LUCA

CARDELLIFEST

Martín Abadi
Philippa Gardner
Andrew D. Gordon
Radu Mardare (Eds.)

Essays for the Luca Cardelli Fest
September 8-9, 2014
Microsoft Research Cambridge
Technical Report MSR-TR-2014-104

Preface

Luca Cardelli has made exceptional contributions to the field of programming languages and beyond. Throughout his career, he has re-invented himself every decade or so, while continuing to make true innovations. His achievements span many areas: software; language design, including experimental languages; programming language foundations; and the interaction of programming languages and biology. These achievements form the basis of his lasting scientific leadership and his wide impact.

Luca was born in Montecatini Terme, Italy, studied at the University of Pisa until 1978, and received a Ph.D. in computer science from the University of Edinburgh, supervised by Gordon Plotkin. He held research positions at Bell Labs, Murray Hill (1982–1985), and at Digital's Systems Research Center, Palo Alto (1985–1997). He joined Microsoft Research in Cambridge in 1997, where for many years he managed the Programming Principles and Tools group. Since 2013, he holds a Royal Society Research Professorship at the University of Oxford, alongside his position as Principal Researcher at Microsoft Research.

Luca Cardelli is a Fellow of the Royal Society, an ACM Fellow, an Elected Member of the Academia Europaea, an Elected Member of AITO (International Association for Object Technologies), and a member of EATCS (European Association for Theoretical Computer Science) and ISCB (International Society for Computational Biology).

A scientific event in honour of Luca Cardelli has been organized in Cambridge (UK) on September 8–9, 2014. This celebration will gather many of Luca's colleagues and friends. It will include talks on a wide variety of topics, corresponding to many of the areas in which Luca has worked. Its program is available at `http://research.microsoft.com/lucacardellifest/`.

Complementing these talks, and as a more lasting reflection of the event, many of Luca's colleagues and friends wrote the short papers included in this informal volume. Luca is always asking "what is new", and is always looking to the future. Therefore, we have asked authors to produce short pieces that would indicate where they are today and where they are going. Some of the resulting pieces are short scientific papers, or abridged versions of longer papers; others are less technical, with thoughts on the past and ideas for the future. We hope that they will all interest Luca.

We thank all the contributors for their work, and Andrew Phillips for his editorial help.

Martín Abadi
Philippa Gardner
Andrew D. Gordon
Radu Mardare

Table of Contents

Naiad Models and Languages

Martín Abadi
Microsoft Research

Abstract

Naiad is a recent distributed system for data-parallel computation. The goal of this note is to review Naiad and the computing model that underlies it, and to suggest opportunities for explaining them and extending them through programming languages, calculi, and semantics.

1 Prologue

Luca Cardelli's research often demonstrates the great value of programming notations and their semantics in understanding and improving a wide variety of computational concepts and phenomena. Some of these concepts and phenomena—for example, linking—are close to programming languages, though often hidden inside implementations. Others, like ambients and chemical reactions, may seem more surprising. Finding and developing an appropriate programming-language perspective on such subjects requires insight and elegance, but it can be quite fruitful, as Luca's work illustrates superbly.

On the occasion of a celebration of Luca's research, his friends and colleagues were invited to write short pieces that would summarize where they are and where they are going. At present I am engaged in several disparate activities in computer security and in programming languages and systems. So, in this note, I chose to focus on one particular project, namely the development of the Naiad system and of the corresponding theory. This subject is related to people and topics that I encountered while working with Luca, such as Kahn networks and the integration of database constructs into programming languages. More importantly, perhaps, it is a subject in which Luca's perspective and approach may prove crucial in the future. (Lately, I sometimes ask myself *"What would Luca do?"*) The subject does not have obvious connections to the many topics on which Luca and I worked together (type systems, objects, explicit substitutions, and more), nor does it have the breadth of Luca's current research interests—but that may all come in due course.

The next section is a brief description of Naiad; further details and references can be found in recent papers (Murray et al. 2013; Abadi et al. 2013; McSherry et al. 2013). The following section outlines some areas of current and future research (joint work with many colleagues, listed below).

2 Naiad in a nutshell

Naiad is a distributed system for data-parallel computation. It aims to offer high throughput and low latency, and to support a range of tasks that includes traditional batch and stream processing and also iterative and incremental computations. For example, a Naiad application may process a stream of events such as tweets (Murray et al. 2013); as they arrive, the events may feed into an incremental connected-components computation and other analyses, which may, for instance, identify the most popular topics in each community of users. Naiad thus aspires to serve as a general, coherent platform for data-parallel applications. In this respect, Naiad contrasts with other systems for data-parallel computation that focus on narrower domains (e.g., graph problems) or on particular styles of programs (e.g., with restrictions on loops).

2.1 Timely dataflow

At the core of Naiad is a model for parallel computing that we call timely dataflow. This model extends dataflow computation with a notion of virtual time. As in Jefferson's Time Warp mechanism (Jefferson 1985), virtual time serves to differentiate between data at different phases of a computation. Unlike in the Time Warp mechanism, however, virtual time is partially ordered (rather than linearly ordered, since a linear order may impose false dependencies).

In Naiad, each communication event is associated with a virtual time. This association enables the runtime system to overlap—but still distinguish—work that corresponds to multiple logical stages in a computation: different input epochs, iterations, workflow steps, and perhaps speculative executions. It also enables the runtime system to notify nodes when they have received their last message for a given virtual time. This combination of asynchronous scheduling and completion notification implies that, within a single application, some components can function in batch mode (queuing inputs and delaying processing until an appropriate notification) and others in streaming mode (processing inputs as they arrive).

As is typical in dataflow models, we specify computations as directed graphs, with distinguished input nodes and output nodes. The graphs may contain loops, even nested loops. Therefore, at each location in a graph, we can define timestamps of the form (input epoch, loop counters), where there is one loop counter for each enclosing loop context at that location. Each loop must contain a feedback node whose function is to increment a timestamp counter. Nodes for loop ingress and egress introduce and remove time coordinates, respectively.

During execution, stateful nodes send and receive timestamped messages, and in addition may request and receive notification that they have received all messages with a certain timestamp. So the runtime system must be able to reason about the possibility or impossibility of such future messages. This reasoning relies on the "could-result-in" relation, which intuitively captures whether an event at graph location l_1 and virtual time t_1 could result in an event at graph location l_2 and virtual time t_2. The reasoning combines a simple static analysis of the graph with a distributed algorithm for tracking progress.

2.2 Higher layers

Building on the timely dataflow substrate, Naiad offers several higher-level programming models. To date, we have the most experience with a model based on language-integrated queries, specifically on a dialect of LINQ. This dialect includes batch-oriented operators that process entire data collections at once, in the spirit of DryadLINQ (Yu et al. 2008) and Spark (Zaharia et al. 2012), incremental operators that accumulate state between invocations, and differential operators that compute in terms of additions and deletions of data records. We have also explored other programming idioms and interfaces, such as Pregel-style Bulk Synchronous Parallel computation (Malewicz et al. 2010) and BLOOM-style asynchronous iteration (Conway et al. 2012).

3 Further work

The remainder of this note outlines a few directions of current and future work (though not a comprehensive list), roughly in top-down order. Some of this work is well underway; other suggestions are more tentative and speculative.

3.1 Other front-end models and languages

As mentioned above, our work to date relies primarily on language-integrated queries, but Naiad can support other models of iterative computation. It may be worthwhile to investigate those in more detail. Going beyond these models, however, we may wish to support recursive dataflow computations, in addition to iterative ones.

Recursion in dataflow is not entirely new (Blelloch 1996), but supporting it in Naiad gives rise to theoretical and practical difficulties, at various levels.

In particular, Naiad's current concrete embodiment of virtual time is based on simple iteration counters. With recursion, stacks would probably have to play a role, and the treatment of the could-result-in relation would need to be revisited.

One possible approach might go as follows. For simplicity, let us consider a dataflow graph that includes distinguished nodes that represent recursive calls to the entire computation, an input node `in`, an output node `out`, and some ordinary nodes (for instance, for selects and joins). We split each recursive-call node `c` into a call part `call-c` and a return part `ret-c`, with an edge between them. A stack is then a finite sequence of recursive-call nodes, and a "pointstamp" a pair of a stack and a node. Finally, the could-result-in relation is the least reflexive, transitive relation on these pointstamps such that: (1) $(s, \texttt{call-c})$ could result in $(\texttt{s.c}, \texttt{in})$; (2) symmetrically, $(s.\texttt{c}, \texttt{out})$ could result in $(s, \texttt{ret-c})$; and (3) if v is not `call-c` and v' is not `ret-c` for any `c`, and there is an edge from v to v', then (s, v) could result in (s, v'). This definition looks principled but too complex to be useful at run-time. Fortunately, it is equivalent to a simpler criterion that requires only finding the first call in which two stacks differ and performing an easy check based on that difference.[1] Special cases might allow further simplifications.

3.2 Differential computation

One of Naiad's distinctive characteristics is its support for differential computation. In computing over collections of data, nodes can transmit deltas, rather than entire collections. Current work aims to generalize differential computation and to put it on a proper semantic foundation.

Differential computation makes sense over any abelian group G (and not just over collections of data). It also makes sense relative to many partial orders T, but some hypotheses on T are necessary or at least convenient. Specifically, the differential version $\delta f : T \rightarrow G$ of a function $f : T \rightarrow G$ should be such that $f(t) = \Sigma_{t' \leq t}(\delta f)(t')$. If, for each t there are only finitely many t' such that $t' \leq t$, the Möbius inversion theorem (Rota 1964), from combinatorics, implies that δf exists and is unique.[2] If there are infinitely many t' below t, on the other hand, the sum may not be meaningful. Alas, Naiad has sometimes relied on lexicographic orders for which the finiteness condition does not immediately hold; one of our

[1]Without loss of generality, suppose that s is of the form $s_1.s_2$ and s' is of the form $s_1.s'_2$, where s_2 and s'_2 start with call nodes `call-c` and `call-c'` respectively if they are not empty. We assume that `call-c` and `call-c'` are distinct if s_2 and s'_2 are both non-empty (so, s_1 is maximal). Let l be `ret-c` if s_2 is non-empty, and be v if it is empty. Let l' be `call-c'` if s'_2 is non-empty, and be v' if it is empty. We can then prove that (s, v) could result in (s', v') if and only if there is a path from l to l'.

[2]Alternatively, one may assume only that there are finitely many elements in each interval, but require that f be 0 below a certain element.

goals is to make sense of this situation.

In programming-language terms, an abelian group of collections may be seen as a base type. Going further, a language with support for differential computation would include type operations to form other abelian groups. Some of these operations may be standard constructions such as products. Others would be more specific to iterative, differential computation. In particular, for each type σ, we may define a type σ^+ that intuitively represents functions from $\mathbf{N}$ to σ, that is, σ indexed by an additional natural-number time coordinate, as required for iteration.

At the level of terms, the language would include common constructs such as let expressions and iteration, and perhaps also differentiation and integration as first-class values. Since some of these constructs do not guarantee termination in general, the language semantics may need to allow partial functions, to which the classic inversion theorem does not immediately apply.

3.3 The essence of timely dataflow

Some of the ideas in timely dataflow seem viable independently of other aspects of Naiad, so we may try to recast and understand them in the general setting of a programming language or calculus. These often include facilities for communication via messages, but unfortunately not completion notifications, which are essential to Naiad. We may however be able to explain completion notifications using extant concepts.

For instance, it may be possible to capture the semantics of completion notifications in terms of the notion of priorities, which has sometimes been studied in process calculi (e.g., Versari et al. (2009); John et al. (2010)). For each virtual time t, we may regard the messages for time t as having higher priority than a notification that time t is complete, so this notification cannot be delivered before the messages.

An important caveat is that this notification should not be delivered before all the messages for time t, even ones not immediately available for transmission. This difficulty may be addressed by imposing, perhaps via the could-result-in relation, that any such future messages for time t be caused by present messages with higher priority than the notification.

3.4 Systems work

Although Naiad is by now a fairly mature prototype, further systems work would be useful. Some of it is pure engineering; some requires new research.

In particular, Naiad's fault-tolerance remains rudimentary. In the last few months, we have been designing more flexible fault-tolerance mechanisms. Reasoning about their correctness is tricky, and might perhaps benefit from work in concurrency theory, more specifically on models of causality and reversibility (e.g., Phillips and Ulidowski (2013)). So far, however, we have been able to make good progress with a straightforward linear-time semantics. We have been writing proofs with prophecy variables—pleasantly reminiscent of DEC SRC, if nothing else. In general, prophecy variables are unusual auxiliary variables for which present values are defined in terms of future values (Abadi and Lamport 1991). In this particular case, the prophecy variables predict which nondeterministic choices will persist despite rollbacks.

4 Conclusion

Naiad remains a work in progress, as this note indicates. This note is based on research with Paul Barham, Rebecca Isaacs, Michael Isard, Frank McSherry, Derek Murray, Gordon Plotkin, Tom Rodeheffer, Nikhil Swamy, and Dimitrios Vytiniotis. I am grateful to them for our collaboration. I am also grateful to Luca Cardelli, who is partly responsible for my interest and research in programming languages and systems.

References

M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

M. Abadi, F. McSherry, D. G. Murray, and T. L. Rodeheffer. Formal analysis of a distributed algorithm for tracking progress. In D. Beyer and M. Boreale, editors, *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013*, volume 7892 of *Lecture Notes in Computer Science*, pages 5–19. Springer, 2013.

G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, Mar. 1996.

N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1:1–1:14, 2012.

D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

M. John, C. Lhoussaine, J. Niehren, and A. M. Uhrmacher. The attributed pi-calculus with priorities. *Transactions on Computational Systems Biology*, 12: 13–76, 2010.

G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146, 2010.

F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research*, 2013.

D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13*, pages 439–455, 2013.

I. Phillips and I. Ulidowski. Reversibility and asymmetric conflict in event structures. In P. R. D'Argenio and H. C. Melgratti, editors, *CONCUR 2013 - Concurrency Theory - 24th International Conference*, volume 8052 of *Lecture Notes in Computer Science*, pages 303–318. Springer, 2013.

G.-C. Rota. On the foundations of combinatorial theory I. Theory of Möbius functions. *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 2 (4):340–368, 1964.

C. Versari, N. Busi, and R. Gorrieri. An expressiveness study of priority in process calculi. *Mathematical Structures in Computer Science*, 19(6):1161–1189, 2009.

Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*, pages 1–14, 2008.

M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 15–28, 2012.

The Behavior of Probabilistic Systems: From Equivalences to Behavioral Distances

Giorgio Bacci Giovanni Bacci Kim G. Larsen Radu Mardare
Department of Computer Science, Aalborg University, Denmark

Abstract

In this paper we synthesize our recent work on behavioral distances for probabilistic systems and present an overview of the current state of the art in the field. We mainly focus on behavioral distances for Markov chains, Markov decision processes, and Segala systems. We illustrate three different methods used for the definition of such metrics: logical, order theoretic, and measure-testing; and we discuss the relationships between them and provide the main arguments in support of each of them.

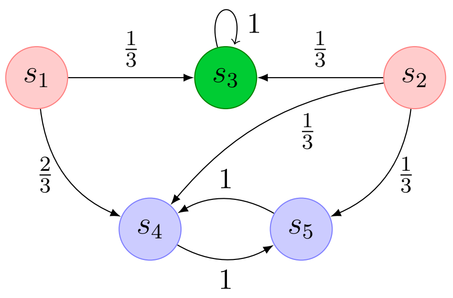
We also overview the problem of computing such distances, both from a theoretical and a practical view point, including the exact and the approximated methods.

1 Introduction

Probabilistic bisimulation of Larsen and Skou (1989) and probabilistic trace equivalence are acknowledged to be the basic equivalences for equating probabilistic systems from the point of view of their behaviors.

An example of probabilistic system is the labelled Markov chain depicted in Figure 1 (left). Here states s_1 and s_2 goes to state s_4 with probability $\frac{2}{3}$ and $\frac{1}{3}$, respectively. Although they move with different probabilities to s_4, states s_1 and s_2 are bisimilar because they reach any bisimilarity class with the same probability (clearly, also s_4 and s_5 are bisimilar).

When the probabilistic models are obtained as approximations of others, e.g., as simplified model abstractions or inferred from empirical data, then an equivalence is too strong a concept. This issue is illustrated in Figure 1 (right), where the states t_1 and t_2 (i.e., the counterpart of s_1 and s_2, respectively, after a perturbation of the transition probabilities) are not bisimilar. This motivated the quest for a robust notion of behavioral equivalence and the development of a theory of behavioral "nearness".



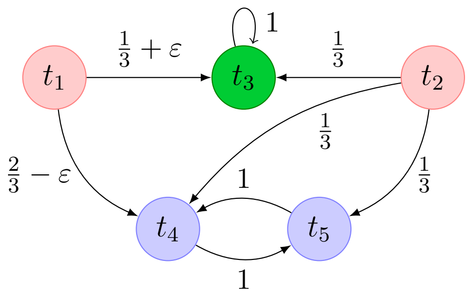


Figure 1: A labelled Markov chain (on the left) and an ε-perturbation of it (on the right), for some $0 < \epsilon < \frac{2}{3}$. Labels are represented by different colors.

To this end, Giacalone et al. (1990) proposed to use *pseudometrics* in place of equivalences aiming at measuring the behavioral similarities between states. Differently from an ordinary metric, a pseudometric allows different states to have distance zero, hence it can be thought of as a quantitative generalization of the notion of equivalence. In this respect, a pseudometric is said *behavioral* if states are at distance zero if and only if they are equivalent w.r.t. some behavioral semantics (e.g., bisimilarity, trace equivalence, etc.).

Behavioral distances do not come only as a robust notion of semantical equivalence, but they can also be used to address some important problems that are challenging computer science nowadays. One of these comes from systems biology and consists in providing analytical tools to help biologists understand the general mechanisms that characterize biological systems.

A considerable step forward in this direction is due to Luca Cardelli with his work on process algebra (Regev et al. 2004; Cardelli 2005, 2013) and on stochastic/ODE semantics (Cardelli 2008; Cardelli and Mardare 2010, 2013; Mardare et al. 2012), etc. Recently, Cardelli and Csikász-Nagy (2012) pointed out how the *Cell Cycle switch* (CC) —a fundamental biomolecular network that regulates the mitosis in eukaryotes— is surprisingly related, both in the structure and in the dynamics, to the *Approximate Majority* (AM) algorithm from distributed computing. The AM algorithm decides which of two populations is in majority by switching the majority into the totality, and it does so in a way that is *fast*, *reliable*, *robust*, and *asymptotically optimal* in the number of reactions required to obtain the result. The comparison between AM and CC is carried out by successive transformations that turn the network structure of the former into that of the latter. The difference is dictated by biological constraints, that are gradually introduced in the structure of AM during its transformation into CC while preserving both the computational and the dynamical properties.

Due to these observations, one may argue that CC, even though constrained by biological limitations, tends to behave as similar as possible to AM, the theoretical "optimal" one. In this respect, behavioral distances seem the appropriate analytical tool to measure the quality of a candidate network model for CC: the closer the network model is to AM the better it is.

In this paper we overview our recent work on behavioral distances in comparison with the current state of the art in the field, focusing on four types of probabilistic systems: discrete-time Markov Chains (MCs), Markov Decision Processes with rewards (MDPs), continuous-time Markov Chains (CTMC), and Segala Systems (SS). We examine the main techniques that have been used in the literature to characterize behavioral distances for probabilistic systems, namely, logical, fixed point, and coupling characterizations. For each technique, we show how they have been applied on the different types of probabilistic systems we are considering, we point out their differences and similarities, and provide practical and theoretical arguments supporting each one.

Finally, we consider the problem of computing such distances, including both the exact and the approximated methods, for which we overview the most recent theoretical complexity results that are known in the literature.

2 Preliminaries

A *probability distribution* over a finite set S is a function $\mu \colon S \to [0,1]$ such that $\sum_{s \in S} \mu(s) = 1$. We denote by $\mathcal{D}(S)$ the set of probability distributions over S.

Let S and L be nonempty finite sets of *states* and *labels*, respectively. A *discrete-time Markov chain* (MC) is a tuple $\mathcal{M} = (S, L, \tau, \ell)$ where $\tau \colon S \to \mathcal{D}(S)$ is a transition probability function, and $\ell \colon S \to L$ a labeling function. Intuitively, labels represent properties that hold at a given state, $\tau(s)(s')$ is the probability to move from state s to a successor state s'. A *continuous-time Markov Chain* (CTMC) $\mathcal{M} = (S, L, \tau, \rho, \ell)$ extends an MC with a rate function $\rho \colon S \to \mathbb{R}_+$ associating with each state s the rate of an exponential distribution representing the residence-time distribution at s. A *Segala system* (SS) extends an MC by adding nondeterminism: is a tuple $\mathcal{M} = (S, L, \theta, \ell)$ where $\theta \colon S \to 2^{\mathcal{D}(S)}$ assigns with each state a set of possible probabilistic outcomes. Finally, a *Markov decision process with rewards* (MDP) is a tuple $\mathcal{M} = (S, A, \vartheta, r)$ consisting a finite nonempty set A of *action labels*, a labeled transition probability function $\vartheta \colon S \times A \to \mathcal{D}(S)$, and a reward function $\rho \colon S \times A \to \mathbb{R}_+$ assigning to each state the reward associated with the chosen action.

A (1-bounded) *pseudometric* on a set S is a map $d \colon S \times S \to [0,1]$ such that for all $s, t, u \in S$, $d(s,s) = 0$, $d(s,t) = d(t,s)$ and $d(s,u) \leq d(s,t) + d(t,u)$. For a set S, the *indiscrete pseudometric* is defined as $\mathcal{I}_S(s,s') = 1$ if $s \neq s'$, otherwise 0.

For a pseudometric $d\colon S \times S \to [0,1]$ on S, we recall two pseudometrics on $\mathcal{D}(S)$

$$\|\mu - \nu\|_{\text{TV}} = \sup_{E \subseteq S} |\mu(E) - \nu(E)| \,, \qquad \text{(Total Variation)}$$
$$\mathcal{K}(d)(\mu,\nu) = \sup\left\{ \left|\int f \,\mathrm{d}\mu - \int f \,\mathrm{d}\nu\right| \mid |f(x) - f(y)| \leq d(x,y) \right\} \,, \qquad \text{(Kantorovich)}$$

and one pseudometric on 2^S defined, for $A, B \subseteq S$, as follows

$$\mathcal{H}(d)(A,B) = \max\{\sup_{a \in A} \inf_{b \in B} d(a,b), \sup_{b \in B} \inf_{a \in A} d(a,b)\} \,. \qquad \text{(Hausdorff)}$$

The set of 1-bounded pseudometrics on S, endowed with the pointwise pre-order $d_1 \sqsubseteq d_2$ iff $d_1(s,s') \leq d_2(s,s')$ for all $s, s' \in S$, forms a complete lattice, with bottom the constant 0 pseudometrics and top the indiscrete pseudometric $\mathcal{I}_S$.

Let $\lambda \in [0,1]$ and $a, b \in \mathbb{R}$, $a \oplus_\lambda b$ denotes the convex combination $\lambda a + (1-\lambda) b$.

3 Behavioral Distances

We overview the three main techniques that have been used in the literature for the characterization of behavioral pseudometrics over probabilistic systems.

3.1 Logical Characterizations

Real-valued logics. The first authentic behavioral pseudometric on probabilistic systems, due to Desharnais et al. (2004), is defined in terms of a family of functional expressions to be interpreted as real-valued modal formulas. Given a probabilistic model $\mathcal{M}$ over the set of states S, a functional $f \in \mathcal{F}$ is interpreted as a function $f_{\mathcal{M}}\colon S \to [0,1]$, and the pseudometric $\delta^{\mathcal{M}}\colon S \times S \to [0,1]$ assigns a distance to any given pair of states of $\mathcal{M}$ according to the following definition:

$$\delta^{\mathcal{M}}(s,s') = \sup_{f \in \mathcal{F}} |f_{\mathcal{M}}(s) - f_{\mathcal{M}}(s')| \,.$$

Their work builds on an idea of Kozen (1985) to generalize logic to handle probabilistic phenomena, and was first developed on MCs. Later, this approach has been adapted to MDPs, by de Alfaro et al. (2007) and Ferns et al. (2014), and extended to Segala systems by Mio (2013). Figure 2 shows the set of functional expressions for the case of MCs. One can think of those as logical formulas: a represents an atomic proposition, $\mathbf{1} - f$ negation, $\min(f_1, f_2)$ conjunction, $\diamond f$ the modal operator, and $f \ominus q$ the "greater then q" test. The key result for such a metric is that two states are at distance 0 iff they are probabilistic bisimilar.

$$
\begin{array}{rlr}
\mathcal{F} \ni f ::= & a & a_{\mathcal{M}}(s) = \mathcal{I}_L(a, \ell(s)) \\
| & \mathbf{1} - f & (\mathbf{1} - f)_{\mathcal{M}}(s) = 1 - f_{\mathcal{M}}(s) \\
| & \min(f, f) & (\min(f_1, f_2))_{\mathcal{M}}(s) = \min\{(f_1)_{\mathcal{M}}(s), (f_2)_{\mathcal{M}}(s)\} \\
| & f \ominus q & (f \ominus q)_{\mathcal{M}}(s) = \max\{f_{\mathcal{M}}(s) - q, 0\} \\
| & \Diamond f & (\Diamond f)_{\mathcal{M}}(s) = \int f_{\mathcal{M}} \, \mathrm{d}\tau(s)
\end{array}
$$

Figure 2: Real-valued logic: syntax (on the left) and interpretation (on the right), where $\mathcal{M} = (S, L, \tau, \ell)$ is an MC, $a \in L$ a label, and $q \in \mathbb{Q} \cap [0, 1]$.

Linear time logics. If the models can be observed by only testing single execution runs, then bisimulation is to stringent as an equivalence and trace equivalence is preferred instead. Similar arguments justify the introduction of behavioral distances that focus on linear time properties only.

In (Bacci et al. 2014) we compared CTMCs against linear real-time specifications expressed as Metric Temporal Logic (MTL) formulas (Alur and Henzinger 1993). MTL is a continuous-time extension of LTL, where the *next* and *until* operators are annotated with a closed time interval $I = [t, t']$, for $t, t' \in \mathbb{Q}_+$.

$$\varphi ::= p \mid \bot \mid \varphi \to \varphi \mid \mathsf{X}^I \varphi \mid \varphi \, \mathsf{U}^I \, \varphi \,,$$

The satisfiability relation $\pi \models \varphi$ is defined over timed paths $\pi = s_0, t_0, s_1, t_1 \dots$, where $t_i \in \mathbb{R}_+$ represents the *residence time* in s_i before moving to s_{i+1}. Modalities are interpreted as in LTL, with the additional requirement that in the next operator the step is taken at a time $t \in I$, and the until is satisfied with total accumulated time in I. We denote by $[\![\varphi]\!]$ the set of timed paths that satisfy φ.

The quantitative model checking of an CTMC $\mathcal{M}$ against an MTL formula φ consists in computing the probability $\mathbb{P}^{\mathcal{M}}_s([\![\varphi]\!])$ that $\mathcal{M}$, starting from the state s, generates a timed path that satisfies φ. Then, the following pseudometric

$$\delta^{\mathcal{M}}_{\mathrm{MTL}}(s, s') = \sup\nolimits_{\varphi \in \mathrm{MTL}} |\mathbb{P}^{\mathcal{M}}_s([\![\varphi]\!]) - \mathbb{P}^{\mathcal{M}}_{s'}([\![\varphi]\!])| \tag{1}$$

guarantees that any result obtained by testing one state against an MTL formula can be reflected to the other with absolute error bounded by their distance[1].

Interestingly, we proved that the measurable sets generated by MTL formulas coincide with those generated by Deterministic Timed Automata (DTAs) specifications. Moreover, we singled out a *dense* subclass of specifications, namely that of resetting single-clock DTAs (1-RDTA), which implies that (i) the probability of satisfying any real-time specification can be approximated arbitrarily close by

[1]Clearly, the *discrete-time* case can be treated analogously by considering LTL formulas.

an 1-RDTA; (ii) the pseudometric (1) can be alternatively characterized letting range the supremum over 1-RDTA specifications only. This has practical applications in the quantitative model checking of CTMCs, since this allows one to exploit algorithms designed by Chen et al. (2011) for single-clock DTAs.

3.2 Fixed Point Characterizations

Often, behavioral distances are defined as fixed points of functional operators on pseudometrics. The first to use this technique were van Breugel and Worrell (2001), who showed the pseudometric of Desharnais et al. (2004) (see §3.1) can be defined as the least fixed point of an operator based on the Kantorovich metric.

This technique is very flexible and adapts easily in different contexts. The key observation is that functional operators can be composed to capture the different characteristics of the system. Next we show some examples from the literature.

Markov Chains. Let $\mathcal{M} = (S, L, \tau, \ell)$ be an MC. The functional operator defined by van Breugel and Worrell (2001) is as follows:

$$\mathcal{F}^{\mathcal{M}}_{\mathrm{MC}}(d)(s,s') = \max\left\{\mathcal{I}_L(\ell(s),\ell(s')), \mathcal{K}(d)(\tau(s),\tau(s'))\right\}. \tag{2}$$

Intuitively, $\mathcal{I}_L$ handles the difference in the labels, whereas the Kantorovich distance $\mathcal{K}(d)$ deals with the probabilistic choices by lifting the underlying pseudometric d over states to distributions over states. The two are combined by taking the maximum between them.

Markov Decision Processes. Let $\mathcal{M} = (S, A, \vartheta, r)$ be an MDP. Ferns et al. (2004) defined a pseudometric using the following operator, for $\lambda \in (0,1)$:

$$\mathcal{F}^{\mathcal{M}}_{\mathrm{MDP}}(d)(s,s') = \max\nolimits_{a\in A}\left\{|r(s,a) - r(s',a)| \oplus_\lambda \mathcal{K}(d)(\vartheta(s,a),\vartheta(s',a))\right\}. \tag{3}$$

The functional mixes in a convex combination the maximal differences w.r.t. the rewards associated with the choice of an action label in the current state and the difference in the probabilities of the succeeding transitions.

Segala Systems. Let $\mathcal{M} = (S, L, \theta, \ell)$ be an SS. van Breugel and Worrell (2014) extended the pseudometric on MCs using the following operator:

$$\mathcal{F}^{\mathcal{M}}_{\mathrm{SS}}(d)(s,s') = \max\left\{\mathcal{I}_L(\ell(s),\ell(s')), \mathcal{H}(\mathcal{K}(d))(\theta(s),\theta(s'))\right\}. \tag{4}$$

This functional extends (2) by handling the additional nonderminism with the Hausdorff metric on sets. In de Alfaro et al. (2007); Mio (2013) a different functional operator is considered; this is obtained from (4) by replacing the sets $\theta(s)$ and $\theta(s')$ with their convex closures.

Continuous-time Markov Chains. Let $\mathcal{M} = (S, L, \tau, \rho, \ell)$ be a CTMC. In Bacci et al. (2014), we proposed the following operator:

$$\mathcal{F}^{\mathcal{M}}_{\mathrm{CTMC}}(d)(s,s') = \max\left\{\mathcal{I}_L(\ell(s),\ell(s')), 1 \oplus_\alpha \mathcal{K}(d)(\tau(s),\tau(s'))\right\}, \tag{5}$$

where $\alpha = \|Exp(\rho(s)) - Exp(\rho(s'))\|_{\text{TV}}$ is the total variation distance between the exponential residence time distributions associated with the current states. Note that α equals the probability that the residence time in the two states differs. Therefore, the operator above can be seen as an extension of (2) that takes into consideration both the probability that the steps occur at different time moments or that the distinction can be seen later on in the probabilistic choices.

3.3 Coupling Characterizations

Given two probability distrubutions $\mu, \nu \in \mathcal{D}(X)$, a *coupling* for (μ, ν) is a joint probability distribution $\omega \in \mathcal{D}(X \times X)$ s.t. for all $E \subseteq X$, $\omega(E \times X) = \mu(E)$ and $\omega(X \times E) = \nu(E)$. Hereafter, $\Omega(\mu, \nu)$ will denote the set of couplings for (μ, ν).

Couplings have come to be used primarily for estimating the total variation distances between measures, since the following equality holds

$$\|\mu - \nu\|_{\text{TV}} = \inf \{\omega(\neq) \mid \omega \in \Omega(\mu, \nu)\} \ , \tag{6}$$

but they also work well for comparing probability distributions in general. Another notable equality is the following, a.k.a. *Kantorovich duality*

$$\mathcal{K}(d)(\mu, \nu) = \inf \left\{ \textstyle\int d \, \mathrm{d}\omega \mid \omega \in \Omega(\mu, \nu) \right\} . \tag{7}$$

Based on these equalities, behavioral pseudometrics have been given alternative characterizations in terms of couplings.

Couplings & linear time logics. In (Bacci et al. 2014), we provided an alternative characterization of (1) that works as follows

$$\delta^{\mathcal{M}}_{\text{MTL}}(s, s') = \sup\nolimits_{E \in \sigma(\text{MTL})} |\mathbb{P}^{\mathcal{M}}_{s}(E) - \mathbb{P}^{\mathcal{M}}_{s'}(E)| \tag{8}$$

$$= \inf \left\{ \omega(\not\equiv_{\text{MTL}}) \mid \omega \in \Omega(\mathbb{P}^{\mathcal{M}}_{s}, \mathbb{P}^{\mathcal{M}}_{s'}) \right\} \ , \tag{9}$$

where $\sigma(\text{MTL})$ denotes the σ-algebra generated by the sets $[\![\varphi]\!]$, for $\varphi \in \text{MTL}$, and $\equiv_{\text{MTL}}$ is the logical equivalence on timed paths. Equation (8) follows by showing that the generator is dense in $\sigma(\text{MTL})$, whereas (9) is proven generalizing (6).

Couplings & fixed points. Due to the Kantorovich duality, to the behavioral distances seen in §3.2 it can be given an alternative characterization based on a notion of *coupling structure* (varying w.r.t. the model) as the following minimum

$$\min \left\{ \gamma^{\mathcal{C}} \mid \mathcal{C} \text{ coupling structure for } \mathcal{M} \right\} . \tag{10}$$

where $\gamma^{\mathcal{C}}$ is the least fixed point of certain operators $\Gamma^{\mathcal{C}}$, that we describe below. A coupling structure $\mathcal{C}$ is said *optimal* if it achieves the minimum in (10).

Markov Chains. A coupling structure for an MC $\mathcal{M} = (S, L, \tau, \ell)$ is a tuple $\mathcal{C} = (\tau_{\mathcal{C}}, \ell)$ where $\tau_{\mathcal{C}} \colon (S \times S) \to \mathcal{D}(S \times S)$ is a probability transition function over pair of states such that, for all $s, s' \in S$, $\tau_{\mathcal{C}}(s, s') \in \Omega(\tau(s), \tau(s'))$, to be though of as a probabilistic pairing of two copies of τ.

Chen et al. (2012) showed that the distance of Desharnais et al. (2004) can be described as in (10) by means of the following operator

$$\Gamma^{\mathcal{C}}_{\text{MC}}(d)(s, s') = \max \left\{ \mathcal{I}_L(\ell(s), \ell(s')), \kappa^{\mathcal{C}}(d)(s, s') \right\}, \tag{11}$$

where $\kappa^{\mathcal{C}}(d)(s, s') = \sum_{u,v \in S} d(u, v) \cdot \tau_{\mathcal{C}}(s, s')(u, v)$. From (7), one may think of $\kappa^{\mathcal{C}}$ as the specialization of $\mathcal{K}$ on a fixed coupling structure $\mathcal{C}$.

Markov Decision Processes. A coupling structure for an MDP $\mathcal{M} = (S, A, \vartheta, r)$ is a tuple $\mathcal{C} = (\vartheta_{\mathcal{C}}, r)$ where $\vartheta_{\mathcal{C}} \colon (A \times S \times S) \to \mathcal{D}(S \times S)$ is a labelled probability transition function over pair of states such that, for all $a \in A$ and $s, s' \in S$, $\vartheta_{\mathcal{C}}(a, s, s') \in \Omega(\vartheta(a, s), \vartheta(a, s'))$. In Bacci et al. (2013a) we showed that the distance of Desharnais et al. (2004), can be described as in (10) using the following operator

$$\Gamma^{\mathcal{C}}_{\text{MDP}}(d)(s, s') = \max_{a \in A} \left\{ |r(s, a) - r(s', a)| \oplus_{\lambda} \kappa^{\mathcal{C}}_a(d)(s, s') \right\}. \tag{12}$$

where $\kappa^{\mathcal{C}}_a(d)(s, s') = \sum_{u,v \in S} d(u, v) \cdot \vartheta_{\mathcal{C}}(a, s, s')(u, v)$. Once again, (12) may be seen the specialization of (3) w.r.t the coupling structure $\mathcal{C}$.

Continuous-time Markov Chains. In Bacci et al. (2014) we extended the case of MCs in a continuous-time setting, by considering also the couplings over residence time distributions. A coupling structure for an CTMC $\mathcal{M} = (S, L, \tau, \rho, \ell)$ is a tuple $\mathcal{C} = (\tau_{\mathcal{C}}, \rho_{\mathcal{C}}, \ell)$ where $\tau_{\mathcal{C}}$ is defined as above, and $\rho_{\mathcal{C}} \colon S \times S \to \mathcal{D}(\mathbb{R}_+ \times \mathbb{R}_+)$ is such that $\rho_{\mathcal{C}}(s, s') \in \Omega(\mathit{Exp}(\rho(s)), \mathit{Exp}(\rho(s')))$, for all $s, s' \in S$.

In this case the functional operator is defined as follows

$$\Gamma^{\mathcal{C}}_{\text{CTMC}}(d)(s, s') = \max \left\{ \mathcal{I}_L(\ell(s), \ell(s')), 1 \oplus_{\beta} \kappa^{\mathcal{C}}(d)(s, s') \right\}, \tag{13}$$

where $\kappa^{\mathcal{C}}$ is defined as above and $\beta = \rho_{\mathcal{C}}(s, s')(\neq)$. Equation (6) justifies the value chosen for β, making (13) a specialization of (5) w.r.t. a coupling structure $\mathcal{C}$.

Notably, due to the characterization above and (9), in Bacci et al. (2014) we have been able to prove that the behavioral distance defined as the least fixed point of $\mathcal{F}^{\mathcal{M}}_{\text{CTMC}}$ is an upper bound of $\delta^{\mathcal{M}}_{\text{MTL}}$ in §3.1(1). In fact, this generalizes from a quantitative point of view the inclusion of probabilistic trace equivalence into probabilistic bisimilarity.

4 Computational Aspects

Historically, behavioral distances have been defined in logical terms, but effective methods for computing distances arose only after the introduction of fixed point characterizations. Indeed, one can easily *approximate* the distance from below by iteratively applying the fixed point operator, and improving the accuracy with the increased number of iterations. To this end, the operator needs to be computed efficiently. For instance, this is the case when the fixed point operator is based on the Kantorovich metric (see §3.2). Indeed, for $\mu, \nu \in \mathcal{D}(S)$ and S *finite*, the value $\mathcal{K}(d)(\mu, \nu)$ is achieved by the optimal solution of the following linear program (a.k.a. *transportation problem*)

$$TP(d)(\mu, \nu) = \arg\min_{\omega \in \Omega(\mu,\nu)} \sum_{u,v \in S} \omega(u, v) \cdot d(u, v) \tag{14}$$

where $\Omega(\mu, \nu)$ describes a transportation polytope. The above problem is in **P** and comes with efficient algorithms (Dantzig 1951; Ford and Fulkerson 1956).

In Bacci et al. (2013b), we proposed an alternative *exact* method that computes the distance of Desharnais et al. (2004) over MCs efficiently, that adopts an on-the-fly strategy to avoid an exhaustive exploration of the state space. Our technique is based on the coupling characterization seen in §3.3. Given an MC $\mathcal{M}$ and an initial coupling structure $\mathcal{C}_0$ for it, we adopt a *greedy search strategy* that moves toward an optimal coupling by updating the current one, say $\mathcal{C}_i$, as $\mathcal{C}_{i+1} = \mathcal{C}_i[(s, s')/\omega]$ by locally replacing, at some pair of states (s, s'), a coupling, which is not optimal, with the optimal solution $\omega = TP(\gamma^{\mathcal{C}_i})(\tau(s), \tau(s'))$. Each update strictly improves the current coupling (i.e., $\gamma^{\mathcal{C}_{i+1}} \sqsubset \gamma^{\mathcal{C}_i}$) and ensures a fast convergence to an optimal one.

The method is sound independently from the initial starting coupling. Moreover, since the update is local, when the goal is to compute the distance only between certain pairs, the construction of the coupling structures can be done *on-the-fly*, delimiting the exploration only on those states that are demanded during the computation. Experimental results show that our method outperforms the iterative one by orders of magnitude even when one computes the distance on all pairs of states.

In Bacci et al. (2013a) we further improved this technique in the case the input model is given as a composition of others. In summary, we identified a well behaved class of operators, called *safe*, for which is it possible to exploit the compositional structure of the system to obtain a heuristic for constructing a good initial coupling to start with the on-the-fly algorithm described above. It is worth noting that this is the first method that exploits the compositionality of the system to compute behavioral distances.

Complexity Results. Most of the theoretical complexity results about the problem of computing behavioral distances relies on fixed point characterizations.

Based on the fixed point characterization of van Breugel and Worrell (2001) (see §3.2(2)), Chen et al. (2012) showed that the bisimilarity distance of Desharnais et al. (2004) over MCs can be computed in polynomial time as the solution of a linear program that can be solved using the ellipsoid method (Schrijver 1986); this result has been later extended in Bacci et al. (2014) to CTMCs (see §3.2(5)).

Not surprisingly, the integration of non determinism on top of the probabilistic behavior, as in MDPs and Segala systems, has consequences also from a complexity perspective. Fu (2012) proved that the fixed point characterization of the bisimilarity pseudometric of de Alfaro et al. (2007) is in $\mathbf{NP} \cap \text{co-}\mathbf{NP}$. This is done by showing that the problem of deciding if a (rational) fixed point is the least one is in **P**, then he provided a nondeterministic procedure to guess (rational) fixed points. Recently, van Breugel and Worrell (2014) proved that the problem of computing the distance on Segala systems (see §3.2(4)) belongs to **PPAD**[2] by using a result by Etessami and Yannakakis (2010) that states that computing fixed points of *polynomial piecewise linear* functionals is in **PPAD**.

In the case of linear (real-)time behavioral distances (see §3.1(1)), the decidability problem is still open. However, in a recent work, we showed that the MTL (resp. LTL) distance on CTMCs (resp. MCs) is **NP**-hard to compute (Bacci et al. 2014, Corollary 23). We proved this result following arguments from Lyngsø and Pedersen (2002), who proved the **NP**-hardness of comparing hidden Markov models with respect to the L_1 norm.

References

R. Alur and T. A. Henzinger. Real-Time Logics: Complexity and Expressiveness. *Information and Computation*, 104(1):35–77, 1993.

G. Bacci, G. Bacci, K. G. Larsen, and R. Mardare. Computing Behavioral Distances, Compositionally. In *MFCS*, volume 8087 of *Lecture Notes in Computer Science*, pages 74–85, 2013a.

G. Bacci, G. Bacci, K. G. Larsen, and R. Mardare. On-the-Fly Exact Computation of Bisimilarity Distances. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 1–15, 2013b.

G. Bacci, G. Bacci, K. G. Larsen, and R. Mardare. Topologies of Stochastic Markov Models: Computational Aspects. *ArXiv e-prints*, Mar. 2014.

L. Cardelli. Brane Calculi. In *CMSB*, volume 3082 of *LNCS*, pages 257–278. Springer, 2005. ISBN 978-3-540-25375-4.

[2]The complexity class **PPAD**, which stands for *polynomial parity argument in a directed graph*, was introduced by Papadimitriou (1994) and it has received increased attention after it has been shown that finding Nash equilibria of two player games is a **PPAD**-complete problem.

L. Cardelli. On process rate semantics. *Theoretical Computer Science*, 391(3): 190–215, 2008. ISSN 0304-3975.

L. Cardelli. Two-domain DNA strand displacement. *MSCS*, 23:247–271, 4 2013. ISSN 1469-8072.

L. Cardelli and A. Csikász-Nagy. The Cell Cycle Switch Computes Approximate Majority. *Scientific Reports*, 2, 2012. doi: 10.1038/srep00656.

L. Cardelli and R. Mardare. The Measurable Space of Stochastic Processes. In *QEST*, pages 171–180, 2010.

L. Cardelli and R. Mardare. Stochastic Pi-calculus Revisited. In *ICTAC*, volume 8049 of *LNCS*, pages 1–21. Springer, 2013. ISBN 978-3-642-39717-2.

