are so organized that the behavior —the semantics— of a system may be composed from the behavior of its components [12]. This means that specifications can be constructed in terms of components —without a global view of the system— and assembled into increasingly complete descriptions.

The second central point is that process algebras have a potent proof principle, yielding a wide range of effective and novel proof techniques [32] [38] [39]

[40]. In particular, *bisimulation* encapsulates an effective notion of process equivalence that has been used in applications as far-ranging as algorithmic games semantics [4] and the construction of model-checkers [7]. The essential notion can be stated in an intuitively recursive formulation: a *bisimulation* between two processes P and Q is an equivalence relation E relating P and Q such that: whatever action of P can be observed, taking it to a new state P', can be observed of Q, taking it to a new state Q', such that P' is related to Q' by E and vice versa. P and Q are *bisimilar* if there is some bisimulation relating them. Part of what makes this notion so robust and widely applicable is that it is parameterized in the actions observable of processes P and Q, thus providing a framework for a broad range of equivalences and up-to techniques [25] all governed by the same core principle [38] [39] [40].

1.1.1 Re-presenting the π-calculus

The modern presentation of a mobile process calculus is heavily influenced by the developments of Milner in [28]. This, in turn follows a modern style of giving algebraic structures in terms of generators and relations on them. The grammar, below, describing term constructors, freely generates the set of process (expressions), $\mathcal{P}(\mathcal{N})$ over a set of names $\mathcal{N}$. This set is then quotiented by a relation known as structural congruence.

What marks these algebras as distinct from other more traditionally studied algebraic structures, e.g. vector spaces or polynomial rings, is the manner in which dynamics is captured. In traditional structures, dynamics is typically expressed through morphisms between such structures, as in linear maps between vector spaces or morphisms between rings. In algebras associated with the semantics of computation, the dynamics is expressed as part of the algebraic structure itself, through a reduction reduction relation typically denoted by $\rightarrow$. Below, we give a recursive presentation of this relation for Milner's calculus.

We note that this particular design choice regarding the representation of dynamics has had huge impact on the development of both families of theories. Ironically, the standard category theoretic machinery – oft employed in programming language semantics – is more readily applied to the traditional division of labor where dynamics is in the morphisms. This choice affords the standard combinators, such as Cartesian or tensor product, to be applied to algebras and extended meaningfully to the dynamics.

Despite advances in Milner's bigraphical theories [18], a proper categorical framework for composing (resp. decomposing) entire process calculi, elucidating the natural combinators is still missing. This is due to the fact that reifying dynamics as a part of algebraic structure itself places a much greater demand on the notion of morphism. To wit, the notion of morphism describes what should be preserved. This leaves any categorical theory in the position of having to address what in the dynamics needs to be preserved in maps from one process calculus to another – essentially before it can get off the ground.

Thus, the extremely elegant structure-function relationships embodied in the process calculi also turn out to be a bit of an Achilles' heel in addressing higher-level notions of composition.

1.1.1.1 Process grammar

The presentation here follows [30]. This presentation is a departure from the one found in [31] or even in [28]. Foreshadowing a little of what is to come, the essential difference lies in the typing of the sequencing term constructor. Is the type of the operator, $action \times process \rightarrow process$, or does it demand some other decomposition? The question can be traced back to Milner's CCS [26]. In CCS an action is essentially *atomic* (even value-passing was interpreted as a summation over atomic actions) in the sense of having no internal structure (as opposed to having any transactional connotation). So, the essential design question is, as actions are developed to have internal structure, how does this relate to process structure? As we will see below, the introdction of binding operators makes a clear commitment to a natural set of relationships amongst these operators. Finally, lest this seem much ado about minutiae, if not nothing, it is important to observe that similar issues about the decomposition of operators – right at the crux where syntax meets semantics – have been pivotal in extremely important developments in the history of logic and computation. All the substructural logics, and the author counts intuitionistic logic among them, enjoy this heritage; and, of course linear logic with its explosive revitalization of logic in computation is regularly sloganized in terms of the decomposition of implication into replication and linear implication.

In this context we present the generators.

SUMMATION
$M, N ::= 0 \mid x.A \mid M + N$

AGENT
$A ::= (\vec{x})P \mid [\vec{x}]P$

PROCESS
$P, Q ::= N \mid P|Q \mid (\nu\ \vec{x})P \mid X\langle\vec{y}\rangle \mid (\mathsf{rec}\ X(\vec{x}).P)\langle\vec{y}\rangle$

FIGURE 1.1: π-calculus grammar

In English, this says that processes are either

summation n-ary sums of sequenced agents, where a 0-ary sum, 0 is the *nil* or stopped process, and a 1-ary sum is just a sequence

agents where agents are either

- abstractions, i.e. maps from (tuples of) names to processes, or
- concretions, i.e. pairs of (tuples of) names and process

$$\mathcal{FN}(0) := \emptyset$$

$$\mathcal{FN}(x?(\vec{y}).P) := \{x\} \cup (\mathcal{FN}(P) \setminus \{\vec{y}\})$$

$$\mathcal{FN}(x!(\vec{y}).P) := \{x\} \cup \{\vec{y}\} \cup \mathcal{FN}(P)$$

$$\mathcal{FN}(P|Q) := \mathcal{FN}(P) \cup \mathcal{FN}(Q) \qquad \mathcal{FN}(P+Q) := \mathcal{FN}(P) \cup \mathcal{FN}(Q)$$

$$\mathcal{FN}((\nu\ \vec{y})P) := \mathcal{FN}(P) \setminus \{\vec{y}\}$$

$$\mathcal{FN}((\mathsf{rec}\ X(\vec{x}).P)\langle\vec{y}\rangle) := \{\vec{y}\} \cup \mathcal{FN}(P) \setminus \{\vec{x}\}$$

FIGURE 1.2: π-calculus free name calculation

composite processes and composite processes, built by

- putting processes in parallel composition, or
- placing them under the scope of a restriction, or
- made from recursive definitions

Note that $\vec{x}$ denotes a vector of names of length $|\vec{x}|$. We adopt the following standard abbreviations.

$$x?(\vec{y}).P := x.(\vec{y})P \qquad x!(\vec{y}).P := x.[\vec{y}]P$$

$$X(\vec{y}) \leftarrow P := (\vec{y})(\mathsf{rec}\ X(\vec{x}).P)\langle\vec{y}\rangle \qquad \Pi_{i=0}^{n-1} P_i := P_0|\ldots|P_{n-1}$$

These abbreviations recover the appearance of the usual decomposition of sequence as typed $action \times process \to process$.

1.1.1.2 Structural congruence

The linear syntax makes too many distinctions. For example, if the intent of the expression $P|Q$ is to model parallel composition, it seems necessary to find a way to identify it with $Q|P$. The equivalence defined below can be seen as a set of relations that cut down the freely generated structure given above, and in the process erase these distinctions.

1.1.1.2.1 Free and bound names and alpha-equivalence. At the core of structural equivalence is alpha-equivalence which identifies process that are the same up to a change of variable. Formally, we recognize the distinction between free and bound (occurrences of) names. The free names of a process, $\mathcal{FN}(P)$, may be calculated recursively as follows:

The bound names of a process, $\mathcal{BN}(P)$, are those names occurring in P that are not free. For example, in $x?(y).0$, the name x is free, while y is bound.