D. Chen, F. van Breugel, and J. Worrell. On the Complexity of Computing Probabilistic Bisimilarity. In *FoSSaCS*, volume 7213 of *Lecture Notes in Computer Science*, pages 437–451. Springer, 2012.

T. Chen, T. Han, J.-P. Katoen, and A. Mereacre. Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications. *Logical Methods in Computer Science*, 7(1), 2011.

G. B. Dantzig. Application of the Simplex method to a transportation problem. In *Activity analysis of production and allocation*, pages 359–373. Wiley, 1951.

L. de Alfaro, R. Majumdar, V. Raman, and M. Stoelinga. Game Relations and Metrics. In *LICS*, pages 99–108, July 2007.

J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangaden. Metrics for labelled Markov processes. *Theoretical Computer Science*, 318(3):323–354, 2004.

K. Etessami and M. Yannakakis. On the Complexity of Nash Equilibria and Other Fixed Points. *SIAM Journal on Computing*, 39(6):2531–2597, 2010.

N. Ferns, P. Panangaden, and D. Precup. Metrics for finite Markov Decision Processes. In *UAI*, pages 162–169. AUAI Press, 2004. ISBN 0-9749039-0-6.

N. Ferns, D. Precup, and S. Knight. Bisimulation for Markov Decision Processes through Families of Functional Expressions. In *Horizons of the Mind. A Tribute to Prakash Panangaden*, volume 8464 of *LNCS*, pages 319–342, 2014.

L. R. Ford and D. R. Fulkerson. Solving the Transportation Problem. *Management Science*, 3(1):24–32, 1956.

H. Fu. Computing Game Metrics on Markov Decision Processes. In A. Czumaj, K. Mehlhorn, A. Pitts, and R. Wattenhofer, editors, *Automata, Languages, and Programming*, volume 7392 of *Lecture Notes in Computer Science*, pages 227–238. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31584-8.

A. Giacalone, C.-C. Jou, and S. A. Smolka. Algebraic Reasoning for Probabilistic Concurrent Games. In *IFIP WG 2.2/2.3*, pages 443–458, 1990.

D. Kozen. A Probabilistic PDL. *J. Comput. Syst. Sci.*, 30(2):162–178, 1985.

K. G. Larsen and A. Skou. Bisimulation Through Probabilistic Testing. In *POPL*, pages 344–352, 1989.

R. B. Lyngsø and C. N. Pedersen. The consensus string problem and the complexity of comparing hidden Markov models. *Journal of Computer and System Sciences*, 65(3):545–569, 2002.

R. Mardare, L. Cardelli, and K. G. Larsen. Continuous Markovian Logics - axiomatization and quantified metatheory. *LMCS*, 8(19):247–271, 2012.

M. Mio. Upper-Expectation Bisimilarity and Real-valued Modal Logics. *CoRR*, abs/1310.0768, 2013.

C. H. Papadimitriou. On the Complexity of the Parity Argument and Other Inefficient Proofs of Existence. *J. Comput. Syst. Sci.*, 48(3):498–532, 1994.

A. Regev, E. M. Panina, W. Silverman, L. Cardelli, and E. Shapiro. BioAmbients: an abstraction for biological compartments. *TCS*, 325(1):141–167, 2004.

A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986. ISBN 0-471-90854-1.

F. van Breugel and J. Worrell. Towards Quantitative Verification of Probabilistic Transition Systems. In *ICALP*, volume 2076 of *LNCS*, pages 421–432, 2001.

F. van Breugel and J. Worrell. The Complexity of Computing a Bisimilarity Pseudometric on Probabilistic Automata. In *Horizons of the Mind. A Tribute to Prakash Panangaden*, volume 8464 of *LNCS*, pages 191–213, 2014.

The Challenges of Attaining Grace (in a Language Definition)

Andrew Black
Portland State University

Kim Bruce
Pomona College

Michael Homer James Noble
Victoria University of Wellington

Abstract

Grace is a new object-oriented language, designed for teaching object-oriented programming to novices. Grace is based on a small number of syntactically and semantically simple constructs that collaborate to provide a flexible and expressive language. Core features include nested scope, generative object constructors, and first-class anonymous functions; classes and most control structures are syntactic sugar. On this core we have been able to build pattern matching, a simple but effective module system, and the infrastructure for supporting pedagogical dialects.

Grace features a *structural* type system that clearly separates interface from implementation. The type system is also *gradual*, allowing programmers to mix static and dynamic typing. These features, which we were able to integrate into the language fairly easily, allow instructors to introduce types as early or late as they see fit.

Surprisingly to us, the more challenging parts of the language design turned out to be those related to inheritance from objects, and object initialization in the presence of inheritance. The challenge was to use generative object constructors to support Java-like class-based inheritance semantics. We thought that these semantics were desirable because students who learn Grace as their first language are likely to transition to Java.

Abadi and Cardelli's foundational book *A Theory of Objects* (1996) sets out the theoretical underpinnings for object-oriented languages, with special emphasis on object-based languages. In their model, as in ours, objects are primitive, and classes are derived from objects. This note discusses the challenges in defining object inheritance and initialization so that the semantics of inheritance and initialization are similar to Java's. We discuss how our work extends that of Abadi and Cardelli.

1 Introduction

Grace is a programming language targeted at novice programmers, and embraces the following high-level goals (Black et al. 2013):

- to integrate proven new ideas in programming languages into a simple object-oriented language;
- to represent the key concepts underlying object-oriented programming *gracefully* — in a way that can be easily explained; and
- to allow students to focus on the essential, rather than the accidental, difficulties of programming, problem solving and system modeling.

Grace is still under development; we plan to begin testing Grace in teaching in the fall of 2014.

The Grace language, which is described in Section 2, is object-based (everything is an object) and supports nested scopes and anonymous first-class functions (which are, consequently, objects). Classes are definable from these more primitive concepts; as discussed in Section refClasses, we include syntactic sugar for classes to make Grace more like other object-oriented languages, and to reduce the syntactic overhead of nesting.

This design makes it remarkably simple to add pattern matching (Homer et al. 2012), a simple module system (Homer et al. 2013), and a system for building pedagogical dialects (Homer et al. 2014), similar to "language levels" in DrScheme (Findler et al. 2002).

Grace includes a structural type system, like Modula-3 (Cardelli et al. 1995) and OCAML (Rémy 2002), rather than a nominal type system, like Java and C#. Type annotations on declarations are optional; we call this *gradual typing*. Gradual typing allows programmers to combine statically and dynamically typed code in a single program. This has pedagogical advantages, because instructors can start teaching using Grace with dynamic typing, and then gradually move students into a statically typed dialect of Grace as the deficiencies of dynamic typing become apparent. This limits the number of new ideas that students must master as they write their first programs.

We have found that these features fit together nicely, and that the resulting language is both simple and pleasant to use. A prototype implementation, *minigrace*, is written in Grace and currently generates either C or JavaScript[1].

In spite of these successes, we did run into some difficulties in the Grace design — difficulties that showed up in sometimes-unexpected places. The core of this paper, found in Section 4, is a discussion of the difficulties related to inheritance from objects,

[1]See http://gracelang.org/applications/minigrace/ for information on how to install and use the compiler. The JavaScript version runs on the web, avoiding the need to download and install any software.

and object initialization. Initialization is routine when inheriting from classes, but as we will see, it is tricky when inheriting objects. We discuss the goals that led to these difficulties, alternative designs, and the present resolution.

2 A Brief Introduction to Grace

This section provides a synopsis of the elements of Grace that are most relevant to the issues addressed in this paper. A more complete description of Grace as a teaching language is available in a paper presented at SIGCSE (Black et al. 2013).

2.1 Objects

A Grace object is self-contained, with a unique identity and its own methods and fields — both constants (**def**s) and variables (**var**s). The outside world (other objects) can interact with an object only through *method requests* (our term for "message sends" or "method calls").

From the beginning, we envisaged that Grace objects would be created by *generative object constructors*: expressions that constructs new objects, whose methods and fields are given in body of the constructor. Emerald's object constructors are generative (Raj et al. 1991), as are OCAML objects (Rémy 2002) and JavaScript object literals. In Grace we can write:

```
object {
    var size := 1
    def threshold = curThreshold
    method grow(n) {
        size := size + n
    }
    print "a new object has been created"
}
```

Evaluating the same generative object constructor multiple times *generates* multiple new objects, with separate identities but the same structure, and potentially different field values. In the above example, the per-instance constant threshold will be defined as the current value of curThreshold, which is presumably defined in an outer scope. This *generative* property differentiates Grace's object constructors from the static object literals of languages like Scala (Odersky 2011) and Self (Ungar and Smith 1991), which are evaluated only once and so create only one object.

Note that Grace object constructors can contain executable code, which may in general have effects. In this example, print has an obvious effect, but curThreshold might also have an effect, if curThreshold is a method. Object constructors are expressions, and may be nested inside other expressions, and thus inside methods.

2.2 Method Requests

Method requests, which we often refer to simply as *requests*, are the basic computational mechanism in Grace. As in Smalltalk, control structures, function application, field access, and primitive operations are all invoked using requests. Method requests are written using the "dot notation" o.m(...), as in many other object-oriented languages. However, method requests can also be written without an explicit receiver, as m(...) (or just m if there are no parameters). The object receiving such a request is determined by the lexical scope. If a method with the same name is available in the most closely enclosing **object**, then the receiver is **self**. Otherwise, the receiver will be found by inserting a sequence of **outer**s to access a method with a matching name in some surrounding **object**.

2.3 Classes

Classes are useful when one wishes to define multiple objects with the same structure. Grace's class syntax combines an object constructor with a method definition:

```
class pointAtx(x')y(y') {
    def x = x'
    def y = y'
    method distanceFromOrigin { ((x^2) + (y^2)).sqrt }
}
```

This defines a method, which can be used to create an object using a method request: pointAtx(5)y(10). We will discuss the meaning of classes in more detail in Section 3.

2.4 Other features

Other components of Grace that are less important to the topic of this paper, but which provide significant expressive power, include the following.

1. A *block* represents an anonymous function, also known as a λ-expression. For example, {x −> x +1} represents the successor function. Blocks can be used to define control structures like *for*...*do* that can be used just like built-in constructs.

2. Methods can have *multi-part names*, where the parts as separated by parameter lists. We have already seen this in the definition of the class pointAtx()y(). Similarly, the method that updates an element in a list has the name at()put(), and is defined like this:

```
method at(n)put(x) {
    boundsCheck(n)
    inner.at(n−1)put(x)
```

```
      self
   }
```

3. *Gradual typing:* Grace can be used as a statically or dynamically typed language, or with a mix. Type annotations that the programmer chooses to insert will be checked, either at compile time or at run time. All actual type errors are caught by the run-time system, whether or not type information is provided by the programmer.

4. *Pattern matching* is supported by match()case()...case() statements. Matches can be based on the identity, type (that is, interface), or other features of the match argument. Basic pattern matching is built into the language, but refined matching criteria can be defined by programmers.

3 Objects or Classes: which should be Primitive?

While one could design a language in which both objects and classes are primitive, the strong connection between the two suggests that only one is necessary. Which should one choose?

3.1 Modeling Object Constructors with Classes

It is quite possible to have syntax for both objects and classes, but to define objects in terms of classes. Java, for example, does not have generative object constructors, and insists that every object is created from a class. However, its anonymous classes can be used to achieve the same effect as object constructors:

```
new Object() {
   final int x = _x;
   final int y = _y;
   float distanceFromOrigin { return sqrt((x^2) + (y^2)) };
}
```

This expression creates a new object with the given fields and method, which inherits from class Object; it assumes that _x and _y are defined in an outer scope. Ruby calls this the "eigenclass" model (Perrotta 2010).

3.2 Modeling Classes with Methods

From a pedagogical point of view, it seems strange to make class the primitive: since classes are used to produce objects, shouldn't objects come first? In particular, it is often the case that in the first few weeks of a course for novices, most classes are used to generate only a single object. It would be simpler to first define objects, and then introduce classes later as a way to generate multiple objects with similar structure. This reasoning led us to ask how we could model class-based systems using objects.

We have explored two ways of representing classes in Grace. Originally, we represented classes as objects; more recently we have moved toward a simpler representation of classes as methods, as we now explain.

The definition of **class** pointAtx(x')y(y') in Section 2.3 is equivalent to:

```
method pointAtx(x')y(y') {
   object {
      def x = x'
      def y = y'
      method distanceFromOrigin { ((x^2) + (y^2)).sqrt }
   }
}
```

This definition lets us construct a new point at (5, 10) by writing pointAtx(5)y(10).

Because classes define methods, it is easy to build factory objects that provide multiple ways of constructing objects. For example, if a programmer wanted to be able to create points by giving either cartesian or polar coordinates, they could define:

```
def point = object {
   class x(x')y(y') {
      method x { x' }
      method y { y' }
      method distanceFromOrigin { ((x^2) + (y^2)).sqrt }
   }
   method r(r)theta(theta) {
       x (r * cos(theta)) y (r * sin(theta))
   }
}
```

They could then create point objects using requests like point.x(3)y(4) or point.r(5)theta (π/6), which both generate objects with similar (cartesian) representations.

3.3 What about other Roles of Classes?

Class-based inheritance and object-based delegation are often considered roughly equivalent in power (Lieberman et al. 1987). Classes can model object constructors, although not necessarily other features of prototype-based languages, such as delegation (Ungar and Smith 1991). And we have just seen how, by repeatedly executing an object constructor, objects and methods can subsume the object-generation aspect of a class.

However, in many languages classes do not just create new objects: they play other roles as well. For example, Borning (1986) lists seven additional roles for classes in Smalltalk, and in languages like Java and C++, classes also function as types. The design of Grace does not conflate classes and types: types in Grace are like Java interfaces rather than classes. Hence, we don't have to worry about supporting this aspect of classes from traditional object-oriented languages.

A key role for classes, however, is as "a means for implementing differential programming (this new object is like some other one, with the following differences. . .)" (Borning 1986). In particular, we wanted to be able to support a notion of inheritance for classes. So we were forced to ask ourselves how we could model class-based inheritance with the sharing mechanisms of object-based languages.

3.3.1 The Traditional Semantics of Objects and Classes

Semanticists have traditionally modeled objects as fixed points and classes as the *generators* of fixed points (Cook 1989; Bruce 2002). The intuition as follows. In an object, the meaning of **self** is the object that contains it: an object is a fixed point in which "the knot has been tied" and **self** refers to the object itself. A class, in contrast, has an *implicit* **self** parameter (it is explicit in the semantics, but not in the syntax): "the knot has not yet been tied". This allows the class to be used both to generate multiple objects, each with its own **self**, and in inheritance, because **self** in the superclass can *later* be given the correct meaning, which is the final (extended) object.

Abadi and Cardelli (1996) illustrate this by defining a class as an object containing a collection of pre-methods — methods where **self** has been abstracted out as a parameter. Essentially, objects are constructed by taking a fixed point with respect to **self**. Because classes contain the *pre*-methods, when a subclass is defined the pre-methods can be altered by adding new methods or overriding existing ones. When new objects are defined from the new subclass, the fixed point is taken with respect to the new or overridden features. As a result, the meaning of **self**, even in inherited methods, is the new object, which includes the new and overriding methods introduced in the subclass.

3.3.2 Inheritance from Objects

So how can we inherit from an object? If the object has only the methods obtained *after* taking the fixed point, then we cannot get the behavior we want, because **self** refers to the wrong object. Thus, it seems that objects will have to contain the *pre*-methods. It is still straightforward to execute methods — just pass in the object to be bound to the **self** parameter as an extra argument before executing the code. This is, of course, exactly what most implementations of object-oriented languages do: they share the pre-methods given in the class definition amongst all of the instances of the class, and bind **self** to the receiver when a method request is executed.

So much for methods. But what about an object's fields — its constants and variables? These must be initialized. And there lies the difficulty.

4 Initializing Objects

Initializing objects is more complex than one might expect. Key issues involve the order of initialization and the meaning of **self** during initialization. In Grace, initialization entails executing the code in the body of the object constructor; there is nothing analogous to Java's "constructor" (which behaves like an initialization method).

Rather than being absolutely precise in our definitions below, we provide sufficient intuition behind the operations to illustrate the issues involved. We start by temporarily ignoring inheritance and looking just at the effect of evaluating an object constructor[2].

When an object constructor is evaluated, the first thing that happens is that space is allocated for all definitions and fields, and their values are set to the special value uninitialized. Second, the closures for methods are made available to the object: because methods are closures, references to definitions or variables in methods are not evaluated at this time.

4.1 A Brief Detour: Representing Methods

While there are many ways in which we might make methods available to an object, we will pick a particular representation here that will allow us to make our discussion more concrete. Rather than having a separate slot in the object for each method, we will choose to represent all the methods together as closures in another structure. Thus, each object o will maintain a reference to a structure M_o that contains all of the closures corresponding to its methods. Moreover, each closure will not only have a parameter for each of the explicit parameters in the method, but will also have a **self** parameter.

We will *not* here assume that the method closures are shared between objects. While a reasonable implementation might well share the *code* of the methods, that is beyond the scope of the present discussion, in which we focus on an explanation of the semantics of inheritance.

When a method request of the form obj.m(e) is evaluated, the system will obtain the closure corresponding to m from M_{obj}. It will then provide the arguments obj and e to the closure, and evaluate the closure.

4.2 Back to Initializing Objects

After the space for the object has been allocated and the methods installed, the identifier **self** is set to refer to this new object. Finally, all the code in the object constructor is evaluated, from top to bottom. By "all the code" we mean statements at the top-level of the object constructor, and initialization expressions for fields. If, during this evaluation — or indeed at any time during the execution of the program — an uninitialized

[2]In Grace, everything inherits at least from the top-level Object; we return to this later.

field is requested, the program will raise a runtime error. Thus, it is the programmer's responsibility to ensure that the code is arranged so that no field is dereferenced before it is initialized.

There are no restrictions on the kind of code that can be executed during initialization: the code may comprise arbitrary method requests, including requests on **self** that access fields that may not yet be initialized. Rather than complicating the core semantics to manage these "self-inflicted" methods, the semantics of Grace simply raises an exception, although dialects may include more restrictive static checks. In contrast, OCAML bans self-inflicted methods altogether during initialization and executes initializers in the scope surrounding the object constructor.

Once completed, the result of evaluating the object constructor is a reference to the newly created object. However, notice that the initialization code might also expose a reference to **self** *before* initialization is complete. For example, the initialization could involve a method request in which **self** is passed as an argument — what Gil and Shragai (2009) call "immodesty". This possibility leads to difficulties because of interactions between inheritance and initialization.

4.3 Inheritance from Classes

Let us turn our attention to inheritance from classes; we defer to Section 4.4 the trickier case of inheriting from objects. Consider the class declaration

```
class d(...) {
   inherits c(...)
   ...
}
```

What happens when we create a new object by evaluating the expression d(...)?

First, as before, space for the new object $dobj$ is allocated. This will include slots for all of the fields, both those inherited from c and those newly defined in d. As before, all these fields are initialised to undefined.

The second step, installing the methods, is more complicated because of inheritance and **super**. The structure containing the closures that represent the methods of $dobj$ is prepared as follows:[3]

1. Allocate the method structure M_{dobj} with slots for both inherited and newly defined methods. As before, all the methods have an explicit **self** parameter.

2. Let M_c refer to the method structure of c. Copy the methods from M_c to the corresponding method slots in M_{dobj}. Next, create and install the closures corresponding to the methods defined in d. Newly introduced methods go into empty slots, while overriding methods replace those from M_c.

[3]We are being very prescriptive of how methods are represented here. Actual implementations will likely differ from this description, though the semantics should be the same.

3. When creating the closures for the methods in d, if a method includes a request of the form **super**.q(...), it is compiled into a statically bound call to the closure corresponding to q in M_c. In addition to the explicit arguments of q, a reference to $dobj$ is passed as the **self** argument in this call.

As before, the final steps of object construction are to install a reference to M_{dobj} in the newly allocated object $dobj$, and set **self** to refer to $dobj$.

To initialize the definitions and variables in the new object, first run the initialization code from c (recursively executing its superclasses' code). In executing this code, the meaning of **self** is the new object $dobj$. Finally, run the initialization code in the object d.

Given the above description, we can see that if the initialization code in c makes a **self**-request for a method that was overridden by d, the overriding method will be executed. In a similar situation in C++, the initialization code from a superclass would always run the method in the superclass. In other words, in C++ the initialization code is run in the original object. Java, in contrast, has a semantics more like that described above, and executes the overriding method.

Why the difference? Meyers (2005) explains that the semantics chosen by C++ will avoid accidental references to uninitialized variables that would occur if overriding code were to refer to the fields of the new object, which have not yet been initialized (though Gil and Shragai (2009) discuss issues with this semantics). Unfortunately, using the (overridden) superclass method also means that any new actions taken by the overriding method will not occur during initialization. For example, if the overriding method logs updates and then requests the super-method, then actions taken during initialization will not be logged.

As discussed in Section 4.2, "immodest" initialization code can expose a reference to the new object. For example, a graphic object may register itself with a display manager so that it can be redrawn as necessary. If this initialization is inherited by a button object, the display manager might request a method on the button (e.g., requesting that it draw itself), before the button is fully initialized.

Because of such problems, researchers and practitioners (Gil and Shragai 2009) have argued for "safe object construction" techniques that restrict or ban both self-infliction and immodesty, especially in the presence of concurrency (Goetz 2002). Qi and Myers (2009) have proposed using type-state to solve object initialization problems in Java.

What is the "right" solution for Grace? After weighing the alternatives, we decided to follow Java, and allow overriding of methods requested during initialization. In part, this is because students need to learn about the dangers of such code, and having them fall into this hole is a good way to create a "teachable moment". But we also realized that Grace is safe here in a way Java isn't. While a Java program can observe a final field going from null (a legal value) to another legal value, a Grace program attempting to access an uninitialized field immediately raises an exception. Moreover, Grace dialects can be used to restrict the Grace language; a dialect could implement a check that allows known safe immodest initialization schemes, while forbidding potentially dangerous ones. This would not be possible if immodesty were banned in the base language.

While we have framed this discussion to explain initialization of objects created using the class syntax, this is unimportant. The description works for all objects that *inherit* from a class, whether or not the new object is created using a class or using an object constructor. The tricky case, discussed below, is when the new object inherits from an object rather than from a class.

4.4 Inheriting from an Object

While Abadi and Cardelli (1996) extensively discuss inheritance from objects in chapter 4, their formal model of inheritance is limited to pre-method reuse, illustrated through their modeling of classes as collections of pre-methods, along with a *new* method to generate new objects. They then discuss how inheritance of classes can be modeled in their object calculus with this design. In the informal discussion of object inheritance and delegation, they mention that it is important that parent objects and classes be statically visible (pp. 40–41), a restriction that we adopt. However, detailed formal models of object inheritance are not provided.

We considered adopting Abadi and Cardelli's approach of defining inheritance only from classes (as does OCAML), but this seemed overly restrictive, and made objects into second-class citizens. In this section, we explore why inheritance from objects can be tricky, and present our solution.

The difficulty with inheriting from existing objects (as opposed to constructor expressions that create new objects) is that *there is no initialization code* to run. While methods are not a problem, how should we initialize the fields of the new object? Our initial thought was to copy the values of the fields of the superobject. But how do we copy them? While a shallow copy can sometimes be the correct solution, too often it is not. For example, suppose an object has a reference to a log object that collects information about its activity. If an inheriting object were simply given a reference to the same log, then activities by the subobject and the superobject would both update the same log, which is probably not the desired behavior. Constructing a new log object requires access to the code that initialized the superobject.

One possible solution is to turn the initialization code for every object into a closure, and have the object carry it around for its whole lifetime, just in case that object is inherited. This seems unattractive, and might have surprising implications for memory usage. Languages like Smalltalk separate object construction from initialization, and make initialization a real method, but that does not work for Grace because **def**s must be given their values when they are declared, not later.

The lack of an obviously good answer to the initialization question led us to restrict the use of inheritance to *fresh* objects: those returned from object constructors and classes. Also permitted are invocations of clone methods, as well as invocations of more general methods that do some work and then return a fresh object.

These restrictions on inheritance are intended to ensure that the initialization code for the superobject is always accessible at the place where it is inherited. They have the side-effect of ensuring that the structure of the object being inherited is statically known. This turns out to be quite useful. Suppose, for example, that a class c has a confidential field f. (Confidential fields are accessible to an object and its subobjects, but are not visible outside.) Then, if d is a subclass of c, the methods defined in d may access f. If we did not have full information about c's structure — including those components that are not externally visible — we might not know how to interpret references to f. One of the consequences of this is that if method m has a parameter p with a method new that creates a new object, then inside m we may construct new objects by requesting p.new, but we may not construct objects that *inherit* from p.new.

4.5 Evaluation

For the most part we are satisfied with our solution to the difficulties involved in inheriting from objects. However, there is one annoying case that is not covered by our current solution, which we would like to fix.

There are many situations where objects are immutable and independent of the surrounding scope. For example, an object may be composed only of methods, or it may contain methods and definitions, but the values of those definitions may also be immutable. Under these circumstances, it is annoying to have to create a (parameterless) class, all of whose instances will be (and will always remain) identical, just so that one can inherit from the class rather than from the immutable object itself.

5 Summary

Grace is an object-oriented language designed for teaching that provides generative object constructors, classes, and first-class functions. Grace makes object constructors, not classes, primary, because they are simpler and more concrete, and so (we hope) easier for novices to understand. While classes as object factories were easily definable from object constructors, it was more challenging to define inheritance from objects, because of the need to initialize the subobject's fields based on the superobject. To avoid initializing those fields via a default clone operation that would frequently provide the wrong result, we instead restrict inheritance to freshly created objects.

References

M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996. ISBN 978-0387947754.

A. P. Black, K. B. Bruce, M. Homer, J. Noble, A. Ruskin, and R. Yannow. Seeking Grace: A new object-oriented language for novices. In *Proc. 44th ACM Technical Symp. on Computer Science Education*, SIGCSE '13, pages 129–134, 2013.

A. Borning. Classes versus prototypes in object-oriented languages. In *ACM/IEE Fall Joint Computer Conf.*, pages 36–40, 1986.

K. B. Bruce. *Foundations of Object-oriented Languages: Types and Semantics*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-02523-X.

L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 reference manual. Research Report 53, DEC Systems Research Center, 1995.

W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, Providence, RI, USA, 1989.

R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: a programming environment for Scheme. *J. Funct. Program.*, 12(2):159–182, 2002.

J. Y. Gil and T. Shragai. Are we ready for a safer construction environment? In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, number 5653 in LNCS, pages 495–519, Berlin, Heidelberg, 2009. Springer-Verlag. URL http://dx.doi.org/10.1007/978-3-642-03013-0_23.

B. Goetz. Java theory and practice: Safe construction techniques. IBM developerWorks, June 2002. URL https://www.ibm.com/developerworks/java/library/j-jtp0618/.

M. Homer, J. Noble, K. B. Bruce, A. P. Black, and D. J. Pearce. Patterns as objects in Grace. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 17–28, New York, NY, USA, 2012. ACM.

M. Homer, K. B. Bruce, J. Noble, and A. P. Black. Modules as gradually-typed objects. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, DYLA '13, pages 1:1–1:8. ACM, 2013.

M. Homer, J. Noble, K. B. Bruce, and A. P. Black. Graceful dialects. In R. Jones, editor, *ECOOP 2014 — Object-Oriented Programming, 28th European Conference*, volume 8586 of *LNCS*, pages 131–156, Uppsala, Sweden, July 2014. Springer-Verlag. URL http://link.springer.com/chapter/10.1007/978-3-662-44202-9_6.

H. Lieberman, L. Stein, and D. Ungar. Treaty of Orlando. *SIGPLAN Not.*, 23(5): 43–44, Jan. 1987.

S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005. ISBN 0321334876.

M. Odersky. The Scala language specification. Technical report, Programming Methods Laboratory, EPFL, 2011.

P. Perrotta. *Metaprogramming Ruby*. Pragmatic Bookshelf, 2010.

X. Qi and A. C. Myers. Masked types for sound object initialization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 53–65, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480890. URL http://doi.acm.org/10.1145/1480881.1480890.

R. K. Raj, E. D. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A general purpose programming language. *SP&E*, 21(1):91–118, 1991.

D. Rémy. Using, Understanding, and Unraveling the OCaml Language. In G. Barthe, editor, *Applied Semantics. Advanced Lectures. LNCS 2395.*, pages 413–537. Springer Verlag, 2002.

D. Ungar and R. B. Smith. SELF: the Power of Simplicity. *Lisp and Symbolic Computation*, 4(3), June 1991.

The gene gate model: some afterthoughts

Ralf Blossey
Interdisciplinary Research Institute USR3078 CNRS
Parc de la Haute Borne, 50 Avenue Halley
59658 Villeneuve d'Ascq, France

Abstract

In 2006, Cardelli, Blossey and Phillips proposed a simple computational model for the dynamics of transcriptional regulation using stochastic π-calculus. Here I show that, to some extent, our path of understanding the properties of this model has run backwards... a recently obtained exact solution of the gene gate model for a self-regulatory gene allows to elucidate some very basic properties which would have been nice to know already in 2006. In this paper I explain the properties of the gene gate model of the self-regulatory gene in the context of the existing literature. Our current understanding of this model - which by all means appears to be the simplest way to model feedback interactions - helps to gain insight into what a computational theory of gene regulation should be able to provide.

The gene gate model. In 2006, Luca Cardelli together with Andrew Phillips and Ralf Blossey proposed a simple model of transcriptional regulation, implemented in stochastic π-calculus (Blossey R, Cardelli L, Phillips A (2006)), with the ambition to be able to compute the dynamic behaviour of 'small' gene networks. The main idea for the description of a gene is to represent it by two states G and G', whereby the gene in state G transcribes protein constitutively at a rate ε,

$$G \to_{\varepsilon} G + A\,. \tag{1}$$

It enters into the state G' through the interaction with a transcription factor, for simplicity in this example let us take its own product (this construct then corresponds to an autoregulated gene):

$$G + A \to_{r} G' + A. \tag{2}$$

Protein A interacts with G and is released after the transition from G to G'. In state G', the gene can now either be repressed, and hence not transcribing at all after which the gene relaxes from state G' back to the active state G via

$$G' \rightarrow_{\eta} G, \tag{3}$$

or G' can be understood as an activated state, and hence transcribing at a higher rate,

$$G' \rightarrow_{\eta} G + A \tag{4}$$

with $\eta > \varepsilon$, and at the same time the gene relaxes back from the highly active state G' to the constitutively active state G. (This is the most concise and simple version, one can separate it in two steps, $G' \rightarrow G' + A$ and $G' \rightarrow G$, but at the cost of adding another parameter to the model.) It is of course noteworthy that within this model we entirely neglected the level of mRNA. This biologically crucial process can, however, be added when needed, so that this omission is not a critical factor.

This model, which the authors called the 'gene gate model', has some peculiarities when compared to other models. First of all, it is regulated, in contrast to probably its major known forefather, the Markov model by Peccoud and Ycart, published a long time ago already in the journal *Theoretical Population Biology* (Peccoud J, Ycart B (1995)). Like the gene gate model, the Peccoud-Ycart model consists of a gene which can be in either of two states, G and G' (we thus inherited this gene), but the PY-gene is not regulated: a change in its state from active to non-active occurs purely at random, without the intervention of a protein A, as in equation (2) above.

The second peculiarity of the gene gate model is that the model is constitutively active, hence it transcribes without the interaction with a transcription factor. And, ultimately, the final peculiarity is that there is no transcription factor-DNA complex in the model, see again reaction (2). This feature has met with some criticism by readers and referees, who sometimes considered this step 'unphysical'. In our view, this is in fact not the only unphysical feature... (see the neglect of mRNA) and the critique is not pertinent. The gene gate model is obviously a caricature of the real process, which involves numerous complex molecules, of which only some are represented explicitly in the model, while others reside somehow in the reaction rates. Just take the RNA polymerase: where is it? It hides in the rate ε, and in η in the activated case. Our motivation for this model was to build something which was a 'minimal' model, which can, however, allow to build complex networks - as those illustrated in Figure 1, something which the Peccoud-Ycart model obviously does not allow. We believed we pushed this modeling to the extreme, while still being rather faithful to reality in this sketchy sense.

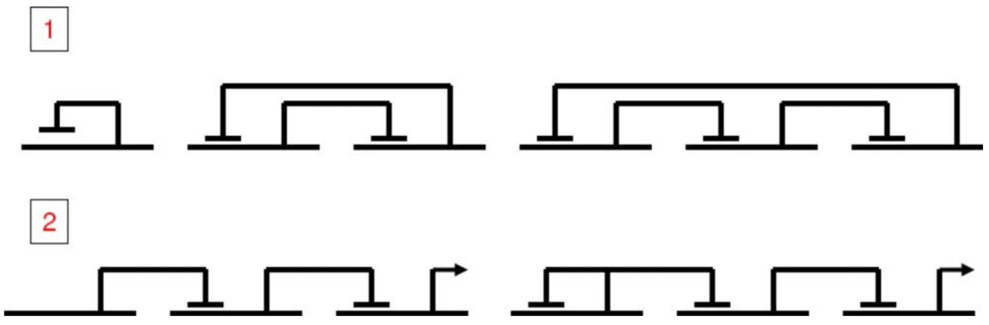


Figure 1: The two main classes of simple gene circuits: Circular (1) and linear (2). Shown are only the repressive circuits; activatory circuits and mixtures of both types can be built in a similar fashion. Circuits shown in (1): The self-repressed circuit, a bistable switch, the repressilator. Circuits shown in (2): A linear array and a linear array with a head feedback, hence a mixture of a circular and a linear circuit. Reprinted with permission from (Blossey R, Giuraniuc V C (2008)). Copyright by the American Physical Society.

The repressilator. In order to build complex networks with this model, we employed stochastic π-calculus, and studied numerous simple networks composed of one, two and three genes (Blossey R, Cardelli L, Phillips A (2006, 2007); Blossey R, Giuraniuc V C (2008)). Within stochastic π-calculus, each gene is represented by a process; as we have two types of genes, there are generically two of them: the repressed gene with process neg(·,·), and the activated gene with process pos(·,·). I do not repeat the definition of these processes here, and refer the interested reader to our papers (Blossey R, Cardelli L, Phillips A (2006, 2007)). A peculiar circuit we studied is the repressilator (Elowitz M B, Leibler S (2000)), a three-gene circuit in which in genes repress each other cyclically, using $\perp$ as the sign of repression (see Figure 1 for the circuit topology)

$$c \perp a \perp b \perp c \tag{5}$$

hence it is given, within stochastic π-calculus, by the beautiful 'compositional' process

$$neg(a, b) \mid neg(b, c) \mid neg(c, a) \tag{6}$$

The repressilator circuit oscillates, see Figure 2. We were quite happy with this result, as the stochastic repressilator computed with the Gillespie algorithm does exactly what the deterministic version does - and, of course, the real one

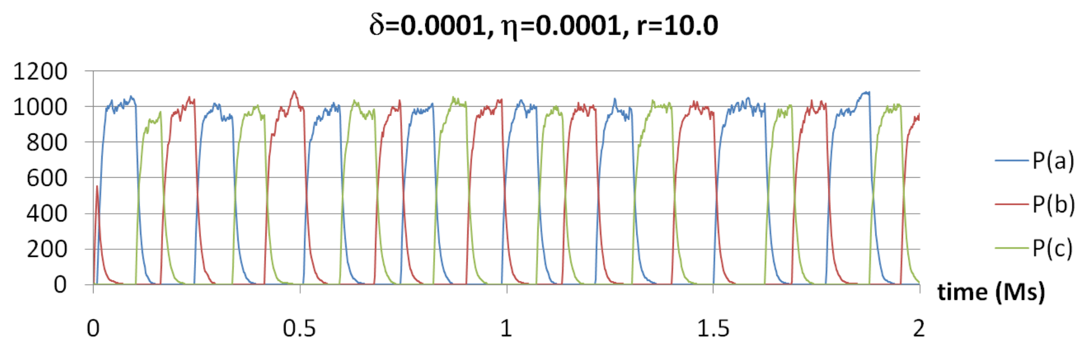


Figure 2: Repressilator oscillations. Shown are protein numbers as a function of time, for a specific choice of parameters, as indicated in the text. Note that δ is the degradation rate of the proteins, corresponding to the reaction $A \rightarrow_\delta 0$. Reprinted with permission from (Blossey R, Cardelli L, Phillips A (2007)).

built in *E. coli* bacteria by Elowitz and Leibler.

Only that it is not at all clear, why.

When we did our work, we were not aware of this problem. Only later did it turn up for us that there is indeed a real issue here. Here it is.

In the deterministic setting, the interaction between the transcription factor proteins and the DNA **must** be cooperative, hence described by a Hill coefficient $h > 1$. In our deterministic version of the repressilator, it actually depends on the model details we include: if we keep the gene states of the three genes as variables, we find a condition $h > 4/3$, while when we ignore the gene states and keep on the gene products, the proteins, we need to have $h > 2$ (Blossey R, Giuraniuc V C (2008)). These values of h are not easily interpreted in physical terms, but one can think of a reaction like

$$A + A + A \rightarrow_{r_a} A^3 \qquad (7)$$

i.e., the formation of a trimer to be on the safe side. Our stochastic π-calculus gene gate repressilator however oscillates with only a monomer protein. [1]

[1]There is also the subtle issue of the interpretation of the number of genes as a continuous variable in our model, see (Blossey R, Cardelli L, Phillips A (2007)). The simplest rationalization is to assume that, when talking about the continuous model, one thinks of its application to an ensemble of synchronized cells which however is not so easy to realize experimentally.

Shortly after our work, Biham *et al.* showed that the stochastic bistable two-gene circuit, in our notation

$$neg(a, b) \mid neg(b, a) \tag{8}$$

switches stochastically between two states without cooperative effects (Lipshtat A, Loinger A, Balaban N Q, Biham O (2006)); they then also showed oscillations it for the repressilator, however for a more complex model (Loinger A, Biham O (2007)).

Here is why the minimal 'gene gate' model is perfectly sufficient to produce oscillations in the stochastic setting. We only understood it recently from an exact solution of the auto-regulated gene, when represented by its Master equations for the gene states G and G', much in the same way as Peccoud and Ycart did it for their model; in fact, the solution is only moderately more complicated to obtain, but physically totally different (Vandecan Y, Blossey R (2012)).

Figure 3 shows the probability distribution of protein A of the auto-regulated gene for a specific range of parameters. As one can see, the distribution is 'bimodal'; actually, we called its shape as being a 'boundary bimodal' since one of the peaks is **always** located at the number of zero proteins. If one wants to have a maximum at a finite small number of proteins, then one is required to take into account the transcription-factor-DNA complex as a state in the model. (That is all it needs, in fact.)

Why would the presence of the bimodality allow for oscillations? It is exactly this property which is needed, as the gene regulatory circuit must be capable of switching between a low-number and a nigh-number state. In the deterministic version of the circuit, the low-number state is simply absent, if the cooperativity (the Hill coefficient) is too low. In this sense we can now precisely state the conditions for oscillations required in the modeling:

- in the deterministic setting, a sufficiently strong cooperatively (Hill coefficient);
- in the stochastic setting, a model allowing for a bimodal (minimally: a boundary bimodal) in the protein distribution (Vandecan Y, Blossey R (2012)).

Figure 4 shows the dynamics of both the stochastic and deterministic versions, here plotted in 'phase space', i.e., the concentration of proteins a,b and c.

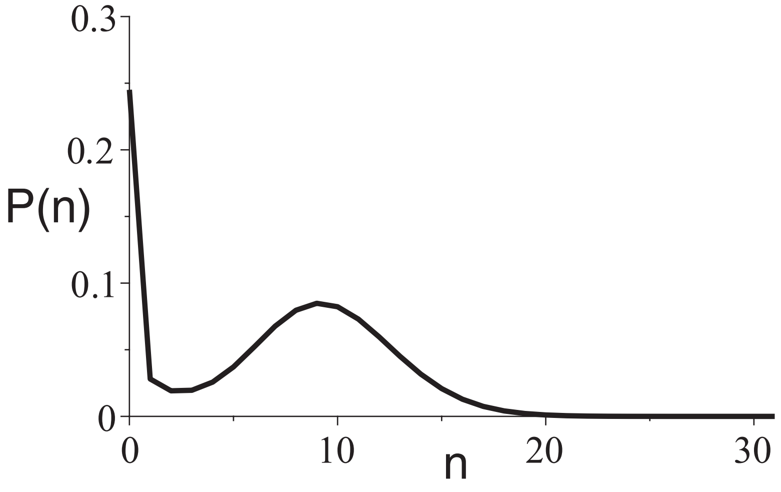
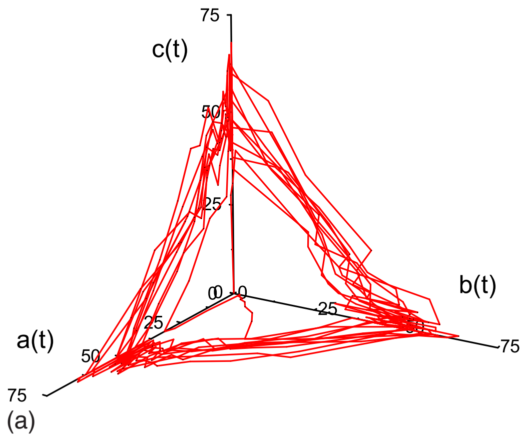


Figure 3: Probability distribution of the produced protein as a function of protein number (not of time, as in Figure 1), for a generic parameter choice. The (boundary) bimodality of the distribution is clearly seen: one maximum lies at $n = 0$ and another one at $n = 9$. Reprinted with permission from (Vandecan Y, Blossey R (2012)). Copyright by the American Physical Society.



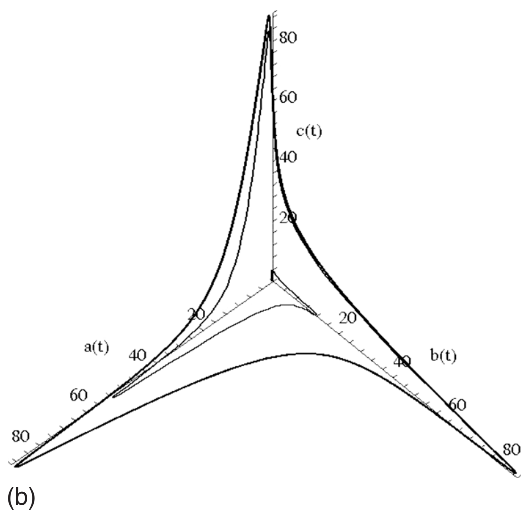


Figure 4: Top: The limit cycle of the stochastic repressilator. Bottom: The deterministic version for comparison. Reprinted with permission from (Blossey R, Giuraniuc V C (2008)). Copyright by the American Physical Society.

A closing remark on a model random path in Science. I feel that the above short story gives a nice illustration of a fairly generic path in Science; we built something whose properties we did not quite understand, and we found something whose importance we didn't immediately understand. As we were beginners in the field, ignorance on some matters may be taken as granted, but in the end one can understand why our model actually does what it did! At the time of our collaboration, we were also probably too much involved with bridging language gaps between computer science and physics, as we tried to apply ideas from both to a biological problem. In fact, we had a hard time to get our second paper published in a physics journal, just because of the language barrier between computer science and physics. My physics colleagues did mostly react quite strongly (in a strictly negative sense) when exposed to the philosophy of stochastic π-calculus, but there also are exceptions, like Joachim Rädler from Ludwig-Maximilians University in Munich, who actually also implemented some of these ideas with his students.

Looking back, I still cherish my discussions with Luca and Andrew and keep them in my memory as a model collaboration on interdisciplinary endeavors, during which we shared our knowledge and tried to overcome our ignorance at the same time. I also wish to thank my two postdoctoral fellows, Claudiu Giuraniuc and Yves Vandecan, for their work on the gene gate model.

References

Blossey R, Cardelli L, Phillips A. A compositional approach to the stochastic dynamics of gene networks. *Transactions in Computer Science*, 2006.

Blossey R, Cardelli L, Phillips A. Compositionality, stochasticity and cooperativity in dynamic models of gene regulation. *HFSP Journal*, 2:17–28, 2007.

Blossey R, Giuraniuc V C. Mean-field vs stochastic models for transcriptional regulation. *Physical Review E*, 78:031909, 2008.

Elowitz M B, Leibler S. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403, 2000.

Lipshtat A, Loinger A, Balaban N Q, Biham O. Genetic toggle switch without cooperative binding. *Physical Review Letters*, 96:188101, 2006.