DEFINITION 1.1 *Then two processes, P, Q, are alpha-equivalent if $P = Q\{\vec{y}/\vec{x}\}$ for some $\vec{x} \in \mathcal{BN}(Q), \vec{y} \in \mathcal{BN}(P)$, where $Q\{\vec{y}/\vec{x}\}$ denotes the capture-avoiding substitution of $\vec{y}$ for $\vec{x}$ in Q.*

DEFINITION 1.2 *The* structural congruence *[38] , $\equiv$, between processes is the least congruence containing alpha-equivalence, satisfying the abelian monoid laws (associativity, commutativity and 0 as identity) for parallel composition $|$ and for summation $+$, in addition to the following axioms:*

$$(\nu\ x)0 \equiv 0 \qquad (\nu\ x)(\nu\ x)P \equiv (\nu\ x)P \qquad (\nu\ x)(\nu\ y)P \equiv (\nu\ y)(\nu\ x)P$$

$$P|(\nu\ x)Q \equiv (\nu\ x)(P|Q),\ \textit{if}\ x \notin \mathcal{FN}(P)$$

$$(\mathsf{rec}\ X(\vec{x}).P)\langle\vec{y}\rangle \equiv P\{\vec{y}/\vec{x}\}\{(\mathsf{rec}\ X(\vec{x}).P)/X\}$$

FIGURE 1.3: π-calculus structural equivalence

1.1.1.3 Operational semantics

Finally, we introduce the computational dynamics. This, too, is in modern accounts a highly structured specification. It may be viewed alternately as composed of a core rule, the comm rule, for the computing the reduction relation, specifying the dynamics of interaction, together with a set of rules saying how the reduction relation may be applied *in context*. Or, it may be viewed as a recursive specification of the reduction relation with the comm rule as the base case of the recursion. Naturally, these views are not mutually exclusive.

$$\textsc{Comm}\ \frac{\vec{y} \cap \vec{v} = \emptyset \qquad |\vec{y}| = |\vec{z}|}{x.(\vec{y})P|x.(\nu\vec{v})[\vec{z}]P \to (\nu\vec{v})(P\{\vec{z}/\vec{y}\}|Q)}$$

$$\textsc{Par}\ \frac{P \to P'}{P|Q \to P'|Q} \qquad \textsc{Equiv}\ \frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q} \qquad \textsc{New}\ \frac{P \to P'}{(\nu\ x)P \to (\nu\ x)P'}$$

FIGURE 1.4: π-calculus reduction relation

We write $\Rightarrow$ for $\to^*$, and $P \to$ if $\exists Q$ such that $P \to Q$.

Our contributions lie in a careful analysis of the structure of the comm rule. Before we present these, however, we complete the presentation of the

standard theory, especially the notions of bisimulation and context.

1.1.1.4 Bisimulation

The computational dynamics described above gives rise to another kind of equivalence, the equivalence of computational behavior. As previously mentioned this is typically captured *via* some form of bisimulation. The notion we use in this paper is derived from weak barbed bisimulation [30]. We must introduce an "up to" [42] [34] strategy to deal with the fact that Reidemeister moves can not only introduce or eliminate crossings (see R_1 R_2), but "reorder" them (see R_3).

DEFINITION 1.3 *An agent B occurs* unguarded *in A if it has an occurence in A not guarded by a prefix x. A process P is observable at x, written here $P \downarrow x$, if some agent $x.A$ occurs unguarded in P. We write $P \Downarrow x$ if there is Q such that $P \Rightarrow Q$ and $Q \downarrow x$.*

DEFINITION 1.4 *A* barbed bisimulation *is a symmetric binary relation $\mathcal{S}$ between agents such that $P\ \mathcal{S}\ Q$ implies:*

1. *If $P \rightarrow P'$ then $Q \Rightarrow Q'$ and $P'\ \mathcal{S}\ Q'$, for some Q'.*

2. *If $P \downarrow x$, then $Q \Downarrow x$.*

P is barbed bisimilar to Q, written $P \simeq Q$, if $P\ \mathcal{S}\ Q$ for a barbed bisimulation $\mathcal{S}$.

One of the principal advantages of this framework is the co-algebraic proof method for proving bisimilarity between two processes: exhibit a bisimulation [41].

1.1.1.5 Contexts

One of the principle advantages of computational calculi like the π-calculus is a well-defined notion of context, contextual-equivalence and a correlation between contextual-equivalence and notions of bisimulation. The notion of context allows the decomposition of a process into (sub-)process and its syntactic environment, its context. Thus, a context may be thought of as a process with a "hole" (written $\Box$) in it. The application of a context M to a process P, written $M[P]$, is tantamount to filling the hole in M with P.

DEFINITION 1.5 contextual application *Given a context M, and process P, we define the* contextual application, *$M[P] := M\{P/\Box\}$. That is, the contextual application of M to P is the substitution of P for $\Box$ in M.*

SUMMATION
$M_M, M_N ::= \Box \mid x.M_A \mid M_M + M_N$

AGENT
$M_A ::= (\vec{x})M_P \mid [\vec{x}]M_P$

PROCESS
$M_P ::= M_N \mid P|M_P \mid (\mathsf{rec}\ X(\vec{x}).M_P)\langle\vec{y}\rangle \mid (\nu\ \vec{x})M_P$

FIGURE 1.5: π-calculus contexts

Again, in terms of the theme of this paper, contexts provide a natural way to reify and calculate concerning questions of the divide between system and environment. This apparatus falls out of the compositional specification of the system. It is a key distinguishing feature that the notion of contextual equivalence, i.e. two systems are equivalent if they may be interchangeably substituted into any given context without a change of behavior of the resulting composite system, has a well-established relation with bisimulation (namely, the two notions conicide).

1.2 Evolution and communication

As noted above, the original comm rule for Milner's π-calculus was of the form

COMM
$$x?(y).P|x!(z).Q \rightarrow P\{z/y\}|Q$$

Later, when Milner analyzed the scoping of input binding he realized that the sequencing operator was given a more hygenic account as synchronization followed by data-exchange. Compare with the productions of the grammar: sequence is effectively a function taking names and agents, while agents is the disjoint sum of two types: abstractions and concretions. In the case of the π-calculus we have abstractions of the form $A \equiv ?(y)P$ (i.e. functions from names to processes) and concretions of the form $C \equiv (\nu u)!(z)Q$ (i.e. pairs of names and processes).

This refactoring leads to a considerably more parsimonious comm rule.

$comm_{AC}$
$$x.A|x.C \rightarrow A \cdot C$$

where $A \cdot C$, called pseudo-application, is then defined in this context as

$$?(y)P \cdot (\nu u)!(z)Q \triangleq (\nu u)P\{z/y\}|Q \tag{1.1}$$

1.2.0.6 Synchronization algebras

We need to make explicit another piece of information implicit in the standard presentation of the π-calculus. Synchronization fundamentally depends on name equality. In the comm rule, the sequenced abstraction may be thought of, roughly, as located at the '?'-end of a channel, x, and the sequenced concretion at the '!'-end. It is name equality, i.e. that $x = x$, that determines that these agents are located at the opposite ends of the *same* channel and thus may communicate. We generalize this notion of co-location by adding a predicate, $\perp$: $name \times name \to bool$, explicitly determining when two names enable synchronization and data-exchange. Our comm rule then becomes

$$\frac{\begin{array}{c} comm_{AC^{\perp}} \\ x_{src} \perp x_{trgt} \end{array}}{x_{src}.A | x_{trgt}.C \to A \cdot C}$$

In this context we note that Milner saw this as an avenue of investigation. In SCCS, for example, [27] he considered actions with the additional structure of a group and a communication rule that depended on action annihilation (i.e. that the channel and co-channel were group inverses of each other). Additionally, other investigators have exploited this possibility. Meredith and Radestock, for example, use it to derive a reflective, two-level comm rule in which co-location is calculated using the process dynamics itself [24].

Intuitively, suppose we had two processes that were written to communicate over protocols that were at different levels in a network stack, e.g. in the network setting, one might be written to accept and emit `TCP/IP` packets while the other was written to accept and emit `HTTP` over `TCP/IP`. It is common practice to think of the process that communicates at the higher layer as essentially 'wrapped' by a set of communications behaviors that have been factored from the core behavior of the process. If P is the process written at the higher layer, we can faithfully represent that division of labor by $K[P]$, where K is a context that constitutes the communications behaviors of packaging up `HTTP` requests and responses over `TCP/IP`.

In terms of our intuitive interpretation, such K represents a protocol mediator, mediating between the layer at which P engages and the layer at which Q engages. The question then becomes how to select these mediators? This is precisely where the expansion of the co-location predicate from boolean-valued to taking values in some other domain comes into play. We use the pair of names (in terms of our networking analogy, the pair of communications port abstractions at the different layers in the networking stack) to determine the mediating context. So, the co-location "predicate" becomes a context-valued function. Thus, the general expression of our comm rule becomes

$$\frac{x_{src} \perp x_{trgt} = K}{x_{src}.A | x_{trgt}.C \rightarrow A \cdot_K C} \; comm_{AC^{\perp}K}$$

Obviously, the relation $x \perp_{\Box} x \triangleq \Box$ with $x \perp_{\Box} y$ undefined if $x \neq y$, recovers the original comm rule. In situations other than the trivial case, this rule must be is instantiated by an appropriate notion of pseudo-application.

Chapter 2

Reflection

2.1 A reflective calculus

The next step in this process is to derive the reflective calculus from the π-calculus. We observe that the calculus presented in the previous section is not a closed theory, but rather a theory dependent upon some theory of names. Operationally, one may think of the π-calculus as a procedure that when handed a theory of names provides a theory of processes that communicate over those names. In this section we present a closed variation of the the π-calculus in which the *codes* of processes supply the set of names. Morally, we are solving a domain equation []. If the set of process generated from the set of names, $\mathcal{N}$, is denoted $\mathcal{P}(\mathcal{N})$, then we are seeking a solution to the equation $\mathcal{N} = \mathcal{P}(\mathcal{N})$. Our solution is entirely syntactic ('free' in the language of category theory): we merely introduce a term constructor for names that encloses process terms in quotes.

A blind application of this idea almost works. For technical convenience, below we present the application to the minimal asynchronous variant of the π-calculus without summation []. But, in the richer calculus aimed at underpinning a general purpose programming language summation is treated. More interestingly, the reflective features make the calculus essentially higher order, allowing replication to be eliminated as an operator, in the same way that the λ-calculus can implement fix point combinators and has no need to introduce them as native constructs. Also, in a higher order setting, it turns out that there are two candidates for the continuation of output: one is the process that remains on the emitting side, the other is the process that is communicated. For simplicity, we opt only to work with the latter. Even more interestingly, it turns out that the restriction operator can be implemented as well. This presentation essentially follows that of [24] of the *r*eflective *h*igher-*o*rder calculus, or rho-calculus for short, or ρ-calculus for even shorter.

2.1.0.7 Notation

As before, we organize the generators along the lines of normal processes, agents and processes.

NORMAL PROCESS	AGENT	PROCESS	NAME
$N ::= 0 \mid x.A \mid \ulcorner x \urcorner$	$A ::= (\vec{y})P \mid \langle\!\langle \vec{Q} \rangle\!\rangle$	$P, Q ::= N \mid P \vert Q$	$x, y ::= \ulcorner P \urcorner$

FIGURE 2.1: ρ-calculus process grammar

2.1.0.7.1 Notation It is useful to adopt analogs of the standard abbreviations, such as

$$x?(\vec{y}).P := x.(\vec{y})P \qquad\qquad x!\langle\!\langle \vec{Q} \rangle\!\rangle := x.\langle\!\langle \vec{Q} \rangle\!\rangle$$

2.1.0.8 Quote

Working in a bottom-up fashion, we begin with names. The technical detail corresponding to the π-calculus' parametricity in a theory of *names* shows up in standard presentations in the grammar describing terms of the language: there is no production for names; names are taken to be terminals in the grammar. Our first point of departure from a more standard presentation of an asynchronous mobile process calculus is here. The grammar for the terms of the language will include a production for names in the grammar. A name is a *quoted* process, $\ulcorner P \urcorner$.

2.1.0.9 Parallel

This constructor is the usual parallel composition, denoting concurrent execution of the composed processes.

2.1.0.10 Lift and drop

As usual we maintain the syntactic category *agent* of abstractions ($(\vec{y})P$) and concretions ($\langle\!\langle \vec{Q} \rangle\!\rangle$). The middle choice of the first production allows agents to be *located* at a channel, e.g. $x.(\vec{y})P$ and $x.\langle\!\langle Q \rangle\!\rangle$. The first form plays the usual role of guarded input, but the second form needs a little discussion. Despite the fact that names are built from (the codes of) processes, we still maintain a careful disinction in kind between process and name; thus, name construction is not process construction. Thus, if we want to be able to generate a name from a given process, there must be a process constructor for a term that creates a name from a process. This is the motivation for the production $x.\langle\!\langle \vec{Q} \rangle\!\rangle$, dubbed here the *lift* operator. The intuitive meaning of this term is that the process P will be packaged up as its code, $\ulcorner P \urcorner$, and ultimately made available as an output at the port x.

A more formal motivation for the introduction of this operator will become clear in the sequel. But, it will suffice to say now that $\ulcorner P \urcorner$ is impervious to substitution. In the ρ-calculus, substitution does not affect the process body between quote marks. On the other hand, $x.\langle\!\langle \vec{P} \rangle\!\rangle$ is susceptible to substitution and as such constitutes a dynamic form of quoting because the process body

ultimately quoted will be different depending on the context in which the $x.\langle\!|\vec{P}|\!\rangle$ expression occurs.

Of course, when a name is a quoted process, it is very handy to have a way of evaluating such an entity. Thus, the $\urcorner x \ulcorner$ operator, pronounced *drop* x, (eventually) extracts the process from a name. We say 'eventually' because this extraction only happens when a quoted process is substituted into this expression. A consequence of this behavior is that $\urcorner x \ulcorner$ is inert except under and input prefix. One way of saying this is that if you want to get something done, sometimes you need to drop a name, but it should be the name of an agent you know.