Loinger A, Biham O. Stochastic simulations of the repressilator circuit. *Physical Review E*, 76:051917, 2007.

Peccoud J, Ycart B. Markovian modelling of gene product synthesis. *Theoretical Population Biology*, 48:222–234, 1995.

Vandecan Y, Blossey R. Self-regulatory gene: An exact solution for the gene gate model. *Physical Review E*, 87:042705, 2012.

Multilinear Programming with Big Data

Mihai Budiu Gordon D. Plotkin
Microsoft Research

Abstract

Systems such as MapReduce have become enormously popular for processing massive data sets since they substantially simplify the task of writing many naturally parallelizable parallel programs. In this paper we identify the computations carried out by such programs as *linear transformations* on distributed collections. To this end we model collections as multisets with a union operation, giving rise to a commutative monoid structure. The results of the computations (e.g., filtering, reduction) also lie in such monoids, (e.g., multisets with union, or the natural numbers with addition). The computations are then modelled as linear (i.e., homomorphic) transformations between the commutative monoids. Binary computations such as join are modelled in this framework by *multilinear* transformations, i.e., functions of several variables, linear in each argument.

We present a typed higher-order language for writing multilinear transformations; the intention is that all computations written in such a programming language are naturally parallelizable. The language provides a rich assortment of collection types, including collections whose elements are negatively or fractionally present (in general it permits modules over any given semiring). The type system segregates computations into linear and nonlinear phases, thereby enabling them to "switch" between different commutative monoids over the same underlying set (for example between addition and multiplication on real numbers). We use our language to derive linear versions of standard computations on collections; we also give several examples, including a linear version of MapReduce.

1 Introduction

As has been famously demonstrated by MapReduce, Dean and Ghemawat (2004), and followed up by related systems, such as DryadLINQ, Yu et al. (2008), big data computations can be accelerated by using massive parallelism. Parallelization is justified for simple mathematical reasons: big data has a natural commutative monoid structure with respect to which the transformations carried out by computations are linear (i.e., homomorphic). We present a programming language which seeks to expose this linearity; we intend thereby to lay the foundations for (multi)linear programming with big data.

Our language manipulates two kinds of types: ordinary and linear. In our setting linear types are commutative monoids (i.e., sets with a commutative associative operation with a zero). A typical example of such a monoid is provided by the positive reals $\mathbf{R}^+$ with addition.

Big data is usually manipulated as collections; these are unordered bags, or multisets, of values (sometimes represented as lists); we write $X^\star$ for the type of collections of elements of a set X. While the elements of a collection may be from an ordinary type, the collection type itself is a commutative monoid if endowed with multiset union (indeed $X^\star$ is the free commutative monoid over X).

Turning to transformations, given a function $f : X \to Y$ between ordinary types the Map operator yields a transformation $\mathrm{Map}(f) : X^\star \to Y^\star$ mapping X-collections to Y-collections. Again, given $g : Y \to \mathbf{R}^+$, the Reduce operator yields a transformation $\mathrm{Reduce}(g) : Y^\star \to \mathbf{R}^+$ mapping Y-collections to $\mathbf{R}^+$. Both of these are linear, preserving the monoid structures, i.e., we have:

$$\begin{array}{lclclcl}
\mathrm{Map}(f)(\emptyset) & = & \emptyset & \mathrm{Map}(f)(c \cup c') & = & \mathrm{Map}(f)(c) \cup \mathrm{Map}(f)(c') \\
\mathrm{Reduce}(g)(\emptyset) & = & 0 & \mathrm{Reduce}(g)(c \cup c') & = & \mathrm{Reduce}(g)(c) + \mathrm{Reduce}(g)(c')
\end{array}$$

(In fact, since $X^\star$ is the free commutative monoid over X, these are the unique such maps extending f and g, respectively.)

These equations justify the use of parallelism. For example, the linearity of Map implies that one can split a collection into two parts, map them in parallel, and combine the results to obtain the correct result.

As another example, suppose we have a binary tree of processors, and a collection c partitioned across the leaves of the tree. We map and then reduce at the leaves, and then reduce the results at the internal nodes. The final result is $\mathrm{Reduce}(g)(\mathrm{Map}(f)(c))$ irrespective of the data distribution at leaves, and of the shape of the tree. This fact depends on both the associativity and commutativity of the monoid operations and the linearity of the transformations; in practice this translates into the ability to do arbitrary load balancing of computations.

Our language is typed and higher-order. The language accommodates binary functions, such as joins, which have multilinear types (they are linear in each of their arguments). The language provides rich collection type constructors: in particular, for any linear types A and (certain) ordinary types X, we can construct the linear type $A[X]$ whose elements can be thought of variously as *A-ary X-collections*, or as *key-value dictionaries*, with X as the type of keys (or indices) and A as the type of values.

For example, by taking A to be the integers, we obtain multisets with elements having positive and negative counts; these are useful in modelling differential dataflow computations, see McSherry et al. (2013). Taking A to be the nonnegative reals, we obtain weighted collections, which are useful for modelling differential privacy, see Prospero et al. (2014). Dictionaries enable one to express GroupBy computations. The language further provides a mechanism for programming computations with both linear and nonlinear phases, possibly switching between different commutative monoids over the same carrier.

In the rest of this paper, after some remarks on commutative monoids, we present the syntax and denotational semantics of our language. We then argue practicality by modelling MapReduce and LINQ distributed computations through a series of examples (see Meijer et al. (2006) for an account of LINQ).

2 Remarks on commutative monoids

We work with commutative monoids $M = (|M|, +, 0)$ and linear (i.e., homomorphic functions) between them. We write $U(M)$ for $|M|$, the *carrier* of M (i.e., its underlying set). For any $n \in \mathbb{N}$ and $m \in M$ we write nm for the sum of m with itself n times.

The product of two commutative monoids is another, with addition and zero defined coordinatewise. Various sets of functions with range a commutative monoid M also form commutative monoids, with addition and zero defined pointwise. Examples include: $M[X]$ the monoid of all functions from a given set X to M which are zero except, possibly, at finitely many arguments; $X \to M$, the monoid of all functions from a given set X to M; and $M_1, \ldots, M_n \multimap M$ the monoid of all multilinear functions from given commutative monoids $M_1, \ldots, M_n$ to M. We write a typical element of $A[X]$ with value 0 except possibly at n arguments $x_1, \ldots, x_n$ as $\{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$.

Categorically, U is (the object part of) the forgetful functor to the category of sets. The product of two commutative monoids is also their sum, and so we employ the biproduct notation $M_1 \oplus M_2$. The commutative monoid $M[X]$ is the categorical sum $\sum_{x \in X} M$ and can also be viewed as the tensor $X \otimes M$ (the corresponding cotensor is $X \to M$).

$$\frac{X = Y}{Y = X} \qquad b = b \qquad |\mathrm{c}| = \mathrm{U}(\mathrm{c})$$

$$\frac{X = Y \quad X' = Y'}{X \times X' = Y \times Y'} \qquad \frac{A = B}{\mathrm{U}(A) = \mathrm{U}(B)} \qquad \frac{X = X' \quad Y = Y'}{X \to Y = X' \to Y'}$$

$$\frac{X = \mathrm{U}(A) \quad Y = \mathrm{U}(B)}{X \times Y = \mathrm{U}(A \oplus B)} \qquad \frac{X = X' \quad Y = \mathrm{U}(A)}{X \to Y = \mathrm{U}(X' \to A)}$$

Figure 1: Definitional equality rules for ordinary types.

$$\mathrm{c} = \mathrm{c} \qquad \frac{A = A' \quad B = B'}{A \oplus B = A' \oplus B'} \qquad \frac{A = A' \quad X = X'}{A[X] = A'[X']}$$

$$\frac{X = X' \quad A = A'}{X \to A = X' \to A'} \qquad \frac{\vec{A} = \vec{A'} \quad B = B'}{\vec{A} \multimap B = \vec{A'} \multimap B'}$$

Figure 2: Definitional equality rules for linear types.

3 The language

Types

The language has two kinds of type expressions: *ordinary* and *linear*, ranged over by $X, Y, \ldots$ and $A, B, \ldots$, respectively. They are given by:

$$X \;::=\; \mathrm{b} \mid X \times Y \mid \mathrm{U}(A) \mid X \to Y$$

$$A \;::=\; \mathrm{c} \mid A \oplus B \mid A[X] \mid X \to B \mid A_1, \ldots, A_m \multimap B$$

where b and c range over given *basic* ordinary and linear types, respectively. The basic ordinary types always contain bool and nat. The basic linear types always contain nat_+; other possibilities are $\mathrm{nat}_{\max}$ and real_+. In $A[X]$ we restrict X to be an *equality type*, meaning one not containing any function types.

We also assume given a syntactic *carrier* function $|\cdot|$, mapping basic linear types c to basic ordinary types $|\mathrm{c}|$. For example $|\mathrm{nat}_+| = |\mathrm{nat}_{\max}| = \mathrm{nat}$. This is used to obtain a notion of definitional equality of types which will enable computations to move between different linear structures on the same carrier; the rules for definitional equality are given in Figures 1 and 2; note the use there of vector notation for sequences of linear types.

Semantics of Types

Ordinary types X denote sets $[\![X]\!]$ and linear types A denote commutative monoids $[\![A]\!]$. The denotations of basic type expressions are assumed to be given. For example, bool and nat would denote, respectively, the booleans and the natural numbers; nat_+, would denote the natural numbers with addition; and $\mathrm{nat}_{\max}$ and real_+ would denote the natural numbers with maximum, and the reals with addition. We assume, for any basic linear type c that $[\![|c|]\!]$ is $U([\![c]\!])$ the carrier of $[\![c]\!]$.

The other type expressions have evident denotations. For example $[\![X \to Y]\!]$ is the set of all functions from $[\![X]\!]$ to $[\![Y]\!]$; $[\![\mathrm{U}(A)]\!]$ is $U([\![A]\!])$; $[\![A[X]]\!]$ is $[\![A]\!][[\![X]\!]]$; $[\![A_1, \ldots, A_n \multimap B]\!]$ is $[\![A_1]\!], \ldots, , [\![A_n]\!] \multimap [\![B]\!]$; and so on. One can check that definitionally equal types have equal denotations.

Terms

The language has *ordinary* terms ranged over by $t, u, \ldots$ and *multilinear* terms ranged over by $M, N, \ldots$. They are given by:

$$\begin{array}{lll}
t & ::= & x \mid \mathrm{d}_X(M) \mid f(t_1, \ldots, t_n) \mid \\
 & & \langle t, u \rangle \mid \ \mathrm{fst}(t) \mid \ \mathrm{snd}(t) \mid \\
 & & \lambda x : X.\, t \mid t(u)
\end{array}$$

$$\begin{array}{lll}
M & ::= & a \mid \mathrm{u}_A(t) \mid g(M_1, \ldots, M_n) \mid \\
 & & 0_A \mid M + N \mid MN \mid \\
 & & \text{if } t \text{ then } M \text{ else } N \mid \text{match}\, x : X, y : Y \text{ as } t \text{ in } M \mid \\
 & & \langle M, N \rangle \mid \ \mathrm{fst}(M) \mid \ \mathrm{snd}(M) \mid \\
 & & M \cdot t \mid \text{sum } a : A, x : X \text{ in } M.\, N \mid \\
 & & \lambda x : X.\, M \mid M(t) \mid \\
 & & \lambda a_1 : A_1, \ldots, a_n : A_m.\, M \mid M(N_1, \ldots, N_n)
\end{array}$$

In the above we use the letters $x, y, \ldots, a, b \ldots$ to range over variables. In the "match" construction x, y have scope extending over M; and in the "sum" construction a, x have scope extending over N. We assume given two signatures: one of ordinary basic functions $f : \mathrm{b}_1, \ldots, \mathrm{b}_n \to \mathrm{b}$ and the other of linear basic functions $g : \mathrm{c}_1, \ldots, \mathrm{c}_n \to \mathrm{c}$.

We introduce three "let" constructions as standard syntactic sugar:

$$\begin{array}{lll}
\text{let } x : X \text{ be } t \text{ in } u & =_{\mathrm{def}} & (\lambda x : X.\, u)(t) \\
\text{let } x : X \text{ be } t \text{ in } M & =_{\mathrm{def}} & (\lambda x : X.\, M)(t) \\
\text{let } \vec{a} : \vec{A} \text{ be } \vec{M} \text{ in } N & =_{\mathrm{def}} & (\lambda \vec{a} : \vec{A}.\, N)(\vec{M})
\end{array}$$

Instead of sum $a : A, x : X$ in $M. N : B$ we may write in a more "mathematical" way:

$$\sum_{a \cdot x \in M} N$$

Finally we may write unary function applications $M(N)$ in an "argument-first" manner, as $N.M$, associating such applications to the left.

Environments

The language has *ordinary* environments ranged over by Γ and *multilinear* environments ranged over by Δ. These environments are sequences of variable bindings of respective forms:

$$\Gamma ::= x_1 : X_1, \ldots, x_m : X_n \qquad \Delta ::= a_1 : A_1, \ldots, a_n : A_n$$

where the x_i are all different, as are the a_j. Below we write $\Delta || \Delta'$ for the set of all merges (interleavings) of the two sequences of variable bindings Δ and Δ'.

Typing Rules

We have two kinds of judgements, *ordinary* and *multilinear*

$$\Gamma \vdash t : X \qquad \text{and} \qquad \Gamma \mid \Delta \vdash M : A$$

where, in the latter, Γ and Δ have no variables in common. The rules are either structural, casting, ordinary, or multilinear, and are as follows:

Structural

$$\Gamma, x : X, \Gamma' \vdash x : X \qquad \Gamma \mid a : A \vdash a : A$$

Casting

$$\frac{\Gamma \mid \vdash M : A \quad \mathrm{U}(A) = X}{\Gamma \vdash \mathrm{d}_X(M) : X} \qquad \frac{\Gamma \vdash t : X \quad X = \mathrm{U}(A)}{\Gamma \mid \vdash \mathrm{u}_A(t) : A}$$

Ordinary

$$\frac{\Gamma \vdash \vec{\mathrm{t}} : \vec{\mathrm{b}}}{\Gamma \vdash f(\vec{\mathrm{t}}) : \mathrm{b}} \; (f : \vec{\mathrm{b}} \to \mathrm{b})$$

$$\frac{\Gamma \vdash t : X \quad \Gamma \vdash u : Y}{\Gamma \vdash \langle t, u \rangle : X \times Y} \qquad \frac{\Gamma \vdash t : X \times Y}{\Gamma \vdash \ \mathrm{fst}(t) : X} \qquad \frac{\Gamma \vdash t : X \times Y}{\Gamma \vdash \ \mathrm{snd}(t) : Y}$$

$$\frac{\Gamma, x : X \vdash t : Y}{\Gamma \vdash \lambda x : X.\, t : X \to Y} \qquad \frac{\Gamma \vdash t : X \to Y \quad \Gamma \vdash u : X}{\Gamma \vdash t(u) : Y}$$

Linear

$$\frac{\Gamma \mid \Delta_i \vdash M_i : \mathrm{c}_i \quad (i = 1, n)}{\Gamma \mid \Delta \vdash g(M_1, \ldots, M_n) : \mathrm{c}} \quad (g : \mathrm{c}_1, \ldots, \mathrm{c}_n \to \mathrm{c}, \Delta \in \Delta_1 || \ldots || \Delta_n)$$

$$\Gamma \mid \Delta \vdash 0_A : A \qquad \frac{\Gamma \mid \Delta \vdash M : A \quad \Gamma \mid \Delta \vdash N : A}{\Gamma \mid \Delta \vdash M + N : A}$$

$$\frac{\Gamma \mid \Delta' \vdash M : \mathrm{nat}_+ \quad \Gamma \mid \Delta'' \vdash N : A}{\Gamma \mid \Delta \vdash MN : A} \quad (\Delta \in \Delta' || \Delta'')$$

$$\frac{\Gamma \vdash t : \mathrm{bool} \quad \Gamma \mid \Delta \vdash M : A \quad \Gamma \mid \Delta \vdash N : A}{\Gamma \mid \Delta \vdash \text{if } t \text{ then } M \text{ else } N : A} \qquad \frac{\Gamma \vdash t : X \times Y \quad \Gamma, x : X, y : Y \mid \Delta \vdash M : A}{\Gamma \mid \Delta \vdash \mathrm{match}\, x : X, y : Y \text{ as } t \text{ in } M : A}$$

$$\frac{\Gamma \mid \Delta \vdash M : A \quad \Gamma \mid \Delta \vdash N : B}{\Gamma \mid \Delta \vdash \langle M, N \rangle : A \oplus B} \qquad \frac{\Gamma \mid \Delta \vdash M : A \oplus B}{\Gamma \mid \Delta \vdash \ \mathrm{fst}(M) : A} \qquad \frac{\Gamma \mid \Delta \vdash M : A \oplus B}{\Gamma \mid \Delta \vdash \ \mathrm{snd}(M) : B}$$

$$\frac{\Gamma \mid \Delta \vdash M : A \quad \Gamma \vdash t : X}{\Gamma \mid \Delta \vdash M \cdot t : A[X]} \qquad \frac{\Gamma \mid \Delta' \vdash M : A[X] \quad \Gamma, x : X \mid \Delta'', a : A \vdash N : B}{\Gamma \mid \Delta \vdash \mathrm{sum}\ a : A, x : X \text{ in } M.\, N : B} \quad (\Delta \in \Delta' || \Delta'')$$

$$\frac{\Gamma, x : X \mid \Delta \vdash M : B}{\Gamma \mid \Delta \vdash \lambda x : X.\, M : X \to B} \qquad \frac{\Gamma \mid \Delta \vdash M : X \to B \quad \Gamma \vdash t : X}{\Gamma \mid \Delta \vdash M(t) : B}$$

$$\frac{\Gamma \mid \Delta, \vec{a} : \vec{A} \vdash M : B}{\Gamma \mid \Delta \vdash \lambda \vec{a} : \vec{A}.\, M : \vec{A} \multimap B}$$

$$\frac{\Gamma \mid \Delta' \vdash M : A_1, \ldots, A_n \multimap B \qquad \Gamma \mid \Delta_i \vdash N_i : A_i \quad (i = 1, n)}{\Gamma \mid \Delta \vdash M(N_1, \ldots, N_n) : B} \quad (\Delta \in \Delta' || \Delta_1 || \ldots || \Delta_n)$$

The use of the merge operator $||$ on linear environments ensures that derivable typing judgments are closed under permutation of linear environments; as may be expected, they are not closed under weakening or duplication. Typing is unique in that for any Γ, Δ and M there is at most one A such that $\Gamma \mid \Delta \vdash M : A$ and similarly for judgments $\Gamma \vdash t : X$. There is also a natural top-down type-checking algorithm.

We sketch the denotational semantics of terms below, but their intended meaning should be clear from the previous section. For example, the term MN with

$M : \mathrm{nat}_+$ indicates the addition of N with itself M times. The terms $\mathrm{d}_X(M)$ and $\mathrm{u}_A(t)$ should be read as "down" and "up" casts, which convert back and forth between a linear type A and an ordinary type X definitionally equal to $\mathrm{U}(A)$. Using terms of the forms $\mathrm{d}_X(M)$, $\mathrm{u}_A(t)$ one can construct conversions between any two definitionally equal types.

Some constructions that may seem missing from the biproduct are in fact definable. The first injection $\mathrm{inl}(M)$ can be defined by $\langle M, 0\rangle$, similarly for the second, and we can define a cases construction by:

$$\begin{array}{lll}\text{cases } K \text{ fst } a : A.\, M,\ \text{snd } b : B.\, N & =_{\text{def}} & \text{let } c : A \oplus B \text{ be } K \text{ in} \\ & & \ (\text{let } a : A \text{ be } \ \mathrm{fst}(c) \text{ in } M) \\ & & + \ (\text{let } b : B \text{ be } \ \mathrm{snd}(c) \text{ in } N)\end{array}$$

So given "product" and "plus" we get "sum"; in fact, given any two of "product", "sum", and "plus" one can define the third.

Semantics of terms

For the basic functions, f, g, one assumes available given functions $\llbracket f \rrbracket$, $\llbracket g \rrbracket$ of suitable types.

For environments $\Gamma = x_1 : X_1, \ldots, x_m : X_m$ and $\Delta = a_1 : A_1, \ldots, a_n : A_n$, we write $\llbracket \Gamma \rrbracket$ for the set $\llbracket X_1 \rrbracket \times \ldots \times \llbracket X_m \rrbracket$, and $\llbracket \Delta \rrbracket$ for the carrier of $\llbracket A_1 \rrbracket \times \ldots \times \llbracket A_n \rrbracket$, respectively. Then, much as usual, the denotational semantics assigns to each typing judgement $\Gamma \vdash t : Y$ a function

$$\llbracket \Gamma \vdash t : Y \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \llbracket Y \rrbracket$$

and to each typing judgement $\Gamma \mid \Delta \vdash M : B$ a function

$$\llbracket \Gamma \mid \Delta \vdash M : B \rrbracket : \llbracket \Gamma \rrbracket \times \llbracket \Delta \rrbracket \longrightarrow \llbracket B \rrbracket$$

linear in each of the Δ coordinates (this is why Δ is called a "multilinear environment"). The definition is by structural definition on the terms; we just illustrate a few cases.

The type conversions are modelled by the identity function, for example:

$$\llbracket \Gamma \mid \vdash \mathrm{d}_X(M) : X \rrbracket(\vec{v}, \vec{\alpha}) = \llbracket \Gamma \vdash M : A \rrbracket(\vec{v})$$

As one might expect the syntactic monoid operations are modelled by the semantic ones, for example:

$$\llbracket \Gamma \mid \Delta \vdash M{+}N : B \rrbracket(\vec{v}, \vec{\alpha}) = \llbracket \Gamma \mid \Delta \vdash M : B \rrbracket(\vec{v}, \vec{\alpha}) +_{\llbracket B \rrbracket} \llbracket \Gamma \mid \Delta \vdash N : B \rrbracket(\vec{v}, \vec{\alpha})$$

For the collection syntax we have first that:

$$[\![\Gamma \mid \Delta \vdash M \cdot t : A[X]]\!](\vec{v}, \vec{\alpha}) = \{[\![\Gamma \vdash t : X]\!](\vec{v}) \mapsto [\![\Gamma \mid \Delta \vdash M : A]\!](\vec{v}, \vec{\alpha})\}$$

Next if $\Gamma \mid\ \Delta \vdash \mathrm{sum}\ a : A, x : X\ \mathrm{in}\ M.\, N : X$ holds then there are, necessarily unique, Δ', Δ'' such that $\Gamma \mid \Delta' \vdash M : A[X]$ and $\Gamma, x : X \mid \Delta'', a : A \vdash N : B$ and $\Delta \in \Delta' || \Delta''$ all hold. We use the fact that $\Delta \in \Delta' || \Delta''$ to obtain canonical projections $\pi' : [\![\Delta]\!] \to [\![\Delta']\!]$ and $\pi'' : [\![\Delta]\!] \to [\![\Delta'']\!]$.

Suppose that

$$[\![\Gamma \mid \Delta' \vdash M : A[X]]\!](\vec{v}, \pi'(\vec{\alpha})) = \{v_1 \mapsto a_1, \ldots, v_n \mapsto a_n\}$$

Then

$$\begin{aligned} [\![\Gamma \mid \vdash \mathrm{sum}\ a : A, x : X\ \mathrm{in}\ M.\, Nt : X]\!](\vec{v}, \vec{\alpha}) \quad = & \\ \textstyle\sum_{i=1,n} [\![\Gamma, x : X \mid \Delta'', a : A \vdash N : B]\!]((\vec{v}, v_i), (\pi''(\vec{\alpha}), a_i)) & \end{aligned}$$

The semantics of the other terms pose no surprises; in cases where linear environments Δ are split up, one again makes use of canonically available projections.

Implementation considerations

It very much remains to be seen how useful our ideas prove. In the meantime, it seems worthwhile saying a little about possible implementation datatypes. One could use lists, possibly spread among different processors, to represent collections. Representations would be recursively defined: if R represented X, and S represented the carrier of A then $(R \times S)^*$ could represent $A[X]$, with $(r_1, s_1) \ldots (r_n, s_n)$ representing $\sum_{i=1,n}\{x_i \mapsto a_i\}$ if r_i, s_i represented x_i, a_i, for all $i \in \{1, \ldots, n\}$.

Such representations have a natural normal form: assuming the r_i and s_i are already in normal form, one adds the s_i together (using a representation of addition on A) to produce a list $(r'_1, s'_1), \ldots, (r'_n, s'_n)$ with the r'_j all different, and then orders the list using a total ordering of S, itself recursively defined.

When evaluating $\mathrm{u}_A(t)$ one needs to have the value of t in normal form, as otherwise the addition implied by the representation relation is that of A, which may not generally be correct (for example, t may itself be $\mathrm{d}_X(M)$ where the (linear) type of M has a different addition from that of A). So when evaluating $\mathrm{d}_X(M)$, one should put the value of M into normal form as the correct addition is then known from the linear type of M.

4 Operators

As stated above, the type $A[X]$ can be regarded variously as that of A-valued X-collections or of key-value dictionaries over A and indexed by X. In particular, taking A to be nat_+ we get the usual unordered collections, i.e., finite multisets of elements of X; we write this type as $X^\star$. We now look at linear versions of standard operators such as Map, Fold, Reduce, GroupBy, and Join. As our notion of collection is more general than that used in traditional programming languages we obtain corresponding generalisations of these operators.

Map

We can define a family of Map operators which operate on both the elements of a collection and their coefficients. Associating function type arrows to the right, they have type

$$(A \multimap B) \multimap (X \to Y) \to (A[X] \multimap B[Y])$$

and are given by:

$$\mathrm{Map}_{X,Y,A,B} =_{\mathrm{def}} \lambda f : (A \multimap B).\, \lambda g : (X \to Y).\, \lambda c : A[X].\, \sum_{a \cdot x \in c} f(a) \cdot g(x)$$

where we are making use of the summation notation introduced above. Note that here, and below, operators are often linear in their function arguments.

Specialising to the case where $B = A$ and $f : A \multimap B$ is the identity id_A (i.e., $\lambda a : A.\, a$), we obtain a family of operators

$$\mathrm{Map}_{X,Y,A} : (X \to Y) \to (A[X] \multimap A[Y])$$

where we are overloading notation. When $A = \mathrm{nat}_+$ these are the usual Map operators, but with their linearity made explicit in their type:

$$(X \to Y) \to (X^\star \multimap Y^\star)$$

Actions and their extensions

We define an *action (term)* of a linear type A on another B to be a term of type $A, B \multimap B$. Such an action always exists when $A = \mathrm{nat}_+$, viz., the term $\lambda n : \mathrm{nat}_+, b : B.\, nb$. In general, we may only be given "multiplication" terms $m_A : A, A \multimap A$ providing an action of A on itself; we may then, as we will see below, use the given multiplication to obtain actions on other linear types.

For example in the case of real_+, the multiplication term would denote the usual multiplication on the positive reals. When we have a multiplication term on a linear type A we may also have a "unit" term $1_A : A$ (e.g., a term denoting the usual unit in the case of the positive reals). The unit provides a generalisation of the multiset singleton map $\{-\} : X \to X^\star$, namely $\lambda x : X.\, 1_A x : X \to A[X]$.[1]

Given an action of A on B we can obtain an action of A on $B[X]$ using a family of Extend operators. They have type

$$(A, B \multimap B) \multimap (A, B[X] \multimap B[X])$$

and are given by:

$$\text{Extend}_{X,A,B} =_{\text{def}} \lambda f : (A, B \multimap B).\, \lambda a : A, c : B[X].\, \sum_{b \cdot x \in c} f(a, b) \cdot x$$

Actions can be extended to other types. In the case of biproducts, given an action of A on both B and C, then there is an action of A on $B \oplus C$; in the case of function types, given an action of A on C, there are actions of A on $X \to C$ and $\overrightarrow{B} \multimap C$. We leave their definition as an exercise for the reader. Combining such extensions, one can build up actions on complex datatypes.

Folding

We define a family of Fold operators with type

$$(A, B \multimap B), (X \to B) \multimap (A[X] \multimap B)$$

They are given by:

$$\text{Fold}_{X,A,B} =_{\text{def}} \lambda m : (A, B \multimap B), f : X \to B.\, \lambda c : A[X].\, \sum_{a \cdot x \in c} m(a, f(x))$$

Note that the fold operator needs an action of of A on B.

SelectMany Using Fold we can define a family of SelectMany operators that generalise those of LINQ analogously to the above Map operators. They have type

$$(A \multimap B), (X \to B[X]) \multimap (A[X] \multimap B[X])$$

[1]We would expect such a multiplication and unit to make A a semiring (i.e., to provide a bilinear associative multiplication operation with a unit) and we would expect the actions of A on other linear types to make them A-modules. If such algebraic assumptions are fulfilled, some natural program equivalences hold.

and are given by:

$$\begin{array}{rcl}\mathrm{SelectMany}_{X,A,B} & = & \lambda f : A \multimap B, g : X \to B[X].\, \lambda c : A[X]. \\ & & \quad \mathrm{let}\ e : A, B[X] \multimap B[X]\ \mathrm{be} \\ & & \quad\quad \mathrm{Extend}(\lambda a : A, b : B.\, m_B(f(a), b)) \\ & & \quad \mathrm{in}\ c.\mathrm{Fold}(e, g)\end{array}$$

where we have made use of the reverse application notation introduced above, and have also assumed available a multiplication term $m_B : B, B \multimap B$.

Taking $B = A$ and specialising $f : A \multimap B$ to the identity, we obtain a family of operators of types

$$\mathrm{SelectMany}_{X,A} : (X \to A[X]) \multimap (A[X] \multimap A[X])$$

again overloading notation. When $A = \mathrm{nat}_+$ these have type

$$(X \to X^\star) \multimap (X^\star \multimap X^\star)$$

and are the usual LINQ SelectMany operators (these are the same as MapReduce's improperly-named Map operators).

Reduction For general A-valued collections, we may already regard Fold as a reduction (or aggregation) operator. We can obtain analogues of the more usual reductions by taking both A and B to be basic linear types where there is an action of A on B; an example would be to take them both to be real_+ and the action to be m_{real_+}.

We can specialise the first argument of Fold to the action of nat_+ on linear type's B and obtain a family of operators

$$\mathrm{Reduce}_{X,B} : (X \to B) \multimap (X^\star \multimap B)$$

In this generality, these include SelectMany, if we take B to be $Y^\star$. Taking B to be a basic linear type such as real_+ we obtain more usual reductions.

Note that in all the cases considered above, the reduction operations are fixed to be the sum operations of the target linear types.

GroupBy

As already indicated, one can regard elements of the linear type $A[X]$ as key-value dictionaries of elements of A, indexed by elements of X. In particular, given a type of keys K, we can regard $A[X][K]$ as the type of K-indexed A-valued X-collections. With this understanding, we have a family of GroupBy operators using of key function $X \to K$. These have type

$$(X \to K) \to (A[X] \multimap A[X][K])$$

and are given by:

$$\mathrm{GroupBy}_{K,X,A} = \lambda k : (X \to K).\, \lambda c : A[X].\, \sum_{a \cdot x \in c} (ax) \cdot k(x)$$

Lookup

Lookup functions extract the element with a given key from a K-indexed dictionary. They have type

$$K \to (A[K] \multimap A)$$

and are given by:

$$\mathrm{Lookup}_{K,X,A} =_{\mathrm{def}} \lambda x : K.\, \lambda c : A[K].\, \sum_{a \cdot x' \in c} \text{if } x' = x \text{ then } a \text{ else } 0$$

where we have assumed available an equality function on K.

Join

We first define cartesian product operations on collections; they require actions of linear types on themselves in order to combine values with the same index. We have a family of operations of type

$$(A, A \multimap A), A[X], A[Y] \multimap A[X \times Y]$$

given by:

$$\mathrm{CartProd}_{X,A} =_{\mathrm{def}} \lambda m : A, A \multimap A, c : A[X], c' : A[X].\, \sum_{a \cdot x \in c} \sum_{a' \cdot y \in c'} m(a, a') \cdot \langle x, y \rangle$$

We further have a family of Join operations which operate on pre-grouped-by collections, with a type of keys K for which an equality function is assumed available. They have type

$$(A, A \multimap A), A[X][K], A[Y][K] \multimap A[X \times Y][K]$$

and are given by:

$$\begin{aligned}\mathrm{Join}_{X,Y,K,A} =_{\mathrm{def}}\ & \lambda m : (A, A \multimap A), d : A[X][K], d' : A[Y][K]. \\ & \textstyle\sum_{c \cdot k \in d} \text{let } c' : A[Y] \text{ be } \mathrm{Lookup}(k)(d') \text{ in } \mathrm{CartProd}(m, c, c') \cdot k\end{aligned}$$

Zip

Our final example is another family of binary functions on key-value dictionaries, which model the LINQ Zip operation:

$$\mathrm{Zip}_{X,A,B} : A[X] \oplus B[X] \multimap (A \oplus B)[X]$$

They take two X-indexed dictionaries and pair entries with the same index. They are given by:

$$\begin{array}{lll} \mathrm{Zip}_{X,A,B} & =_{\mathrm{def}} & \lambda d : A[X] \oplus B[X]. \\ & & \quad \mathrm{Map}(\mathrm{inl})(\mathrm{id}_X)(\mathrm{fst}(d)) +_{A\oplus B} \mathrm{Map}(\mathrm{inr})(\mathrm{id}_X)(\mathrm{snd}(d)) \end{array}$$

equivalently:

$$\begin{array}{lll} \mathrm{Zip}_{X,A,B} & =_{\mathrm{def}} & \lambda d : A[X] \oplus B[X].\, \mathrm{cases}\ d\ \mathrm{fst}\ a : A[X].\, \mathrm{Map}(\mathrm{inl})(\mathrm{id}_X)(\mathrm{a}), \\ & & \qquad\qquad\qquad\qquad\quad \mathrm{snd}\ b : B[X].\, \mathrm{Map}(\mathrm{inr})(\mathrm{id}_X)(\mathrm{b}) \end{array}$$

5 Some Example Programs

In this section we give some example programs. The first two compute non-linear functions. However they are composed from linear subcomputations, and these are exposed as linear subterms. The last example is a linear version of MapReduce.

5.1 Counting

Our first example illustrates the utility of being able to move non-linearly between different monoids with the same carrier. Given a collection, we can count its elements, taking account of their multiplicity, using $\mathrm{Count}_X : X^\star \multimap \mathrm{nat}_+$, given by:

$$\lambda c : X^\star.\, \mathrm{Reduce}_{X,\mathrm{nat}_+}(\lambda x : X.\, \mathrm{u}_{\mathrm{nat}_+}(1))(c)$$

However, if we instead want to count ignoring multiplicity (i.e., to find the number of distinct elements), the computation proceeds in two linear phases separated by a non-linear one, as follows:

- the input collection, read as a nat_+-collection by a type conversion from an ordinary to a linear type, is mapped to a $\mathrm{nat}_{\mathrm{max}}$-collection c' to record only the presence or absence of an item (by a 1 or a 0),
- c' is then transformed nonlinearly to a nat_+-collection c'', using the type conversions, and, only then,

- the Count function is applied to c''.

To do this we use a term $\mathrm{SetCount}_X : \mathrm{U}(X^\star) \to \mathrm{nat}_+$, namely

$$\begin{array}{l}\lambda x : \mathrm{U}(X^\star).\\ \quad \text{let } c' : \mathrm{nat}_{\max}[X] \text{ be } \mathrm{u}_{X^\star}(x).\mathrm{Map}(\mathrm{Conv})(\mathrm{id}_X) \text{ in}\\ \quad \text{let } c'' : X^\star \text{ be } \mathrm{u}_{X^\star}(\mathrm{d}_{\mathrm{U}(X^\star)}(c')) \text{ in } c''.\mathrm{Count}\end{array}$$

where $\mathrm{Conv} : \mathrm{nat}_+ \multimap \mathrm{nat}_{\max}$ is a linear term converting between nat_+ and $\mathrm{nat}_{\max}$, namely $\lambda n : \mathrm{nat}_+ .\, n(\mathrm{u}_{\mathrm{nat}_{\max}}(1))$ (it sends 0 to 0 and all other natural numbers to 1), and where we have dropped operator indices.

5.2 Histograms

Our second example is a simple histogram computation. Suppose we have a collection c of natural numbers and wish to plot a histogram of them spanning the range 0 to m, the maximum element in the collection (which we assume to be > 0). The histogram is to have $k > 0$ buckets, starting at 0, so each bucket will have width m/k. We model histograms by multisets of natural numbers, where each element corresponds to a bucket, and has multiplicity corresponding to the number of values in that bucket.

The following function $\mathrm{Hist}^k_{\max} : \mathrm{nat} \to (\mathrm{nat}^\star \multimap \mathrm{nat}^\star)$ is provided the maximum element value and computes the histogram linearly over c:

$$\lambda m : \mathrm{nat}.\, \lambda c : \mathrm{nat}^\star .\, c.\mathrm{SelectMany}(\lambda n : \mathrm{nat}.\, \{\lfloor kn/m \rfloor\})$$

The maximum element can be found linearly from c using a reduction:

$$c.\mathrm{Reduce}(\lambda n : \mathrm{nat}.\, \mathrm{u}_{\mathrm{nat}_{\max}}(n))$$

Putting these two together, we obtain a function $\mathrm{Hist}^{\mathrm{k}} : \mathrm{nat}^\star \to \mathrm{nat}^\star$ computing the required histogram:

$$\begin{array}{l}\lambda c : \mathrm{U}(\mathrm{nat}^\star).\\ \quad \text{let } m : \mathrm{nat} \text{ be } \mathrm{d}_{\mathrm{nat}}(\mathrm{u}_{\mathrm{nat}^\star}(c).\mathrm{Reduce}(\lambda n : \mathrm{nat}.\, \mathrm{u}_{\mathrm{nat}_{\max}}(n))) \text{ in}\\ \quad \mathrm{u}_{\mathrm{nat}^\star}(c).\mathrm{Hist}^k_{\max}(m)\end{array}$$

Note the double occurrence of c, signalling nonlinearity.

5.3 A linear MapReduce

We present a linear version of MapReduce. It models the distributed nature of the data by using a dictionary indexed by machine names to model partitioned collections. The MapReduce computation begins with the initial collection distributed over the machines, carries out a computation in parallel on each machine, redistributes the data between the machines by a shuffle operation, and then performs a final reduction.

We begin with the computation carried out on each machine. This applies to a collection c, and consists of a SelectMany, then a GroupBy using a basic type K of keys, and then a Reduce at each key. It is given by the following term MR:

$$\lambda c : X^\star .\ c.\mathrm{SelectMany}(m).\mathrm{GroupBy}(k).\mathrm{Map}(\mathrm{Reduce}(r))(\mathrm{id}_{\mathrm{K}})$$

which has typing:

$$k : Y \to \mathrm{K} \mid m : X \to Y^\star, r : Y \to A \vdash \mathrm{MR} : X^\star \multimap A[\mathrm{K}]$$

We next need to model data of any given type B spread across machines. To do this we assume available a basic type M of machine names and model such data by an M-indexed dictionary of type $B[\mathrm{M}]$. With this in mind the parallel computation is given by the following term PMR:

$$\mathrm{Map}(\mathrm{MR})(\mathrm{id}_{\mathrm{M}})$$

which maps MR across the machines and which has typing

$$k : Y \to \mathrm{K} \mid m : X \to Y^\star, r : Y \to A \vdash \mathrm{PMR} : X^\star[\mathrm{M}] \multimap A[\mathrm{K}][\mathrm{M}]$$

The shuffle operation employs a key-to-machine function, $h : \mathrm{K} \to \mathrm{M}$, and is given by the following term SH:

$$\begin{array}{l} \lambda e : A[\mathrm{K}][\mathrm{M}]. \\ \quad \mathrm{sum}\ d : A[\mathrm{K}], m : \mathrm{M}\ \mathrm{in}\ e. \\ \quad \mathrm{sum}\ a : A, k : \mathrm{K}\ \mathrm{in}\ d.\ (\{a\} \cdot k) \cdot h(k) \end{array}$$

which has typing:

$$h : \mathrm{K} \to \mathrm{M} \mid\ \vdash \mathrm{SH} : A[\mathrm{K}][\mathrm{M}] \multimap A^\star[\mathrm{K}][\mathrm{M}]$$

The final reduction is carried out in parallel on each machine and is given by the following term FR:

$$\mathrm{Map}(\mathrm{Map}(\mathrm{Reduce}(\mathrm{id}_A))(\mathrm{id}_{\mathrm{K}}))(\mathrm{id}_{\mathrm{M}})$$

which maps the reduction at each key across the machines and which has typing

$$\vdash \mathrm{FR} : A^\star[\mathrm{K}][\mathrm{M}] \multimap A[\mathrm{K}][\mathrm{M}]$$

Putting everything together we obtain the entire MapReduce computation. It is given by the following term MapReduce:

$$\lambda b : X^\star[\mathrm{M}].\, b.\mathrm{PMR.SH.FR}$$

which has typing:

$$k : Y \to \mathrm{K}, h : \mathrm{K} \to \mathrm{M} \mid f : X \to Y^\star, r : Y \to A \vdash \mathrm{MapReduce} : X^\star[\mathrm{M}] \multimap A[\mathrm{K}][\mathrm{M}]$$

One could evidently abstract on the various functions, or choose specific ones.

6 Discussion

One can imagine a number of extensions and developments. Most immediately, as well as rules for type-checking, one would like an equational system, as is usual in type theories. This would open up the possibility of proving programs such as MapReduce correct. Regarding the language design, the reduction facilities depend on the built-in monoid structures. However in, e.g., LINQ, programmers can choose their own. In order to continue exposing linearity, it would be natural to introduce linear types of the form (X, z, m) where $z : X$ and $m : X^2 \to X$ are intended to provide X with a commutative monoid structure.

The mathematics suggests further possibilities. For example when working with A-valued collections (but not dictionaries) it is natural to suppose one has a semiring structure on A. Perhaps it would be worthwhile to add a kind of semirings (possibly even programmable) and to have separate linear types of collections and dictionaries.

Again, commutative monoids have a tensor product $A \otimes B$ classifying bilinear functions. One wonders if this would provide a useful datatype for big data programming. The tensor product enjoys various natural isomorphisms, for example: $A[X] \otimes B[Y] \cong (A \otimes B)[X \times Y]$, in particular $X^\star \otimes Y^\star \cong (X \times Y)^\star$.

The mathematics suffers if one were to drop commutativity, and just work with monoids, as in Nesl — see Blelloch (2011). One no longer has linear function spaces or tensor products. However it is not clear that one would not thereby enjoy benefits for programming with big data.

There are yet other possibilities for further development. It would be useful to add probabilistic choices to the language, however the interaction between probability and linearity is hardly clear. It would be interesting to consider differential aspects, as in McSherry et al. (2013). This would involve passing from monoids and semirings to abelian groups and rings. Compilers might well benefit from language facilities to indicate intended parallelism; an example is the use of machine-indexed collections used above to model MapReduce. One could imagine a programmer-specified machine architecture, with machine-located datatypes $A@m$, see Jia and Walker (2004) and Murphy VII (2008).

References

G. E. Blelloch. Nesl. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1278–1283. Springer, 2011. ISBN 978-0-387-09765-7.

J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th OSDI*, pages 137–150. ACM, 2004. URL `http://labs.google.com/papers/mapreduce.html`.

L. Jia and D. Walker. Modal proofs as distributed programs (extended abstract). In *ESOP*, pages 219–233, 2004.

F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*. www.cidrdb.org, 2013.

E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Proc. SIGMOD Int. Conf. on Manage. Data*, page 706. ACM, 2006. ISBN 1-59593-434-0. doi: http://doi.acm.org/10.1145/1142473.1142552. URL `http://doi.acm.org/10.1145/1142473.1142552`.

T. Murphy VII. *Modal types for mobile code.* PhD thesis, CMU, 2008.

D. Prospero, S. Goldberg, and F. McSherry. Calibrating data to sensitivity in private data analysis, a platform for differentially-private analysis of weighted datasets. To appear in VLDB14, 2014.

Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. Gunda, K. Pradeep, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. 8th OSDI*, pages 1–14. ACM, 2008.

Types and Logic, Concurrency and Non-Determinism

Luís Caires
NOVA LINCS/CITI and Departamento de Informática, FCT UNL

Abstract

Behavioural types are becoming an increasingly useful instrument to reason about the behaviour of complex concurrent and interactive computing systems. While type systems for traditional programming models have been for long rooted on pure logical principles, the connections between interactive and concurrent programming models and their logically motivated type disciplines started to be better understood only recently. In this note, we motivate an approach to accommodate internal non-determinism, a phenomenon pervasively present both in the behaviour of artificial and natural computing systems but which seems to have escaped logical analysis, in a type system for session-based communication that preserves logical compatibility by supporting a Curry-Howard correspondence.