REMARK 2.1 The lift operator turns out to play a role analogous to $(\nu\ x)P$. As mentioned in the introduction, it is essential to the computational completeness of the calculus, playing a key role in the implementation of replication. It also provides an essential ingredient in the compositional encoding of the ν operator. □

REMARK 2.2 It is well-known that replication is not required in a higher-order process algebra [38]. While our algebra is *not* higher-order in the traditional sense (there are not formal process variables of a different type from names) it has all the features of a higher-order process algebra. Thus, it turns out that there is no need for a term for recursion. To illustrate this we present below an encoding of $!P$ in this calculus. Intuitively, this will amount to receiving a quoted form of a process, evaluating it, while making the quoted form available again. The reader familiar with the λ-calculus will note the formal similarity between the crucial term in the encoding and the paradoxical combinator [5]. □

2.1.0.11 Input and output

The input constructor is standard for an asynchronous name-passing calculus. Input blocks its continuation from execution until it receives a communication. Lift is a form of output which – because the calculus is asynchronous – is allowed no continuation. It also affords a convenient syntactic sugar, which we define here.

$$x![y] \triangleq x!\langle\!|\urcorner y \ulcorner|\!\rangle$$

2.1.0.12 The null process

As we will see below, the null process has a more distinguished role in this calculus. It provides the sole atom out of which all other processes (and the names they use) arise much in the same way that the number 0 is the sole number out of which the natural numbers are constructed; or the empty set is

the sole set out of which all sets are built in ZF-set theory [20]; or the empty game is the sole game out of which all games are built in Conway's theory of games and numbers [**?**]. This analogy to these other theories draws attention, in our opinion, to the foundational issues raised in the introduction regarding the design of calculi of interaction.

2.1.1 The name game

Before presenting some of the more standard features of a mobile process calculus, the calculation of free names, structural equivalence, etc., we wish to consider some examples of processes and names. In particular, if processes are built out of names, and names are built out of processes, is it ever possible to get off the ground? Fortunately, there is one process the construction of which involves no names, the null process, 0. Since we have at least one process, we can construct at least one name, namely $\ulcorner 0 \urcorner$ [1]. Armed with one name we can now construct at least two new processes that are evidently syntactically different from the 0, these are $\ulcorner 0 \urcorner[\ulcorner 0 \urcorner]$ and $\ulcorner 0 \urcorner(\ulcorner 0 \urcorner) \,.\, 0$. As we might expect, the intuitive operational interpretation of these processes is also distinct from the null process. Intuitively, we expect that the first outputs the name $\ulcorner 0 \urcorner$ on the channel $\ulcorner 0 \urcorner$, much like the ordinary π-calculus process $x![x]$ outputs the name x on the channel x, and the second inputs on the channel $\ulcorner 0 \urcorner$, much like the ordinary π-calculus process $x?(x) \,.\, 0$ inputs on the channel x.

Of course, now that we have two more processes, we have two more names, $\ulcorner\ulcorner 0 \urcorner[\ulcorner 0 \urcorner]\urcorner$ and $\ulcorner\ulcorner 0 \urcorner(\ulcorner 0 \urcorner) \,.\, 0 \urcorner$. Having three names at our disposal we can construct a whole new supply of processes that generate a fresh supply of names, and we're off and running. It should be pointed out, though, that as soon as we had the null process we also had $0|0$ and $0|0|0$ and consequently, we had the names $\ulcorner 0|0 \urcorner$, and $\ulcorner 0|0|0 \urcorner$, and .... But, since we ultimately wish to treat these compositions as merely other ways of writing the null process and not distinct from it, should we admit the codes of these processes as distinct from $\ulcorner 0 \urcorner$?

2.1.2 Free and bound names

The syntax has been chosen so that a binding occurrence of a name is sandwiched between round braces, $(\cdot)$. Thus, the calculation of the free names of a process, P, denoted $\mathcal{FN}(P)$ is given recursively by

An occurrence of x in a process P is *bound* if it is not free. The set of names occurring in a process (bound or free) is denoted by $\mathcal{N}(P)$.

[1] pun gratefully accepted ;-)

$$\mathcal{FN}(0) := \emptyset \qquad \mathcal{FN}(x(y).P) := \{x\} \cup \mathcal{FN}(P) \setminus \{y\}$$

$$\mathcal{FN}(x(\!|P|\!)) := \{x\} \cup \mathcal{FN}(P) \qquad \mathcal{FN}(\ulcorner x \urcorner) := \{x\}$$

$$\mathcal{FN}(P|Q) := \mathcal{FN}(P) \cup \mathcal{FN}(Q)$$

FIGURE 2.2: ρ-calculus free name calculation

2.1.3 Structural congruence

The *structural congruence* of processes, noted $\equiv$, is the least congruence, containing α-equivalence, $\equiv_\alpha$, that satisfies the following laws:

$$P|\,0 \equiv P \;\equiv 0|P \qquad P|Q \equiv Q|P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

FIGURE 2.3: ρ-calculus structural equivalence

2.1.4 Name equivalence

We take name equivalence, written $\equiv_N$, to be the smallest equivalence relation generated by the following rules.

$$\frac{}{\ulcorner \urcorner x \ulcorner \urcorner \equiv_N x} \qquad \frac{P \equiv Q}{\ulcorner P \urcorner \equiv_N \ulcorner Q \urcorner}$$

FIGURE 2.4: ρ-calculus name equivalence

2.1.5 Syntactic substitution

Now we build the substitution used by α-equivalence. We use $Proc$ for the set of processes, $\ulcorner Proc \urcorner$ for the set of names, and $\{\vec{y}/\vec{x}\}$ to denote partial maps, $s : \ulcorner Proc \urcorner \to \ulcorner Proc \urcorner$. A map, s lifts, uniquely, to a map on process terms, $\widehat{s} : Proc \to Proc$ by the following equations.

where

$$(x)\{\ulcorner Q \urcorner / \ulcorner P \urcorner\} = \begin{cases} \ulcorner Q \urcorner & x \equiv_N \ulcorner P \urcorner \\ x & otherwise \end{cases}$$

and z is chosen distinct from $\ulcorner P \urcorner$, $\ulcorner Q \urcorner$, the free names in Q, and all the names in R. Our α-equivalence will be built in the standard way from this

substitution.

2.1.6 Dynamic quote: an example

Anticipating something of what's to come, consider applying the substitution, $\widehat{\{u/z\}}$, to the following pair of processes, $w!(\!|y![z]|\!)$ and $w![\ulcorner y![z]\urcorner]$.

$$w!(\!|y![z]|\!)\widehat{\{u/z\}} = w!(\!|y![u]|\!)$$
$$w![\ulcorner y![z]\urcorner]\widehat{\{u/z\}} = w![\ulcorner y![z]\urcorner]$$

Because the body of the process between quotes is impervious to substitution, we get radically different answers. In fact, by examining the first process in an input context, e.g. $x?(z).w!(\!|y![z]|\!)$, we see that the process under the lift operator may be shaped by prefixed inputs binding a name inside it. In this sense, the lift operator will be seen as a way to dynamically construct processes before reifying them as names.