"Each time a man is confronted with several alternatives, he chooses one and eliminates the others; in the fiction of Tsui Płn, he chooses simultaneously all of them. He creates, in this way, diverse futures, diverse times which themselves also proliferate and fork." (in *The Garden of Forking Paths*, Jorge Luis Borges)

Dedicated to Luca Cardelli on the occasion of his 60th birthday.

1 Introduction

Type systems for programming languages have its foundational roots in logic, as recalled in the famous Typeful Programming paper by Luca Cardelli (1991): "... one can say that typeful programming is just a special case of program specification, and one can read type as a synonym for specification, and typechecking as a synonym for verification [...]. This view fits well with the types as propositions paradigm of axiomatic semantics, and the propositions as types paradigm of intuitionistic logic.". Nevertheless, while type systems for traditional programming models are known to be rooted in pure logical principles, the connections between interactive and concurrent programming models and their logically motivated type disciplines started to be better understood only recently, building on several relationships between substructural logics and process calculi (e.g., Abramsky (1993); Caires and Cardelli (2004)). This expected development can be seen also as yet another realisation of the fact that "one can also extrapolate this correspondence and turn it into a predictive tool: if a concept is present in type theory but absent in programming, or vice versa, it can be very fruitful to both areas to investigate and see what the corresponding concept might be in the other context." (Cardelli 1991). In this note, we motivate an approach to accommodate the concept of internal non-determinism, a phenomenon pervasively present both in artificial and natural computing systems, and in particular in stochastic models of biological systems (Cardelli 2008). Quite interestingly, internal non-determinism seems to keep escaping for some time a reasonable logical analysis, due to the apparent incompatibility between the confluence property of cut-elimination and the collapsing effect inherent of internal non-determinism.

In this note, we investigate a session-based type system for interactive processes that preserves compatibility of internal non-determinism with logic in the sense of a Curry-Howard correspondence. The approach involves two key ingredients, both clearly related to familiar concepts in concurrency and programming languages. The first one, involves admitting processes denoting alternative potential behaviours among the various process forms under consideration: these processes are subject to behaviour preserving reduction rules, but are compatible with cut-elimination, and support compositional reasoning about process behavior. The second ingredient involves capturing non-determinism *in* the type structure, thus cleanly separating, as a proper conservative extension of the basic type system, non-deterministic behaviour from the basic deterministic behaviors. To achieve this, we encapsulate non-determinism inside a monadic presentation supported by two type operators $\&A$ and $\oplus A$, related by duality.

2 Session Types and Linear Logic

The session discipline (Honda 1993; Honda et al. 1998) applies to distributed processes which communicate through point-to-point private channels (e.g., such as TCP sockets). In such a practically relevant setting communications must always occur in matching pairs: when one partner sends, the other receives; when one partner offers a selection, the other chooses; when a partner closes the session, no further interactions may be initiated from either side in the same channel. Sessions are initiated when a participant invokes a server. A server acts as a shared service provider, with the capability of unboundedly spawning fresh sessions between the invoking client and a newly created service instance process. A service name may be publicly shared by any clients in the environment. In general, a session based system exhibits concurrency and parallelism in the sense that many sessions may be executing simultaneously and independently. Both session and server names may be passed around in communications. Although no races in communications within a session or even between different sessions can occur; processes may also concurrently invoke shared servers. It is easily understood that session channels are subject to a linear usage discipline, conforming to a specific state dependent protocol, while server channels can be freely shared, and invoked according to a simple new session creation protocol.

Caires and Pfenning (2010) introduced a type system for π-calculus processes that exactly corresponds to a linear logic proof system, revealing the first Curry-Howard interpretation of session types as linear logic propositions, a line of research further developed by Toninho, Perez and others. Unlike with traditional session type systems, the logical interpretation ensures unrestricted progress, meaning that well-typed processes never get stuck, as well as livelock freedom. The interpretation can be developed within either intuitionistic or classical linear logic, with certain subtle differences in expressiveness (Caires et al. 2012; Wadler 2012). We base the developments in this paper on the classical interpretation, as it represents rather directly the symmetries present both at the level of types and process interactions, given the presence of negation. The presented system thus establishes a Curry-Howard interpretation with classical linear logic, via the presentation Σ_2 of Andreoli (1992) extended with mix principles, which we also explore here for the first time. The structure is given by the types

$$A, B \ ::= \ \bot \ \mid \ \mathbf{1} \ \mid \ !A \ \mid \ ?A \ \mid \ A \otimes B \ \mid \ A \parr B \ \mid \ A \oplus B \ \mid \ A \,\&\, B$$

For each type A we define the dual $\overline{A}$ corresponding to the negation operator of linear logic $(\cdot)^{\perp}$, following basic de Morgan laws. Intuitively, the type of a session end-point is the dual of the type of the opposite session end-point.

$$\begin{array}{llll}
\overline{\mathbf{1}} = \bot & \overline{\bot} = \mathbf{1} & \overline{!A} = ?\overline{A} & \overline{?A} = !\overline{A} \\
\overline{A \otimes B} = \overline{A} \parr \overline{B} & \overline{A \parr B} = \overline{A} \otimes \overline{B} & \overline{A \oplus B} = \overline{A} \,\&\, \overline{B} & \overline{A \,\&\, B} = \overline{A} \oplus \overline{B}
\end{array}$$

Session Termination $\bot$ and $\mathbf{1}$ type terminated sessions, seen from each end-point partner. No well-typed partner will make further use of a terminated session. Both types are represented in traditional session type systems by a single type end. In the presence of mix principles, as we do here, we have $\bot \multimap \mathbf{1}$ and $\mathbf{1} \multimap \bot$, so we also consider $\bot = \mathbf{1}$, and write $\bullet$ for either one, hence $\overline{\bullet} = \bullet$ (recall $A \multimap B \triangleq \overline{A} \parr B$). We have the rules

$$\frac{}{\mathbf{0} \vdash x{:}\bullet; \Theta}\,(\text{T}\mathbf{1}) \qquad \frac{P \vdash \Delta; \Theta}{P \vdash x{:}\bullet, \Delta; \Theta}\,(\text{T}\bot)$$

The associated principal cut reduction corresponds to the structural congruence law $\mathbf{0} \mid P \equiv P$ rather than to a reduction step expressed by process synchronisation (structural congruence $\equiv$ is the basic identity on processes).

Send and Receive $A \otimes B$ is the type of a session that starts by sending a session of type A and then continues as a session of type B. It thus corresponds to the session type $A!.B$ (Honda 1993). Notice that $A \otimes B$ is essentially a pair of non-interfering types, which is what is essentially put to use in our session model. $A \parr B$ is the type of a session that starts by receiving a session of type A and then continues as a session of type B. It corresponds to the session type $\overline{A}?.B$ (so $A?.B$ corresponds to $A \multimap B$).

Session send and receive are respectively represented by π-calculus input and output guarded processes. An input process has the form $x(y).R$, representing a process that receives on session x a session n, passed in parameter y, and then proceeds as specified by R. The continuation R will use both the received session and other open sessions (including x). An output process has the form $\overline{x}(y).M$, where y is a freshly created name (this corresponds, without loss of expressiveness, to the output mechanism of the π-calculus with internal mobility). The behaviour of such a process is to send session y on x and then proceed as defined by M. In our typed language, M always has the form $P \mid Q$ (that is $M \equiv P \mid Q$ where P defines the behaviour of session y being sent, and Q the behaviour of the continuation session on x). This separation is key to ensure lock freedom, and does not limit expressiveness. Notice that y is bound both in $\overline{x}(y).M$ and in $x(y).R$. We then have the following typing rules.

$$\frac{P \vdash \Delta, y{:}A; \Theta \quad Q \vdash \Delta', x{:}B; \Theta}{\overline{x}(y).(P \mid Q) \vdash \Delta, \Delta', x{:}A \otimes B; \Theta}\ (\mathrm{T}\otimes) \qquad \frac{R \vdash \Gamma, y{:}C, x{:}D; \Theta}{x(y).R \vdash \Gamma, x{:}C \parr D; \Theta}\ (\mathrm{T}\parr)$$

The associated principal cut reduction corresponds to the expected (session passing) process synchronisation. If we consider $C = \overline{A}$ and $D = \overline{B}$ we obtain

$$\frac{\overline{x}(y).(P \mid Q) \vdash \Delta, \Delta', x{:}A \otimes B; \Theta \quad x(y).R \vdash \Delta, x{:}\overline{A} \parr \overline{B}; \Theta}{(\nu x)(\overline{x}(y).(P \mid Q) \mid x(y).R) \vdash \Delta, \Delta', \Gamma; \Theta}\ (\mathsf{Tcut})$$

which reduces to

$$\frac{\dfrac{P \vdash \Delta, y{:}A; \Theta \quad R \vdash \Gamma, y{:}\overline{A}, x{:}\overline{B}; \Theta}{(\nu y)(P \mid R) \vdash \Delta, \Gamma, x{:}\overline{B}; \Theta}\ (\mathsf{Tcut}) \quad \dfrac{}{Q \vdash \Delta', x{:}B; \Theta}}{(\nu x)((\nu y)(P \mid R) \mid Q) \vdash \Delta, \Delta', \Gamma; \Theta}\ (\mathsf{Tcut})$$

This corresponds to the standard input/output synchronisation in the (internal) π-calculus, expressing communication reduction (recall that $M \equiv P \mid Q$).

$$(\nu x)(\overline{x}(y).M \mid x(y).R) \to (\nu x)(\nu y)(M \mid R)$$

Offer and Choice $A \oplus B$ types a session that first chooses (from the partner menu offer) either “left” or “right”, and then continues as a session of type respectively either A or B. It thus corresponds to a pure, binary version, of the (choice) session type $\oplus_{i \in I}\{l_i : A_i\}$. $A \& B$ types a session that first offers (for partner to choose) both“left” or “right” menu options and then continues as a session of type respectively A or B, depending on the choice made. It corresponds to a pure, binary version, of the (offer) session type $\&_{i \in I}\{l_i : A_i\}$. The offer and choice primitives are typed by the additive connectives $\&$ and $\oplus$, namely linear conjunction and linear disjunction.

$$\frac{R \vdash \Delta, x{:}A; \Theta}{x.\mathtt{inl}; R \vdash \Delta, x{:}A \oplus B; \Theta}\ (\mathrm{T}\oplus_1) \quad \frac{R \vdash \Delta, x{:}B; \Theta}{x.\mathtt{inr}; R \vdash \Delta, x{:}A \oplus B; \Theta}\ (\mathrm{T}\oplus_2)$$

$$\frac{P \vdash \Delta, x{:}A; \Theta \quad Q \vdash \Delta, x{:}B; \Theta}{x.\mathtt{case}(P, Q) \vdash \Delta, x{:}A \mathbin{\&} B; \Theta}\ (\mathrm{T}\&)$$

The associated principal cut reductions corresponds to the session offer and choice process synchronization, as expressed by the reductions:

$$(\nu x)(x.\mathtt{case}(P, Q) \mid x.\mathtt{inl}; R) \to (\nu x)(P \mid R)$$
$$(\nu x)(x.\mathtt{case}(P, Q) \mid x.\mathtt{inr}; R) \to (\nu x)(Q \mid R)$$

In general, we may consider instead n-ary labeled sums, which are closer to the offer and choice constructs found in more traditional session types.

$$\frac{R \vdash \Delta, x{:}A; \Theta}{x.\mathtt{l_i}; R \vdash \Delta, x{:} \oplus_{i \in I} \{\mathtt{l_i} : A_i\}; \Theta} \quad \frac{P_i \vdash \Delta, x{:}A_i; \Theta \quad (\text{all } \ i \in I)}{x.\mathtt{case}_{i \in I}(\mathtt{l_i}.P_i) \vdash \Delta, x{:} \&_{i \in I} \{\mathtt{l_i} : A_i\}; \Theta}$$

In this case, the principal cut reduction corresponds to the process reduction

$$(\nu x)(x.\mathtt{case}_{i \in I}(\mathtt{l_i}.P_i) \mid x.\mathtt{l}_i; R) \to (\nu x)(P_i \mid R)$$

Shared Service Definition and Invocation Shared service definition and invocation are typed by the linear logic exponentials ! and ?. $!A$ types any shared channel that persistently publishes a replicated service which whenever invoked spawns a fresh session of type A (from the server behaviour perspective). Dually, type $?A$ types any shared channel on which requests to a persistently replicated service of type A can be unboundedly issued, from the client's perspective. We consider the rules:

$$\frac{P \vdash \Delta, y{:}A; u{:}A, \Theta}{\overline{u}(y).P \vdash \Delta; u{:}A, \Theta}\ (\mathsf{Tcopy}) \quad \frac{P \vdash \Delta; x{:}A, \Theta}{P \vdash \Delta, x{:}?A; \Theta}\ (\mathrm{T}?) \quad \frac{Q \vdash y{:}A; \Theta}{!x(y).Q \vdash x{:}!A; \Theta}\ (\mathrm{T}!)$$

The associated principal cut reduction corresponds to shared service invocation

$$(\nu x)(\overline{x}(y).P \mid !x(y).Q) \to (\nu x)((\nu y)(P \mid Q) \mid !x(y).Q)$$

Notice that rule T? is silent on the process, as it essentially corresponds to a book-keeping principle, moving the shared session channel to the exponential context, not corresponding to an actual reduction step in the process language.

Composition Rules Typed process composition principles are presented in our system by a set of orthogonal typing rules, which correspond to cut and mix principles.

$$\frac{}{\mathbf{0} \vdash ; \Theta}\ (\mathrm{T}\cdot) \quad \frac{P \vdash \Delta; \Theta \quad Q \vdash \Delta'; \Theta}{P \mid Q \vdash \Delta, \Delta'; \Theta}\ (\mathrm{T} \mid)$$

$$\frac{P \vdash \Delta, x{:}\overline{A}; \Theta \quad Q \vdash \Delta', x{:}A; \Theta}{(\nu x)(P \mid Q) \vdash \Delta, \Delta'; \Theta}\ (\mathsf{Tcut}) \quad \frac{P \vdash y{:}\overline{A}; \Theta \quad Q \vdash \Delta; u{:}A, \Theta}{(\nu u)(!u(y).P \mid Q) \vdash \Delta; \Theta}\ (\mathsf{Tcut}^?)$$

The mix rules (T$\cdot$) and (T $|$) express independent composition principles (grouping non-interfering sub-systems). The cut rules correspond to dependent composition principles (connecting subsystems through a selected communication channel). The cut principle appears in two forms, one for plugging a linear (session) channel (Tcut), other for plugging with a exponential (shared) channel (Tcut$^?$). For typing "source code" only the linear cut is required, but the exponential cut is required for the full system to enjoy cut-elimination (it thus corresponds to a "run-time" typing rule, in terms of operational semantics jargon). Notice that session type systems not based on logical principles (Honda et al. 1998; Gay and Hole 2005) usually embody composition principles quite differently, and fail to ensure a general progress property.

Forwarding We conveniently interpret the identity axiom by a primitive forwarder process $[x \leftrightarrow y]$, with several advantages (Caires et al. 2012), particularly when considering polymorphism (Caires et al. 2013). The forwarder at type A is typed by

$$[x \leftrightarrow y] \vdash x{:}A, y{:}\overline{A}; \Theta$$

The associated cut reduction $(\nu x)(P \mid [x \leftrightarrow y]) \to P\{y/x\}$ corresponds to explicit substitution application (y is not free in P). We assume $[x \leftrightarrow y] \equiv [y \leftrightarrow x]$ as a structural congruence axiom. We may represent π-calculus "free" output $\overline{x}y.P$ by $\overline{x}(z).([y \leftrightarrow z] \mid P)$ (cf. the internal mobility translation of Boreale (1998)).

Process reduction semantics The operational semantics of our typed session calculus exhibits a precise correspondence with cut elimination at the logic level. It is defined by a relation of reduction ($P \to Q$) expressing dynamic evolution, and a relation of structural congruence, which equates processes with the same spatial (or static) structure. While most cut-reduction steps directly correspond to process reductions, other cut-reduction steps are better expressed in the process world as structural congruence principles or behavioural equivalence principles, the same remark also applies to the so-called commuting conversions (which typically express typed behavioural equivalences (Pérez et al. 2012)). Behavioural equivalence equates processes presenting the same behaviour under all contexts (even if they differ in spatial structure).

Examples We consider a toy scenario involving a movie server and some clients. The first example models a single session (on channel s) implemented by a client $\mathit{Alice}(s)$ and a server instance $\mathit{SBody}(s)$. The server instance offers two options, a "buy movie" option (`inl`), and a "preview trailer" option (`inr`). Alice selects the "preview trailer" option from the server menu.

$$\begin{aligned}
&\mathit{SBody}(s) \triangleq s.\mathtt{case}(s(\mathit{title}).s(\mathit{card}).s\langle \mathit{movie}\rangle.\mathbf{0}, s(\mathit{title}).s\langle \mathit{trailer}\rangle.\mathbf{0})\\
&\mathit{Alice}(s) \triangleq s.\mathtt{inr}; s\langle \text{“}\mathit{solaris}\text{”}\rangle.s(\mathit{preview}).\mathbf{0}\\
&\mathit{System} \triangleq (\nu s)(\mathit{SBody}(s) \mid \mathit{Alice}(s))
\end{aligned}$$

Assuming some given types for movie titles (T), credit card data (C) and movie files M, types and type assignments for the various components are given by

$$\begin{array}{l} \mathit{SBT} \triangleq (T \multimap C \multimap M \otimes \mathbf{1}) \;\&\; (T \multimap M \otimes \mathbf{1}) \\ \mathit{SBody}(s) \vdash s : \mathit{SBT} \qquad \mathit{Alice}(s) \vdash s : \overline{\mathit{SBT}} \qquad \mathit{System} \vdash \cdot \end{array}$$

We may also consider a shared movie server and two concurrent clients. Now Alice still selects the "preview trailer" option, but Bob selects the "buy movie" option.

$$\begin{array}{l} \mathit{MOVIES}(\mathit{srv}) \triangleq !\mathit{srv}(s).\mathit{SBody}(s) \\ \mathit{Alice}(\mathit{srv}) \triangleq \overline{\mathit{srv}}(s).s.\mathtt{inr}; s\langle\text{“}\mathit{solaris}\text{”}\rangle.s(\mathit{preview}).\mathbf{0} \\ \mathit{Bob}(\mathit{srv}) \triangleq \overline{\mathit{srv}}(s).s.\mathtt{inl}; s\langle\text{“}\mathit{inception}\text{”}\rangle.s\langle \mathit{bobscard}\rangle.s(\mathit{moviefile}).\mathbf{0} \\ \mathit{System} \triangleq (\nu \mathit{srv})(\mathit{MOVIES}(\mathit{srv}) \mid \mathit{Alice}(\mathit{srv}) \mid \mathit{Bob}(\mathit{srv})) \end{array}$$

The following types and type assignments are now derivable

$$\begin{array}{l} \mathit{MOVIES}(\mathit{srv}) \vdash \mathit{srv} : !\mathit{SBT}; \quad \mathit{Alice}(\mathit{srv}) \vdash ; \mathit{srv} : \overline{\mathit{SBT}} \quad \mathit{Bob}(\mathit{srv}) \vdash ; \mathit{srv} : \overline{\mathit{SBT}} \\ \mathit{Alice}(\mathit{srv}) \mid \mathit{Bob}(\mathit{srv}) \vdash \mathit{srv} : ?\overline{\mathit{SBT}} \quad \mathit{System} \vdash \cdot \end{array}$$

We leave to the reader the fun exercise of reconstructing the various typing derivations.

3 Non-Determinism

The main obstacle to the conciliation of "internal" non-determinism within a logically motivated system interpreting proofs as processes is related to the fact that even if reduction steps at the programming language level may directly map into cut-elimination steps at the level of proofs, such cut-elimination steps express, in the first place, proof simplification or explicitation identities, towards a normal form. For example, in traditional functional interpretations, proofs are expressed in some λ-calculus. Then cut-elimination or proof reduction maps directly into some form of β-reduction, a computationally oriented version of some form of β-conversion, which is, in turn, a behavioural equivalence relation. Thus, a Curry-Howard correspondence directly relates proof reduction with program reduction, since both notions are coherent with conversion, which denotes proof equality. In general, we may understand every proof in the given logical system as a process or program satisfying the property denoted by the proposition it is a proof of, and proof reduction / cut-elimination, as the process that brings up explicitly the single underlying object of behaviour (the "normal form").

It is remarkable that the Curry-Howard interpretations of session types in linear logic (Caires and Pfenning 2010; Wadler 2012; Caires et al. 2012) follow these general principles, even if now the language is concurrent and expresses message passing distributed systems, and not functional computation (but see Toninho et al. (2012)). In fact, due to typing, reduction on session typed processes preserves observational equivalence, given their deterministic and deadlock free nature. It is important to highlight that the fact that reduction preserves observational equivalence not only already holds for traditional session type systems, but also for other typed fragments of π-calculi, where (at least certain sequences of) internal reduction steps preserve observational equivalence, e.g., if $P \to Q$ then $P \approx Q$ (cf. τ-inertness of Groote and Sellink (1996)). In the world of deterministic session typed processes internal reduction does not change its future, externally observable behaviour, which by type safety must conform with the prescribed session types of its free session channels. In particular, in the absence of additives (offer and choice), it is clear that the behaviour of a process is fully determined by its type. Even in the presence of additives, languages for pure session types (e.g. Honda et al. (1998)) forbid the possibility of truly non-deterministic session behaviour. Consider the process $S \triangleq x.\mathtt{case}(P, Q)$ so that $S \vdash x{:}\bullet \mathbin{\&} \bullet, y{:}\bullet \oplus \bullet$; where $P \triangleq y.\mathtt{inl}; \mathbf{0}$ and $Q \triangleq y.\mathtt{inr}; \mathbf{0}$. S can "non-deterministically" choose between `inl` or `inr` on session y. But, in fact, such choice is determined by the selection made by the environment on x, guarded by the $x.\mathtt{case}(P, Q)$ construct, excluding the possibility of "internal" choice. So, although the typing rule for offer

$$\frac{P \vdash \Delta, x{:}A; \Theta \quad Q \vdash \Delta, x{:}A; \Theta}{x.\mathtt{case}(P, Q) \vdash \Delta, x{:}A \mathbin{\&} A; \Theta} \text{(T\&)}$$

can express the alternative between processes P and Q, possibly of the same type as in S above (where $A = \bullet$, $P \vdash x{:}\bullet, y{:}\bullet \oplus \bullet$ and $Q \vdash x{:}\bullet, y{:}\bullet \oplus \bullet$) it cannot express real non-determinism: for any well-typed process communicating on x process S will always get a deterministic behaviour on y. A conceivable way to express "true" non-determinism at the logical level would be through a typed construct denoting (unguarded) internal choice. A possible rendering would be:

$$\frac{P \vdash \Delta, x{:}A; \Theta \quad Q \vdash \Delta, x{:}A; \Theta}{P \oplus Q \vdash \Delta, x{:}A; \Theta}$$

where $P \oplus Q$ would denote internal choice between behaviours P and Q. Intuitively, this typing (or proof) rule would express superposition of behaviours P and Q in the space of possibilities, as (T&) does, but replacing explicit external selection by internally decided non-deterministic choice. The question then arises about what should be the computational behaviour of $P \oplus Q$, usually defined by rules expressing the non-deterministic collapse of the space of possibilities into one singled out choice.

$$P \oplus Q \to P \qquad\qquad P \oplus Q \to Q$$

It is clear that reduction principles such as these cannot be accepted as adequate proof reduction principles, as one would loose preservation of observational equivalence under cut-elimination of the proof objects (processes in our case), crucial to obtain a sound and fully compositional logical interpretation of process behaviour. The counterpart of the two reduction principles depicted above as cut-elimination steps would be non-deterministic proof reductions

$$\frac{\dfrac{P \vdash \Delta, x{:}A; \quad Q \vdash \Delta, x{:}A;}{P \oplus Q \vdash x{:}A, \Delta;} \qquad R \vdash \Delta', x{:}\overline{A};}{(\nu x)(P \oplus Q \mid R) \vdash \Delta, \Delta';} \quad \begin{array}{l} \to \dfrac{P \vdash \Delta, x{:}A; \quad R \vdash \Delta', x{:}\overline{A};}{(\nu x)(P \mid R) \vdash \Delta, \Delta';} \\[2em] \to \dfrac{Q \vdash \Delta, x{:}A; \quad R \vdash \Delta', x{:}\overline{A};}{(\nu x)(Q \mid R) \vdash \Delta, \Delta';} \end{array}$$

But, course there is no room for obviously unsound behavioural equivalences such as $(\nu x)(P \mid Q) \approx (\nu x)(P \mid R)$ when $Q \not\approx R$, which excludes this naive approach. The fact that non-determinism may induce degeneracy in the computational interpretation is well known, and usually considered a serious, if not unsurmountable, obstacle to a Curry-Howard interpretation of non-determinism and concurrency. In this note we develop a seemingly unexplored avenue to frame this challenge, in the context of our session types interpretation of linear logic. The main idea involves two key ingredients, clearly related to familiar concepts in concurrency and programming languages.

The first ingredient, involves admitting processes denoting alternative potential behaviours among the various process forms under consideration. These processes are subject to behaviour preserving reduction rules, which are compatible with cut-elimination, and support compositional reasoning about process behavior. A typed process then represents a possibly non-deterministic behaviour, denoting all the possible alternatives, necessary for compositionally and compatibility with equational reasoning. Such alternatives "overlap" in the sense that they share linear resources - this sharing is not unsound since only some alternatives will be "actual", cf. the rules for the additive $A \,\&\, B$. We should then interpret an "actual" execution step as a choice among the possible paths in the non-deterministic space of possibilities, a concrete observation that eliminates other competing alternatives. Such a non-deterministic execution step should not however be regarded at the same level as the laws that govern the global description of the non-deterministic system, which are essentially captured by rules compatible with cut-elimination, but instead as something that falls outside the logical explanation. By analogy, we recall the mathematical structure of models of non-determinism, e.g., the power domain or power set constructions in denotational (compositional) accounts of non-deterministic computation (Plotkin 1976), or even the wave function model of quantum systems, which does not attempt to explain the (non-deterministic) collapse that occurs at the concrete observation step. As in such settings, we should nevertheless be obliged to formally relate the concrete non-deterministic behaviour, each possible observed behaviour, within the intended general model, without confusing the role of both.

The second ingredient, involves capturing non-determinism in the type structure, thus clearly and cleanly separating, as a proper conservative extension of the basic system, non-deterministic behaviour from the basic deterministic one. To achieve this, we encapsulate non-determinism within a monadic presentation supported by the type operators $\&A$ and $\oplus A$, naturally related by duality ($\overline{\&A} = \oplus\overline{A}$). The basic intuition is that $\&A$ represents the type of a session that may *non-deterministically* choose to produce some behaviour conforming to type A. On the other hand, $\oplus A$ represents the type of a session that may safely interact with (or consume, say) any non-deterministic behaviour of type A. Non-determinism is thus encapsulated inside the $\&A$ monad. In particular, any process typed in the basic (deterministic) fragment will be subject to the usual discipline of (deterministic) session typed processes discussed above, in a precise sense. We present the typing rules for these new logical / type operators.

$$\frac{P \vdash \Delta, x{:}A; \Theta}{P \vdash \Delta, x{:}\&A; \Theta}\,(\mathrm{T}\&_1) \quad \frac{P \vdash \&\Delta; \Theta \quad Q \vdash \&\Delta; \Theta}{P \oplus Q \vdash \&\Delta; \Theta}\,(\mathrm{T}\&_2) \quad \frac{P \vdash \&\Delta, x{:}A; \Theta}{P \vdash \&\Delta, x{:}\oplus A; \Theta}\,(\mathrm{T}\oplus)$$

The T$\&_1$ rule expresses that any (possibly linear) session can be coerced to a non-deterministic one, corresponding to the monadic unit $A \multimap \&A$. The rule is silent on the typed process P, it just acts at the level of types: although it would be possible to formulate a non-silent interpretation of our non-deterministic types, we prefer not to do so in this note, as that may seem a bit artificial when viewed from the process model perspective, and of doubtful pedagogical utility. The crucial mechanism allowing true non-determinism in the system is encapsulated in the T$\&_2$ typing rule, which also allows the internal choice operator to be introduced at the level of processes. The T$\&_2$ rule requires all linear sessions to be assigned a (producing) non-deterministic type $\&A$; this seems essential for cut-elimination (and type preservation) and overall soundness of our interpretation. We will get back to this point below. An informal explanation of the T$\oplus$ typing rule can be given in fairly intuitive terms: to soundly accept a non-deterministic behaviour at session x, process P must be already willing to offer non-deterministic behaviour at every other open session.

The cut elimination step for the $\&_1/\oplus$ redex essentially advances the session type of x from the non-deterministic view (of type $\&A$) to the deterministic view (of type A). As a design option, we interpret this reduction silently at the level of processes.

$$\frac{\dfrac{P \vdash \&\Delta, x{:}\overline{A}; \Theta}{P \vdash \&\Delta, x{:}\oplus\overline{A}; \Theta} \quad \dfrac{Q \vdash \Delta', x{:}A; \Theta}{Q \vdash \Delta', x{:}\&A; \Theta}}{(\nu x)(P \mid Q) \vdash \&\Delta, \Delta'; \Theta} \quad \to \quad \frac{P \vdash \&\Delta, x{:}\overline{A}; \Theta \quad Q \vdash \Delta', x{:}A; \Theta}{(\nu x)(P \mid Q) \vdash \&\Delta, \Delta'; \Theta}$$

The cut reduction step involving the $\&_2/\oplus$ pair is far more interesting. We thus have

$$\frac{P \vdash \&\Delta, x{:}\oplus\overline{A}; \Theta \quad \dfrac{Q \vdash \&\Delta', x{:}\&A; \Theta \quad R \vdash \&\Delta', x{:}\&A; \Theta}{Q \oplus R \vdash \&\Delta', x{:}\&A; \Theta}}{(\nu x)(P \mid (Q \oplus R)) \vdash \&\Delta, \&\Delta'; \Theta}$$

$$\to$$

$$\frac{\dfrac{P \vdash \&\Delta, x{:}\oplus\overline{A}; \Theta \quad Q \vdash \&\Delta', x{:}\&A; \Theta}{(\nu x)(P \mid Q) \vdash \&\Delta, \&\Delta'; \Theta} \quad \dfrac{P \vdash \&\Delta, x{:}\oplus\overline{A}; \Theta \quad R \vdash \&\Delta', x{:}\&A; \Theta}{(\nu x)(P \mid R) \vdash \&\Delta, \&\Delta'; \Theta}}{(\nu x)(P \mid Q) \oplus (\nu x)(P \mid R) \vdash \&\Delta, \&\Delta'; \Theta}$$

In our correspondence between proofs and processes, this cut reduction step gives rise to a structural congruence principle rather than to a dynamic reduction step: it clearly expresses a behavioural equivalence law (distribution of parallel composition over choice), not explicitly involving any process interaction.

$$(\nu x)(P \mid (Q \oplus R)) \equiv (\nu x)(P \mid Q) \oplus (\nu x)(P \mid R)$$

This principle resembles the expansion law of CCS equational theory, which also distributes parallel compositions over choices. The structural congruence $\mathbf{0} \oplus \mathbf{0} \equiv \mathbf{0}$ also follows from the expected proof reduction; remarkably, from other available commuting conversions (Caires and Pfenning 2010; Pérez et al. 2012) we may derive the equation $P \oplus P \approx P$, for any well-typed P (idempotence of choice).

The conditions enforced by our typing rules for the non-deterministic fragment seem difficult to relax. For example, let $\mathbf{B} = \bullet \oplus \bullet$ (cf., a type of "booleans"), and consider the following typing

$$\dfrac{\dfrac{x.\texttt{inl}; y.\texttt{inr}; \mathbf{0} \vdash x{:}\mathbf{B}, y{:}\mathbf{B}}{x.\texttt{inl}; y.\texttt{inr}; \mathbf{0} \vdash x{:}\&\mathbf{B}, y{:}\&\mathbf{B}} \quad \dfrac{x.\texttt{inl}; y.\texttt{inl}; \mathbf{0} \vdash x{:}\mathbf{B}, y{:}\mathbf{B}}{x.\texttt{inl}; y.\texttt{inl}; \mathbf{0} \vdash x{:}\&\mathbf{B}, y{:}\&\mathbf{B}}}{x.\texttt{inl}; y.\texttt{inr}; \mathbf{0} \oplus x.\texttt{inl}; y.\texttt{inl}; \mathbf{0} \vdash x{:}\&\mathbf{B}, y{:}\&\mathbf{B}}\ (\mathrm{T}\&_2)$$

The behaviour on session y is clearly non-deterministic, it can "collapse" on either $y.\texttt{inl}$ or $y.\texttt{inr}$: one may think of $\&\mathbf{B}$ as a type of (linear) non-deterministic booleans. Although the behaviour on session x is deterministic, the composition rule requires all sessions to be conservatively typed as non-deterministic, so that the whole process is assigned type $x{:}\&\mathbf{B}, y{:}\&\mathbf{B}$. For the sake of argument, suppose the conditions on our typing rules where relaxed, allowing non-deterministic typing of single sessions

$$\dfrac{\dfrac{x.\texttt{inl}; y.\texttt{inr}; \mathbf{0} \vdash x{:}\mathbf{B}, y{:}\mathbf{B}}{x.\texttt{inl}; y.\texttt{inr}; \mathbf{0} \vdash x{:}\mathbf{B}, y{:}\&\mathbf{B}} \quad \dfrac{x.\texttt{inl}; y.\texttt{inl}; \mathbf{0} \vdash x{:}\mathbf{B}, y{:}\mathbf{B}}{x.\texttt{inl}; y.\texttt{inl}; \mathbf{0} \vdash x{:}\mathbf{B}, y{:}\&\mathbf{B}}}{x.\texttt{inl}; y.\texttt{inr}; \mathbf{0} \oplus x.\texttt{inl}; y.\texttt{inl}; \mathbf{0} \vdash x{:}\mathbf{B}, y{:}\&\mathbf{B}}\ (\mathrm{T}\&_2)$$

Then, in general we would also need to accept the typing

$$\dfrac{\dfrac{x.\texttt{inl}; y.\texttt{inr}; \mathbf{0} \vdash x{:}\mathbf{B}, y{:}\mathbf{B}}{x.\texttt{inr}; y.\texttt{inr}; \mathbf{0} \vdash x{:}\mathbf{B}, y{:}\&\mathbf{B}} \quad \dfrac{x.\texttt{inl}; y.\texttt{inl}; \mathbf{0} \vdash x{:}\mathbf{B}, y{:}\mathbf{B}}{x.\texttt{inr}; y.\texttt{inl}; \mathbf{0} \vdash x{:}\mathbf{B}, y{:}\&\mathbf{B}}}{x.\texttt{inl}; y.\texttt{inr}; \mathbf{0} \oplus x.\texttt{inr}; y.\texttt{inl}; \mathbf{0} \vdash x{:}\mathbf{B}, y{:}\&\mathbf{B}}\ (\mathrm{T}\&_2)$$

After composition with a process of type $y{:}\oplus\overline{\mathbf{B}}$, say $Q \triangleq y.\texttt{case}(\mathbf{0}, \mathbf{0})$, we would obtain $(\nu y)(x.\texttt{inl}; y.\texttt{inr}; \mathbf{0} \oplus x.\texttt{inr}; y.\texttt{inl}; \mathbf{0} \mid Q) \vdash x{:}\mathbf{B}$. This typing would be invalid, since we would be giving a non-deterministic process a deterministic type (outside the monad $\&-$). A similar reasoning justifies the form of typing rule (T$\oplus$). Notice however that one may also express the intended x-deterministic / y-non-deterministic type anyway, but using a different process, as follows:

$$\dfrac{\dfrac{\dfrac{y.\texttt{inr}; \mathbf{0} \vdash y{:}\&\mathbf{B} \quad y.\texttt{inl}; \mathbf{0} \vdash y{:}\&\mathbf{B}}{y.\texttt{inr}; \mathbf{0} \oplus y.\texttt{inl}; \mathbf{0} \vdash y{:}\&\mathbf{B}}\ (\mathrm{T}\&_2)}{y.\texttt{inr}; \mathbf{0} \oplus y.\texttt{inl}; \mathbf{0} \vdash x{:}\bullet, y{:}\&\mathbf{B}}\ (\mathrm{T}\bot)}{x.\texttt{inl}; (y.\texttt{inr}; \mathbf{0} \oplus y.\texttt{inl}; \mathbf{0}) \vdash x{:}\mathbf{B}, y{:}\&\mathbf{B}}\ (\mathrm{T}\oplus_1)$$

Moreover, in some cases it is possible to assign a purely deterministic type to a process with actual, but hidden, non-deterministic computations: of course, soundness of our interpretation ensures that no non-deterministic behaviour can be externally observable on such a process. This fine grained compositional control of non-determinism is a natural consequence of our monadic interpretation. Consider some basic laws enjoyed by the non-deterministic type constructs, given by the following derivable typings

$$\begin{array}{ll}
[x \leftrightarrow y] \vdash x{:}\oplus\oplus\overline{A}, y{:}\&A; & (\text{cf. } \&\&A \multimap \&A) \\
[x \leftrightarrow y] \vdash x{:}\overline{A}, y{:}\&A; & (\text{cf. } A \multimap \&A) \\
[x \leftrightarrow y] \vdash y{:}\&\overline{A}, x{:}A, ; & (\text{cf. } \oplus A \multimap A) \\
[x \leftrightarrow y] \vdash x{:}\oplus\overline{A}, y{:}\&A; & (\text{cf. } \&A \multimap \&A) \\
[x \leftrightarrow y] \vdash x{:}\&\overline{A}, y{:}\oplus A; & (\text{cf. } \oplus A \multimap \oplus A) \\
\mathbf{0} \vdash x{:}\oplus\bullet, y{:}\bullet; & (\text{cf. } \&\bullet \multimap \bullet)
\end{array}$$

Notice that although we have $\&\bullet \multimap \bullet$ and $\bullet \multimap \oplus\bullet$, we cannot in general (that is, for any type A) derive a typing of the form $R \vdash z{:}\&A \multimap A$, as that would signal a "leak" in the non-deterministic monad. The special case of the unit $\bullet$ is sound since no behaviour apart from the trivial one is involved. This relates to the absence of information flow at the unit type in the context of information flow type systems (e.g, Crary et al. (2005)), where a high security value of unit type can be safely declassified to low security, since no interference may arise at that particular type. In our setting, there is also no real non-deterministic behaviour of $\bullet$ type (since there is no observable behaviour of $\bullet$ type at all). The fact that several of the laws above are realised by a simple forwarder $[x \leftrightarrow y]$ should not come as a surprise, given the silent interpretation (at the level of the processes) of our non-deterministic (monadic) type operators.

It is interesting to note the (at least superficial) similarity between the rules for $\oplus A$ and $\&A$ and those for the exponentials $!A$ and $?A$ respectively. A key difference is visible in the "contraction principle" expressed by T$\&_2$, which works along the space of possible alternatives (in terms of grouping overlapping states, possibly sharing linear resources), instead of along the space of shared usages (in terms of fusing replicated behaviours, not depending on linear resources). This fact also fundamentally separates our approach from the differential linear logic of Ehrhard and Regnier (2006), which may also support a logical interpretation of non-determinism by interpreting all proofs as sets (more precisely as linear combinations) of "simple" proofs. Our logic is based on quite different principles, as we encapsulate internal non-determinism in new connectives related to the additives of linear logic, while differential linear logic modifies the interpretation of the exponentials, namely $!A$, by adding the rule of co-contraction. That suggests a computational model in which non-determinism only applies to replicated servers, thus quite different from ours. In our model, co-contraction may conceivably be represented in a finer grained way by deriving $\&!A$ via T$\&_2$.

Examples We get back to our movie server scenario, and illustrate how to model a system with a client $\mathit{Randy}(s)$ that non-deterministically decides between either actually buying a movie or just seing its trailer. Essentially, we have

$$Randy(s) \triangleq Alice(s) \oplus Bob(s) \qquad USystem \triangleq (\nu s)(SBody(s) \mid Randy(s))$$

where the suitable types and type assignments are now given by

$$\begin{array}{l} SBT \triangleq (T \multimap C \multimap M \otimes \mathbf{1}) \ \& \ (T \multimap M \otimes \mathbf{1}) \\ SBody(s) \vdash s : \oplus SBT \qquad Randy(s) \vdash s : \&\overline{SBT} \qquad USystem \vdash \cdot \end{array}$$

Consider now a variant server that logs requests on a log service l of type $\mathbf{B}$. We then obtain the following typings, where of course the visible behaviour at the log channel l must be given the non-deterministic type $\&\mathbf{B}$.

$$\begin{array}{l} SBodyL(s) \triangleq s.\texttt{case}(s(title).s(card).s\langle movie\rangle.l.\texttt{inl};\mathbf{0}, s(title).s\langle trailer\rangle.l.\texttt{inr};\mathbf{0}) \\ SBodyL(s) \vdash s{:}SBT, l{:}\mathbf{B} \qquad SBodyL(s) \vdash s{:} \oplus SBT, l{:} \& \mathbf{B} \qquad USystem \vdash l{:} \& \mathbf{B} \end{array}$$

It would be easy to extend this system to a shared setting, to illustrate other interesting non-deterministic behaviours. For example, we could consider a shared movie server that non-deterministically offers to the client a (possibly) randomly chosen trailer of the movie title asked for. Such a shared movie server would then be given type

$$SRBT \triangleq !(T \multimap C \multimap M \otimes \mathbf{1}) \ \& \ (T \multimap (\&M) \otimes \mathbf{1})$$

In our next example, we illustrate in a very simple setting how some systems encapsulating non-deterministic behaviour can nevertheless be given a globally deterministic type, thus showing that the given internal non-determinism is not observable at public channels. Consider the following processes and typings:

$$\begin{array}{ll} Some(y) \triangleq y.\texttt{inl};\mathbf{0} \oplus y.\texttt{inr};\mathbf{0} & Some(y) \vdash y : \&\mathbf{B} \\ Prod \triangleq \overline{x}(y).(Some(y) \mid b\langle\text{“}done\text{”}\rangle.\mathbf{0}) & Prod \vdash x : (\&\mathbf{B}) \otimes \bullet, b : String \otimes \bullet \\ Cons \triangleq x.\texttt{case}(\mathbf{0},\mathbf{0}) & Cons \vdash x{:}(\oplus\mathbf{B}) \parr \bullet \\ Plug \triangleq (\nu x)(Prod \mid Cons) & Plug \vdash b : String \otimes \bullet \end{array}$$

Notice that the although the producer $Prod$ sends (on x) a non-deterministic boolean $Some(y)$ (on y) to the consumer $Cons$, the type of system $Plug$ is $b : String \otimes \bullet$, a deterministic type. In fact, we may easily verify that $Plug$ always reduces to $b\langle\text{“}done\text{”}\rangle.\mathbf{0}$. In Fig. 1 we summarise our process language, reduction, and structural congruence.

4 Main Results

We collect in this section a preliminary analysis of our non-deterministic linear logic-based type system for session process behaviour. First, our system enjoys the cut-elimination property, which we may express in our setting as follows, given a suitable defined observational congruence $\cong_s$ on processes which includes reduction, structural congruence, and some necessary commuting conversions (along the lines of Caires and Pfenning (2010); Pérez et al. (2012); Caires et al. (2012)).

(Processes)

$$\begin{array}{rcl} P & ::= & [x \leftrightarrow y] \mid P|Q \mid (\nu y)P \mid \overline{x}(y).P \mid x(y).P \mid !x(y).P \\ & \mid & x.\texttt{case}(P,Q) \mid x.\texttt{inr};P \mid x.\texttt{inl};P \mid P \oplus Q \mid \mathbf{0} \end{array}$$

(Reduction)

$$\begin{array}{ll} \overline{x}(y).Q \mid x(y).P \to (\nu y)(Q \mid P) & \overline{x}(y).Q \mid !x(y).P \to (\nu y)(Q \mid P) \mid !x(y).P \\ (\nu x)([x \leftrightarrow y] \mid P) \to P\{y/x\} & Q \to Q' \Rightarrow P \mid Q \to P \mid Q' \\ P \to Q \Rightarrow (\nu y)P \to (\nu y)Q & P \equiv P',\, P' \to Q',\, Q' \equiv Q \Rightarrow P \to Q \\ x.\texttt{inr};P \mid x.\texttt{case}(Q,R) \to P \mid R & x.\texttt{inl};P \mid x.\texttt{case}(Q,R) \to P \mid Q \end{array}$$

(Structural Congruence)

$$\begin{array}{c} P \mid \mathbf{0} \equiv P \qquad P \equiv_\alpha Q \Rightarrow P \equiv Q \qquad (\nu x)\mathbf{0} \equiv \mathbf{0} \qquad P \mid Q \equiv Q \mid P \\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad x \notin \mathit{fn}(P) \Rightarrow P \mid (\nu x)Q \equiv (\nu x)(P \mid Q) \\ [x \leftrightarrow y] \equiv [y \leftrightarrow x] \qquad (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \\ \mathbf{0} \oplus \mathbf{0} \equiv \mathbf{0} \qquad (\nu x)(P \mid (Q \oplus R)) \equiv (\nu x)(P \mid Q) \oplus (\nu x)(P \mid R) \end{array}$$

Figure 1: The Process Language.

THEOREM 1. *If $P \vdash \Delta; \Theta$ then there is a process Q such that $P \cong_s Q$ and $Q \vdash \Delta; \Theta$ is derivable without using the rules (T*cut*) and (T*cut$^?$*).*

Then, we state type safety, which is witnessed by theorems of preservation and progress (for closed systems).

THEOREM 2 (PRESERVATION). *If $P \vdash \Delta; \Theta$ and $P \to Q$ then $Q \vdash \Delta; \Theta$.*

We say that process P is live, noted $live(P)$ if and only if $P \equiv (\nu \overline{n})(\pi.Q \mid R)$ for some $\pi.Q, R, \overline{n}$ where $\pi.Q$ is a non-replicated action prefixed process (e.g, π is a simple session input, output, offer, or choice prefix). We then have

THEOREM 3 (PROGRESS). *If $P \vdash\, ; \Theta$ and $live(P)$ then there is Q such that $P \to Q$.*

The following results clarify some key features of the our type system. We say that a process P is prime if it is not structurally congruent to a process of the form $Q \oplus R$ with non-trivial (e.g, equivalent to $\mathbf{0}$) Q and R. We also denote by $P \Rightarrow Q$ the reflexive-transitive closure of $P \to Q$. We can prove the following property:

PROPOSITION 4. *Let $P \vdash \Delta; \Theta$ where types in Δ are deterministic (do not contain $\&A$ or $\oplus A$ types at the top level, and let $P \Rightarrow Q \not\to$. Then Q is prime.*

Being based on a logical system in which reduction is deeply related with cut-elimination, it turns out that typed processes enjoy the confluence property, and in fact also strong normalisation. The proof of these results can be established using for instance logical relations, along the lines of Pérez et al. (2012).

Confluence holds because, as discussed above, non-determinism is captured without losing information, by means of delaying choice in processes of the form $P \oplus Q$, which express alternative (overlapping in time) states. It is nevertheless interesting to relate our system with extensions of it with reduction rules explicitly collapsing non-deterministic states in prime states. For that purpose, we consider the extension of the basic reduction relation defined in Figure 1 with standard rules for internal choice, namely $P \oplus Q \to P$ and $P \oplus Q \to Q$. We denote by $P \to_c Q$ the extended reduction relation, which still satisfies preservation and progress in the sense of Theorems 2 and 3. We may then show the following result, expressing postponing of internal non-deterministic collapse of non-deterministic states into prime states.

THEOREM 5 (POSTPONING). *Let* $P \vdash \Delta; \Theta$. *We have*

1. If $P \Rightarrow P_1 \oplus \ldots \oplus P_n \not\to$ *with* P_i *prime for all* i*, then* $P \Rightarrow_c P_i$ *for all* $0 < i \leq n$.

2. Let $\mathcal{C} = \{P_i \mid P \Rightarrow_c P_i \not\to_c$ *and* P_i *is prime* $\}$. *Then* $\mathcal{C}$ *is finite up to* $\equiv$*, with* $\#\mathcal{C} = n$*, and for all* $0 < i \leq n$, $P \Rightarrow P_1 \oplus \ldots \oplus P_n \to_c P_i$.

We can therefore tightly relate the system based on pure logically motivated reduction with the system extended with the standard (non-logical, non-confluent) reduction rules for internal choice, in the sense that the former precisely captures the set of observable alternatives defined by the latter, while preserving compositional and equational reasoning about the system behavior.

5 Concluding Remarks

We have sketched an approach to accommodate internal non-determinism in a logically motivated behavioural type system for concurrent processes, in the setting of a Curry-Howard correspondence of session types and linear logic. Distinguishing aspects of our contribution is the embedding of non-determinism inside a logical system by the introduction of superposed states motivated by the additive rules of linear logic, disciplined by specific type operators $\&A$ and $\oplus A$. Apart from the foundational contribution, we should also mention that our approach also adds to the flexibility of the standard session paradigm, in which deterministic and (externally determined) non deterministic phases are sharply distinguished. Our system allows lock-free, confluent programs with richer combinations of determinism and non determinism, preserving compatibility with observable collapsing non-determinism (in the sense of Theorem 5). We expect our development to raise many other interesting questions, not only about expressiveness, but also about induced observational equivalences, its combination with recursion, and its compatibility with stochastic models of non-determinism.

Acks. Thanks to M. Abadi, P. Gardner, A. Gordon, and R. Mardare for inviting this contribution, and to Pfenning, Toninho, and Perez for many related discussions.

References

S. Abramsky. Computational Interpretations of Linear Logic. *Theoret. Comput. Sci.*, 111(1–2):3–57, 1993.

J.-M. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *J. Log. Comput.*, 2(3):297–347, 1992.

M. Boreale. On the Expressiveness of Internal Mobility in Name-Passing Calculi. *Theor. Comput. Sci.*, 195(2):205–226, 1998.

L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part II). *Theor. Comput. Sci.*, 3(322):517–565, 2004.

L. Caires and F. Pfenning. Session Types as Intuitionistic Linear Propositions. In *CONCUR'10*, number 6269 in LNCS, pages 222–236, 2010.

L. Caires, F. Pfenning, and B. Toninho. Linear Logic Propositions as Session Types. *Math. Struct. in Comp. Sci.*, 2012. to appear.

L. Caires, J. A. Pérez, F. Pfenning, and B. Toninho. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *ESOP'13*, number 7792 in LNCS, 2013.

L. Cardelli. Typeful Programming. *IFIP State-of-the-Art Reports: Formal Description of Programming Concepts*, pages 431–507, 1991.

L. Cardelli. On Process Rate Semantics. *Theor. Comput. Sci.*, 391(3):190–215, 2008.

K. Crary, A. Kliger, and F. Pfenning. A Monadic Analysis of Information Flow Security with Mutable State. *J. Funct. Program.*, 15(2):249–291, 2005.

T. Ehrhard and L. Regnier. Differential Interaction Nets. *Theor. Comput. Sci.*, 364(2): 166–195, 2006.

S. Gay and M. Hole. Subtyping for Session Types in the Pi Calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

J. F. Groote and M. P. A. Sellink. Confluence for Process Verification. *Theor. Comput. Sci.*, 170(1-2):47–81, 1996.

K. Honda. Types for Dyadic Interaction. In *CONCUR'93*, number 715 in LNCS, pages 509–523, 1993.

K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP'98*, number 1381 in LNCS, 1998.

J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho. Linear Logical Relations for Session-Based Concurrency. In *ESOP'12*, number 7211 in LNCS, 2012.

G. D. Plotkin. A Powerdomain Construction. *SIAM J. Comput.*, 5(3):452–487, 1976.

B. Toninho, L. Caires, and F. Pfenning. Functions as Session-Typed Processes. In *FoSSaCS'12*, number 7213 in LNCS, 2012.

P. Wadler. Propositions as Sessions. In *ICFP'12*, ACM, pages 273–286, 2012.

What makes a biological clock efficient?

Attila Csikász-Nagy[1,2], Neil Dalchau[3]

[1] Randall Division of Cell and Molecular Biophysics and Institute for Mathematical and Molecular Biomedicine, King's College London, London SE1 1UL, United Kingdom

[2] Department of Computational Biology, Research and Innovation Center, Fondazione Edmund Mach, San Michele all'Adige 38010, Italy

[3] Microsoft Research, Cambridge CB1 2FB, United Kingdom

Abstract

Biological clocks regulate the proper periodicity of several processes at the cellular and organismal level. The cell cycle and circadian rhythm are the best characterized among these but several other biological clocks function in cells at widely variable periodicity. The underlying molecular networks are controlled by delayed negative feedbacks, but the role of positive feedbacks and substrate-depletion has been also proposed to play crucial roles in the regulation of these processes. Here we will investigate which features of biological clocks might be important for their efficient timekeeping.

Evolution of biological clocks

The ability of organisms to temporally co-ordinate their physiology is evolutionarily advantageous, and is therefore ubiquitous in nature. Organisms have evolved numerous so-called biological clocks to optimise their fitness, by co-ordinating their physiology with the availability of resources. Simple experiments monitoring the growth of bacteria against varying nutrient availability illustrate an enormous flexibility in deciding how frequently cells choose to divide. Yet, the developmental programs that lead to the replication of entire organisms (mammals) are relatively inflexible. The cell cycle, which culminates in cell division, is controlled by regulatory networks that have numerous conserved features and components across the eukaryotic kingdom (Harashima *et al.*, 2013).

Circadian clocks allow organisms to co-ordinate their physiology with the external time of day, enabling anticipation of changes in temperature, light availability, predator activity, etc. In contrast to the cell cycle, circadian clocks have evolved multiple times and have many different features (though some shared components) across the eukaryotic kingdom (Dalchau and Webb, 2011). The overall structure of circadian networks involves input pathways, a core oscillator, and output pathways (Dunlap, 1999). The core oscillator comprises multiple feedback loops that sustain circadian rhythms with a period of approximately 24 h. Input pathways enable the oscillator to maintain synchrony with external time, while output pathways provide the biochemical means of the oscillator to regulate downstream physiology, including gene expression, metabolism and signalling.

Synthesizing biological clocks

In recent years, there has been a large rise in the number of attempts to engineer biological systems. The field of *synthetic biology* seeks to improve understanding of biological functions by attempting to re-create specific systems and their behaviours, using existing cells and their housekeeping components (RNA polymerase, ribosomes, proteasomes, etc.) as a chassis. The creation of *biological devices* is beginning to open new opportunities in industry, for example using bacteria to produce biofuels and medicines. A seminal work in this field was the construction of a biological clock, termed the *repressilator*, in which three transcriptional repressors were taken from non-oscillatory networks and inserted into *Escherichia coli* on a plasmid, but arranged as a cycle of repression (Elowitz and Leibler, 2000). Briefly, TetR was placed under the control of a LacI-repressible promoter, LacI was placed under the control of a CI-repressible promoter, and CI under the control of a TetR-repressible promoter. It was demonstrated that oscillations in the abundance of the constituent proteins could be generated when the strengths of the interactions between the repressor proteins and their cognate DNA-binding domains were tuned to appropriate levels.

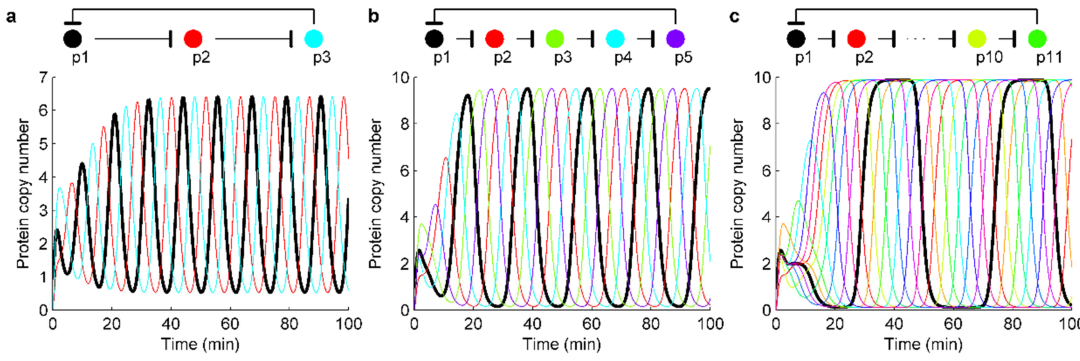


Figure 1. Deterministic simulation of ring oscillators. Sequential inhibition of transcriptional repressors around a single feedback loop produces oscillations. **a.** Simulation of the repressilator (3-component ring oscillator) model in Elowitz & Leibler. **b,c.** Extending the model to 5 and 11 components also yields oscillations in protein copy number. Protein 1 is plotted with a thickened black line to emphasize differences in oscillation waveform between different degree ring oscillators.

The repressilator network is an example of a 3-stage *ring oscillator*. Ring oscillators are often used in electrical engineering for generating oscillations. However, only rings with an odd number of components can give oscillatory dynamics ((Sprinzak and Elowitz, 2005); examples in Figure 1). This is because each regulator inverts the gradient of the following regulator, which for an even number of components would result in an equilibrium ON-OFF-ON-OFF-…-ON-OFF. Using an odd number of components breaks this pattern, and can yield oscillations. Increasing the length of the feedback loop with additional components leads to more square-like waveforms (Figure 1). It has recently been established that a repressilator motif also exists in nature, lying at the heart of the circadian clock in plants (Pokhilko *et al.*, 2012) and in the core of transcriptional regulation of the cell cycle (Sriram *et al.,* 2007).

Oscillators have also been created in mammals (Tigges *et al.*, 2009) as well as in cell-free conditions, mixing chemical compounds in such a way as to recapitulate interaction networks that can exhibit oscillatory behaviour, for instance using negative feedback. In various works dating back to the 1950s, the famous Belousov-Zhabotinsky reaction was shown to both oscillate in time and propagate over excitable media (Field *et al.*, 1972). More recently, the construction of chemical oscillators made from DNA has been demonstrated, inspired by predator-prey (PP) cycles (Fujii and Rondelez, 2012). A system of 3 reactions is sufficient to generate PP cycles: i. an autocatalytic growth of the prey species, ii. an autocatalytic predation of the prey, and iii. decay of the predator species. The DNA-based PP network relies on DNA polymerization-depolymerization reactions to recapitulate these reactions, and is capable of sustaining many (>20) cycles before eventual depletion of necessary cofactors.

Mathematical analysis of biological clock architectures

There is a long history on the mathematical analysis of biological clocks (Goldbeter, 1997, 2002), still it is not fully understood what makes such a periodic system efficient. Biological clocks can run with a period of seconds (neural, cardiac, calcium rhythms) to months and years (ovarian, annual and ecological rhythms) and are regulated by delayed negative feedback loops that cause oscillations in the activities of system components (Goldbeter, 2008). Direct negative feedback loops lead to stabilization of steady states but delay in the loop and non-linearity in the interactions can induce oscillations (Goodwin, 1965; Griffith, 1968). This generic rule that delayed negative feedback loops form the basis of biological oscillations is now very well established for many biological clocks (Fig. 2A). The daily rhythms of the circadian clock might be the best example, where it was established that the existence of a direct time delay caused by a transcriptional-translational loop is driving the periodic appearance of a transcriptional repressor (Dunlap, 1999). Interestingly it was recently revealed that even in the absence of the delay caused by transcription-translation the circadian clock is robustly ticking (Nakajima *et al.*, 2005; O’Neill *et al.*, 2011). Later it was proposed that a positive feedback loop might play a crucial role in the control of this reduced system (Mehra *et al.*, 2006). Indeed the importance of positive feedback loops in the robustness of circadian clock regulation was proposed at other places as well (Tyson *et al.*, 1999; Becker-Weimann *et al.*, 2004; Hong *et al.*, 2009). These led to the conclusion that the circadian clock is controlled by interactions of positive and negative feedback loops.

Another highly investigated biological clock is driven by the cell cycle regulatory network. The controlled timing of DNA replication and cell division is determined by this clock and again earliest models considered a delayed negative feedback loop to drive this system (Goldbeter, 1991) and later results revealed the importance of positive feedback loops as well (Pomerening *et al.*, 2005; Tsai *et al.*, 2014). Thus it is a reoccurring pattern that crucial biological clocks are regulated by interlinked positive and negative feedback loops (Tsai *et al.*, 2008; Ferrell Jr *et al.*, 2011).

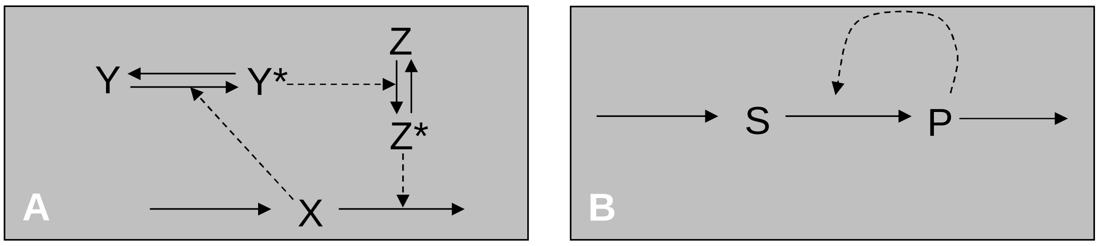


Figure 2. Feedback loops leading to oscillations. A, negative feedback loop, where protein X activates Y, which activates Z which is eventually inducing the degradation of X. **B,** Substrate-depletion, where a substrate S is produced and piling up in this form until the product P cannot turn on its autocatalytic loop converting most S into P. As P is less stable than S, the system runs out of both S and P, thus S will pile up again and oscillations emerge.

In the case of glycolytic oscillations of the metabolic system it was proposed very early that a positive feedback loop has a crucial role in controlling this biological clock and the oscillations appear as a result of the depletion of the substrate of an autocatalytic process (Higgins, 1964; Sel'kov, 1968). In this system a stable substrate is produced and converted into an unstable product in an autocatalytic manner (Fig. 2B) leading to oscillations where S is slowly increasing until P reaches a threshold and quickly converts all S into P (Fig. 3). The requirements for this system to oscillate are: i, non-linear autocatalysis on the S→P transition, ii, a background S→P conversion independent of P to allow P reaching the threshold and iii, removal rate of P has to be much higher than that of S. Note that this system shows high resemblance to the above mentioned predator-prey cycles. In both cases the pile up of one species is followed by the conversion of this species to another species by an autocatalytic step and eventual removal of the second species.

Interestingly one of the earliest cell cycle models was also working as a substrate-depletion oscillator (Tyson, 1991) and since then it was further established that the kinetics of the substrate-depletion model resemble that of the negative feedback with positive feedback model (Fall *et al.*, 2002). Indeed one can see the delayed negative feedback in the substrate-depletion model as P removes its activator S (by converting it to P). Thus we could state again that interlocked positive and negative feedbacks regulate glycolytic oscillations. It is also important to mention that the substrate-depletion mechanism that leads to oscillations in time can drive spatial biological clocks such as pattern formation and emergence of travelling waves (Meinhardt, 1982).

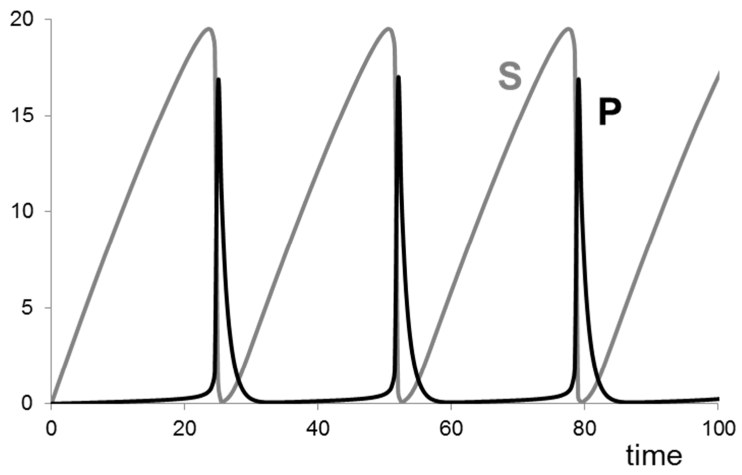


Figure 3. Deterministic simulation of a substrate-depletion oscillator. The substrate S is produced and first slowly converted into P. When P reaches a threshold it converts all S to P, which gets quickly destroyed. Leading to a bursting-like pattern in P oscillations.

Efficiency of biological clocks

Going back to the original question in the title: what makes biological clocks efficient? In fact, how to measure the efficiency of biological clocks? The robustness of the periodicity of biological clocks were investigated in the context of the circadian rhythm (Barkai and Leibler, 2000; Gonze *et al.*, 2002) and the cell cycle (Steuer, 2004; Mura and Csikász-Nagy, 2008). Both were found to be quite robust to parameter perturbations and also to intrinsic noise resulting from the low molecular numbers present in the system. So far we have seen many parallels between the circadian clock and cell cycle regulatory systems. There is one major point where they differ. The period of the circadian clock is quite insensitive for temperature changes whereas the cell cycle time can be greatly influenced by alterations in temperature (Klevecz and King, 1982). This result might suggest that the circadian rhythm regulatory network is a more efficient time keeper, while the cell cycle regulatory systems is more efficient in adjusting its period to adapt to environmental changes (Zámborszky *et al.*, 2007; Hong *et al.*, 2014). Changes in temperature affect chemical reactions exponentially, following the Arrhenius equation. How such changes in reaction rates do not influence the period of the circadian clock is a debated question (Tyson *et al.*, 2008). Several models have been worked out to understand what causes the temperature compensation in the circadian clock (Ruoff and Rensing, 1996; Leloup and Goldbeter, 1997; Gould *et al.*, 2006; Hong *et al.*, 2007; François *et al.*, 2012) and some more generic models of temperature compensation in biochemical reaction networks have also been proposed (Ruoff *et al.*, 1997; Hatakeyama and Kaneko, 2012). Recently even a synthetic temperature compensated oscillator was created (Hussain *et al.*, 2014), interestingly containing both a positive and a negative feedback loop. Furthermore non-biological ring oscillators on semiconductors were also designed to be temperature compensated

(Hayashi and Kondoh, 1993). Despite all of these results and theoretical ideas we still lack a coherent generic picture of what makes biological oscillators temperature compensated and in general robust in proper periodicity.

Conclusions

Recently it was established by Cardelli and Csikász-Nagy that a class of biological switches follow the dynamical features of an efficient computational algorithm (Cardelli and Csikász-Nagy, 2012). The Approximate Majority (AM) algorithm is used in distributed computing as a population protocol computing the majority of two finite populations by converting the minority population into the majority population (Angluin *et al.*, 2008). It was shown that AM can mimic the dynamics of the cell cycle switch regulatory network that induces the transition between stable cell cycle states. It was also postulated that the cell cycle switch efficiency is maximal only when its dynamics fully captures that of the AM algorithm (Cardelli and Csikász-Nagy, 2012) and later this prediction was experimentally verified (Hara *et al.*, 2012). We have seen that it is very well established that reliable biological time keeping mechanisms are regulated by the interconnection of such switch generating positive feedback loops with oscillation inducing negative feedback loops. The existence of negative feedback is essential and in almost all highly investigated systems the role of the positive feedback is important for the robust behaviour of the biological clock. It was established that the positive feedback module of the cell cycle regulatory network behaves like an efficient algorithm (Cardelli and Csikász-Nagy, 2012). Later, Cardelli (2014) established a theory to identify kinetically identically behaving regulatory networks. A future challenge will be the elucidation of which aspects of real life biological oscillators are important for their proper ticking and how far their kinetics could be associated to a minimalistic oscillator model.

References

Angluin, D., Aspnes, J., and Eisenstat, D. (2008). A simple population protocol for fast robust approximate majority. Distributed Computing *21*, 87-102.

Barkai, N., and Leibler, S. (2000). Biological rhythms: Circadian clocks limited by noise. Nature *403*, 267-268.

Becker-Weimann, S., Wolf, J., Herzel, H., and Kramer, A. (2004). Modeling Feedback Loops of the Mammalian Circadian Oscillator. Biophysical journal *87*, 3023-3034.

Cardelli, L. (2014). Morphisms of Reaction Networks that Couple Structure to Function. BMC Systems Biology *in press*

Cardelli, L., and Csikász-Nagy, A. (2012). The cell cycle switch computes approximate majority. Scientific reports *2*, 656.

Dalchau, N., and Webb, A.A. (2011). Dalchau, N., and Webb, A.A. (2011). Ticking over. Circadian systems across the kingdoms of life. Biochemist *February issue*, 12-15.

Dunlap, J.C. (1999). Molecular bases for circadian clocks. Cell *96*, 271-290.

Elowitz, M.B., and Leibler, S. (2000). A synthetic oscillatory network of transcriptional regulators. Nature *403*, 335-338.

Fall, C.P., Marland, E.S., Wagner, J.M., and Tyson, J.J. (2002). Computational cell biology, volume 20 of Interdisciplinary Applied Mathematics: Springer Verlag.

Ferrell Jr, J.E., Tsai, T.Y.-C., and Yang, Q. (2011). Modeling the cell cycle: why do certain circuits oscillate? Cell *144*, 874-885.

Field, R.J., Koros, E., and Noyes, R.M. (1972). Oscillations in chemical systems. II. Thorough analysis of temporal oscillation in the bromate-cerium-malonic acid system. Journal of the American Chemical Society *94*, 8649-8664.

François, P., Despierre, N., and Siggia, E.D. (2012). Adaptive Temperature Compensation in Circadian Oscillations. PLoS Comput Biol *8*, e1002585.

Fujii, T., and Rondelez, Y. (2012). Predator–prey molecular ecosystems. Acs Nano *7*, 27-34.

Goldbeter, A. (1991). A minimal cascade model for the mitotic oscillator involving cyclin and cdc2 kinase. Proceedings of the National Academy of Sciences *88*, 9107-9111.

Goldbeter, A. (1997). Biochemical oscillations and cellular rhythms. Biochemical Oscillations and Cellular Rhythms, by Albert Goldbeter, Foreword by MJ Berridge, Cambridge, UK: Cambridge University Press, 1997 *1*.

Goldbeter, A. (2002). Computational approaches to cellular rhythms. Nature *420*, 238-245.

Goldbeter, A. (2008). Biological rhythms: clocks for all times. Current biology *18*, R751-R753.

Gonze, D., Halloy, J., and Goldbeter, A. (2002). Robustness of circadian rhythms with respect to molecular noise. Proceedings of the National Academy of Sciences *99*, 673-678.

Goodwin, B.C. (1965). Oscillatory behavior in enzymatic control processes. Advances in enzyme regulation *3*, 425-437.

Gould, P.D., Locke, J.C., Larue, C., Southern, M.M., Davis, S.J., Hanano, S., Moyle, R., Milich, R., Putterill, J., and Millar, A.J. (2006). The molecular basis of temperature compensation in the Arabidopsis circadian clock. The Plant Cell Online *18*, 1177-1187.

Griffith, J. (1968). Mathematics of cellular control processes I. Negative feedback to one gene. Journal of Theoretical Biology *20*, 202-208.

Hara, M., Abe, Y., Tanaka, T., Yamamoto, T., Okumura, E., and Kishimoto, T. (2012). Greatwall kinase and cyclin B-Cdk1 are both critical constituents of M-phase-promoting factor. Nature communications *3*, 1059.

Harashima, H., Dissmeyer, N., and Schnittger, A. (2013). Cell cycle control across the eukaryotic kingdom. Trends in cell biology *23*, 345-356.

Hatakeyama, T.S., and Kaneko, K. (2012). Generic temperature compensation of biological clocks by autonomous regulation of catalyst concentration. Proceedings of the National Academy of Sciences *109*, 8109-8114.

Hayashi, I., and Kondoh, H. (1993). Temperature-compensated ring oscillator circuit formed on a semiconductor substrate: US patent 5180995 A.

Higgins, J. (1964). A chemical mechanism for oscillation of glycolytic intermediates in yeast cells. Proceedings of the National Academy of Sciences of the United States of America *51*, 989.

Hong, C.I., Conrad, E.D., and Tyson, J.J. (2007). A proposal for robust temperature compensation of circadian rhythms. Proceedings of the National Academy of Sciences *104*, 1195-1200.

Hong, C.I., Zámborszky, J., Baek, M., Labiscsak, L., Ju, K., Lee, H., Larrondo, L.F., Goity, A., Chong, H.S., Belden, W.J., and Csikasz-Nagy, A. (2014). Circadian rhythms synchronize mitosis in Neurospora crassa. Proceedings of the National Academy of Sciences *111*, 1397-1402.

Hong, C.I., Zámborszky, J., and Csikász-Nagy, A. (2009). Minimum criteria for DNA damage-induced phase advances in circadian rhythms. PLoS computational biology *5*, e1000384.

Hussain, F., Gupta, C., Hirning, A.J., Ott, W., Matthews, K.S., Josić, K., and Bennett, M.R. (2014). Engineered temperature compensation in a synthetic genetic clock. Proceedings of the National Academy of Sciences *111*, 972-977.

Klevecz, R.R., and King, G.A. (1982). Temperature compensation in the mammalian cell cycle. Experimental cell research *140*, 307-313.

Leloup, J.-C., and Goldbeter, A. (1997). Temperature compensation of circadian rhythms: control of the period in a model for circadian oscillations of the PER protein in Drosophila. Chronobiology international *14*, 511-520.

Mehra, A., Hong, C.I., Shi, M., Loros, J.J., Dunlap, J.C., and Ruoff, P. (2006). Circadian rhythmicity by autocatalysis. PLoS computational biology *2*, e96.

Meinhardt, H. (1982). Models of biological pattern formation. Academic Press London.

Mura, I., and Csikász-Nagy, A. (2008). Stochastic Petri Net extension of a yeast cell cycle model. Journal of Theoretical Biology *254*, 850-860.

Nakajima, M., Imai, K., Ito, H., Nishiwaki, T., Murayama, Y., Iwasaki, H., Oyama, T., and Kondo, T. (2005). Reconstitution of circadian oscillation of cyanobacterial KaiC phosphorylation in vitro. Science *308*, 414-415.

O'Neill, J.S., Van Ooijen, G., Dixon, L.E., Troein, C., Corellou, F., Bouget, F.-Y., Reddy, A.B., and Millar, A.J. (2011). Circadian rhythms persist without transcription in a eukaryote. Nature *469*, 554-558.

Pokhilko, A., Fernández, A.P., Edwards, K.D., Southern, M.M., Halliday, K.J., and Millar, A.J. (2012). The clock gene circuit in Arabidopsis includes a repressilator with additional feedback loops. Molecular systems biology *8*.

Pomerening, J.R., Kim, S.Y., and Ferrell Jr, J.E. (2005). Systems-level dissection of the cell-cycle oscillator: bypassing positive feedback produces damped oscillations. Cell *122*, 565-578.

Ruoff, P., and Rensing, L. (1996). The temperature-compensated Goodwin model simulates many circadian clock properties. Journal of Theoretical Biology *179*, 275-285.

Ruoff, P., Rensing, L., Kommedal, R., and Mohsenzadeh, S. (1997). Modeling temperature compensation in chemical and biological oscillators. Chronobiology international *14*, 499-510.

Sel'kov, E. (1968). Self-Oscillations in Glycolysis. European Journal of Biochemistry *4*, 79-86.

Sprinzak, D., and Elowitz, M.B. (2005). Reconstruction of genetic circuits. Nature *438*, 443-448.

Sriram, K., Bernot, G., and Kepes, F. (2007). A minimal mathematical model combining several regulatory cycles from the budding yeast cell cycle. IET systems biology *1*, 326-341.

Steuer, R. (2004). Effects of stochasticity in models of the cell cycle: from quantized cycle times to noise-induced oscillations. Journal of Theoretical Biology *228*, 293-301.

Tigges, M., Marquez-Lago, T.T., Stelling, J., and Fussenegger, M. (2009). A tunable synthetic mammalian oscillator. Nature *457*, 309-312.

Tsai, T.Y.-C., Choi, Y.S., Ma, W., Pomerening, J.R., Tang, C., and Ferrell, J.E. (2008). Robust, tunable biological oscillations from interlinked positive and negative feedback loops. Science *321*, 126-129.

Tsai, T.Y.C., Theriot, J.A., and Ferrell, J.E., Jr. (2014). Changes in Oscillatory Dynamics in the Cell Cycle of Early *Xenopus laevis* Embryos. PLoS Biol *12*, e1001788.

Tyson, J.J. (1991). Modeling the cell division cycle: cdc2 and cyclin interactions. Proceedings of the National Academy of Sciences *88*, 7328-7332.

Tyson, J.J., Albert, R., Goldbeter, A., Ruoff, P., and Sible, J. (2008). Biological switches and clocks. Journal of The Royal Society Interface *5*, S1-S8.

Tyson, J.J., Hong, C.I., Dennis Thron, C., and Novak, B. (1999). A simple model of circadian rhythms based on dimerization and proteolysis of PER and TIM. Biophysical journal *77*, 2411-2417.

Zámborszky, J., Hong, C.I., and Csikász-Nagy, A. (2007). Computational analysis of mammalian cell division gated by a circadian clock: quantized cell cycles and cell size control. Journal of biological rhythms *22*, 542-553.

Two possibly alternative approaches to the semantics of stochastic process calculi[*]

Rocco De Nicola (IMT Lucca) Diego Latella (ISTI-CNR, Pisa)
Michele Loreti (Univ. Firenze) Mieke Massink (ISTI-CNR, Pisa)

Abstract

In a recent paper, published in ACM Computing Surveys, we introduced a unifying framework to describe the semantics of process algebras, including their variants useful for modeling quantitative aspects of behaviors. In parallel with our work Luca Cardelli and Radu Mardare advocated a new approach to the semantics of stochastic process algebras based on measure theory. In this note, we briefly introduce the two approaches and contrast them by using both of them to describe the semantics of PEPA, one of the most known and used stochastic process algebra.

1 Introduction

Process Algebras have been successfully used over the last thirty years to model and analyze the behavior of concurrent distributed systems. They are based on mathematically rigorous *process description languages* with well-defined semantics that provide models of processes, regarded as agents that perform actions (act) and communicate (interact) with similar agents and with their environment. Process behavior is modeled by Labelled Transition Systems (LTSs) and these LTSs are then compared according to behavioral relations, giving rise to so-called *process calculi*. In some cases, the behavioral relations also have complete axiomatizations, in forms of equations, that exactly capture the relevant equivalences induced by the abstract operational semantics; then process calculi are also called *process algebras*. Nowadays, *process algebras*, *process calculi* and *process description languages* are often used interchangeably.

[*]Research partially funded by projects EU ASCENS (nr. 257414), EU QUANTICOL (nr. 600708), IT MIUR CINA and PAR FAS 2007-2013 Regione Toscana TRACE-IT.

Initially, process calculi were mainly designed to model functional (extensional) system behavior. However, it was soon recognized that, in order to capture other important features of concurrent systems, variants were needed to take quantitative features into account. This led to the development of *timed* process calculi, *probabilistic* process calculi, and *stochastic(-ally timed)* process calculi. The latter have proven to be particularly suitable for capturing important properties related to performance and quality of service, and even for modeling biological systems.

Terms of *stochastic process calculi* have been used to model systems and then to generate Continuous-Time Markov Chains (CTMCs) for performing systems analyses. In CTMCs, delays are modelled as *random variables* with negative exponential distributions; each of them is thus characterized by its unique parameter, the rate $\lambda > 0$, and has expected value λ^{-1}. CTMC-based process calculi associate time with actions, annotating them with rates; by means of a *rated action prefix* $(a, \lambda).P$, with λ being the rate associated with a.

While the target domains of the semantics functions of stochastic process calculi have in many cases been CTMCs, the same uniformity cannot be found in the source PDLs and in the approach taken to associate CTMCs to terms.

Some differences are *conceptual*; for instance, *multi-party* process synchronization is used in most SPCs, but there are notable examples of *one-to-one* process synchronization use, like stochastic π-calculus of Priami (1995) and stochastic CCS of Klin and Sassone (2008).

Other differences, instead, are purely *technical*, a prominent example of such a technical difference is the modeling of the *race condition* principle and its relationship to the issue of *transition multiplicity*. To take transition multiplicity into account and guarantee that the behavior of terms like $(a, \lambda).P + (a, \lambda).P$ is the same as that of $(a, 2 \cdot \lambda).P$, and not as $(a, \lambda).P$, several, significantly different, approaches have been proposed. They range from the use of *multi-relations* (Hillston 1996; Hermanns 2002) to *proved transition systems* (Priami 1995; Gotz et al. 1993) and from LTSs with *numbered transitions* (Hermanns et al. 2002) to *unique rate names* (De Nicola et al. 2007), to mention just a few. The feature that unites them all is that they require two steps to obtain the 'right' rate: first an enriched LTS is built and then it is manipulated to properly combine (e.g. to add up) rates.

In order to provide a uniform account of the many stochastic calculi, in De Nicola et al. (2009a,b) we proposed a variant of LTSs, namely *Rate Transition Systems* (RTSs). In LTSs, a transition is a triple (P, α, P') where P is the source state, α is the label of the transition, and P' is the target state reached from P via α. In RTSs a transition is a triple of the form $(P, \alpha, \mathscr{P})$, whose first and second component are again the source state and the transition label, but the third component $\mathscr{P}$ is the *continuation function* that associates a real non-negative value with each state P'. A non-zero value represents the rate of the exponential distribution characterizing the time needed for the execution of the action represented by α, necessary to reach P' from P via the transition. We have that $\mathscr{P}(P') = 0$, indicates that P' is not reachable from P via α. RTSs elegantly solve the issue of transition multiplicity; the rates of equal transitions, among those derivable from the semantics rules, are simply added via operations on continuation functions. Furthermore, RTSs make it relatively easy to define *associative* parallel composition operators for calculi adopting the one-to-one interaction paradigm.

In De Nicola et al. (2013), we introduced *State to Function Labeled Transition Systems*, FuTSs for short, a generalization of RTSs based on the parameterization of the co-domain of the continuation functions, which enables us to consider more models and to take non-deterministic systems into account. The co-domains of FuTSs are generic *commutative semi-rings*, and not just the set of non-negative reals. Continuation functions are equipped with a rich set of (generic) operations, making FuTSs very well suited as a semantic domain for the *compositional* definition of the operational semantics of stochastic process calculi. Such operations induce an algebraic structure on the set of continuation functions, which we systematically exploit for the compositional definition of the FuTS semantics of PCs. In De Nicola et al. (2013) we showed that FuTSs can be effectively used as a semantic domain for the compositional definition of the operational semantics of a calculus with both non-deterministic behavior and stochastic delays, and for an extension including probabilistic discrete (sub-)distributions over processes. By defining appropriate operators on continuation functions, we provided a compositional operational semantics of the key fragments of major stochastic process calculi including TIPP, EMPA, PEPA, StoCCS, IML and MAL, a language for Markov Automata. We thus provided a uniform, clean and powerful framework which helps in identifying differences and similarities among the many stochastic process calculi proposed in the literature.

Almost in the same period we were working on RTSs and FuTSs, Cardelli and Mardare proposed another semantic framework that can be used to express structural operational semantics of stochastic process calculi in terms of *measure theory*, see Cardelli and Mardare (2010, 2014). In their framework, a σ-algebra generated by the syntax of processes is used to organise processes as a measurable space. The structural operational semantics associates to each process a set of measures over the space of processes. The measures encode the rates of the transitions from a process to a measurable set of processes.

In this note we give a light-weight presentation of the two frameworks presented in De Nicola et al. (2013) and in Cardelli and Mardare (2014) together with an example of their application for the definition of the formal semantics of a PEPA, a typical stochastic process calculus. We conclude with a short discussion about the relationships between the two approaches.

2 A simple stochastic Process Calculus: PEPA

In this section, we consider a stochastically timed variant of CSP called Performance Evaluation Process Algebra (PEPA) introduced in Hillston (1996). In this calculus, every action is equipped with a rate $\lambda \in \mathbb{R}_{>0}$ that uniquely characterizes the exponentially distributed random variable quantifying the duration of the action itself (the expected duration is $1/\lambda$). The choice among the actions that are enabled in each state is governed by the race policy: the action to execute is the one that samples the least duration. As a consequence, (i) the sojourn time in each state is exponentially distributed with rate given by the sum of the rates of the transitions departing from that state, (ii) the execution probability of each transition is proportional to its rate, and (iii) the alternative and parallel composition operators are implicitly probabilistic.

The set $\mathcal{P}_{PEPA}$ of the PEPA terms we consider is defined by the grammar in Fig. 1, where a is a generic element of $\mathcal{L}_{PEPA}$, a given action set, λ is a *positive* real number, L is finite subset of $\mathcal{L}_{PEPA}$. Moreover, a suitable equation $X = P$ is assumed for each process constant X.

$$P ::= (a, \lambda).P \mid P + P \mid X \mid P \bowtie_L P$$

Figure 1: PEPA Syntax

Term $(a, \lambda).P$ denotes the process that performs action a and then evolves to P. The duration of this activity is a random variable exponentially distributed with parameter λ. The choice $P + Q$ describes a process that behaves like P or like Q, where the possible enabled actions are selected according to the *race condition* principle. Finally, *cooperation* $P \bowtie_L Q$ is used to combine behaviours of two processes. In $P \bowtie_L Q$, processes P and Q behave independently for actions not appearing in L while a synchronization is needed to execute actions occurring in L. The principle regulating the synchronization rate of PEPA processes is the so-called *minimal rate*, that assigns as rate of an action resulting from the synchronization of two processes the MIN of the rates of the synchronizing actions. Whenever a component process may perform the same action in several different ways, the cumulative, so-called *apparent*, rate has to be considered. Given a PEPA process P and an action α, the *apparent rate* of α in P, denoted by $r_\alpha(P)$, is inductively defined as follows:

$$\begin{aligned}
r_\alpha((\beta, \lambda).P) &=_{\text{def}} \begin{cases} \lambda, & \text{if } \beta = \alpha \\ 0, & \text{if } \beta \neq \alpha \end{cases} \\
r_\alpha(P + Q) &=_{\text{def}} r_\alpha(P) + r_\alpha(Q) \\
r_\alpha(P \bowtie_L Q) &=_{\text{def}} \begin{cases} min(r_\alpha(P), r_\alpha(Q)) & \text{if } \alpha \in L \\ r_\alpha(P) + r_\alpha(Q), & \text{if } \alpha \notin L \end{cases}
\end{aligned}$$

$$\frac{}{(a,\lambda).P \xrightarrow{\alpha,\lambda} P} \qquad \frac{P \xrightarrow{\alpha,\lambda} R}{P+Q \xrightarrow{\alpha,\lambda} R} \qquad \frac{Q \xrightarrow{\alpha,\lambda} R}{P+Q \xrightarrow{\alpha,\lambda} R} \qquad \frac{P \xrightarrow{\alpha,\lambda} Q, X{:=}P}{X \xrightarrow{\alpha,\lambda} Q}$$

$$\frac{P \xrightarrow{\alpha,\lambda} P', \alpha \notin L}{P \bowtie_L Q \xrightarrow{\alpha,\lambda} P' \bowtie_L Q} \qquad \frac{Q \xrightarrow{\alpha,\lambda} Q', \alpha \notin L}{P \bowtie_L Q \xrightarrow{\alpha,\lambda} P \bowtie_L Q'}$$

$$\frac{P \xrightarrow{\alpha,\lambda_1} P', Q \xrightarrow{\alpha,\lambda_2} Q', \alpha \in L}{P \bowtie_L Q \xrightarrow{\alpha, r(\alpha,\lambda_1,\lambda_2,P,Q)} P' \bowtie_L Q'}$$

Figure 2: SOS Rules for PEPA.

The stochastic operational semantics of PEPA processes (Hillston 1996) is given by means of definition of the least multi-relation satisfying the rules given in Fig. 2 where:

$$r(\alpha, \lambda_1, \lambda_2, P, Q) =_{\text{def}} \frac{\lambda_1}{r_\alpha(P)} \cdot \frac{\lambda_2}{r_\alpha(Q)} \cdot min(r_\alpha(P), r_\alpha(Q))$$

Please notice that the use of a *multi-relation* is crucial to guarantee the correct computation of the rate associated to a process transition. Unfortunately, this notion is *vague*. Indeed, no way is provided to formally compute this *multi-relation*. In the next section we will show how, by using FuTS, one can elegantly solve the issue of transition multiplicity.

3 FuTS semantics of PEPA

In this section we show how FuTSs can be used to define stochastic semantics of PEPA processes.

3.1 FuTSs in a nutshell

As anticipated in Sec. 1, the key ingredients of *State to Function Labeled Transition Systems*, FuTSs for short, are the *continuation functions* $\mathscr{P}$ (in the sequel often abbreviated with *continuations*), used as process transition targets, *and* a rich set of continuation *operators*, which facilitate the *compositional* definition of process calculi.