2.1.7 Semantic substitution

The substitution used in α-equivalence is really only a device to formally recognize that binding occurrences do not depend on the specific names. It is not the engine of computation. The proposal here is that while synchronization is the driver of that engine, the real engine of computation is a semantic notion of substitution that recognizes that a dropped name is a request to run a process. Which process? Why the one whose code has been bound to the name being dropped. Formally, this amounts to a notion of substitution that differs from syntactic substitution in its application to a dropped name.

$$(0)\{\ulcorner\widehat{Q\urcorner/\ulcorner P\urcorner}\} := 0$$

$$(R|S)\{\ulcorner\widehat{Q\urcorner/\ulcorner P\urcorner}\} := (R)\{\ulcorner\widehat{Q\urcorner/\ulcorner P\urcorner}\}|(S)\{\ulcorner\widehat{Q\urcorner/\ulcorner P\urcorner}\}$$

$$(x(y)\,.\,R)\{\ulcorner\widehat{Q\urcorner/\ulcorner P\urcorner}\} := (x)\{\ulcorner Q\urcorner/\ulcorner P\urcorner\}(z)\,.\,((R\widehat{\{z/y\}})\{\ulcorner\widehat{Q\urcorner/\ulcorner P\urcorner}\})$$

$$(x(\!|R|\!))\{\ulcorner\widehat{Q\urcorner/\ulcorner P\urcorner}\} := (x)\{\ulcorner Q\urcorner/\ulcorner P\urcorner\}(\!|R\{\ulcorner\widehat{Q\urcorner/\ulcorner P\urcorner}\}|\!)$$

$$\frac{x \equiv_N \ulcorner P\urcorner}{(\urcorner x\ulcorner)\{\ulcorner\widehat{Q\urcorner/\ulcorner P\urcorner}\} :=\urcorner\ulcorner Q\urcorner\ulcorner} \qquad \frac{x \not\equiv_N \ulcorner P\urcorner}{(\urcorner x\ulcorner)\{\ulcorner\widehat{Q\urcorner/\ulcorner P\urcorner}\} :=\urcorner x\ulcorner}$$

FIGURE 2.5: ρ-calculus semantic substitution

$$(\ulcorner x \urcorner)\{\ulcorner \widehat{Q \urcorner / \ulcorner P} \urcorner\} = \begin{cases} Q & x \equiv_N \ulcorner P \urcorner \\ \ulcorner x \urcorner & otherwise \end{cases}$$

In the remainder of the paper we will refer to semantic and syntactic substitutions simply as substitutions and rely on context to distinguish which is meant. Similarly, we will abuse notation and write $\{y/x\}$ for $\widehat{\{y/x\}}$.

Finally equipped with these standard features we can present the dynamics of the calculus.

2.1.8 Operational Semantics

The reduction rules for ρ-calculus are

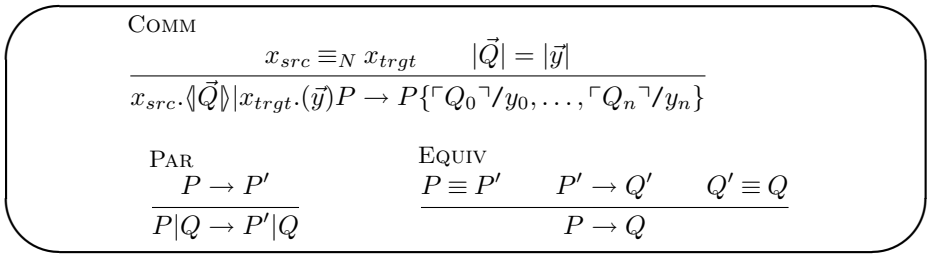

FIGURE 2.6: ρ-calculus reduction relation

The context rules are entirely standard and we do not say much about them, here. The communication rule does what was promised, namely make it possible for agents to synchronize and communicate processes packaged as names. For example, using the comm rule and name equivalence we can now justify our syntactic sugar for output.

$$x![z]|x?(y).P = x!\langle\!|\ulcorner z \urcorner|\!\rangle|x?(y).P \to P\{\ulcorner \ulcorner z \urcorner \urcorner / y\} \equiv P\{z/y\}$$

But, it also provides a scheme that identifies the role of name equality in synchronization. We explore this family of calculi in a forthcoming paper. For the rest of this paper, however, we restrict our attention to the calculus with the less exotic communication rule, using $\to$ for reduction according to that system and $\Rightarrow$ for $\to^*$.

2.2 Replication

As mentioned before, it is known that replication (and hence recursion) can be implemented in a higher-order process algebra [38]. As our first example of calculation with the machinery thus far presented we give the construction explicitly in the ρ-calculus.

$$D(x) := x?(y).(x![y]|\ulcorner y\urcorner) \qquad !P(x) := x\langle\!\langle D(x)|P\rangle\!\rangle|D(x)$$

$$\begin{array}{rc}
 & !P(x) \\
= & x\langle\!\langle D(x)|P\rangle\!\rangle|D(x) \\
= & x\langle\!\langle (x?(y).(x![y]|\ulcorner y\urcorner))|P\rangle\!\rangle|x?(y).(x![y]|\ulcorner y\urcorner) \\
\rightarrow & (x![y]|\ulcorner y\urcorner)\{\ulcorner (x?(y).(\ulcorner y\urcorner|x![y]))|P\urcorner/y\} \\
= & x![\ulcorner (x?(y).(x![y]|\ulcorner y\urcorner))|P\urcorner]|(x?(y).(x![y]|\ulcorner y\urcorner))|P \\
= & x![\ulcorner D(x)|P\urcorner]|D(x)|P \\
\rightarrow & \ldots \\
\rightarrow^* & P|P|\ldots
\end{array}$$

Of course, this encoding, as an implementation, runs away, unfolding $!P$ eagerly. A lazier and more implementable replication operator, restricted to input-guarded processes, may be obtained as follows.

$$!u?(v).P \triangleq x\langle\!\langle u?(v).(D(x)|P)\rangle\!\rangle|D(x)$$

It is worth noting that the lift operator is essential to get computational completeness.

2.3 Bisimulation

Having taken the notion of restriction out of the language, we carefully place it back into the notion of observation, and hence into the notion of program equality, i.e. bisimulation. That is, we parameterize the notion of barbed bisimulation by a set of names over which we are allowed to set the barbs. The motivation for this choice is really comparison with other calculi. The set of names of the ρ-calculus is *global*. It is impossible, in the grammar of processes, to guard terms from being placed into contexts that can potentially observe

communication. So, we provide a place for reasoning about such limitations on the scope of observation in the theory of bisimulation.

REMARK 2.3 Because this point about names and observation is somewhat subtle, it is worthwhile comparing this situation to Milner's original calculus. In the π-calculus we have the ν operator that scopes names and ostensibly limits observations. But, if we review the situation carefully, we see that Milner's presentation demands *both* a countably infinite set of names *and* an effective equality on these names. This pair of demands cannot be realized on a computer for *atomic* names, i.e. names without structure. To achieve both demands on a computer the structure of names and how those names are compared for equality *has* to be revealed. Thus, for *any* effective realization of the π-calculus the name set is *always global*. This means that in any effective presentation of Milner's calculus ν scopes are meaningless in terms of protection against barbs. To bring this point home, in an effective presentation of the π-calculus the term $(\nu\ x)x?(y).0$ cannot be bisimilar to 0 because the program has to pick a binding for x and after finite guesses for x an attacker of the form $x!(z).exploit!(x)$ will signal the binding of x on the channel *exploit*. □

DEFINITION 2.1 *An* observation relation, $\downarrow_{\mathcal{N}}$, *over a set of names,* $\mathcal{N}$, *is the smallest relation satisfying the rules below.*

$$\frac{y \in \mathcal{N},\ x \equiv_N y}{x![v] \downarrow_{\mathcal{N}} x} \qquad (\textsc{Out-barb})$$

$$\frac{P \downarrow_{\mathcal{N}} x \text{ or } Q \downarrow_{\mathcal{N}} x}{P|Q \downarrow_{\mathcal{N}} x} \qquad (\textsc{Par-barb})$$

We write $P \Downarrow_{\mathcal{N}} x$ *if there is* Q *such that* $P \Rightarrow Q$ *and* $Q \downarrow_{\mathcal{N}} x$.