We recall that, for stochastic process calculi with CTMC semantics, the co-domain of any continuation $\mathscr{R}$ is $\mathbb{R}_{\geq 0}$, the set of non-negative real numbers. A transition $R \overset{\alpha}{\rightarrowtail} \mathscr{R}$ explicitly states that, whenever $\mathscr{R}(R') = 0$, process R' is not reachable in one step from R by performing action α, while, if $\mathscr{R}(R') = \lambda > 0$, then λ is the rate of a jump from state R to state R' performing action α.

In order to be able to treat different kinds of process calculi in a uniform way, an obvious step is the generalization of the codomain of continuations to any set of values.

Since, for the formalization of the semantics of process calculi it is necessary to be able (at least) to *sum* and *multiply* relevant values, possibly retaining useful properties like associativity, commutativity and distributivity, the generic continuations are functions from processes to *commutative semi-rings*. Actually, for the purposes of our work, it is sufficient to consider total functions with finite support, i.e. total functions which yield a non zero value only on a finite set of input values. $\mathbf{FTF}(S, \mathbb{C})$ will denote the class of finite support functions from set S to $\mathbb{C}$. As mentioned above, it is convenient to equip $\mathbf{FTF}(S, \mathbb{C})$ with operators. We will briefly recall them referring to De Nicola et al. (2013) for detailed formal definitions. The relevant operators are derived from those of $\mathbb{C}$. We lift $+_{\mathbb{C}}$ to $\mathbf{FTF}(S, \mathbb{C})$ by letting $(\mathscr{F}_1 + \mathscr{F}_2)(s) = \mathscr{F}_1(s) +_{\mathbb{C}} \mathscr{F}_2(s)$, for $\mathscr{F}_1$ and $\mathscr{F}_2$ in $\mathbf{FTF}(S, \mathbb{C})$; furthermore, for injective binary function $\cdot : S \times S \to S$ we let $(\mathscr{F}_1 \cdot \mathscr{F}_2)(s)$ yield $\mathscr{F}_1(s_1) \cdot_{\mathbb{C}} \mathscr{F}_2(s_2)$, if there exist (unique, due to injectivity) s_1, s_2 such that $s = s_1 \cdot s_2$, and $0_{\mathbb{C}}$ otherwise. We use the notation $[s \mapsto c]$ for the function associating c with s and 0 with any other $s' \neq s$, letting $[]$ denote the degenerate function yielding $0_{\mathbb{C}}$ everywhere. Finally, by $\oplus\mathscr{F}$ we mean $\sum_{s \in S} \mathscr{F}(s)$ while for any $C \subseteq S$, $\mathscr{F}(C) = \sum_{s \in C} \mathscr{F}(s)$, where $\sum$ is to be intended as the n-ary extension of $+_{\mathbb{C}}$, noting that the sum exists and is finite since $\mathscr{F}$ has finite support.

To define the formal semantics of mono-dimensional process calculi, i.e. process calculi with a single "kind" of transition relation, like most of the stochastic process calculi proposed in the literature, *simple total deterministic* FuTS*s* are sufficient.

Definition 1. *A* simple total deterministic state to function $\mathcal{L}$-labelled transition system *(simple deterministic* FuTS*) over* $\mathbb{C}$ *is a tuple* $(S, \mathcal{L}, \mathbb{C}, \rightarrowtail)$ *where* S *is a countable, non-empty, set of* states, $\mathcal{L}$ *is a countable, non-empty, set of transition* labels, $\mathbb{C}$ *is a commutative semi-ring, and* $\rightarrowtail$ *is a total function in* $S \to \mathcal{L} \to \mathbf{FTF}(S, \mathbb{C})$. •

We use the standard SOS semantics (i) notation $s \overset{\alpha}{\rightarrowtail} \mathscr{F}$ for $\rightarrowtail(s)(\alpha) = \mathscr{F}$ and (ii) terminology saying that $\rightarrowtail$ is the "transition relation"; finally, $(s \overset{\alpha}{\rightarrowtail})(s') = \mathscr{F}(s')$ if $s \overset{\alpha}{\rightarrowtail} \mathscr{F}$. Intuitively, $s_1 \overset{\alpha}{\rightarrowtail} \mathscr{F}$ and $(\mathscr{F}\, s_2) = \gamma \neq 0_{\mathbb{C}}$ means that s_2 is reachable from s_1 via (the execution of) α with a value $\gamma \in \mathbb{C}$. $(\mathscr{F}\, s_2) = 0_{\mathbb{C}}$ means that s_2 is not reachable from s_1 via the above α-transition. In the sequel we will omit "simple total deterministic" when referring to simple deterministic FuTSs.

A notion of bisimilarity is readily defined for FuTSs. Let $\mathcal{F} = (S, \mathcal{L}, \mathbb{C}, \rightarrowtail)$ be a FuTS. An equivalence relation $R \subseteq S \times S$ is called an $\mathcal{F}$-bisimulation if $s_1 \, R \, s_2$ implies

$$\sum_{s' \in [s]_R} (s_1 \overset{\alpha}{\rightarrowtail})(s') = \sum_{s' \in [s]_R} (s_2 \overset{\alpha}{\rightarrowtail})(s') \tag{1}$$

for all $s \in S$ and $\alpha \in \mathcal{L}$, where $[s]_R$ is the equivalence class of R which s belongs to. Two elements $s_1, s_2 \in S$ are called $\mathcal{F}$-bisimilar if $s_1 \, R \, s_2$ for some $\mathcal{F}$-bisimulation R for $\mathcal{F}$. Notation $s_1 \sim_{\mathcal{F}} s_2$. Again, note that the sums in equation (1) exist and are finite since, by definition, function $(s \overset{\alpha}{\rightarrowtail})$ has finite support for all s and α.

$$\frac{}{(a,\lambda).P \overset{a}{\rightarrowtail} [P\mapsto\lambda]} \qquad \frac{b\neq a}{(a,\lambda).P \overset{b}{\rightarrowtail} []_{\mathbb{R}_{\geq 0}}} \qquad \frac{P \overset{a}{\rightarrowtail} \mathscr{P}, Q \overset{a}{\rightarrowtail} \mathscr{Q}}{P+Q \overset{a}{\rightarrowtail} \mathscr{P}+\mathscr{Q}} \qquad \frac{P \overset{a}{\rightarrowtail} \mathscr{P}, X:=P}{X \overset{a}{\rightarrowtail} \mathscr{P}}$$

$$\frac{P \overset{a}{\rightarrowtail} \mathscr{P}, Q \overset{a}{\rightarrowtail} \mathscr{Q}, a\notin L}{P\bowtie_L Q \overset{a}{\rightarrowtail} (\mathscr{P}\bowtie_L(\mathcal{X}\,Q)) + ((\mathcal{X}\,P)\bowtie_L\mathscr{Q})} \qquad \frac{P \overset{a}{\rightarrowtail} \mathscr{P}, Q \overset{a}{\rightarrowtail} \mathscr{Q}, a\in L}{P\bowtie_L Q \overset{a}{\rightarrowtail} \mathscr{P}\bowtie_L\mathscr{Q}\cdot\frac{\textsc{min}\{\oplus\mathscr{P},\oplus\mathscr{Q}\}}{\oplus\mathscr{P}\cdot\oplus\mathscr{Q}}}$$

Figure 3: Semantics Rules for PEPA

3.2 An operational semantics of PEPA

In this section we show the use of our FuTS approach for the definition of the semantics of a major fragment of PEPA (Hillston 1996). For the study of the FuTS semantics of all of the most prominent SPCs we refer the reader to De Nicola et al. (2013); Latella et al. (2012).

Fig. 3 shows the semantics rules for (the fragment of) PEPA. The relevant semiring is $\mathbb{R}_{\geq 0}$. The rules for rated action-prefix establish that process P is reachable from $(a,\lambda).P$ via an a-transition with rate λ, while no process is reachable from $(a,\lambda).P$ via the execution of any action (b) different from a.

The rule for choice is such that the rates at which a process is reachable in one step via an action a from P or Q are *accumulated* in the rate at which the process is reachable in one step via a from $P+Q$. So, for instance, let R be the term $(a_1,\lambda_1).P_1 + (a_2,\lambda_2).P_2$; then $R \overset{a_1}{\rightarrowtail} [P_1 \mapsto \lambda_1]$ and $R \overset{a_2}{\rightarrowtail} [P_2 \mapsto \lambda_2]$; in particular, if $a_1 = a_2 = a$, we have that $R \overset{a}{\rightarrowtail} [P_1 \mapsto \lambda_1] + [P_2 \mapsto \lambda_2]$, i.e. $R \overset{a}{\rightarrowtail} \mathscr{R}$ with $\mathscr{R}(P_1) = \lambda_1$, $\mathscr{R}(P_2) = \lambda_2$, and $\mathscr{R}(P) = 0_{\mathbb{R}_{\geq 0}}$ for $P \notin \{P_1, P_2\}$; if P_1 and P_2 are *the same* process P, we get $R \overset{a}{\rightarrowtail} [P \mapsto \lambda_1 +_{\mathbb{R}_{\geq 0}} \lambda_2]$ which encodes the *race condition* principle of CTMCs.

Let us now consider the semantics of the PEPA cooperation operator. The cooperation syntax constructor $\bowtie_L$ is clearly injective, thus we have $(\mathscr{P}_1\bowtie_L\mathscr{P}_2)(P_1\bowtie_L P_2) = \mathscr{P}_1(P_1)\cdot_{\mathbb{R}_{\geq 0}}\mathscr{P}(Q_2)$ while $(\mathscr{P}_1\bowtie_L\mathscr{Q}_2)$ returns 0 when applied to a process which is not of the form $P_1\bowtie_L P_2$ for some P_1 and P_2. The rule for the case $a \in L$ encodes the PEPA *minimal rate synchronization* principle where $\oplus\mathscr{P}$ (resp. $\oplus\mathscr{Q}$) is exactly the *apparent rate* of a in P (resp. Q) defined in Hillston (1996). $\textsc{min}\{r_1, r_2\}$ is the minimum between r_1 and r_2 and we define $r_1/0 = 0$ and r_1/r_2 as the inverse of the product of $\mathbb{R}_{\geq 0}$, if $r_2 \neq 0$. For example, for R as before, assuming $a_1 = a_2 = a$ and $\lambda_1 + \lambda_2 > \lambda_3$, we get $R\bowtie_{\{a\}}(a,\lambda_3).P_3 \overset{a}{\rightarrowtail} \mathscr{P}$ where $\mathscr{P}(P_1\bowtie_{\{a\}}P_3) = \lambda_1\cdot\lambda_3\cdot\frac{\lambda_3}{(\lambda_1+\lambda_2)\cdot\lambda_3} = \frac{\lambda_1\cdot\lambda_3}{\lambda_1+\lambda_2}$, $\mathscr{P}(P_2\bowtie_{\{a\}}P_3) = \lambda_2\cdot\lambda_3\cdot\frac{\lambda_3}{(\lambda_1+\lambda_2)\cdot\lambda_3} = \frac{\lambda_2\cdot\lambda_3}{\lambda_1+\lambda_2}$ and $\mathscr{P}(P) = 0$ for any other P. The rule for the case $a \notin L$ makes use of the characteristic function $\mathcal{X}\,R$ defined as $[R \mapsto 1]$ and computes the rates for the interleaving case. In fact $(\mathscr{P}_1\bowtie_L\mathcal{X}\,P_2)(P_1\bowtie_L P_2) =$

$\mathscr{P}_1(P_1) \cdot 1 = \mathscr{P}_1(P_1)$ while $(\mathscr{P}_1 \bowtie_L \mathcal{X}\, P_2)(R) = 0$ if R is not of the form $P_1 \bowtie_L P_2$ for some P_1; this essentially means that in the rule, the function $(\mathscr{P} \bowtie_L (\mathcal{X}\, Q))$ assigns a non-zero rate $\mathscr{P}(P')$ only to processes $P' \bowtie_L Q$ such that P' is reachable in one step from P via action a and Q remains unchanged. Similar reasoning applies to the symmetric case and, as usual, the rates for the resulting a-steps are accumulated together.

It is easy to see that for all $P \in \mathcal{P}_{PEPA}$, $a \in \mathcal{L}_{PEPA}$, and $\mathscr{P}$ function from $\mathcal{P}_{PEPA}$ to $\mathbb{R}_{\geq 0}$, if $P \overset{a}{\rightarrowtail} \mathscr{P}$ can be derived using the set of rules of Fig. 3, then we have $\mathscr{P} \in \mathbf{FTF}(\mathcal{P}_{PEPA}, \mathbb{R}_{\geq 0})$; furthermore, the least relation $\rightarrowtail \subseteq \mathcal{P}_{PEPA} \times \mathcal{L}_{PEPA} \times \mathbf{FTF}(\mathcal{P}_{PEPA}, \mathbb{R}_{\geq 0})$ which satisfies the set of rules of Fig. 3 is indeed a function in $\mathcal{P}_{PEPA} \to \mathcal{L}_{PEPA} \to \mathbf{FTF}(\mathcal{P}_{PEPA}, \mathbb{R}_{\geq 0})$. We can thus *define* the FuTS semantics of PEPA as the FuTS $\mathcal{F}_{PEPA} = (\mathcal{P}_{PEPA}, \mathcal{L}_{PEPA}, \mathbb{R}_{\geq 0}, \rightarrowtail_{PEPA})$, where $\rightarrowtail_{PEPA}$ is the *least* relation satisfying the rules in Fig. 3.

For the fragment of PEPA we considered, one can easily prove the formal correspondence between the FuTS semantics and the original SOS, as in Hillston (1996) where an action-rate indexed family of transition *multi*-relations $\xrightarrow{a,\lambda}$ is defined on processes[1]. In particular, for all $P, Q \in \mathcal{P}_{PEPA}$, $a \in \mathcal{L}_{PEPA}$, and $\mathscr{P} \in \mathbf{FTF}(\mathcal{P}_{PEPA}, \mathbb{R}_{\geq 0})$ such that $P \rightarrowtail_{PEPA} \mathscr{P}$ it holds that: $(\mathscr{P}\, Q) = \sum_{\lambda \in \{\!|\, \lambda' | P \xrightarrow{a,\lambda'} Q \,|\!\}} \lambda$, where the notation $\{\!|\;|\!\}$ is used for multi-sets.

Clearly, the FuTS semantics simply abstracts from the *different* SOS-transitions from a state to the next one, via a certain action—including possible *copies* of the same SOS-transition, which may originate from different derivations in the SOS—making them *collapse* into a single FuTS-transition, while accumulating all the rates labeling the SOS-transitions and embedding the cumulative rate in the continuation. The interesting thing is that such a accumulation/collapse process is performed while computing the continuations in a *compositional* and *incremental* way, as established by the semantics rules. Furthermore, the relevant behavioral properties of processes, as implied by the SOS, are preserved in the FuTS semantics. This is also witnessed by the fact that the bisimilarity defined on FuTS coincides with the classical PEPA strong equivalence $\cong$ (Hillston 1996). Indeed, for all processes $P, Q \in \mathcal{P}_{PEPA}, P \cong Q$ if and only if $P \sim_{\mathcal{F}_{PEPA}} Q$.

4 Measurable space of PEPA Processes

In this section we show how the approach proposed in Cardelli and Mardare (2014) can be used to define stochastic semantics of PEPA processes.

[1]We conventionally call such transitions the SOS-transitions, as opposed to the FuTS-transition relation $\rightarrowtail_{PEPA}$.

4.1 A short introduction to measurable spaces

We first recall notions of measure theory and we introduce the terminology and notations used in Cardelli and Mardare (2014) and in the rest of this section.

Let M be a set; a set $\Sigma \subseteq 2^M$ is a σ-algebra over M if and only if Σ contains M and it is closed under complement and countable union. When Σ is a σ-algebra over M, the pair (M, Σ) is called a *measurable space*, the elements of Σ *measurable sets* and M the *support set*.

A set $\Omega \subseteq 2^M$ is a *generator* for the σ-algebra Σ on M if Σ is the closure of Ω under complement and countable union. A generator with disjoint elements is called a *base* for Σ.

A *measure* on a measurable space $\mathcal{M} = (M, \Sigma)$ is a function $\mu : \Sigma \to \mathbb{R}_{\geq 0}$ such that: $\mu(\emptyset) = 0$; for any $\{N_i\}_{i\in I}$ countable sets of pairwise disjoint elements, $\mu(\{N_i\}_{i\in I}) = \sum_{i\in I} \mu(N_i)$. We let ω to denote the *null measure* on (M, Σ), i.e. the measure such that $\omega(M) = 0$. Let Ω be a base for (M, Σ), $N \in \Omega$ and $r \in \mathbb{R}_{>0}$, then $D(r, N)$ denotes:

$$D(r, N)(N') = \begin{cases} r & N = N' \\ 0 & N \neq N' \end{cases}$$

We also let $\Delta(M, \Sigma)$ be the measurable space of the measures on (M, Σ) with the σ-algebra generated, for any set $S \in \Sigma$ and $r > 0$, by the set $\{\mu \in \Delta(M, \Sigma) | \mu(S) \geq r\}$.

Given two measurable spaces (M, Σ) and (N, Θ), a mapping $f : M \to N$ is *measurable* if and only if for any $T \in \Theta$, $f^{-1}(\Theta) \in \Sigma$. We let $[\![M \to N]\!]$ be the class of measurable mapping from (M, Σ) to (N, Θ). Let (M, Σ) be a measurable space and A a denumerable set of *actions*. An *A-Markov kernel* is a tuple $\mathcal{M} = (M, \Sigma, \theta)$, with $\theta : A \to [\![M \to \Delta(M, \Sigma)]\!]$. Let $a \in A$, $m \in M$ and S a measurable set of states, $\theta(\alpha)(m)$ is a measure on the state space; in particular $\theta(\alpha)(m)(S) \in \mathbb{R}_{>0}$ represents the rate of an exponentially distributed random variable that characterizes the duration of an α-transition from m to an arbitrary $s \in S$.

4.2 A measure oriented semantics of PEPA Processes

In order to apply the approach proposed in Cardelli and Mardare (2014) we have first to define a σ-algebra generated by the syntax of processes. Then, this σ-algebra is used to define a $\mathcal{L}_{PEPA}$*-Markov kernel* that models stochastic behaviour of PEPA processes.

A σ-algebra for the set $\mathcal{P}_{PEPA}$ of PEPA processes can be defined by considering the *structural congruence relation* which equates processes that, in spite of their different syntactic form, represent the same system. We let $\equiv \subseteq \mathcal{P}_{PEPA} \times \mathcal{P}_{PEPA}$ be the smallest relation satisfying the following conditions:

1. $\equiv$ is an equivalence relation on $\mathcal{P}_{PEPA}$;

2. for each $P, Q, R \in \mathcal{P}_{PEPA}$ and for each set of actions $L \subseteq \mathcal{L}_{PEPA}$:

   $$P \bowtie_L Q \equiv Q \bowtie_L P \quad P+Q \equiv Q+P \quad (P+Q)+R \equiv P+(Q+R) \quad X := P \Longrightarrow X \equiv P$$

3. $\equiv$ is a congruence with respect to the algebraic structure of $\mathcal{P}_{PEPA}$, i.e. if $P \equiv Q$ then:

   $$P + R \equiv Q + R \qquad Q \bowtie_L R \equiv P \bowtie_L R \qquad (a, \lambda).P \equiv (a, \lambda).Q$$

Since $\equiv$ is an equivalence relation, we can consider the set $\mathcal{P}^{\equiv}_{PEPA}$ of $\equiv$-classes on $\mathcal{P}_{PEPA}$. Moreover, given a PEPA process P we let $P^{\equiv}$ the $\equiv$-class of P. $\mathcal{P}^{\equiv}_{PEPA}$ is a denumerable partition of $\mathcal{P}_{PEPA}$, hence it generates a σ-algebra Π over $\mathcal{P}_{PEPA}$ and $(\mathcal{P}_{PEPA}, \Pi)$ is a measurable space.

PEPA operators can be lifted to the elements of Π. Let $\mathcal{P}$ and $\mathcal{Q}$ be arbitrary elements in Π and $P \in \mathcal{P}_{PEPA}$ we let $\mathcal{P} \bowtie_L \mathcal{Q}$ and $\mathcal{P}_{\bowtie_L P}$ be the following measurable sets:

$$\mathcal{P} \bowtie_L \mathcal{Q} = \bigcup_{P \in \mathcal{P}, Q \in \mathcal{Q}} (P \bowtie_L Q)^{\equiv} \qquad \mathcal{P}_{\bowtie_L P} = \bigcup_{P \bowtie_L R \in \mathcal{P}} R^{\equiv}$$

The measure space $(\mathcal{P}_{PEPA}, \Pi)$ is the starting point to define our $\mathcal{L}_{PEPA}$-Markov kernel. Indeed, the latter is defined as the tuple $(\mathcal{P}_{PEPA}, \Pi, \theta)$ where $\theta : \mathcal{L}_{PEPA} \to [\![\mathcal{P}_{PEPA} \to \Delta(\mathcal{P}_{PEPA}, \Pi)]\!]$ is inductively defined on the structure of $P \in \mathcal{P}_{PEPA}$ as follow:

- $P = (a, \lambda).Q$: for any $b \in \mathcal{L}_{PEPA}$

  $$\theta(b)((a, \lambda).Q) = \begin{cases} D(\lambda, Q^{\equiv}) & (b = a) \\ \omega & (b \neq a) \end{cases}$$

- $P = Q + R$: For any $a \in \mathcal{L}_{PEPA}$ and $\mathcal{P} \in \Pi$:

  $$\theta(a)(Q + R)(\mathcal{P}) = \theta(a)(Q)(\mathcal{P}) + \theta(a)(R)(\mathcal{P})$$

- $P = Q \bowtie_L R$, for any $a \notin L$ and $\mathcal{P} \in \Pi$:

  $$\theta(a)(Q \bowtie_L R)(\mathcal{P}) = \theta(a)(Q)(\mathcal{P}_{\bowtie_L R}) + \theta(a)(R)(\mathcal{P}_{\bowtie_L Q})$$

  for any $a \in L$ and $\mathcal{P} \in \Pi$:

  $$\theta(a)(Q \bowtie_L R)(\mathcal{P}) = \frac{\textsc{min}\{r_a(Q), r_a(R)\}}{r_a(Q) \cdot r_a(R)} \cdot \sum_{\mathcal{P}_1 \bowtie_L \mathcal{P}_1 \subseteq \mathcal{P}} \theta(a)(Q)(\mathcal{P}_1) \cdot \theta(a)(R)(\mathcal{P}_2)$$

  where $r_a(P)$ (resp. $r_a(Q)$) is the *apparent rate* of a in P (resp. Q) (Hillston 1996).

- $P = X$: if $X := Q$, for any $a \in \mathcal{L}_{PEPA}$ and $\mathcal{P} \in \Pi$:

$$\theta(a)(X)(\mathcal{P}) = \theta(a)(Q)(\mathcal{P})$$

The stochastic behaviour induced by the $\mathcal{L}_{PEPA}$-Markov kernel defined above is exactly the same induced by the FuTS semantics considered in the previous section. Indeed, it is easy to prove that for any $P \in \mathcal{P}_{PEPA}$, $a \in \mathcal{L}_{PEPA}$ and $\mathcal{P} \in \Pi$:

$$\theta(a)(P)(\mathcal{P}) = v \Leftrightarrow \exists \mathscr{P} : P \overset{a}{\rightarrowtail} \mathscr{P} \text{ and } \mathscr{P}(\mathcal{P}) = v$$

According to Cardelli and Mardare (2014) we can now define the structural operational semantics that associates to each process P an infinite measurable set of processes. First, we have to introduce some extra notation. We let $[^{a,\lambda}P]$ denote the function $\mathcal{L}_{PEPA} \to \Delta(\mathcal{P}_{PEPA}, \Pi)$ defined as follows:

$$[^{a,\lambda}P](b) = \begin{cases} D(\lambda, P^{\equiv}) & (b = a) \\ \omega & (b \neq a) \end{cases}$$

Let $\mu' : \mathcal{L}_{PEPA} \to \Delta(\mathcal{P}_{PEPA}, \Pi)$ and $\mu'' : \mathcal{L}_{PEPA} \to \Delta(\mathcal{P}_{PEPA}, \Pi)$, $\mu' \oplus \mu'' \mathcal{L}_{PEPA} \to \Delta(\mathcal{P}_{PEPA}, \Pi)$ is the function such that for any $a \in \mathcal{L}_{PEPA}$:

$$(\mu' \oplus \mu'')(a) = \mu'(a) + \mu''(a)$$

Moreover, for any $P, Q \in \mathcal{P}_{PEPA}$ and $L \subseteq \mathcal{L}_{PEPA}$, $\mu'^{P} \bowtie_{L}^{Q} \mu'' \mathcal{L}_{PEPA} \to \Delta(\mathcal{P}_{PEPA}, \Pi)$ is the function such that, if $a \notin L$:

$$(\mu'^{P} \bowtie_{L}^{Q} \mu'')(a)(\mathcal{R}) = \mu'(a)(\mathcal{R}_{\bowtie_L Q}) \; + \; \mu''(a)(\mathcal{R}_{\bowtie_L P})$$

while if $a \in L$:

$$(\mu'^{P} \bowtie_{L}^{Q} \mu'')(a)(\mathcal{R}) = \frac{\textsc{min}\{r_a(Q), r_a(R)\}}{r_a(Q) \cdot r_a(R)} \cdot \sum_{\mathcal{P}_1 \bowtie_L \mathcal{P}_1 \subseteq \mathcal{R}} \mu'(a)(\mathcal{P}_1) \cdot \mu''(\mathcal{P}_2)$$

Finally, the *stochastic transition relation* of PEPA processes is the smallest transition relation $\to\ \subseteq \mathcal{P}_{PEPA} \times [\mathcal{L}_{PEPA} \to \Delta(\mathcal{P}_{PEPA}, \Pi)]$ satisfying the following rules:

$$\frac{}{(a,\lambda).P \to [^{a,\lambda}P]} \qquad \frac{P \to \mu' \quad Q \to \mu''}{P+Q \to \mu'+\mu''} \qquad \frac{P \to \mu' \quad Q \to \mu''}{P \bowtie_L Q \to \mu'^{P} \bowtie_{L}^{Q} \mu''} \qquad \frac{X:=P \quad P \to \mu}{X \to \mu}$$

The transition relation $\to$ can be used to define a *stochastic bisimulation* on PEPA processes (Cardelli and Mardare 2014) . A *rate bisimulation relation* on $\mathcal{P}_{PEPA}$ is an equivalence relation $\mathcal{R} \subseteq \mathcal{P}_{PEPA} \times \mathcal{P}_{PEPA}$ such that for each $P, Q \in \mathcal{P}_{PEPA}$ with $P \to \mu$ and $Q \to \mu'$, $(P, Q) \in \mathcal{R}$ if and only if for any $C \in \Pi(\mathcal{R})$[2] and $a \in \mathcal{L}_{PEPA}$,

[2]If (M, Σ) is a measurable space and $\mathcal{R} \subseteq M \times M$, $\Sigma(\mathcal{R})$ denotes the set of measurable $\mathcal{R}$-closed subsets of M.

$\mu(a)(C) = \mu'(a)(C)$. Two PEPA processes P and Q are stochastic bisimilar, written $P \sim Q$, if and only if there exists a rate bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$. Relation $\sim$ coincides with the classical PEPA strong equivalence $\cong$ (Hillston 1996). Indeed, for all processes $P, Q \in \mathcal{P}_{PEPA}$, $P \cong Q$ if and only if $P \sim Q$.

The transition relation $\rightarrow$ is in fact *equivalent* to the one considered in the FuTS semantics of the previous section.

Theorem 2. *For any $P \in \mathcal{P}_{PEPA}$:*

$$P \overset{a}{\rightarrowtail} \mathscr{P} \Leftrightarrow P \rightarrow \mu \text{ and for any } \mathcal{P} \in \Pi\text{: } \mu(a)(\mathcal{P}) = \mathscr{P}(\mathcal{P})$$

5 Concluding Remarks

In this paper we have provided a light-weight presentation of the two frameworks presented in De Nicola et al. (2013) and in Cardelli and Mardare (2014) together with an example of their application for the definition of the formal semantics of PEPA, one of the protypical stochastic process calculi.

The key feature of the FuTSs model introduced in De Nicola et al. (2013) is the fact that each transition is a triple of the form $(s, \alpha, \mathscr{P})$. The first and the second components are the source state and the label of the transition, while the third component, $\mathscr{P}$, is the *continuation function*, which associates a value of a suitable type with each state, say s'. The only requirement on the co-domains of the continuation functions is that they must be *commutative semi-rings*, which make FuTSs a very general framework. In De Nicola et al. (2013) the FuTS framework has been applied to the major stochastic process calculi proposed in the literature, ranging from CCS to Stochastic π-calculus, from PEPA to TIPP, but including also those process calculi that deal with both non-deterministic and probabilistic/stochastic behavior. Furthermore, in Latella et al. (2012) the basis are set for a systematic study of FuTS within the coalgebraic framework that calls for further investigations of the relationship between general Weighted Transition Systems and FuTSs.

The framework proposed in Cardelli and Mardare (2014) relies on a σ-algebra generated by the syntax of processes to structure processes as measurable spaces. The structural operational semantics associates to each process a set of measures over the space of processes. The measures encode the rates of the transitions from a process to a measurable set of processes.

We could say that the two approaches aim at describing the same set of systems by taking a slightly different approach. FuTS generalize LTS and associate to pairs *(state, label)* a weight function, and then define operators to combine, in a structural oriented approach, such functions in order to give semantics to the different operators of an calculus. Cardelli and Mardare define operators on measure spaces and rely on them to provide a meaning to the operators of the considered calculus. They associate to each state a mapping from labels to measures and use operations on measures to provide the meaning of composite terms.

Our approach seems to be more flexible and its expressivity is vindicated by the rich set of calculi that have been modeled with FuTS. For example, these have been also used to model stochastic process calculi with non determinism like IML presented in Hermanns (2002). We doubt this calculus can be accounted by the approach proposed by Cardelli and Mardare. Indeed, the latter approach, up to now, has been only used to deal with classical SPCs.

We also feel that our approach is more natural, but of course *"Every ugly child is nice for his mom"* or like they say in Naples *"every cockroach is beautiful in the eyes of his mother"*. Anyway, for what concerns the choice between the two approaches that we have briefly outlined in this note, we leave the final word to Luca Cardelli. Thank you Luca for all contributions and for all inspiring conversations.

References

L. Cardelli and R. Mardare. The Measurable Space of Stochastic Processes. In *Seventh International Conference on the Quantitative Evaluation of Systems (QEST 2010)*, pages 171–180. IEEE Computer Society Press, 2010.

L. Cardelli and R. Mardare. The Measurable Space of Stochastic Processes. *Fundam. Inform.*, 131(3-4):351–371, 2014.

R. De Nicola, J.-P. Katoen, D. Latella, M. Loreti, and M. Massink. Model Checking Mobile Stochastic Logic. *Theoret. Computer Science, Elsevier*, 382(1):42–70, 2007.

R. De Nicola, D. Latella, M. Loreti, and M. Massink. MarCaSPiS: a Markovian Extension of a Calculus for Services. In M. Hennessy and B. Klin, editors, *Proc. of SOS 2008*, vol. 229, *ENTCS*, pages 11–26. Elsevier, 2009a.

R. De Nicola, D. Latella, M. Loreti, and M. Massink. Rate-based Transition Systems for Stochastic Process Calculi. In A. S., A. Marchetti-Spaccamela, et al., editors, *Proc. of ICALP 2009*, vol. 5556, *LNCS*, pages 435–446. Springer, 2009b.

R. De Nicola, D. Latella, M. Loreti, and M. Massink. A Uniform Definition of Stochastic Process Calculi. *ACM Computing Surveys*, 46(1):5:1–5:35, 2013.

N. Gotz, U. Herzog, and M. Rettelbach. Multiprocessor and distributed systems design: The integration of functional specification and performance analysis using stochastic process algebras. In L. Donatiello and R. Nelson, eds, *Performance Evaluation of Computer and Communication Systems*, vol. 729, *LNCS*. Springer, 1993.

H. Hermanns. *Interactive Markov Chains*. Springer, 2002. LNCS 2428.

H. Hermanns, U. Herzog, and J.-P. Katoen. Process algebra for performance evaluation. *Theoretical Computer Science. Elsevier.*, 274(1-2):43–87, 2002.

J. Hillston. A compositional approach to performance modelling, 1996. Distinguished Dissertation in Computer Science. Cambridge University Press.

B. Klin and V. Sassone. Structural Operational Semantics for Stochastic Process Calculi. In R. Amadio, editor, *FoSSaCS 2008*, vol. 4962, *LNCS*, pages 428–442. Springer, 2008.

D. Latella, M. Massink, and de Vink. Bisimulation of Labeled State-to-Function Transition Systems of Stochastic Process Languages. In U. Golas and T. Soboll, editors, *Proc. of ACCAT 2012*, vol. 93, *EPTCS*, pages 23–43, 2012.

C. Priami. Stochastic π-Calculus. *The Computer Journal. Oxford University Press*, 38 (7):578–589, 1995.

Refining Objects
(Preliminary Summary)

Robert Harper
Computer Science Department, Carnegie Mellon University

Rowan Davies
School of Computer Science, University of Western Australia

Abstract

Inspired by Cardelli's pioneering work, many type disciplines for object-oriented programming are based on enrichments of structural type theories with constructs such as subtyping and bounded polymorphism. A principal benefit of such a formulation is that the absence of "message not understood" errors is an immediate corollary of the type safety theorem. A principal drawback is that the resulting type systems tend to be rather complex in order to accommodate the methodology of object-oriented programming.

We consider another approach based on a simple structural type theory enriched with a system of type refinements with which we may express behavioral requirements such as the absence of "message not understood" errors. Ensuring this property is viewed as a verification condition on programs that use dynamic dispatch, which we construe as an abstract type of objects supporting instantiation and messaging operations. At the structural level dynamic dispatch may fail, but at the behavioral level this possibility is precluded.

To validate this approach we give an interpretation of Featherweight Java (**FJ**), a widely-used model of object-oriented programming, that comprises a compilation into dynamic dispatch, and an interpretation of the class table as a system of type refinements. We show that well-typed **FJ** programs translate to well-typed and well-refined programs, from which we deduce the same safety guarantees as are provided by **FJ**. More importantly, the behavioral formulation may be scaled to verify the absence of other behaviors, such as down-cast errors, that are not easily handled using only structural types.

1 Introduction

In fairness, designers of object-oriented languages did not simply "forget" to include properties such as good type systems and good modularity: the issues are intrinsically more complex than in procedural languages. - Cardelli (1996)

A great deal of effort has gone into the design of type systems for object-oriented programming. A prime objective, formulated by Cardelli in the 1980's, is to devise type systems for object-oriented languages that preclude "message not understood" errors at run-time (see, for example, Cardelli (1988)). Achieving this objective proved quite challenging, stimulating a large body of research on type systems that could account for a rich variety of programming practices while ensuring that such run-time errors are precluded. Numerous new techniques were introduced, ranging from relatively simple concepts such as subtyping to more advanced concepts such as higher-kinded bounded quantification (see, for example, Bruce et al. (1999) and Fisher and Mitchell (1996)).

These type systems are notoriously complex, to the point that their uptake in practice has been more limited than one might have hoped. Negative results, such as the discovery of unsoundness in extant languages such as Eiffel, have had scant influence on their design or use (see Cook (1989)). Positive results, such as the development of comprehensive theories of objects by Abadi and Cardelli (1996), have had only limited influence on the design of new languages. Although languages such as Modula-3 (Cardelli et al. 1989) have benefited from the theories, newer object-oriented languages, such as Scala (Odersky and Rompf 2014), have only weakly developed theoretical foundations. The situation is in sharp contrast to the direct and continuing influence of type theory on the design and implementation of functional languages, including notable examples such as Standard ML (Milner et al. 1997) and Haskell (Jones 2003), and their more recent evolutes such as Agda (Norell 2008) and Idris (Brady 2013).

It is reasonable to ask why this is the case. One response might be to conclude that the complexity of the type theories involved is an indication that the concepts of object-oriented programming are overly complex, perhaps even conceptually and methodologically suspect. Another reaction might be to argue that type systems are simply not up to the task, and should either be made substantially more powerful (and complicated), or be abandoned entirely (by reversion to untyped languages). But, as Scott (1976) made clear decades ago, untyped languages are uni-typed languages, so there is really no possibility of abandoning types; it is only a matter of how they are to be deployed.

In this paper we propose an alternative approach to typing object-oriented languages that exploits the distinction between *structural*, or *intrinsic*, typing from *behavioral*, or *extrinsic*, typing (Reynolds 1985). Briefly, a structural type system is a context-sensitive grammar that determines what are the well-formed programs, and, via Gentzen's inversion principle, how they are executed. A behavioral type system is a system of predicates or relations (propositional functions), called *type refinements*, or just *refinements* for short, that describe the execution properties of well-typed programs (Freeman and Pfenning 1991; Davies and Pfenning 2000; Davies 2005; Dunfield 2007). Whereas showing that a program is (structurall) well-typed is usually decidable, showing that a program satisfies a refinement is, by Rice's Theorem, a matter of verification requiring proof. In many cases one can derive efficient and effective decision procedures for certain behaviors, such as the ones we shall consider here, but of course one cannot expect to have fully automatic verification of such conditions.

Since Cardelli's orginal work in the area (Cardelli 1988), the structural approach has drawn the most attention for formulating type disciplines for object-oriented programming. One reason is that structural type disciplines induce behavioral properties of programs from general properties of the language in which they are written. Most importantly, a properly formulated structural type discipline enjoys the *type safety* property (Milner 1978; Wright and Felleisen 1994; Harper 2012), which guarantees that certain forms of run-time errors cannot arise. It makes sense, then, to build on this foundation to derive desirable properties of object-oriented programs, such as the absence of "not understood" errors, from the safety theorem for the type discipline. This goal has usually been achieved by regarding objects as analogous to labelled tuples and messages as analogous to projections, so that type safety ensures that no message may be sent to an object that does not recognize it. Achieving this goal, while ensuring that the type system is not too restrictive, requires concepts such as structural subtyping and bounded quantification (see Abadi and Cardelli (1996) for a thorough discussion of the techniques required). The result is an impressive array of typing concepts for relatively little pay-off. Moreover, from a structural point of view, these concepts are, to an extent, questionable. (For example, width subtyping for tuples relies on the assumption that projections are meaningful independently of the tuple type, a property that is not guaranteed by the universal properties of products, but which can often be arranged to hold in specific implementations.)

The difficulty with the structural approach is that it does not scale well to ensure other desirable properties of programs, such as the absence of "down-cast errors," or to the enforcement of behavioral subtyping conditions (Liskov and Wing 1994). To better address these issues we propose another approach to typing object-oriented programs that is based on distinguishing the structural concept of dynamic dispatch (Cook 2009; Aldrich 2013) from the behavioral concept of avoidance of run-time errors. According to our view, dynamic dispatch is simply an application of data abstraction in which an abstract type of objects is equipped with introduction operations that *instantiate* a class with instance data and elimination operations that *message* to invoke a method on an instance. Thus, dynamic dispatch amounts to *heterogeneous programming* in which we have a variety of operations (methods) acting on data of a variety of forms (classes). Such a setup can be envisioned as a *dispatch matrix* whose rows are classes, whose columns are methods, and whose entries determine the behavior of each method on each class. The dispatch matrix gives rise to two equivalent implementations of dynamic dispatch that arise from the duality between sums and products in type theory. This implies that there is no inherent reason to prefer a product-based realization of objects; one may just as well use a sum-based representation. (See Section 3 for further discussion of this point.) This description leaves open what we mean by the behavior of a method on an instance of a class. When well-defined, a method determines a result as a function of the instance data of the object on which it acts. But a method may also be undefined on certain classes, and would, if invoked, incur a "not understood" error. Thus, at the structural level, it is possible for dynamic dispatch to fail, even in a well-typed program, just as it is possible to incur an arithmetic fault in a well-typed numeric computation.

To rule out this possibility we introduce a behavioral type discipline that allows us to express the expectation that certain methods are well-defined on certain classes (or, equivalently, that certain classes admit certain methods as well-defined on their instances). Specifically, we will use a semantic form of *type refinements* of the kind introduced by Freeman and Pfenning (1991) and further developed by Davies and Pfenning (2000); Davies (2005). According to the semantic viewpoint, a type refinement is a predicate (or, more generally, a relation) on a structural type that respects observational equivalence, so that expressions that behave the same way enjoy the same properties. The behavior of dynamic dispatch may be specified by refining the type of the dispatch matrix to express, for example, the expectation that certain methods are well-defined on certain classes. Richer properties of dynamic dispatch may be specified in a similar manner. For example, we may express invariants on the instance data of certain classes (for example, that an integer is always positive) or properties of the results of certain methods (for example, that it return a non-negative number). The critical subsumption property (Cardelli 1988) of type disciplines for object-oriented programming is expressible using logical entailments between refinements, allowing us to support verification in the presence of a class hierarchy.

To assess the viability of our approach, we give an interpretation of Featherweight Java (Igarashi et al. 1999) in terms of the structural formulation of dynamic dispatch to account for its dynamics. We then introduce a system of type refinements derived from the Featherweight Java class table to express the expectation that certain methods are well-defined on certain classes. We then prove that well-typed and well-refined programs cannot incur a "not understood" error, but may still incur a "down-cast error", replicating the guarantees provided by the Featherweight Java type system. Previous work (Davies 2005; Dunfield 2007; Xi and Pfenning 1998) suggests that other conditions, such as absence of down-cast errors or array bounds errors, may be verified in a similar manner. By generalizing from predicates to binary relations it also appears possible to verify equational properties of programs, such as the Liskov-Wing subtyping criterion, in a similar manner. In this respect our approach coheres with the trend to integrate verification of program properties into the development process, allowing us to express a variety of properties of programs that are not easily achievable using purely structural techniques.

2 Background

We will work in a background structural type theory with finite products and sums; function types; general recursive types; predicative polymorphic types; and an error monad with two forms of error. Detailed descriptions of these standard typing constructs may be found in (Harper 2012). We make no use of subtyping, of higher kinds, or of any of the more advanced forms of polymorphism found in the literature (not even impredicativity). Our treatment of the error monad follows the judgmental formulation given by Pfenning and Davies (2001) in which there is a modal separation between *expressions* of a type, which may diverge, but otherwise evaluate to a value of that type, and *commands* of a type, which may incur a run-time error (that is, an uncaught exception) when evaluated. We confine ourselves to functional behavior, and do not consider mutation in this brief account.

The syntactic skeleton of our language, **L**, is given by the following grammar:

Type	τ	$::=$	t	type variable
			$\langle\tau\rangle_{i\in I}$	finite product
			$[\tau_i]_{i\in I}$	finite sum
			$\tau_1 \rightharpoonup \tau_2$	partial function
			$\mu\, t.\tau$	type recursion
			$\forall\, t\,.\,\tau$	type abstraction
			$\tau\ \mathtt{cmd}$	encapsulated command
Expression	e	$::=$	x	value variable
			$\mathtt{cmd}\, k$	encapsulated command
			$\ldots$	

Command	k	::=	$\texttt{ret}\,e$	return a value
			$\texttt{bnd}\,x \leftarrow e\,;k$	sequence
			`error`	signal an error
			`fail`	signal a failure

The finite product $\langle\tau\rangle_{i\in I}$ and sum $[\tau_i]_{i\in I}$ types are indexed by a finite set, I, of indices, which may be construed as position numbers or field labels. The finite product and sum types are often written in the display forms $\prod_{i\in I}\tau$ and $\sum_{i\in I}\tau$. Function types, $\tau_1 \rightharpoonup \tau_2$ classify partial (possibly divergent) functions so as to be compatible with general recursive types, $\mu\, t.\tau$. Polymorphic types, $\forall\, t\,.\,\tau$, express (predicative) type abstraction. Existentials are definable from polymorphic types in the usual way, and are sufficient for our purposes. Much of the syntax of expressions is elided for the sake of brevity, but is largely standard.[1]

The command type $\tau\;\texttt{cmd}$ represents an error monad formulated in the style of Pfenning and Davies (2001). We consider two forms of error, one that is deemed permissible in a normal execution, and one that is deemed impermissible and should be ruled out by verification. (In Section 4 down-cast errors are considered permissible, and not-understood errors are considered impermissible.) A permissible error is signaled by `error`, and an impermissible error is signaled by `fail`. A non-error return is effected by the command $\texttt{ret}\,e$, where e is a pure expression, rather than another command. The command $\texttt{bnd}\,x \leftarrow e\,;k$ evaluates e to an encapsulated command, evaluates it, possibly incurring a failure or an error, which are propagated, and otherwise passes the return value to the command k. The command type $\tau\;\texttt{cmd}$ is equivalent to the delayed sum type

$$\langle\rangle \rightharpoonup [\texttt{ret} \hookrightarrow \tau, \texttt{error} \hookrightarrow \langle\rangle, \texttt{fail} \hookrightarrow \langle\rangle] \qquad \text{(i.e., equivalent to } 1 \rightharpoonup \tau + 2\text{).}$$

The monadic bind is then an implied three-way case analysis in which the error cases are propagated implicitly, and the return case is handled by the continuation of the bind. This simplifies programming, and is sufficient for our purposes. We note that an error monad does not incur the complications with refinements in the presence of general computational effects considered by Davies and Pfenning (2000) and Dunfield and Pfenning (2003).
The static semantics of **L** is given by three forms of typing judgment:

$\Delta \vdash \tau\ \mathsf{type}$	type formation
$\Gamma \vdash_\Delta e : \tau$	expression typing
$\Gamma \vdash_\Delta k \mathrel{\dot{\sim}} \tau$	command typing

The definitions of these judgments are largely standard, and omitted here. For the sake of clarity, we give the rules for the command types, which are less familiar.

[1]See, for example, Harper (2012) for more details.

$$\frac{\Gamma \vdash_\Delta k \mathrel{\dot{\sim}} \tau}{\Gamma \vdash_\Delta \texttt{cmd}\, k : \tau\ \texttt{cmd}}$$

$$\frac{}{\Gamma \vdash_\Delta \texttt{error} \mathrel{\dot{\sim}} \tau} \qquad \frac{}{\Gamma \vdash_\Delta \texttt{fail} \mathrel{\dot{\sim}} \tau} \qquad \frac{\Gamma \vdash_\Delta e : \tau}{\Gamma \vdash_\Delta \texttt{ret}\, e \mathrel{\dot{\sim}} \tau}$$

$$\frac{\Gamma \vdash_\Delta e : \tau_1\ \texttt{cmd} \quad \Gamma, x : \tau_1 \vdash_\Delta k \mathrel{\dot{\sim}} \tau_2}{\Gamma \vdash_\Delta \texttt{bnd}\, x \leftarrow e\, ; k \mathrel{\dot{\sim}} \tau_2}$$

The dynamic semantics of **L** is given by the following judgments:

e val	evaluated expression
$e \mapsto e'$	expression transition
k err	run-time error
k fail	run-time failure
k final	completed computation
$k \mapsto k'$	command transition

The first two define the final states and transition of expression evaluation. The second two define the error states and transition for commands. The expression `cmd` k is a value, regardless of the form of k; it represents a suspended expression that may incur a failure or error when executed. The command `ret` e is fully executed when e val; any errors or failures arising within a command are propagated as such.[2] Note that error and failure are observable outcomes of complete programs; these are used in the definition of Kleene equivalence, which states that two pograms either both diverge or have the same observable outcomes.

We now formulate a system of type refinements in the style of Freeman and Pfenning (1991) and Davies (2005). A refinement, ρ, of a type, τ, is, in general, a relation on the elements of τ. Davies and Pfenning considered only unary relations, which is all that are required here, but is useful to consider binary relations to express deeper properties of programs, as in Denney (1998). We depart from Davies and Pfenning, however, in treating refinements semantically, rather than syntactically. In their work refinements are formulated as a syntactic type discipline, with emphasis on decidability of refinement checking. Here we stress the semantics of refinements, leaving mechanical verification as a separate, albeit but important, practical matter.

[2]The full definition of the static and dynamic semantics of **L**, and the proof of its type safety, may be found in Harper (2012).

The syntax of refinements is given as follows:

ρ	$::=$	r	variable	$\rho_1 \rightharpoonup \rho_2$	partial function
		$\top$	truth	$\forall\,(t \sqsupseteq \vec{r} : \theta) \mathbin{.} \rho$	generic family
		$\bot$	falsity	$i \cdot \rho$	summand
		$\rho_1 \wedge \rho_2$	conjunction	$\mu\,\vec{r}.\vec{\rho}\ \mathtt{in}\ r$	recursive
		$\rho_1 \vee \rho_2$	disjunction	$\mathtt{ret}\,\rho$	normal return
		$\langle\rho\rangle_{i \in I}$	product	$\mathtt{error}$	error
				$\mathtt{fail}$	failure

The logical refinements represent finite conjunctions and disjunctions of properties of any fixed type. The product, function, and command refinements represent the action of their corresponding types on predicates. The summand refinements specify, for a finite sum type, a summand and refinement of its underlying value. The finite sum refinement may be defined by the equation

$$\sum_{i \in I} \rho_i \triangleq \bigvee_{i \in I} (i \cdot \rho_i),$$

the disjunction of all of its summand refinements. The recursive refinement $\mu\,\vec{r}.\vec{\rho}$ `in` r specifies one of a set of mutually recursive properties of the recursive unrolling of a value of a recursive type. The command refinements, `error`, `fail`, and `ret` ρ, are just summand refinements for the sum type underlying the command refinement, as discussed earlier.

The generic refinement requires further explanation. Following the treatment of abstract refinements for Standard ML modules by Davies (2005), the refinement $\forall\,(t \sqsupseteq \vec{r} : \theta) \mathbin{.} \rho$ refines the polymorphic type $\forall\, t \mathbin{.} \tau$ by introducing a type variable, t, a finite set, $\vec{r}$, of variables refining t, and a finite set of *entailment assumptions*, θ, involving the variables $\vec{r} \sqsubseteq t$. The generic refinement may be seen as the behavioral analogue of bounded quantification (Cardelli and Wegner 1985), but with the freedom to introduce a finite set of abstract refinements satisfying a specified set of entailments.

The *entailment judgment* $\rho_1 \leq_\tau \rho_2$ between two refinements of τ states any closed expression of type τ that satisfies ρ_1 also satisfies ρ_2. Entailment may be seen as the behavioral analogue of structural subtyping. If θ is a finite set of refinement assumptions $\rho_i \leq_\tau \rho_i'$, then the hypothetical judgment $\theta \vdash_\tau \rho \leq_\tau \rho'$ states that whenever the entailments in θ are valid, then so is $\rho \leq_\tau \rho'$. We write Θ for a family of refinement assumptions θ_τ indexed over types τ. This notation often arises when the types τ range over a given set of type variables Δ.

The *expression refinement* judgment has the form

$$x_1 \in_{\tau_1} \rho_1, \ldots, x_n \in_{\tau_n} \rho_n \vdash_\Theta e \in_\tau \rho,$$

where Θ is a family of refinement assumptions for Δ, $\Theta \vdash \rho_i \sqsubseteq \tau_i$ (for each i), $\Theta \vdash_\Delta \rho \sqsubseteq \tau$, and $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash_\Delta e : \tau$.

The semantics of the basic refinement judgments is given by assigning to each refinement $\rho \sqsubseteq \tau$ a subset of the closed expressions, modulo observational equivalence, of the type τ, and similarly for closed commands, which are expressions of a distinguished sum type. The details of the construction of such an interpretation are too involved to present here, but the required techniques are well-understood.[3] The semantics of refinements enjoys these properties:

$$
\begin{array}{rll}
e \in_\tau \top & \text{iff} & \text{always} \\
e \in_\tau \bot & \text{iff} & \text{never} \\
e \in_\tau \rho_1 \wedge \rho_2 & \text{iff} & e \in_\tau \rho_1 \text{ and } e \in_\tau \rho_2 \\
e \in_\tau \rho_1 \vee \rho_2 & \text{iff} & e \in_\tau \rho_1 \text{ or } e \in_\tau \rho_2 \\
e \in_{\langle \tau \rangle_{i \in I}} \langle \rho \rangle_{i \in I} & \text{iff} & e \cdot i \in_{\tau_i} \rho_i \ (\forall i \in I) \\
e \in_{\tau_1 \rightharpoonup \tau_2} \rho_1 \rightharpoonup \rho_2 & \text{iff} & e_1 \in_{\tau_1} \rho_1 \text{ implies } e\texttt{(}e_1\texttt{)} \in_{\tau_2} \rho_2 \\
i \cdot e_i \in_{[\tau_i]_{i \in I}} i \cdot \rho_i & \text{iff} & e_i \in_{\tau_i} \rho_i \\
\texttt{fold(}e\texttt{)} \in_{\mu\, t.\tau} \mu\, \vec{r}.\vec{\rho} \ \texttt{in}\ r_i & \text{iff} & e \in_{[\mu\, t.\tau/t]\tau} [\mu\, \vec{r}.\vec{\rho}\ \texttt{in}\ r_1/r_1, \ldots, \mu\, \vec{r}.\vec{\rho}\ \texttt{in}\ r_n/r_n]\rho_i \\
e \in_{\forall\, t.\tau} \forall\,(t \sqsupseteq \vec{r} : \theta)\,.\,\rho & \text{iff} & e\texttt{[}\sigma\texttt{]} \in_{[\sigma/t]\tau} [\vec{\rho}/\vec{r}]\rho \quad \text{for all} \quad \sigma, \vec{\rho} \sqsubseteq \sigma \ \textit{sat.}\ \theta \\
\texttt{cmd}\, k \in_{\tau\, \texttt{cmd}} \rho & \text{iff} & k \in_\tau \rho \\
\texttt{ret}\, e \in_\tau \texttt{ret}\, \rho & \text{iff} & e \in_\tau \rho \\
\texttt{error} \in_\tau \texttt{error} & \text{iff} & \text{always} \\
\texttt{fail} \in_\tau \texttt{fail} & \text{iff} & \text{always}
\end{array}
$$

In the clause for refinements of $\forall\, t.\tau$ we quantify over refinements $\vec{\rho} = \{\rho_1, \ldots, \rho_n\}$ of the (monomorphic) type σ such that the entailments $[\vec{\rho}/\vec{r}]\theta$ over σ are all valid. Generally, an entailment $\rho_1 \leq_\tau \rho_2$ is valid iff whenever $e \in_\tau \rho_1$, then $e \in_\tau \rho_2$, and this extends to sets of entailments conjunctively.

3 Dynamic Dispatch

3.1 Structural Typing

Consider a system defining a finite set, M, of methods acting on data objects classified by a finite set, C, of classes. Associated to each class $c \in C$ is a type τ^c, called the *instance type* of c, the type of c, that classifies the instance data of that class. Associated to each method $m \in M$ is a type τ_m, called the *result type*

[3]The main difficulty is with recursive types, for which see Pitts (1996); Crary and Harper (2007); Harper (2012)). Using only predicative polymorphism considerably simplifies the construction. The refinements of the concrete sum monad for errors are interpreted the same as refinements for functions returning sums, similar to a simple, unary version of the PER semantics for monadic refinements for exceptions given by Benton and Buchlovsky (2007).

of m, that classifies the result of that method when applied to some data object. Such a system may be concisely described as an element of the type

$$\tau_{\mathsf{het}} \triangleq (\sum_{c \in C} \tau^c) \rightharpoonup (\prod_{m \in M} \tau_m)$$

parameterized by the choice of classes and methods and their associated instance and result types. It describes a collection of methods each acting on data of one of a collection of classes, which is an instance of the general concept of heterogeneous programming available in any language with products and sums.

By a de Morgan-type duality there is an isomorphism between τ_{het} and the type τ_{dm} defined by the equation

$$\tau_{\mathsf{dm}} \triangleq \prod_{c \in C} \prod_{m \in M} (\tau^c \rightharpoonup \tau_m).$$

The type τ_{dm} describes a *dispatch matrix* of dimension $|C| \times |M|$, with rows indexed by classes and columns indexed by methods. The entry, e^c_m, of the dispatch matrix defines the *behavior* of method m on instances of c as a function of type $\tau^c_m \triangleq \tau^c \rightharpoonup \tau_m$ mapping τ^c, the instance type of c, to τ_m, the result type of m. The class c and method m may be thought of as the *coordinates* of the behavior of method m on instances of class c.

Any matrix may be seen as a row of columns or a column of rows. In the case each row $c \in C$ of the dispatch matrix determines the behavior of methods M on instances of the class c. Thus the dispatch matrix may be seen as C-indexed column of methods acting on the instance data of c:

$$\tau_{\mathsf{dm}} \cong \prod_{c \in C} (\tau^c \rightharpoonup (\prod_{m \in M} \tau_m)).$$

Dually, each column $m \in M$ of the dispatch matrix determines the behavior of m on the instances of each of the classes C. Thus the dispatch matrix may also be seen as an M-indexed column of results for each possible instance:

$$\tau_{\mathsf{dm}} \cong \prod_{m \in M} (\sum_{c \in C} \tau^c) \rightharpoonup \tau_m.$$

In view of these isomorphisms neither organization can be seen as more significant than the other. They are, rather, equivalent descriptions of the information encoded in the dispatch matrix.

Dynamic dispatch is an implementation of the abstract type

$$\tau_{\mathsf{dd}} \triangleq \exists(t_{\mathsf{obj}} \,.\, \langle \mathtt{new} \hookrightarrow \prod_{c \in C} \tau^c \rightharpoonup t_{\mathsf{obj}}, \mathtt{snd} \hookrightarrow \prod_{m \in M} t_{\mathsf{obj}} \rightharpoonup \tau_m \rangle),$$

which specifies a type, t_{obj}, of objects on which are defined two families of operations, *instantiation* and *messaging*, which are, respectively, the introductory and eliminatory forms of the object type. The intended behavior is that sending a message m to an instance of class c engenders the behavior given by the dispatch matrix with coordinates c and m. Clients of this package are equipped with instantiation and messaging operations

$$\frac{\Gamma \vdash_\Delta e : \tau^c}{\Gamma \vdash_\Delta \texttt{new[}c\texttt{](}e\texttt{)} : t_{\mathsf{obj}}} \qquad \frac{\Gamma \vdash_\Delta e : t_{\mathsf{obj}}}{\Gamma \vdash_\Delta \texttt{snd[}m\texttt{](}e\texttt{)} : \tau_m}$$

Given a dispatch matrix, e_{dm}, we may implement dynamic dispatch τ_{dd} in two equivalent ways, by defining a representation type, τ_{obj}, and an associated *class*, or *constructor, vector*, e_{cv}, of type

$$\tau_{\mathsf{cv}}(\tau_{\mathsf{obj}}) \triangleq \prod_{c \in C} (\tau^c \rightharpoonup \tau_{\mathsf{obj}}),$$

and a *method*, or *message, vector*, e_{mv}, of type

$$\tau_{\mathsf{mv}}(\tau_{\mathsf{obj}}) \triangleq \prod_{m \in M} (\tau_{\mathsf{obj}} \rightharpoonup \tau_m).$$

We will consider two equivalent implementations of the dynamic dispatch abstraction. The *method-based*, or *sum*, form of dynamic dispatch is given by the following definitions:

$$\begin{aligned}
\tau^{\Sigma}_{\mathsf{obj}} &\triangleq \sum_{c \in C} \tau^c \\
e_{\mathsf{cv}} &\triangleq \langle c \hookrightarrow \lambda\,(x : \tau^c)\ [c \hookrightarrow x]\rangle_{c \in C} \\
e_{\mathsf{mv}} &\triangleq \langle m \hookrightarrow \lambda\,(\mathit{this} : \tau^{\Sigma}_{\mathsf{obj}})\ \texttt{case}\ \mathit{this}\,\{[c \hookrightarrow x] \Rightarrow e_{\mathsf{dm}} \cdot c \cdot m(x)\}_{c \in C}\rangle_{m \in M}.
\end{aligned}$$

The *class-based*, or *product*, form of dynamic dispatch is given by the following definitions:

$$\begin{aligned}
\tau^{\Pi}_{\mathsf{obj}} &\triangleq \prod_{m \in M} \tau_m \\
e_{\mathsf{cv}} &\triangleq \langle c \hookrightarrow \lambda\,(x : \tau^c)\ \langle m \hookrightarrow e_{\mathsf{dm}} \cdot c \cdot m(x)\rangle_{m \in M}\rangle_{c \in C} \\
e_{\mathsf{mv}} &\triangleq \langle m \hookrightarrow \lambda\,(x : \tau^{\Pi}_{\mathsf{obj}})\ x \cdot m\rangle_{m \in M}.
\end{aligned}$$

For either choice of implementation the instantiation and messaging operations behave by deferral to the constructor and messaging vectors, respectively:

$$\begin{aligned}
\texttt{new[}c\texttt{](}e\texttt{)} &\longmapsto^* (e_{\mathsf{cv}} \cdot c)(e) \\
\texttt{snd[}m\texttt{](}e\texttt{)} &\longmapsto^* (e_{\mathsf{mv}} \cdot m)(e),
\end{aligned}$$

whenever e is a value of appropriate type. Then, by construction, we have in either case that

$$\texttt{snd}[m]\texttt{(new}[c]\texttt{(}e\texttt{))} \mapsto^* \texttt{(}e_{\mathsf{dm}} \cdot m \cdot c\texttt{)(}e\texttt{)},$$

again under the condition that e is a value. This property may be seen as characterizing dynamic dispatch (Igarashi et al. 1999) in that sending a message m to an instance of class c engenders the behavior assigned to m on c by the dispatch matrix.

This basic model of dynamic dispatch may be elaborated to account for several forms of self-reference found in object-oriented languages:

1. Any method may call any other, including itself.
2. Any class may create an instance of any other, including itself.
3. The instance type of a class may involve any object.
4. The result type of a method may involve any object.

Scaling up to allow for these behaviors is largely a matter of generalizing the type τ_{dm}, choosing τ_{obj} to be a recursive type, and making corresponding changes to the class and method vectors, based on the choice of τ_{obj}. The details of the construction can be found in Harper (2014), but may be briefly summarized as follows.

The types of the components of the dispatch matrix must be changed so that they have access to the class vector (for creating new instances) and the method vector (for sending messages to instances). Moreover, the instance type of each class and the result type of each method may involve instances created in this manner. Thus, the components of the dispatch matrix are given the (predicative) polymorphic type

$$\tau_m^c \triangleq \forall\, t_{\mathsf{obj}} \,.\, \tau_{\mathsf{cv}}(t_{\mathsf{obj}}) \rightharpoonup \tau_{\mathsf{mv}}(t_{\mathsf{obj}}) \rightharpoonup \tau^c(t_{\mathsf{obj}}) \rightharpoonup \tau_m(t_{\mathsf{obj}}).$$

The type variable, t_{obj}, is the abstract type of objects with which the behaviors interact via the class- and method vectors.behaviors provided as arguments.

In the method-based (sum) form the type τ_{obj} of objects is defined by the equation

$$\tau_{\mathsf{obj}}^{\Sigma} \triangleq \mu\, t_{\mathsf{obj}}.\sum_{c \in C} \tau^c(t_{\mathsf{obj}}),$$

whereas in the class-based (product) form the type τ_{obj} is defined by the equation

$$\tau_{\mathsf{obj}}^{\Pi} \triangleq \mu\, t_{\mathsf{obj}}.\prod_{m \in M} \tau_m(t_{\mathsf{obj}}).$$

The implementations of the method and class vectors in terms of the dispatch matrix are slightly more involved than before, because the object types are recursive (requiring folding and unfolding operations), and either the method vector (in the sum form) or the class vector (in the product form) must be self-referential using standard fixed-point operations.

Finally, we observe that it is not necessary for every method to be meaningfully defined on every class of object. More precisely, an ill-defined situation may be defined as one that signals a run-time error corresponding to the "message not understood" error described in the introduction to this paper. This amounts to choosing τ_m, the result type of method m, to admit the possibility of a run-time fault, which may be accomplished using the error monad described in Section 2. Once this possiblity is allowed, it becomes important to specify and verify that certain method and class combinations are sensible, which we view as a behavioral, rather than structural, property of a program.

3.2 Behavioral Typing

As we have seen in the preceding section, dynamic dispatch is a form of heterogeneous programming in which the behavior of a collection of methods is defined on the instances of a collection of classes. In some cases the behavior is to give rise to a "not understood" error, reflecting that the particular combination is ill-defined. The expectation that a method m be defined on every instance of a class c is not inherent in the idea of dynamic dispatch, but is rather a *methodological consideration* imposed from the outside, much as one might insist as a matter of methodology that other forms of run-time fault are to be precluded. Indeed, following Cardelli's principle, one might say that what makes dynamic dispatch, a mode of use of recursive products and sums, be "object-oriented" is just that such expectations are stated and and enforced for each program (for example, by decalarations that form a "class table" for a program). More generally, one may wish to enforce many other methodological conditions, such as absence of "down-cast" errors, or avoidance of "bound check" errors, not all of which can be anticipated in a particular structural type system.

In Section 4 we will carry out a full-scale verification of the absence of "not understood" messages for an interpretation of **FJ** as an application of dynamic dispatch. Here we outline the general approach to verification of properties of dynamic dispatch using type refinements. For the sake of clarity , we first consider the non-self-referential case of dynamic dispatch; this makes it easier to explain the generalization to admit self-reference. To carry out a verification of the properties of dynamic dispatch involves the following ingredients:

1. A family of refinements $\rho^c_m \sqsubseteq \tau^c_m$, which constrains the behavior of the

entries of the dispatch matrix. This family determines a refinement $\rho_{\mathsf{dm}} \sqsubseteq \tau_{\mathsf{dm}}$ given by

$$\rho_{\mathsf{dm}} \triangleq \prod_{c \in C} \prod_{m \in M} \rho^c_m \sqsubseteq \prod_{c \in C} \prod_{m \in M} \tau^c_m.$$

2. A family of refinements $\rho^c_{\mathsf{obj}} \sqsubseteq \tau_{\mathsf{obj}}$, for each $c \in C$, and $\rho^m_{\mathsf{obj}} \sqsubseteq \tau_{\mathsf{obj}}$, for each $m \in M$. In Section 4 we will choose the refinement ρ^c_{obj} to express that an object is an instance of class c, and the refinement ρ^m_{obj} to express that an object understands method m.

3. A refinement $\rho^c \sqsubseteq \tau^c$ characterizing the instance data of class c. The instance refinement determines a refinement of the class vector type given by

$$\rho_{\mathsf{cv}} \triangleq \prod_{c \in C} (\rho^c \rightharpoonup \rho^c_{\mathsf{obj}}) \sqsubseteq \prod_{c \in C} (\tau^c \rightharpoonup \tau_{\mathsf{obj}}).$$

   The refinement ρ_{cv} states that if the instance data satisfies ρ^c, then the resulting instance will be an object that satisfies ρ^c_{obj}.

4. A refinement $\rho_m \sqsubseteq \tau_m$ characterizing the result of method m. The result refinement determines a refinement of the method vector type given by

$$\rho_{\mathsf{mv}} \triangleq \prod_{m \in M} (\rho^m_{\mathsf{obj}} \rightharpoonup \rho_m) \sqsubseteq \prod_{m \in M} (\tau_{\mathsf{obj}} \rightharpoonup \tau_m).$$

   The refinement ρ_{mv} states that if an object satisfies ρ^m_{obj}, then the result of method m will satisfy ρ_m.

5. Because τ^c_m is $\tau^c \rightharpoonup \tau_m$, the refinement ρ^c_m must satisfy the entailment $\rho^c_m \leq \rho^c \rightharpoonup \rho_m$ so that if ρ^c_m holds for matrix entry e^c_m, instances satisfying ρ^c are mapped to results satisfying ρ_m.

These choices determine verification conditions that ensure that dynamic dispatch is well-behaved. We must ensure that $e_{\mathsf{dm}} \in \rho_{\mathsf{dm}}$, which is to say that $e^c_m \in \rho^c_m$ for each behavior e^c_m, and then we must show $e_{\mathsf{cv}} \in \rho_{\mathsf{cv}}$ and that $e_{\mathsf{mv}} \in \rho_{\mathsf{mv}}$, making use of this fact. In sum form the method vector condition follows directly from the fact that $e_{\mathsf{dm}} \in \rho_{\mathsf{dm}}$, but the class vector condition must be checked for the choice of ρ^c and ρ^c_{obj}. In product form the dual situation obtains: the class vector condition follows from the verification of the dispatch matrix, and the method vector condition must be verified for the choice of ρ^m_{obj} and ρ_m.

These conditions ensure that dynamic dispatch satisfies the following properties:

1. if $e \in \rho^c$, then `new[`c`](`e`)` $\in \rho^c_{\mathsf{obj}}$, and

2. if $e \in \rho^{m}_{\mathsf{obj}}$, then $\texttt{snd[}m\texttt{](}e\texttt{)} \in \rho_m$.

In the case that τ_m is a command type $\tau'_m\ \texttt{cmd}$, indicating that method m may fail when invoked, then some additional conditions are required to ensure that "message not understood" errors are avoided. Specifically, if instances of c are to admit method m, then we require the following conditions:

1. Failure is not an option: $\rho^{c}_{m} \leq \rho^{c} \rightharpoonup \texttt{ret}\,\rho'_m \vee \texttt{error}$, for some ρ'_m such that $\rho'_m \sqsubseteq \tau'_m$.

2. Any object satisfying ρ^{c}_{obj} must satisfy ρ^{m}_{obj}: $\rho^{c}_{\mathsf{obj}} \leq \rho^{m}_{\mathsf{obj}}$.

These further conditions ensure that if $e \in \rho^c$, then

$$\texttt{snd[}m\texttt{](new[}c\texttt{](}e\texttt{))} \in \texttt{ret}\,\rho'_m \vee \texttt{error},$$

which is to say that sending m to an instance of c cannot fail.

The self-referential case is handled similarly, with some additional complications arising because the entries in the dispatch matrix are polymorphic in the object type and abstracted with respect to the class and method vectors. The ingredients are as follows:

1. As before, a family of refinements $\rho^{c}_{m} \sqsubseteq \tau^{c}_{m}$ characterizing the behavior of method m on instances of class c as specified by the dispatch matrix.

2. As before, a family of refinements, $\vec{\rho}$, consisting of refinements $\rho^{c}_{\mathsf{obj}} \sqsubseteq \tau_{\mathsf{obj}}$, for each $c \in C$, and $\rho^{m}_{\mathsf{obj}} \sqsubseteq \tau_{\mathsf{obj}}$, for each $m \in M$.

3. Variable refinements $\vec{r}$ consisting of refinements r^c and r_m of the abstract object type t_{obj} for each c and m. These are to be thought of as abstract correlates of the refinements ρ^c and ρ_m of τ_{obj} that will instantiate them when the dispatch implementation is chosen. The refinement variables $\vec{r}$ are governed by a finite set of entailment assumptions, θ, that must be true when τ_{obj} instantiates t_{obj} and $\vec{\rho}$ instantiates $\vec{r}$.

4. As before, instance and result refinements, stated parametrically in t_{obj} and $\vec{r} \sqsubseteq t_{\mathsf{obj}}$, and object refinements for each class and method, also parametrically in the same variables.

5. We require that

$$\rho^{c}_{m} \leq \forall\,(t_{\mathsf{obj}} \sqsupseteq \vec{r} : \theta)\,.\,\rho_{\mathsf{cv}}(\vec{r}) \rightharpoonup \rho_{\mathsf{mv}}(\vec{r}) \rightharpoonup \rho^{c}(\vec{r}) \rightharpoonup \rho_{m}(\vec{r}),$$

   where $\vec{r}$ and θ are the refinement variables and their governing entailments described above.

The last requirement ensures that e^c_m satisfies the instantiation of the polymorphic refinement

$$\rho_{\mathsf{cv}}(\vec{\rho}) \rightharpoonup \rho_{\mathsf{mv}}(\vec{\rho}) \rightharpoonup \rho^c(\vec{\rho}) \rightharpoonup \rho_m(\vec{\rho}),$$

where τ_{obj} is the object type refined by the refinements $\vec{\rho}$ specified above.

A detailed example is given in the next section in which we give an interpretation of **FJ** into **L**, and use refinements to state and prove that "not understood" errors are precluded in well-refined programs.

4 Refining Featherweight Java

4.1 Overview

To demonstrate the suggested separation of structural from behavioral typing, we give a relatively straightforward translation of Featherweight Java (Igarashi et al. 1999) into **L**, then equip it with a system of refinements that ensures that "message not understood" failures cannot arise in a well-refined program. End-to-end we achieve the same safety guarantees as were ensured by the original formulation; our goal here is to show that the proposed reorganization is adequate to achieve the same ends. But, as we shall outline in Section 5, the separation permits consideration of significantly more elaborate verifications than are feasible by increasing the complexity of the structural type system that determines the operational semantics of the language.

The main idea is very simple, particularly if we (temporarily) ignore the self-referential aspects of **FJ** (to which we shall return momentarily). The key step is to translate the **FJ** class table into a dispatch matrix whose entries are commands that either return the behavior of method m on class c when it is defined by the class table, or signal a failure to indicate that it is not defined.

The main idea of the verification hinges on the definition of two forms of refinement particular to the problem at hand, `inst[`c`]` and `recog[`m`]`, which refine the type τ_{obj}, regardless of whether it is chosen to be of sum or product form. Informally, an object $o : \tau_{\mathsf{obj}}$ is an instance of class c is defined *semantically* to mean that for every method m associated to class c in the **FJ** class table, the object o does not fail when method m is invoked on it. Notice that o is not required to have arisen from the constructor of class c, but could be any object that behaves in the way that such an object would (that is, it could be an instance of a subclass of c). This semantic instance property is certainly not decidable, but this is not relevant to our purposes. Similarly, an object $o : \tau_{\mathsf{obj}}$ recognizes the method m if any instance of any class declared to have m in the class table does not fail when sent message m. This, too, is not decidable, but this is not relevant for our purposes. What is relevant is that the semantic definition of `inst[`c`]` is not defined by declaration, and does not reflect the history of how the object was created, but is instead a description of its behavior when executed. This ensures that the subsumption principle of **FJ** is validated under our interpretation.

Following the methodology outlined in Section 3, we will set up a system of refinements that ensures that no "message not understood" errors can arise. The instance refinement, $\rho^c \sqsubseteq \tau^c$, is chosen as the product of the class types declared for the instance variables $\prod_{cf \in F^c} \rho^c_{\mathsf{obj}}$. The result refinement ρ_m is chosen to be

$$\mathtt{ret}\,((\prod_{c\,i \in \overline{\mathsf{arg}}_m} \rho^c_{\mathsf{obj}}) \rightharpoonup (\mathtt{ret}\,\rho^{\mathsf{res}_m}_{\mathsf{obj}} \vee \mathtt{error}))$$

expressing the typing conditions augmented with the possibility of a "downcast" error as outcome of the method body. In the case that the class table does not associate m with c, we instead choose ${\rho^c_m}'$ as

$$\mathtt{fail} \sqsubseteq \tau_m$$

reflecting that it is a "not understood" message. In either case we have

$$\rho^c \rightharpoonup \rho_m \sqsubseteq \tau^c \rightharpoonup \tau_m.$$

The refinements ρ^c_{obj} are chosen to be $\mathtt{inst}[c]$ for each $c \in C$, and the refinements ρ^m_{obj} are chosen to be $\mathtt{recog}[m]$ for each m in M. These choices ensure that the class vector entry at c creates objects that are, semantically, instances of c, and that the method vector entry at m delivers a non-error result when applied to the instance data of a class for which it is defined. We note that if m occurs in the class table entry for c, then $\mathtt{inst}[c] \leq \mathtt{recog}[m]$, which states that an instance of c admits the message m, as would be expected. Similarly, **FJ** has the property that if $c <: c'$ then every method m is the class table entry for c if it is in the class table entry for c', thus $\mathtt{inst}[c] \leq \mathtt{inst}[c']$. These choices determine the refinements of the class and method vectors, as described in Section 3.

To complete the verification we need only check that the dispatch matrix derived from the **FJ** class table, and the associated class and method vectors satisfy the stated refinements, which they shall do by construction. This guarantees the well-behavior of dynamic dispatch in the sense described in Section 3, which ensures that "message not understood" cannot arise.

The main additional complication to account for self-reference is that we must, as outlined in Section 3, choose abstract refinement variables r^c and r_m that refine the abstract type t_{obj} of objects, together with a set of entailments, θ, that will be true for ρ^c_{obj} and ρ^m_{obj} when t_{obj} is instantiated to τ_{obj}. Within the dispatch matrix the code makes use of these assumptions, just as it would have made use of the refinement entailments that are true of $\mathtt{inst}[c]$ and $\mathtt{recog}[m]$ in the non-self-referential case. The result proceeds along similar lines to those outlined above.

4.2 Compiling **FJ** to **L**

The following presentation of **FJ** follows the CBV version by Pierce (2002). One difference is that we use $\boldsymbol{m}$ for **FJ** method names, because we need to distinguish these from the method names m in the interpretation in **L**. For classes c the two coincide.

The syntax of **FJ** is a subset of Java, aside from top-level programs, which consist of a class table and an expression to evaluate.

class declarations	CL	$::=$	$\texttt{class}\, c\, \texttt{extends}\, c\ \{\overline{c}\ \overline{f};K\ \overline{ME}\}$
constructor declarations	K	$::=$	$c(\overline{c}\ \overline{f})\ \{\texttt{super}(\overline{f});\ \mathit{this}.\overline{f} = \overline{f};\}$
method declarations	ME	$::=$	$c\ \boldsymbol{m}(\overline{c}\ \overline{x})\ \{\texttt{return}\ T;\}$
terms	T	$::=$	$x \mid T.f \mid \texttt{new}\, c(\overline{T}) \mid T \cdot \boldsymbol{m}(\overline{T}) \mid (c)\ T$
values	V	$::=$	$\texttt{new}\, c(\overline{V})$
class table	CT	$::=$	$\overline{CL}$
program	P	$::=$	(CT, T)

We now instantiate our framework to show a relatively straightforward compilation of **FJ** programs to **L** types and expressions. To ease the presentation and aid comparison we adopt some similar notation to **FJ**, including having a single implicit global class table CT.

We rely on the following auxiliary definitions from the presentation of **FJ** by Pierce (2002):

$$\texttt{mtype}(\boldsymbol{m}, c) \quad \texttt{mbody}(\boldsymbol{m}, c) \quad \texttt{fields}(c) \quad c <: d$$

We also require the following definitions derived from these, which depend on some standard properties of **FJ** like unique fields. For notational convenience we treat method arguments like records with integers as the field names. Also, the Π record type below with $c\,i \in \overline{\mathsf{arg}}_{\boldsymbol{m}}$ constructs a record type with indices i, and in what follows similarly the index excludes the c type declaration.

$$\left.\begin{aligned} \overline{\mathsf{arg}}_{\boldsymbol{m}} &\triangleq (c_1\,1), \ldots, (c_n\,n) \\ \mathsf{res}_{\boldsymbol{m}} &\triangleq c \end{aligned}\right\} \quad \begin{aligned} &\text{where } \texttt{mtype}(\boldsymbol{m}, c') = \overline{c} \to c \text{ for some } c' \\ &\qquad\qquad\qquad (\text{this maps each } \boldsymbol{m} \text{ uniquely}) \end{aligned}$$

$$F \triangleq \{cf \mid cf \in F^{c'} \text{ for some } c'\} \quad (\text{this maps each } f \text{ uniquely})$$

$$F^c \triangleq \texttt{fields}(c)$$

$$\tau^{\mathsf{arg}}_{\boldsymbol{m}}(t_{\mathsf{obj}}) \triangleq \prod_{c\,i \in \overline{\mathsf{arg}}_{\boldsymbol{m}}} t_{\mathtt{obj}}$$

Each **FJ** method $\boldsymbol{m}$ gives rise to a method $\boldsymbol{m}$ in the interpretation. Fields and casting are implemented by adding extra methods: for each field we have a method $\texttt{get}[f]$ and for each class we have a method $\texttt{cast}[c]$. We write m to indicate a method in the interpretation, which may be any of the three forms $\boldsymbol{m}$, $\texttt{get}[f]$ or $\texttt{cast}[c]$.

In Figure 1 we instantiate the framework in **L** in the previous sections by defining the sets of classes and methods, C and M, and the associated types $\tau^c(t_{\mathsf{obj}})$ and $\tau_m(t_{\mathsf{obj}})$. This instantiates the types of the dispatch entries and the types of the method and class vectors from Section 3. Following the convention for **FJ**, these definitions implicitly depend on the **FJ** class table CT for a particular program, and much of the remainder of this section assumes similarly that there is a fixed "global" class table.

Class Types	**Method Types**
$C \triangleq \{c \mid c \text{ is declared in } CT\}$	$M \triangleq \{\boldsymbol{m} \mid \boldsymbol{m} \text{ is declared in } CT\} \cup \{\texttt{get}[f] \mid f \in F\} \cup \{\texttt{cast}[c] \mid c \in C\}$
$\tau^c(t_{\mathsf{obj}}) \triangleq \prod_{c'f \in F^c} t_{\mathsf{obj}}$	$\tau_m(t_{\mathsf{obj}}) \triangleq \tau'_m(t_{\mathsf{obj}})\ \texttt{cmd}$ where $\tau'_{\boldsymbol{m}}(t_{\mathsf{obj}}) \triangleq \tau^{\mathsf{arg}}_{\boldsymbol{m}}(t_{\mathsf{obj}}) \rightharpoonup (t_{\mathsf{obj}}\ \texttt{cmd})$, $\tau'_{\mathtt{get}[f]}(t_{\mathsf{obj}}) \triangleq t_{\mathsf{obj}}$, $\tau'_{\mathtt{cast}[c']}(t_{\mathsf{obj}}) \triangleq t_{\mathsf{obj}}$
$\tau_{\mathsf{cv}}(t_{\mathsf{obj}}) \triangleq \prod_{c \in C} (\tau^c(t_{\mathsf{obj}}) \rightharpoonup t_{\mathsf{obj}})$	$\tau_{\mathsf{mv}}(t_{\mathsf{obj}}) \triangleq \prod_{m \in M} (t_{\mathsf{obj}} \rightharpoonup \tau_m(t_{\mathsf{obj}}))$

$$\boxed{\tau^c_m \triangleq \forall\, t_{\mathsf{obj}}\,.\, \tau_{\mathsf{cv}}(t_{\mathsf{obj}}) \rightharpoonup \tau_{\mathsf{mv}}(t_{\mathsf{obj}}) \rightharpoonup \tau^c(t_{\mathsf{obj}}) \rightharpoonup \tau_m(t_{\mathsf{obj}})}$$

Figure 1: Compiling **FJ** syntax to **L** types

The dispatch matrix entries e^c_m are defined in Figure 2 via an auxiliary command $k^{c,\Gamma}_m$, with the context Γ explicitly indicating which type variables and typed expressions variables are allowed to be free relative to the command, in this case t_{obj}, cv, mv, u and *this*. We adopt the convention that each **FJ** variable has a corresponding **L** variable with the same name. This is particularly convenient in the translation of terms $|T|^{\Gamma}$. Also in what follows we often use substitutions Ψ, to replace free variables with types and expressions as appropriate, such as the substitution for *this* here.

More generally, a context Γ can specify allowed free type variables t and

$$
\begin{array}{rcll}
e^c_m & \triangleq & \Lambda(t_{\mathsf{obj}})\,\lambda\,(cv\!:\!\tau_{\mathsf{cv}}(t_{\mathsf{obj}}))\,\lambda\,(mv\!:\!\tau_{\mathsf{mv}}(t_{\mathsf{obj}})) & \\
 & & \qquad\quad \lambda\,(u\!:\!\tau^c(t_{\mathsf{obj}}))\,\mathtt{cmd}\,\Psi(k^{c,\Gamma}_m) & \\
\text{where } \Gamma & \triangleq & \left(\begin{array}{l} t_{\mathsf{obj}}, cv : \tau_{\mathsf{cv}}(t_{\mathsf{obj}}), mv : \tau_{\mathsf{mv}}(t_{\mathsf{obj}}), \\ u : \tau^c(t_{\mathsf{obj}}), \mathit{this} : t_{\mathsf{obj}} \end{array}\right) & \\
\Psi & \triangleq & \mathit{this} \mapsto cv \cdot c(u) & \\
\text{and } k^{c,\Gamma}_{\boldsymbol{m}} & \triangleq & \mathtt{ret}\,(\lambda\,(\overline{x}\!:\!\tau^{\mathsf{arg}}_{\boldsymbol{m}}(t_{\mathsf{obj}}))\,\|T\|^{\Gamma,\overline{x}:t_{\mathsf{obj}}}) & \text{if } \mathtt{mbody}(\boldsymbol{m}, c) = (\overline{x}, T) \\
 & \triangleq & \mathtt{fail} & \text{if } \mathtt{mbody}(\boldsymbol{m}, c) \text{ is undefined} \\
k^{c,\Gamma}_{\mathtt{get}[f]} & \triangleq & \mathtt{ret}\,(u \cdot f) & \text{if } f \in F^c \\
 & \triangleq & \mathtt{fail} & \text{otherwise} \\
k^{c,\Gamma}_{\mathtt{cast}[c']} & \triangleq & \mathtt{ret}\,\mathit{this} & \text{if } c <: c' \\
 & \triangleq & \mathtt{error} & \text{otherwise}
\end{array}
$$

Figure 2: Dispatch entries for FJ methods $\boldsymbol{m}$, plus $\mathtt{get}[f]$ and $\mathtt{cast}[c']$

allowed free expression variables along with their type which can depend on type variables earlier in the context. This means that t_{obj} is an abstract type in the rest of Γ. Indeed, we consider the initial part of Γ with $t_{\mathsf{obj}}, cv : \tau_{\mathsf{cv}}(t_{\mathsf{obj}}), mv : \tau_{\mathsf{mv}}(t_{\mathsf{obj}})$ to correspond to a client's view of an existential package with type τ_{dd} from Section 3, with cv and mv being the `new` and `snd` components. This is appropriate here because with self-reference the bodies of the dispatch entries are themselves clients of dynamic dispatch abstract type. Γ is augmented with $u : \tau^c(t_{\mathsf{obj}})$ so that the instance data of the object is available and $\mathit{this} : t_{\mathsf{obj}}$ to appropriately allow *this* to appear in the FJ method bodies.

In the definition of $k^{c,\Gamma}_m$ in Figure 2, for each class c the we interpret each of three kinds of methods using the `fail` command appropriately for undefined method bodies and undefined fields and `error` for casts to non-superclasses.

Defined method bodies are translated via an inductive translation $\|T\|^\Gamma$ which we will see shortly. We use a little syntactic sugar here, following FJ, writing $\lambda\,(\overline{x}\!:\!\tau^{\mathsf{arg}}_{\boldsymbol{m}}(t_{\mathsf{obj}}))\,T$ to abbreviate a function binding the variables in $\overline{x}$ to the corresponding components of the argument.

To interpret FJ we only need to use the instance data u in two ways:

- Each get method $\mathtt{get}[f]$ for class c is interpreted as a command returning the f field of $u : \tau^c(t_{\mathsf{obj}})$. f must be present in $\tau^c(t_{\mathsf{obj}})$ if $f \in F^c$, assuming that we've correctly dispatched to $e^c_{\mathtt{get}[f]}$ due to sending method $\mathtt{get}[f]$ to an instance of (exactly) class c. (See the definition of e'_{mv} below.)

- The FJ special variable *this* is substituted by the expression $cv \cdot c(u)$ with type t_{obj} which is equivalent to the object on which the method was called, assuming that we've correctly dispatched to e^c_m due to sending method m to an instance of c. This is then available for the translation of recursive method calls in method bodies (via $\| T \|^{\Gamma, \overline{x}:t_{\mathsf{obj}}}$) and is also used for successful casts.