Notice that $x?(y).P$ has no barb. Indeed, in ρ-calculus as well as other asynchronous calculi, an observer has no direct means to detect if a message sent has been received or not.

DEFINITION 2.2 *An* $\mathcal{N}$-barbed bisimulation *over a set of names,* $\mathcal{N}$, *is a symmetric binary relation* $\mathcal{S}_{\mathcal{N}}$ *between agents such that* $P\ \mathcal{S}_{\mathcal{N}} Q$ *implies:*

1. *If* $P \rightarrow P'$ *then* $Q \Rightarrow Q'$ *and* $P'\ \mathcal{S}_{\mathcal{N}} Q'$.
2. *If* $P \downarrow_{\mathcal{N}} x$, *then* $Q \Downarrow_{\mathcal{N}} x$.

P *is* $\mathcal{N}$*-barbed bisimilar to* Q, *written* $P \dot{\approx}_{\mathcal{N}} Q$, *if* $P\ \mathcal{S}_{\mathcal{N}} Q$ *for some* $\mathcal{N}$*-barbed bisimulation* $\mathcal{S}_{\mathcal{N}}$.

Chapter 3

Logic

3.1 Namespace logic

Namespace logic resides in the subfamily of Hennessy-Milner logics discovered by Caires and Cardelli and known as spatial logics [**?**]. Thus, as is seen below, in addition to the action modalities, we also find formulae for *separation*, corresponding, at the logical level, to the structural content of the parallel operator at the level of the calculus. Likewise, we have quantification over names.

In this connection, however, we find an interesting difference between spatial logics investigated heretofore and this one. As in the calculus, we find no need for an operator corresponding to the ν construction. However, revelation in spatial logic, is a structural notion [**?**]. It detects the *declaration* of a new name. No such information is available in the reflective calculus or in namespace logic. The calculus and the logic can arrange that names are used in a manner consistent with their being declared as new in the π-calculus, but it cannot detect the declaration itself. Seen from this perspective, revelation is a somewhat remarkable observation, as it seems to be about detecting the programmer's intent.

Boolean	Spatial	Behavioral	Recursive
$\phi, \psi ::= true \mid \neg\phi \mid \phi \& \psi$	$\mid 0 \mid \phi \mid \psi$	$\mid a(\!\langle \phi \rangle\!) \mid \langle a?b \rangle \phi$	$\mid \mathsf{rec}\ X\ .\ \phi$

Nominal

$\mid \ulcorner b \urcorner \mid \forall n : \psi\ .\ \phi \qquad a ::= \ulcorner \phi \urcorner \mid b \qquad b ::= \ulcorner P \urcorner \mid n$

FIGURE 3.1: namespace logic formulae

We let $PForm$ denote the set of formulae generated by the ϕ-production, $QForm$ denote the set of formulae generated by the a-production and $\mathcal{V}$ denote the set of propositional variables used in the rec production.

Inspired by Caires' presentation of spatial logic [**?**], we give the semantics in terms of sets of processes (and names). We need the notion of a valuation

$v : \mathcal{V} \to \wp(Proc)$, and use the notation $v\{\mathcal{S}/X\}$ to mean

$$v\{\mathcal{S}/X\}(Y) = \begin{cases} S & Y = X \\ v(Y) & otherwise \end{cases}$$

The meaning of formulae is given in terms of two mutually recursive functions,

$$[\![-]\!](-) : PForm \times [\mathcal{V} \to \wp(Proc)] \to \wp(Proc)$$
$$(\!(-)\!)(-) : QForm \times [\mathcal{V} \to \wp(Proc)] \to \wp(\ulcorner Proc \urcorner)$$

taking a formula of the appropriate type and a valuation, and returning a set of processes or a set of names, respectively.

$$[\![true]\!](v) = Proc$$

$$[\![\neg\phi]\!](v) = Proc \setminus [\![\phi]\!](v) \qquad [\![\phi \& \psi]\!](v) = [\![\phi]\!](v) \cap [\![\psi]\!](v)$$

$$[\![0]\!](v) = \{P : P \equiv 0\}$$

$$[\![\phi|\psi]\!](v) = \{P : \exists P_0, P_1 . P \equiv P_0|P_1, P_0 \in [\![\phi]\!](v), P_1 \in [\![\psi]\!](v)\}$$

$$[\![a\langle\!|\phi|\!\rangle]\!](v) = \{P : \exists Q, P' . P \equiv Q|x\langle\!|P'|\!\rangle,\ x \in (\!(a)\!)(v),\ P' \in [\![\phi]\!](v)\}$$

$$[\![\langle a?b\rangle\phi]\!](v) = \{P : \exists Q, P' . P \equiv Q|x(y).P', x \in (\!(a)\!)(v), \forall c . \exists z . P'\{z/y\} \in [\![\phi\{c/b\}]\!](v)\}$$

$$[\![\mathsf{rec}\ X\ .\ \phi]\!](v) = \cup\{\mathcal{S} \subseteq Proc : \mathcal{S} \subseteq [\![\phi]\!](v\{\mathcal{S}/X\})\}$$

$$[\![\ulcorner b \urcorner]\!](v) = \{P : \exists Q, P' . P \equiv Q|\ulcorner x \urcorner,\ x \in (\!(b)\!)(v)\}$$

$$[\![\forall n : \psi\ .\ \phi]\!](v) = \cap_{x \in (\!(\ulcorner \psi \urcorner)\!)(v)} [\![\phi\{x/n\}]\!](v)$$

$$(\!(\ulcorner \phi \urcorner)\!)(v) = \{x : x \equiv_N \ulcorner P \urcorner, P \in [\![\phi]\!](v)\} \qquad (\!(\ulcorner P \urcorner)\!)(v) = \{x : x \equiv_N \ulcorner P \urcorner\}$$

FIGURE 3.2: namespace logic semantics

We say P witnesses ϕ (resp., x witnesses $\ulcorner \phi \urcorner$), written $P \models \phi$ (resp., $x \models \ulcorner \phi \urcorner$) just when $\forall v . P \in [\![\phi]\!](v)$ (resp., $\forall v . x \in [\![\ulcorner \phi \urcorner]\!](v)$).

THEOREM 3.1 Equivalence
$P \dot{\approx} Q \Leftrightarrow \forall \phi . P \models \phi \Leftrightarrow Q \models \phi$.

The proof employs an adaptation of the standard strategy. As noted in the introduction, this theorem means that there is no algorithm guaranteeing that a check for the witness relation will terminate.

3.1.0.1 Syntactic sugar

In the examples below, we freely employ the usual DeMorgan-based syntactic sugar. For example,

$$\phi \Rightarrow \psi \triangleq \neg(\phi \& \neg\psi)$$
$$\phi \vee \psi \triangleq \neg(\neg\phi \& \neg\psi)$$

Also, when quantification ranges over all of $Proc$, as in $\forall n : \ulcorner true \urcorner \,.\, \phi$, we omit the typing for the quantification variable, writing $\forall n \,.\, \phi$.

3.1.1 Examples

3.1.1.1 Controlling access to namespaces

Suppose that $\ulcorner \phi \urcorner$ describes some namespace, i.e. some collection of names. We can insist that a process restrict its next input to names in that namespace by insisting that it witness the formula

$$\langle \ulcorner \phi \urcorner ? b \rangle \, true \, \& \neg \langle \ulcorner \neg \phi \urcorner ? b \rangle true$$

which simply says the the process is currently able to take input from a name in the namespace $\ulcorner \phi \urcorner$ and is not capable of input on any name not in that namespace. In a similar manner, we can limit a server to serving only inputs in $\ulcorner \phi \urcorner$ throughout the lifetime of its behavior [1]

$$\mathsf{rec}\ X \,.\, \langle \ulcorner \phi \urcorner ? b \rangle X \& \neg \langle \ulcorner \neg \phi \urcorner ? b \rangle true$$

This formula is reminiscent of the functionality of a firewall, except that it is a *static* check. A process witnessing this formula will behave as though it were behind a firewall admitting only access to the ports in $\ulcorner \phi \urcorner$ without the need for the additional overhead of the watchdog machinery.

3.1.1.2 Validating the structure of data

Of course, the previous example might make one wonder what a useful namespace looks like. The relevance of this question is further amplified when

[1] Of course, this formula also says the server never goes down, either – or at least is always willing to take such input...;-)

we observe that processes pass names as messages as well as use them to govern synchronization. The next example, therefore, considers a space of names that might be seen as well-suited to play the role of data, for their structure loosely mimics the structure of the infoset model [**?**] of `XML` (sans schema).

$$\phi_{info} = \ulcorner \mathsf{rec}\ X\ .\ (\forall m\ .\ m\langle\!|\forall n\ .\ 0 \vee n\langle\!|X|\!\rangle \vee \mathsf{rec}\ Y\ .\ (\forall n'\ .\ \langle n'?b\rangle(X \vee Y)) \vee (X|X)|\!\rangle)\urcorner$$

The formula is essentially a recursive disjunction selecting names that are first of all rooted with an enclosing lift operation – reminiscent of the way an `XML` document has a single enclosing root; and then are either

- the empty 'document'; or
- an 'element'; or
- a sequence of documents each 'located' at an input action; or
- an unordered group.

Notice that it is possible to parameterize this namespace on names for rooting 'documents' or 'elements'. Currently, these are typed as coming from the whole namespace, $\ulcorner true \urcorner$, but they could come from any subspace.

Moreover, the formula is itself a template for the interpretation of schema specifications [**?**]. If we boil `XSD` schema down to its essential type constructors, we have a recursive specification in which a schema is a

- a sequence, or
- a choice, or
- a group, or
- a recursion, in which a type name is bound to a schema definition

of element-tagged schema or schema references, with the recursive specification bottoming out at the simple and builtin types. Abstractly, then essential structures of `XSD` schema are captured by the grammar

$$S ::= ESeq \mid ESum \mid EGrp \mid \mathsf{rec}\ N\ .\ S$$

$$ESeq ::= \epsilon \mid E, ESeq \qquad ESum ::= \epsilon \mid E + ESum \qquad EGrp ::= \epsilon \mid E|EGrp$$

$$E ::= tag(N \mid S)$$

FIGURE 3.3: `XML`-like schema

We use s to range over schema, σ, χ and γ to range over sequences, choices and groups, respectively.

The encoding below, which for clarity makes liberal – but obvious – use of polymorphism and elides the standard machinery for treating recursion variables, illustrates that we can view this grammar as essentially providing a high-level language for carving out namespaces in which the names conform to the schema.

$$\begin{aligned}
[\![\epsilon]\!] &= \ulcorner 0 \urcorner \\
[\![tag(s), \sigma]\!] &= \ulcorner \forall n : [\![tag]\!] \,.\, \langle n?b \rangle ([\![s]\!] | [\![\sigma]\!]) \urcorner \\
[\![tag(s) + \chi]\!] &= \ulcorner \forall n : [\![tag]\!] \,.\, (\langle n?b \rangle [\![s]\!]) \vee [\![\chi]\!] \urcorner \\
[\![tag(s) \mid \gamma]\!] &= \ulcorner \forall n : [\![tag]\!] \,.\, (\langle n?b \rangle [\![s]\!]) | [\![\gamma]\!] \urcorner \\
[\![\mathsf{rec}\ N \,.\, s]\!] &= \ulcorner \mathsf{rec}\ N \,.\, [\![s]\!] \urcorner
\end{aligned}$$

We emphasize that the example is not meant to be a complete account of `XML` schema. Rather, it is intended to suggest that with the reflective capabilities the logic gives a fairly intuitive treatment of names as structured data. The simplicity and intuitiveness of the treatment is really brought home, however, when employing the framework analytically. As an example, from a commonsense perspective it should be the case that any `XML` document that observes a schema automatically also corresponds to an infoset. The reader is encouraged to try her hand at using the framework to establish that if s is a schema, then

$$x \models [\![s]\!] \Rightarrow x \models \phi'_{info}$$

where ϕ'_{info} a suitably modified version of ϕ_{info}.

Part II

From calculi to language design

Chapter 4

Data and control

4.1 Names and messages

Having derived a closed account of names it is hard to miss the fact that names have an incredible amount of structure. In fact, remarkably the structure is not terribly far from that of `XML` documents as noted in []. The real issue is in making this structure accessible to processes. In point of fact, we already have mechanism to construct names. The drop operator acts as a kind of splicing mechanism (ala `Lisp` or `Scheme` macros), while the lift operator actually produces names from process terms (that may have been partially constructed from splicing). What is missing is a means of taking them apart. Taking a page from a robust and increasingly popular language (`Erlang`) which, in turn, took a page from the functional languages, the natural place to introduce this *destructuring* mechanism is in *input*. In turn, wrapping this *pattern-matching* behavior in a case-like construct – which just happens to fit with the notion of summation – is also a natural design choice. The only real wrinkle in this approach is the issue of pattern-matching patterns.

4.1.1 Grammar

4.2 Conclusions and future work

TBD

4.2.0.0.1 Acknowledgments. TBD

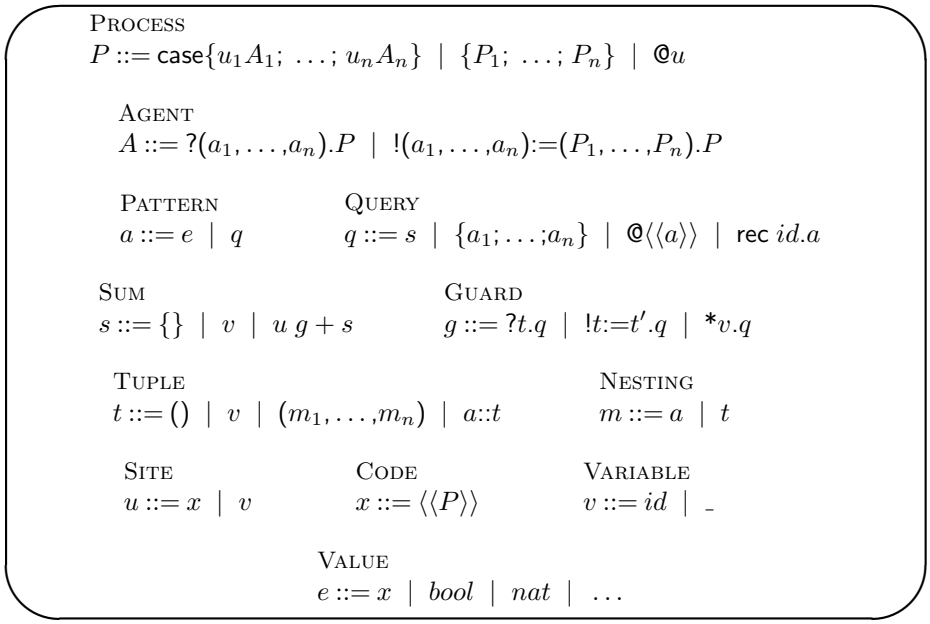

FIGURE 4.1: holOgram abstract syntax

References

[1] Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. In *POPL*, pages 33–44, 2002.

[2] Martín Abadi and Bruno Blanchet. Secrecy types for asymmetric communication. *Theor. Comput. Sci.*, 3(298):387–415, 2003.

[3] Samson Abramsky. Computational interpretations of linear logic. *Theor. Comput. Sci.*, 111(1&2):3–57, 1993.

[4] Samson Abramsky. Algorithmic game semantics and static analysis. In *SAS*, page 1, 2005.

[5] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[6] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. Assoc. Comput. Mach.*, 31(3):560–599, 1984.

[7] Luís Caires. Behavioral and spatial observations in a logic for the pi-calculus. In *FoSSaCS*, pages 72–89, 2004.

[8] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part i). *Inf. Comput.*, 186(2):194–235, 2003.

[9] Luís Caires and Luca Cardelli. A spatial logic for concurrency - ii. *Theor. Comput. Sci.*, 322(3):517–565, 2004.

[10] Luca Cardelli. Brane calculi. In *CMSB*, pages 257–278, 2004.

[11] Vincent Danos and Cosimo Laneve. Core formal molecular biology. In Pierpaolo Degano, editor, *ESOP*, volume 2618 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2003.

[12] Wan Fokkink, Rob van Glabbeek, and Paulien de Wind. Compositionality of Hennessy-Milner logic through structural operational semantics. In *Fundamentals of computation theory*, volume 2751 of *Lecture Notes in Comput. Sci.*, pages 412–422. Springer, Berlin, 2003.

[13] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. Jocaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, pages 129–158, 2002.

[14] Cédric Fournet, Georges Gontheir, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR 1996*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer-Verlag, 1996.

[15] C. A. R. Hoare. *Communicating sequential processes*. Prentice Hall International Series in Computer Science. Prentice Hall International, Englewood Cliffs, NJ, 1985. With a foreword by Edsger W. Dijkstra.

[16] C. A. R. Hoare. Notes on communicating sequential systems. In *Control flow and data flow: concepts of distributed programming (Marktoberdorf, 1984)*, volume 14 of *NATO Adv. Sci. Inst. Ser. F Comput. Systems Sci.*, pages 123–204. Springer, Berlin, 1985.

[17] C. A. R. Hoare. Algebraic specifications and proofs for communicating sequential processes. In *Logic of programming and calculi of discrete design (Marktoberdorf, 1986)*, volume 36 of *NATO Adv. Sci. Inst. Ser. F Comput. Systems Sci.*, pages 277–300. Springer, Berlin, 1987.

[18] Ole Høgh Jensen and Robin Milner. Bigraphs and transitions. In *POPL*, pages 38–49, 2003.

[19] Allen L. Brown Jr., Cosimo Laneve, and L. Gregory Meredith. Piduce: A process calculus with native xml datatypes. In Mario Bravetti, Leïla Kloul, and Gianluigi Zavattaro, editors, *EPEW/WS-FM*, volume 3670 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2005.

[20] Jean-Louis Krivine. The curry-howard correspondence in set theory. In Martin Abadi, editor, *Proceedings of the Fifteenth Annual IEEE Symp. on Logic in Computer Science, LICS 2000*. IEEE Computer Society Press, June 2000.

[21] Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In *FoSSaCS*, pages 282–298, 2005.

[22] Greg Meredith and Steve Bjorg. Contracts and types. *Commun. ACM*, 46(10):41–47, 2003.

[23] L. Gregory Meredith and Matthias Radestock. A reflective higher-order calculus. *Electr. Notes Theor. Comput. Sci.*, 141(5):49–67, 2005.

[24] L.G. Meredith and Matthias Radestock. A reflective higher-order calculus. In Mirko Viroli, editor, *ETAPS 2005 Satellites*. Springer-Verlag, 2005.

[25] R. Milner and D. Sangiorgi. Techniques of weak bisimulation up-to, 1992.

[26] Robin Milner. *A Calculus of Communicating Systems*. Springer Verlag, 1980.

[27] Robin Milner. Calculi for synchrony and asynchrony. *Journal of Theoretical Computer Science*, 25:267–310, 1983.

[28] Robin Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.

[29] Robin Milner. Elements of interaction - turing award lecture. *Commun. ACM*, 36(1):78–89, 1993.

[30] Robin Milner. The polyadic π-calculus: A tutorial. *Logic and Algebra of Specification*, Springer-Verlag, 1993.

[31] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. I, II. *Inform. and Comput.*, 100(1):1–40, 41–77, 1992.

[32] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In Werner Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer, 1992.

[33] Andrew Phillips and Luca Cardelli. A correct abstract machine for the stochastic pi-calculus. In *Bioconcur'04*. ENTCS, August 2004.

[34] Damien Pous. Weak bisimulation up to elaboration. In Christel Baier and Holger Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 390–405. Springer, 2006.

[35] Corrado Priami. Stochastic pi-calculus. *Comput. J.*, 38(7):578–589, 1995.

[36] Corrado Priami, Aviv Regev, Ehud Y. Shapiro, and William Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Inf. Process. Lett.*, 80(1):25–31, 2001.

[37] Amitai Regev and Ehud Y. Shapiro. Cells as computation. In Corrado Priami, editor, *CMSB*, volume 2602 of *Lecture Notes in Computer Science*, pages 1–3. Springer, 2003.

[38] David Sangiorgi and David Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[39] Davide Sangiorgi. On the proof method for bisimulation (extended abstract). In Jirí Wiedermann and Petr Hájek, editors, *MFCS*, volume 969 of *Lecture Notes in Computer Science*, pages 479–488. Springer, 1995.

[40] Davide Sangiorgi. Bisimulation in higher-order process calculi. *Information and Computation*, 131:141–178, 1996.

[41] Davide Sangiorgi. Bisimulation: From the origins to today. In *LICS*, pages 298–302. IEEE Computer Society, 2004.

[42] Davide Sangiorgi and Robin Milner. The problem of "weak bisimulation up to". In Rance Cleaveland, editor, *CONCUR*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 1992.

[43] Pawel T. Wojciechowski and Peter Sewell. Nomadic pict: Language and infrastructure design for mobile agents. In *ASA/MA*, pages 2–12, 1999.