It's important here that the instance data for a class c is only directly accessed from the $\mathtt{get}[f]$ method implementation for exactly the class c. All other uses of the instance data are via method calls to *this* which dispatch appropriately, hence no subtyping or similar constraints are ever required between the types of instance data of different classes.[4]

We compile **FJ** expressions T as corresponding **L** commands, in a relatively direct way, aside from making the propagation of errors explicit via the monadic bind. The compilation is parameterized by a context Γ that includes **FJ** expression variables in scope to corresponding **L** expression variables, including *this* which is always in scope in **FJ** method bodies. Γ also maps cv and mv to corresponding **L** expressions for the class and method vectors (renaming as necessary to avoid clashes with **FJ** variable names).

$$\begin{array}{rcll}
\| T \|^{\Gamma} & \triangleq & \mathtt{cmd}\, | T |^{\Gamma} & \\[1ex]
| x |^{\Gamma} & \triangleq & \mathtt{ret}\, x & \text{if } x \text{ is in } \Gamma \text{ (including when } x \text{ is } \textit{this}) \\
| \mathtt{new}\, c\mathtt{(}\overline{T}\mathtt{)} |^{\Gamma} & \triangleq & \mathtt{bnd}\, \overline{x} \leftarrow \| \overline{T} \|^{\Gamma} \,;\, \mathtt{ret}\, \mathtt{(}cv \cdot c\mathtt{)}\mathtt{(}\overline{x}\mathtt{)} & \\
| T \cdot m\mathtt{(}\overline{T}\mathtt{)} |^{\Gamma} & \triangleq & \mathtt{bnd}\, x \leftarrow \| T \|^{\Gamma} \,;\, \mathtt{bnd}\, y \leftarrow \mathtt{(}mv \cdot m\mathtt{)}\mathtt{(}x\mathtt{)} \,; & \\
 & & \mathtt{bnd}\, \overline{x} \leftarrow \| \overline{T} \|^{\Gamma} \,;\, \mathtt{bnd}\, w \leftarrow y\mathtt{(}\overline{x}\mathtt{)} \,;\, \mathtt{ret}\, w & \\[1ex]
| T \cdot f |^{\Gamma} & \triangleq & \mathtt{bnd}\, x \leftarrow \| T \|^{\Gamma} \,;\, \mathtt{bnd}\, y \leftarrow \mathtt{(}mv \cdot \mathtt{get}[f]\mathtt{)}\mathtt{(}x\mathtt{)} \,;\, \mathtt{ret}\, y & \\
| \mathtt{(}c\mathtt{)}\; T |^{\Gamma} & \triangleq & \mathtt{bnd}\, x \leftarrow \| T \|^{\Gamma} \,;\, \mathtt{bnd}\, y \leftarrow \mathtt{(}mv \cdot \mathtt{cast}[c]\mathtt{)}\mathtt{(}x\mathtt{)} \,;\, \mathtt{ret}\, y &
\end{array}$$

At this point we can use the expressions e^c_m to interpret the class table CT of an **FJ** program as a typed **L** expression for the dispatch matrix:

$$e_{\mathsf{dm}} = \langle \langle e^c_m \rangle_{m \in M} \rangle_{c \in C}$$

We sketch here two parts of the type correctness theorem for the compilation to **L** of the **FJ** syntax. Because every **FJ** class/type is interpreted as t_{obj}, type correctness in **L** corresponds to **FJ** syntactic correctness, including scoping of variables and consistency of argument counts. We omit some syntactic lemmas such as that subclasses have all the fields of their superclasses with the same

[4]An alternative approach to *this* is to pass it as a separate argument to e^c_m, with an invariant that u and *this* must correspond. This makes little difference here when interpreting **FJ**, but appears to scale better to certain kinds of extensions like run-time inheritance.

type. Note that the **FJ** object types of the fields are not yet relevant, and the presence of specific object types in the lemma statement is simply because **FJ** lacks a judgment witnessing syntactic correctness without also requiring specific object types.

We write $\vdash_{\mathsf{FJ}} \mathit{ME}\,\mathtt{OK}\,\mathtt{in}\,c$ for the **FJ** judgment "ME is a valid method declaration for class c", which implicitly depends on the types declared in CT; see Pierce (2002).

Lemma 1.

1. *If* $\overline{x} : \overline{c} \vdash_{\mathsf{FJ}} T : c$ *and* $\Gamma = t_{\mathsf{obj}}, cv : \tau_{\mathsf{cv}}(t_{\mathsf{obj}}), mv : \tau_{\mathsf{mv}}(t_{\mathsf{obj}})$
   then $t_{\mathsf{obj}}, \overline{x} : t_{\mathsf{obj}} \vdash |T|^{\Gamma} \mathrel{\dot{\sim}} t_{\mathsf{obj}}$.

2. *If* $\mathtt{mtype}(\boldsymbol{m}, c) = \overline{c} \to c_0$ *and* $\mathtt{mbody}(\boldsymbol{m}, c) = (\overline{x}, T_0)$
   and $\vdash_{\mathsf{FJ}} c_0\ m\,(\overline{c}\,\overline{x})\{\mathtt{return}\ \ T_0;\}\ \mathtt{OK}\ \mathtt{in}\,c$
   then $\vdash e^{c}_{m} : \tau^{c}_{m}$.

A consequence of our compilation is that the structure of an **FJ** value V is observable via its translation $k = |V|^{\Gamma}$ in **L**. This is because it is possible to observe errors and non-errors for calls to $\mathtt{cast}[c']$ on v for each c', from which we can determine its class c and then determine (inductively) its fields by calling $\mathtt{get}[f]$. If this observability seems suspect, note that it is required by the definition of **FJ** due to the class and fields of object values being observable everywhere, including in top-level expressions. Further, the compilation can easily be modified to accommodate similar languages which allow the exact class of an object to be hidden (by omitting or restricting the cast methods) or which have private fields (by omitting the get methods and instead using direct access to the instance data).

We now characterize the translations of values via the following inductive definition. (Note that all **FJ** values have the form $\mathtt{new}\,c(\overline{V})$.)

$$|\mathtt{new}\,c(\overline{V})|^{\Gamma}_{\mathsf{val}} \quad \triangleq \quad cv \cdot c(|\overline{V}|^{\Gamma}_{\mathsf{val}})$$

The following lemma shows that for values this is equivalent to the previous translation $|\cdot|^{\Gamma}$, modulo some evaluation steps that reduce $\mathtt{bnd}\,x \leftarrow \mathtt{cmd}\ (\mathtt{ret}\,v)\,;k$ to $[v/x]k$. This is a standard evaluation rule for monadic commands, and easily derived from the sum interpretation of commands described in Section 2. We show some details of the proof just to give the flavor of such proofs.

Lemma 2. *(value translation)*
If $\Gamma = t_{obj}, cv : \tau_{cv}(t_{obj}), mv : \tau_{mv}(t_{obj}), \Gamma'$ *and* $\Psi : \Gamma$ *with both* $\Psi(cv) = e_{cv}$ *and* $\Psi(mv) = e_{mv}$ *terminating (and closed)*
then for all V *we have* $\Psi(|V|^\Gamma) \mapsto^* k$ *and* k *final*
iff $\Psi(|V|^\Gamma_{\mathsf{val}}) \mapsto^* v$ *and* v *val and* $k = \mathtt{ret}\,v$.

Proof. (sketch) By induction on V. We have just one case.
Case: $V = \mathtt{new}\,c(\overline{V})$. Then

$$\boxed{\begin{aligned}&\Psi(|V|^\Gamma)\\ &= \Psi(|\mathtt{new}\,c(\overline{V})|^\Gamma)\\ &= \mathtt{bnd}\,\overline{x} \leftarrow \mathtt{cmd}\,\Psi(|\overline{V}|^\Gamma)\,;\,\mathtt{ret}\,(e_{\mathsf{cv}}\cdot c)\,(\overline{x})\end{aligned}} \qquad \boxed{\begin{aligned}&\Psi(|V|^\Gamma_{\mathsf{val}})\\ &= \Psi(|\mathtt{new}\,c(\overline{V})|^\Gamma_{\mathsf{val}})\\ &= (e_{\mathsf{cv}}\cdot c)\,(\Psi(|\overline{V}|^\Gamma_{\mathsf{val}}))\end{aligned}}$$

[Left $\Longrightarrow$ right] (the other direction is similar)

If LHS $\mapsto^* k$ and k final, then, by inversion on the evaluation, for each V_i there is k_i s.t. $\Psi(|V_i|^\Gamma) \mapsto^* k_i$ and k_i final.

Applying the I.H. to each V_i yields $\Psi(|V_i|^\Gamma_{\mathsf{val}}) \mapsto^* v_i$ and v_i val and $k_i = \mathtt{ret}\,v_i$.

Then LHS $\mapsto^* \mathtt{ret}\,(e_{\mathsf{cv}}\cdot c)\,(\overline{v})$ and RHS $\mapsto^* (e_{\mathsf{cv}}\cdot c)\,(\overline{v})$.

But then $\mathtt{ret}\,(e_{\mathsf{cv}}\cdot c)\,(\overline{v}) \mapsto^* k$ (since LHS $\mapsto^* k$, and $\mapsto$ is deterministic).

Thus $k = \mathtt{ret}\,v$ for some v with v val and RHS $\mapsto^* v$, as required. □

4.3 Class and method vectors (sum-based)

Now, so far nothing in our **FJ** compilation is specific to the sum-based or product-based organization. But, to have a concrete verification of a complete framework, we now consider the full implementation of self-referential class and method vectors. This subsection isn't specific to **FJ**, but applies generally to any e_{dm} with self-reference via cv and mv parameters in dispatch entries. We have delayed the full details until now so that they can be considered in a more concrete context than in Section 3.

We focus on the method-based (sum) organization because it is the road (much) less traveled, and leads to some novel views of some aspects, but everything that follows also works out dually for the class-based (product) organization.

The appropriate sum-based recursive object type $\tau^\Sigma_{\mathsf{obj}}$ is as in Section 3 and the corresponding self-referential class and method vectors are as follows. Following Harper (2012) **L** uses $\mathtt{fold}(e)$ and $\mathtt{unfold}(e)$ for recursive types, and $\mathtt{self}\,x\,\mathtt{is}\,e$ and $\mathtt{unroll}(e)$ for recursive expressions.

$$\begin{aligned}
\tau^{\Sigma}_{\mathsf{obj}} &\triangleq \mu\, t_{\mathsf{obj}}.\sum_{c\in C} \tau^{c}(t_{\mathsf{obj}}) \\
e^{\Sigma}_{\mathsf{cv}} &\triangleq \langle c \hookrightarrow \lambda\,\texttt{(}u\texttt{:}\tau^{c}(\tau_{\mathsf{obj}})\texttt{)}\,\texttt{fold(}c\cdot u\texttt{)}\rangle_{c\in C} : \tau_{\mathsf{cv}}(\tau_{\mathsf{obj}}). \\
e^{\Sigma}_{\mathsf{mv}} &\triangleq \texttt{unroll(}e^{\Sigma\,\prime}_{\mathsf{mv}}\texttt{)} : \tau_{\mathsf{mv}}(\tau_{\mathsf{obj}}) \\
e^{\Sigma\,\prime}_{\mathsf{mv}} &\triangleq \texttt{self}\; mv\, \texttt{is}\, \langle m \hookrightarrow \lambda\,\texttt{(}\mathit{this}\texttt{:}\tau_{\mathsf{obj}}\texttt{)}\;\texttt{case}\,\texttt{unfold(}\mathit{this}\texttt{)}\;\texttt{\{}c\cdot u \Rightarrow e^{c\,\prime}_{m}\,\texttt{\}}_{c\in C}\rangle_{m\in M} \\
&\qquad \text{where} \quad e^{c\,\prime}_{m} \triangleq e_{\mathsf{dm}}\cdot c\cdot m\,\texttt{[}\tau_{\mathsf{obj}}\texttt{]}\,\texttt{(}e_{\mathsf{cv}}\texttt{)}\,\texttt{(}e_{\mathsf{mv}}\texttt{)}\,\texttt{(}u\texttt{)}
\end{aligned}$$

Lemma 3. *(Dynamic dispatch) If $v : \tau^{c}$ and v val then*

$$e^{\Sigma}_{mv}\cdot m\,\texttt{(}e^{\Sigma}_{cv}\cdot c\,\texttt{(}v\texttt{))} \mapsto^{*} e_{dm}\cdot c\cdot m\,\texttt{[}\tau_{obj}\texttt{]}\,\texttt{(}e^{\Sigma}_{cv}\texttt{)}\,\texttt{(}e^{\Sigma}_{mv}\texttt{)}\,\texttt{(}v\texttt{)}$$

This lemma exactly characterizes correctness of e_{cv} and e_{mv} as an implementation of dynamic dispatch, and there is a dual proof for the product-based organization. What follows generally doesn't depend the implementation, just on this lemma, except where noted. Hence we generally omit the Σ superscripts in what follows.

4.4 Top-level and compilation correctness

We compile the top-level "external" term T in a program (CT, T) as $\Psi_{\mathsf{ex}}(|T|^{\Gamma_{\mathsf{ex}}})$, via Ψ_{ex} and Γ_{ex} (below) which appropriately omit *this*, and have both $\Psi_{\mathsf{ex}}(cv)$ and $\Psi_{\mathsf{ex}}(mv)$ closed and terminating.

$$\begin{aligned}
\Gamma_{\mathsf{ex}} &\triangleq t_{\mathsf{obj}}, cv : \tau_{\mathsf{cv}}(t_{\mathsf{obj}}), mv : \tau_{\mathsf{mv}}(t_{\mathsf{obj}}) \\
\Psi_{\mathsf{ex}} &\triangleq t_{\mathsf{obj}} \mapsto \tau^{\Sigma}_{\mathsf{obj}}, cv \mapsto e^{\Sigma}_{\mathsf{cv}}, mv \mapsto e^{\Sigma}_{\mathsf{mv}}
\end{aligned}$$

Then using the earlier lemmas we can show that the compilation is operationally sound. This has two parts: one for ordinary **FJ** evaluation steps and one for invalid downcasts. The theorem statement and proof involve **FJ** evaluation contexts $E\{\}$, defined as follows, as in Pierce (2002).

$$\begin{aligned}
E\{\} ::= \{\}\ |E\{\}.f \mid \texttt{new}\, c\texttt{(}\overline{V}, E\{\}, \overline{T}\texttt{)} \\
|E\{\}\cdot \boldsymbol{m}\texttt{(}\overline{T}\texttt{)} \mid V\cdot \boldsymbol{m}\texttt{(}\overline{V}, E\{\}, \overline{T}\texttt{)} \mid \texttt{(}c\texttt{)}\, E\{\}
\end{aligned}$$

Theorem 4. *(compilation correctness) Suppose for a particular **FJ** class table CT we have $\vdash_{\boldsymbol{FJ}} T : c$. Then*

1. *if $T \mapsto_{\boldsymbol{FJ}} T'$ then $\Psi_{ex}(|T|^{\Gamma_{ex}}) \mapsto^{*} \Psi_{ex}(|T'|^{\Gamma_{ex}})$*

2. *if T has the form $E\{(c)\,\texttt{new}\,c'\,(\overline{V})\}$ and not $c' <: c$ then $\Psi_{\mathsf{ex}}(|T|^{\Gamma_{\mathsf{ex}}}) \mapsto^* \texttt{error}$.*

Proof. (Sketch.) By induction on (closed) T for part 1, using Lemma 3 (dynamic dispatch) to emulate **FJ** calls to $\boldsymbol{m}$ on instances of c with instance data V via $e_m^c\,\texttt{[}\tau_{\mathsf{obj}}\texttt{]}\,\texttt{(}e_{\mathsf{cv}}\texttt{)}\,\texttt{(}e_{\mathsf{mv}}\texttt{)}\,\texttt{(}v\texttt{)}$ where $|V|^{\Gamma}_{\mathsf{val}} \mapsto^* v$. We similarly use Lemma 2 (value translation) to produce corresponding **L** values when the **FJ** evaluation rule requires certain subterms of T to be values.

4.5 Interpreting FJ types as refinements

Figure 3 shows the details of our interpretation of **FJ** types, instantiating the setup in Section 3. Firstly we define M^c and dual C_m which we take as our specification of what class-method combinations are required to dispatch to valid implementations. This is in fact derived from the class table of the **FJ** program here, since **FJ** lacks a mechanism to separately specify such requirements, and we wish to provide the same guarantees as the the **FJ** type system in regards to "method not understood" failures. We still consider this specification as conceptually prior to the actual code implementing classes and methods, and in general it could be derived from a separate specification.

Unlike for types, the refinements indicated for the results of dispatch entries e_m^c for a single method m can differ between classes due to unrequired class-method combinations. So, we choose $\rho_m(\vec{r})$ as the appropriate refinement for required combinations, and then (below in Figure 3) we choose $\rho_m^c(\vec{r})$ as $\top$ (which includes all commands, hence `fail`) when the combination c, m isn't specified as required.

Next Figure 3 introduces type variables r^c and r_m which conceptually indicate "instances of c" and "recognizers of m". However, taking a behavioral view, we actually characterize r^c in terms of behavior, namely the methods that instances of r^c recognize. Thus, r^c includes all objects that recognize all methods that instances of c do.

The refinement r^m directly indicates that method m is recognized. θ_0 is a set of entailments that are safe based directly on what class-method combinations are required. However, this directness may exclude some combinations that behaviorally should be included, based on the above characterization of r^c. Thus θ is constructed so that it is a superset of the entailments in θ_0, closed with respect to the behavioral view. As we shall see, θ is sufficient to justify subsumption between class types in **FJ** (which is built into some of the **FJ** typing rules), also called subclassing, while θ_0 is not.

The last part of the union in the definition of θ represents a dual concern: that given a specified set of required class-method combinations, knowledge that a particular object recognizes a particular method may be sufficient to deduce that the object also recognizes some other methods. We call this dual concept supermethoding, and include it here to emphasize that it is the natural dual of subclassing. Further, we note that what appears to be essentially the same concept has been studied significantly in the mature field of *formal concept analysis*, for an overview see the text by Ganter et al. (1997).

We now verify that these definitions satisfy the conditions in Section 3.2

Lemma 5. *For all classes c, c', if $CT \vdash_{\mathsf{FJ}} c' <: c$ then $(r^{c'} \leq_{t_{obj}} r^{c})$ is in θ.*

Proof. (sketch) Roughly by construction: in FJ, subclassing c to form c' leads to each method of c either being inherited or overridden in c' (with the same type), and so on transitively, hence c' has all methods that c does.

Lemma 6. *For all $c \in C$ and $m \in M$ we have $e^c_m \in_{\tau^c_m} \rho^c_m$*

Proof. (sketch) We show that for all τ_{obj}, $\vec{\rho} \sqsubseteq \tau_{\mathsf{obj}}$ with $\vec{\rho}$ *sat.* θ, and all $e_{\mathsf{cv}} \in \rho_{\mathsf{cv}}(\vec{\rho})$, $e_{\mathsf{mv}} \in \rho_{\mathsf{mv}}(\vec{\rho})$, $v_u \in \rho^c(\vec{\rho})$
that $[\tau_{\mathsf{obj}}/t_{\mathsf{obj}}][\vec{\rho}/\vec{r}][e_{\mathsf{cv}}/cv][e_{\mathsf{mv}}/mv][v_u/u]k^{c,\Gamma}_m \in {\rho^c_m}'(\vec{r})$ in each case.

The case for a defined $\boldsymbol{m}$ involves the translation $|T|^{\Gamma}$ of the method body $\vdash_{\mathsf{FJ}} T : \mathsf{res}_{\boldsymbol{m}}$ which we treat by induction on the FJ typing derivation, generalizing appropriately.

Lemma 7. *$\vec{\rho}_{\Sigma}$ satisfies θ.*

Proof. From the definitions of $\vec{\rho}$, $\rho^c_{\mathsf{obj}}(\vec{r})$ and $\rho^m_{\mathsf{obj}}(\vec{r})$ we have each required $\rho^c \leq \rho_m$ and $\rho^c \leq \rho^{c'}$ simply by inclusions between the sets M^c and C_m. (No entailments for $\rho^c(\vec{\rho})$ are involved.)

Lemma 8. *$e^{\Sigma}_{\mathsf{cv}} \in \rho_{\mathsf{cv}}(\vec{\rho}_{\Sigma})$ and $e^{\Sigma}_{\mathsf{mv}} \in \rho_{\mathsf{mv}}(\vec{\rho}_{\Sigma})$.*

Proof. (sketch) By the properties of refinements in Section 2.

Lemma 9.
If ($m = \boldsymbol{m}$ and $\mathtt{mbody}(c, m)$ defined), or ($m = \mathtt{get}[f]$ and $f \in \mathtt{fields}(c)$), or $m = \mathtt{cast}[c]$ then $\rho^c_m \leq \forall\,(t_{obj} \sqsupseteq \vec{r} : \theta)\,.\, \rho_{\mathsf{cv}}(\vec{r}) \rightharpoonup \rho_{\mathsf{mv}}(\vec{r}) \rightharpoonup \rho^c(\vec{r}) \rightharpoonup \rho_m(\vec{r})$

Proof. For these cases the definitions of the two refinements coincide.

Theorem 10. *If $\vdash_{\mathsf{FJ}} T{:}c$ then $\Psi_{ex}(|T|^{\Gamma_{ex}}) \in (\mathtt{ret\,inst}\,[c]) \vee \mathtt{error}$.*

Proof. (sketch) By induction on the typing derivation for T, and using lemma 6 with the subsequent lemmas discharging the assumptions of that lemma.

(Alternatively, the result follows from the type preservation and progress theorems of FJ sketched by Pierce (2002) and our earlier lemma that FJ reduction can be simulated via the interpretation in L, but this is perhaps less convincing as a demonstration of reasoning using semantic refinements.)

Specification of required class-method combinations

$$M^c \triangleq \{\boldsymbol{m}{\in}M \mid \mathtt{mtype}(\boldsymbol{m}, c) \text{ is defined}\} \cup \{\mathtt{get}[f] \mid f \in F^c\} \cup \{\mathtt{cast}[c] \mid c \in C\}$$

$$C_m \triangleq \{c \in C \mid m \in M^c\}$$

Refinement Variables and Entailment Constraints

$$\vec{r} \triangleq \{r^c\}_{c \in C} \cup \{r_m\}_{m \in M} \quad \text{(with the } r^c \text{ and } r_m \text{ all distinct)}$$

$$\theta_0 \triangleq \{r^c \leq_{t_{\mathsf{obj}}} r_m \mid c \in C, m \in M^c\}$$

$$\theta \triangleq \theta_0 \cup \{\, r^c \leq_{t_{\mathsf{obj}}} r^{c'} \mid \text{for all } r_m.\ r^c \leq_{t_{\mathsf{obj}}} r_m \text{ in } \theta_0 \text{ if } r^{c'} \leq_{t_{\mathsf{obj}}} r_m \text{ in } \theta_0\}$$
$$\cup \{r_m \leq_{t_{\mathsf{obj}}} r_{m'} \mid \text{for all } r^c.\ r^c \leq_{t_{\mathsf{obj}}} r_{m'} \text{ in } \theta_0 \text{ if } r^c \leq_{t_{\mathsf{obj}}} r_m \text{ in } \theta_0\}$$

Class Refinements	Method Refinements
$\rho^c(\vec{r}) \triangleq \prod_{(c' f) \in F^c} r^{c'}$	$\rho_{\boldsymbol{m}}(\vec{r}) \triangleq \mathtt{ret}\,((\prod_{c\ i \in \overline{\mathsf{arg}}_{\boldsymbol{m}}} r^c) \rightharpoonup \begin{pmatrix} \mathtt{ret}\, r^{\mathsf{res}_{\boldsymbol{m}}} \\ \vee\, \mathtt{error} \end{pmatrix})$ $\rho_{\mathsf{get}[f]}(\vec{r}) \triangleq \mathtt{ret}\, r^c$ for each $c\, f \in F$ $\rho_{\mathsf{cast}[c]}(\vec{r}) \triangleq \mathtt{ret}\, r^c \vee \mathtt{error}$ for each $c \in C$
$\rho_{\mathsf{cv}}(\vec{r}) \triangleq \prod_{c \in C} (\rho^c(\vec{r}) \rightharpoonup r^c)$	$\rho_{\mathsf{mv}}(\vec{r}) \triangleq \prod_{m \in M} (r_m \rightharpoonup \rho_m(\vec{r}))$

Dispatch Entry Refinements

$$\rho^c_m \triangleq \forall\, (t_{\mathsf{obj}} \sqsupseteq \vec{r} : \theta)\ .\ \rho_{\mathsf{cv}}(\vec{r}) \rightharpoonup \rho_{\mathsf{mv}}(\vec{r}) \rightharpoonup \rho^c(\vec{r}) \rightharpoonup {\rho^c_m}'(\vec{r})$$

$$\text{where } {\rho^c_m}'(\vec{r}) \triangleq \begin{cases} \rho_m(\vec{r}) & \text{if } m \in M^c \\ \top & \text{otherwise} \end{cases}$$

Refinements of $\tau^{\Sigma}_{\mathsf{obj}}$ (sum-based)

$$\boxed{\rho^c_{\mathsf{obj}}(\vec{r}) \triangleq \bigwedge_{m \in M^c} \rho^m_{\mathsf{obj}}(\vec{r})} \qquad \boxed{\rho^m_{\mathsf{obj}}(\vec{r}) \triangleq \bigvee_{c \in C_m} (c \cdot \rho^c(\vec{r}))}$$

$$\vec{\rho}_{\Sigma} \triangleq \mu\, \vec{r}.(\rho^c_{\mathsf{obj}}(\vec{r}))_{c \in C}, (\rho^m_{\mathsf{obj}}(\vec{r}))_{m \in M}\ \mathtt{in}\ \vec{r}$$

$$\mathtt{inst}\,[c] \triangleq \rho^c_{\mathsf{obj}}(\vec{\rho}_{\Sigma}) \qquad \mathtt{recog}\,[m] \triangleq \rho^m_{\mathsf{obj}}(\vec{\rho}_{\Sigma})$$

Figure 3: Interpreting **FJ** types as **L** refinements

Corollary 11. *If* $\vdash_{\textsf{FJ}} T{:}c$ *then* $\Psi_{ex}(|T|^{\Gamma_{ex}})$ *will not evaluate to* `fail` *indicating "message not understood".*

Thus, our interpretation of the type system of **FJ** as semantic refinements will correctly accept the translations of all well-typed **FJ** programs. As well it will of course accept any program that doesn't result in a "message not understood" or "field not understood" error even if the **FJ** type system rejects it. It can also rule out downcast failures in essentially the same way, or better, characterize exactly what conditions will lead to downcast failures.

Of course, a disadvantage is that the semantic approach generally does not as directly lead to practical tools such as refinement checkers. But, a refinement checker like that studied (and built) by Davies (2005) can be considered a proof checker for certain quite restricted language of proofs of semantic properties that can be conveniently expressed via a few annotations within or alongside a program. Making semantic refinements the primary notion not only leads to some technical simplifications, it clarifies the nature of syntactic refinements and the exact limitations that should be expected when using a refinement checker.

5 Conclusion

By separating structural from behavioral considerations we have repositioned the problem of typing for object-oriented programming from one of designing languages (structural type theories) to one of designing specifications (behavioral type theories). Rather than privileging the "message not understood" error, we instead treat it on a par with other conditions, such as "down-cast errors", that naturally arise when using dynamic dispatch, and which are much more difficult to account for in a purely structural framework. More broadly, avoiding the characteristic errors associated with dynamic dispatch becomes a particular instance of avoiding a broader class of errors, such as array bounds check errors. The emphasis on the semantic interpretation of behavioral typing may be further generalized to account for richer properties, such as the equational properties inherent in the Liskov-Wing behavioral notion of behavioral subtyping (Liskov and Wing 1994), by passing from predicates to binary relations defined over a structural type system.

In our main example we have derived the key safety property provided by the **FJ** type system through a combination of structural and behavioral typing. Being semantically defined, behavioral typing is, in general, not mechanically checkable; whether a program exhibits (or fails to exhibit) a particular behavior is a matter of proof. In this respect our formulation is coherent with the general trend toward the integration of program verification as part of standard software development practices. For this to be practical, it is necessary to develop tools that can, in common cases, perform automatic verification, or semi-automatic verification via modest "proof hints" such as annotations specifying expected invariants. For example, it appears that the existing tool SML CIDRE developed by Davies (2005) is sufficiently expressive and efficient to handle SML code corresponding to our main example, including refinements of abstract types via refinements in SML modules. More broadly, it would be interesting to integrate structural and behavioral typing in a single dependent type theory in which one may regard type refinements as propositional functions, and then apply automated reasoning systems, such as Coq (Bertot and Castéran 2004), to perform the verification. It would appear that in such a framework the **FJ** type checker would emerge as a tactic that handles the verification of the absence of "not understood" errors. This should naturally extend to full Java type checking, and other languages involving dispatch, including more involved aspects such as the variance of generics which we expect to fit well with behavioral refinements. Further, we expect this to suggest some natural extensions, for example enriching subtyping of generics with *strictness* of type parameters, or the more general *constrained inclusions* considered by Davies (2005), with the formulation of even more precise tactics and refinement checking tools being naturally open-ended.

The semantic foundations for behavioral typing suggest other interesting directions for research. As mentioned earlier, by passing to a relational interpretation of refinements we may express properties, such as parametricity properties, that hold of a particular language, or to verify properties such as the behavioral subtyping condition mentioned earlier, that hold of particular programs. Another direction is to observe that the structural treatment of dynamic dispatch naturally gives rise to a semantic account of object-oriented concepts such as subclassing. Briefly, rather than consider subclass relationships to be a matter of declaration or construction, as they are in **FJ**, we may instead define such relationships behaviorially in terms of the dispatch matrix. For example, one may consider c to be a subclass of c' whenever every method that is well-defined on instances of c' is also well-defined on instances of c, a semantic formulation of what is stated by declaration in **FJ**. It would also be interesting to extend our methods to concepts such as multiple dispatch (pattern matching on tuples of objects), or more exotic programming concepts such as predicate dispatch. These seem ripe for consideration from a behavioral/verification viewpoint, without requiring substantial changes to the underlying structural type theory.

Acknowledgement The authors are grateful to Martín Abadi, Stephanie Balzer and Matthias Felleisen for their very helpful comments on this work.

References

M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

J. Aldrich. The power of interoperability: Why objects are inevitable. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 101–116, 2013.

N. Benton and P. Buchlovsky. Semantics of an effect analysis for exceptions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 15–26. ACM, 2007.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, Nov. 1999.

L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Summary in *Semantics of Data Types*, Kahn, MacQueen, and Plotkin, eds., Springer-Verlag LNCS 173, 1984.

L. Cardelli. Bad engineering properties of object-oriented languages. *ACM Computing Surveys (CSUR)*, 28(4es):150, 1996.

L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, Dec. 1985.

L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula-3 type system. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 202–212, Jan. 1989.

W. R. Cook. A proposal for making eiffel type-safe. *The Computer Journal*, 32 (4):305–311, 1989.

W. R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 557–572, 2009.

K. Crary and R. Harper. Syntactic logical relations for polymorphic and recursive types. *Electronic Notes in Theoretical Computer Science*, 172:259–299, 2007.

R. Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University School of Computer Science, May 2005. Available as Technical Report CMU–CS–05–110.

R. Davies and F. Pfenning. Intersection types and computational effects. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 198–208, 2000.

E. Denney. Refinement types for specification. In *Programming Concepts and Methods PROCOMET'98*, pages 148–166. Springer, 1998.

J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, August 2007.

J. Dunfield and F. Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Foundations of Software Science and Computation Structures*, FOSSACS'03, pages 250–266, 2003.

K. Fisher and J. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1996.

T. Freeman and F. Pfenning. Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario*, June 1991.

B. Ganter, R. Wille, and C. Franzke. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag New York, Inc., 1997.

R. Harper. *Practical foundations for programming languages*. Cambridge University Press, 2012.

R. Harper. Practical foundations for programming languages (second edition). Available at `http://www.cs.cmu.edu/~rwh/plbook/2nded.`, 2014.

A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Oct. 1999.

S. L. P. Jones. Haskell 98: Introduction. *J. Funct. Program.*, 13(1):0–6, 2003.

B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML,* Revised edition. MIT Press, 1997.

U. Norell. Dependently typed programming in Agda. In *In Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.

M. Odersky and T. Rompf. Unifying functional and object-oriented programming with scala. *Commun. ACM*, 57(4):76–86, 2014.

F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

A. M. Pitts. Relational properties of domains. *Information and Computation*, 127(2):66–90, 1996.

J. Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development.* Springer-Verlag, 1985. Lecture Notes in Computer Science No. 185.

D. Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov. 1994.

H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, pages 249–257, 1998.

A Theory of Model Equivalence

Ozan Kahramanoğulları
The Microsoft Res. – Uni. of Trento
COSBI, Rovereto, Italy

James F. Lynch
Department of Computer Science,
Clarkson University, Potsdam, NY, USA

Abstract

We propose a theory for quantitative comparison of models in terms of flux networks obtained from stochastic simulations. The technique is applicable to a range of models from chemical reaction networks to rule-based models. The fluxes of the networks are given by the flow of species instances in stochastic simulations (Kahramanoğulları and Lynch (2013)). This makes it possible to define a quantitative notion of equivalence, which includes graph isomorphism of flux networks as a special case. We use the technique for comparing models with respect to their simulations at arbitrary time intervals with varying degrees of accuracy, and for simplifying models when a larger model produces the same behavior as the smaller one. Other more involved queries that we aim to address include queries on emulation of a complex model by a simpler one.

Introduction

In systems biology, models are commonly refined and extended, and often compared for their capability to produce a behavior of interest. Despite the limited number of formal means, drawing parallels between various models of biological systems is central to many investigations in this field. Furthermore, existing efforts are often limited by the measurement of ad hoc model signals, as it is inherently challenging to provide a general method for the task.

To this end, we propose a methodology for comparing models in relation to the dynamic behavior that is produced by the model components. For this purpose, we use the fluxes generated by a model as a summary of the dynamic behavior, where flux is given by the flow of resources during stochastic simulations (Kahramanoğulları and Lynch (2013)). The graphs that we obtain by computing the fluxes display how many of the model species instances flow between which model components during which intervals of the simulations. As this provides a mathematical structure that quantifies the model dynamics, we use this information as a summary of the model that can be compared to other structures obtained in the same way. Moreover, the quantitative observations made during the comparisons are useful in contrasting the stronger components of the models that account for most of the dynamical behavior with the weaker components.

In the following, we illustrate our approach on examples. We provide an introduction as discussed in Kahramanoğulları and Lynch (2013) to the computations of fluxes in stochastic simulations with models that are typically defined as chemical reaction networks as well as their more compact representations in the form of rule-based descriptions. We describe the notions of our method, first on a simple example, and then on published models from the literature. We illustrate the notion of flux equivalence (Kahramanoğulları and Lynch (2011)) on a model of GTP-binding proteins (Goryachev and Pokhilko (2006); Cardelli et al. (2009)). We then extend our discussion to the cases where models with varying sizes and structures are compared. For this purpose, we resort to the cell cycle and approximate majority models, which were previously compared in Cardelli and Csikász-Nagy (2012) by using stochastic and deterministic simulations together with probabilistic model checking. While providing analyses that are consistent with these studies, our method gives rise to promising observations that have a potential as a formal method for model comparison.

Models and Flux

The method for stochastic flux analysis presented in Kahramanoğulları and Lynch (2013) can be applied to any discrete or continuous time discrete event simulation that implements reaction networks as Markov chains. Rule based modeling languages with a stochastic simulation engine as well as implementations of stochastic Petri nets fall into this category. The method can thus be applied to all such languages; here, for simplicity, we use the chemical reaction network representation of the models.

The flux analysis method is based on marking individuals that are transformed by the reactions during the simulations, and using the markings to track the causal dependencies between reaction instances as in event structures (Kahramanoğulları (2009); Kahramanoğulları (2006)). We process this causality information to obtain a quantification of the flow of resources between reactions, and thereby quantify the network fluxes at chosen time intervals. This is easily implemented by assigning a unique identifier to each network species instance in the initial state and to each reaction product of every reaction instance. In our case, these identifiers are integers.

The formal definitions of our method for obtaining the flux graphs are described in detail in Kahramanoğulları and Lynch (2013). Below, we thus give a

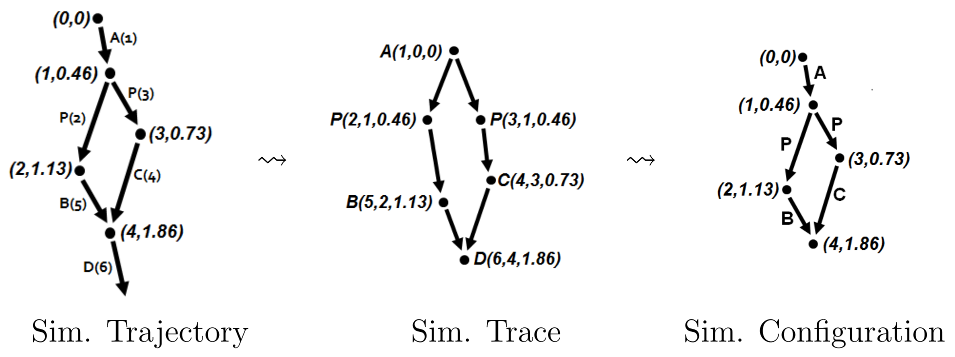


Figure 1: The transformation from a simulation trajectory generated by the network in the introductory example to its simulation trace, and the transformation from the simulation trace to the simulation configuration.

textual description that summarizes the method and the structures it uses.

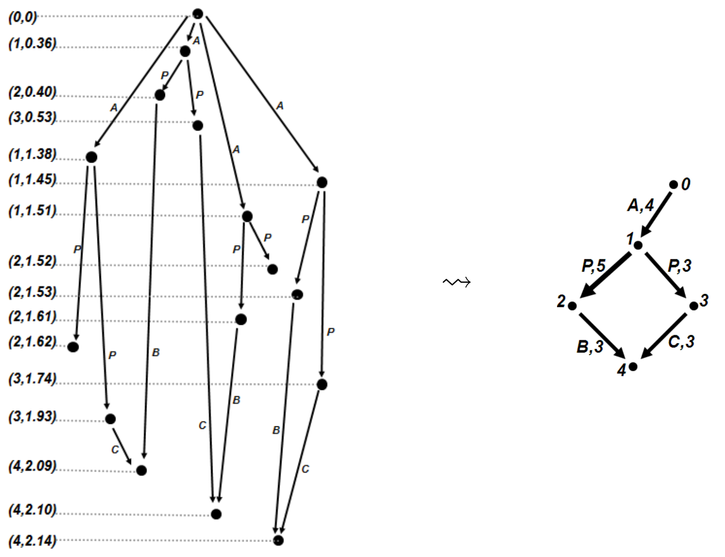
A reaction instance is a random event whose probability is determined by the current state of the network. A reaction can be applied at a state to obtain a reaction instance if its reactants are available at that state and the reaction is picked by the simulation algorithm from all the applicable reactions. Whenever a reaction is applied at a state the simulation algorithm updates the resulting state with the reaction products and their unique identifiers in a structure that we call simulation trajectory. Because this information can be recorded in a bounded amount of time during simulation in real time, the method does not introduce any additional complexity to the simulation algorithm.

Example. Consider the chemical reaction network below, where each reaction is named with an integer.

$$1 : A \to P + P, \qquad 2 : P \to B, \qquad 3 : P \to C, \qquad 4 : B + C \to D$$

The initial state is $\{A(1)\}$, where 1 is the unique identifier of the species instance A. A possible 4-step simulation trajectory is depicted on the left-hand-side of Figure 1, where each node of the graph is a reaction instance: the first parameter of the label of the node is the reaction name, and the second parameter is the reaction instance time. The edges are the species instances that are produced by the source node and consumed by the target node. Each edge has a unique integer identifier.

By using the unique identifiers of the species instances in a reaction trajectory, which indicate the production-consumption relationship between reaction instances of the simulation, we construct a directed graph structure. This graph structure, called the simulation trace, makes the causality relationship explicit.



Sim. configuration

Flux configuration

Figure 2: The simulation configuration of a simulation with the network in the introductory example and the initial state $\{A(1), A(2), A(3), A(4)\}$. In the simulation configuration, each edge is additionally decorated with its species for illustration purposes. We first obtain the simulation configuration and then the flux configuration.

Example. Consider the simulation trace in the middle in Figure 1, which is obtained from the simulation trajectory of the example above. The nodes denote the species instances: the first parameter of the triple is the unique identifier of that species instance, the second parameter is the identifier of the reaction that created it, and the third parameter is the time it is created. The edges denote the causality relationship between the species instances in the sense that the node at the source of an edge is required for the production of the node at the target.

By further processing this graph, we obtain an edge-labeled directed multigraph that reveals the independence and causality information of the transitions with respect to the flow of specific resources between reactions. The information displayed by this graph is different from that given by the simulation trace, where the evolution of the species instances with respect to the reactions is shown. In this graph, which we call simulation configuration, each node is a pair that contains the reaction that is applied and its time in the simulation.

Example. Consider the simulation configuration on the right-hand-side of Figure 1. Each edge is labeled with the species that is produced by the reaction at the source node of that edge and consumed by the reaction at the target node.

In order to quantify the flow of resources between the reactions within given time intervals of the simulation, we compress simulation configurations into structures that we call flux configurations. A flux configuration is a graph, where the nodes are the reactions of the network. We obtain a flux configuration first by merging the edges of the simulation configuration such that all the edges with a certain species within the given time interval are mapped to a single edge by filtering out their time stamps. For each label that denotes a network species instance, we then count in the simulation configuration the number of edges from each node (which corresponds to a reaction of the network) to other nodes within the given time interval. The number of such edges is then used to decorate the edge for that species between the respective reactions.

The flux configurations give a summary of the specific resource flows between specific reactions at arbitrary time intervals during the simulation, and thereby they provide a narrative for the essence of the dynamic behavior.

Example. Consider the chemical reaction network given in the example above. A simulation trace for the initial state $\{A(1), A(2), A(3), A(4)\}$ is depicted on the left-hand-side of Figure 2. The figure demonstrates the simulation configuration and the flux configuration obtained from this trace. In the flux configuration the nodes are reactions, and the edges are the pairs of species names and their counts.

The time and space complexity of generating the above data structures is linear in the number of simulation steps, which follows from the facts that there is a fixed number of reactions, and each reaction involves a fixed number of species. It is also evident that the flux graphs can be generated in linear time and space. Because the steps of this algorithm do not modify the generation of the individual events, the algorithm can be included in any discrete events simulator of chemical reaction networks.

Model Equivalence as Flux Equivalence

In order to demonstrate our concept of equivalence, we use the chemical reaction network depicted in Figure 3. This network models Rho GTP-binding protein activation. For detailed dynamic analysis of this network, we refer to Goryachev and Pokhilko (2006) with respect to ordinary differential equations and we refer to Cardelli et al. (2009); Kahramanoğulları and Lynch (2013) with respect to stochastic simulations.

In this network, all the reactions except 18, 20, and 22 are reversible. Here, we consider the regime with the initial conditions given with $\mathsf{R}_0 = 1000$, $\mathsf{E}_0 = 776$ and $\mathsf{A}_0 = 1$; the analysis on regimes with other initial conditions can be found

$$
\begin{array}{lll}
\mathit{1}: \mathsf{A}+\mathsf{R} \xrightarrow{1.0} \mathsf{RA} & \mathit{9}: \mathsf{RA} \xrightarrow{500} \mathsf{A}+\mathsf{R} & \mathit{17}: \mathsf{RT} \xrightarrow{0.02} \mathsf{R} \\
\mathit{2}: \mathsf{A}+\mathsf{RD} \xrightarrow{1.0} \mathsf{RDA} & \mathit{10}: \mathsf{RD} \xrightarrow{0.02} \mathsf{R} & \mathit{18}: \mathsf{RT} \xrightarrow{0.02} \mathsf{RD} \\
\mathit{3}: \mathsf{A}+\mathsf{RT} \xrightarrow{1.0} \mathsf{RTA} & \mathit{11}: \mathsf{RDA} \xrightarrow{500} \mathsf{A}+\mathsf{RD} & \mathit{19}: \mathsf{RTA} \xrightarrow{3.0} \mathsf{A}+\mathsf{RT} \\
\mathit{4}: \mathsf{E}+\mathsf{R} \xrightarrow{0.43} \mathsf{RE} & \mathit{12}: \mathsf{RDE} \xrightarrow{0.136} \mathsf{E}+\mathsf{RD} & \mathit{20}: \mathsf{RTA} \xrightarrow{2104} \mathsf{RDA} \\
\mathit{5}: \mathsf{E}+\mathsf{RD} \xrightarrow{0.0054} \mathsf{RDE} & \mathit{13}: \mathsf{RDE} \xrightarrow{6.0} \mathsf{RE} & \mathit{21}: \mathsf{RTE} \xrightarrow{76.8} \mathsf{E}+\mathsf{RT} \\
\mathit{6}: \mathsf{E}+\mathsf{RT} \xrightarrow{0.0075} \mathsf{RTE} & \mathit{14}: \mathsf{RE} \xrightarrow{1.074} \mathsf{E}+\mathsf{R} & \mathit{22}: \mathsf{RTE} \xrightarrow{0.02} \mathsf{RDE} \\
\mathit{7}: \mathsf{R} \xrightarrow{0.033\times D} \mathsf{RD} & \mathit{15}: \mathsf{RE} \xrightarrow{0.033\times D} \mathsf{RDE} & \mathit{23}: \mathsf{RTE} \xrightarrow{0.02} \mathsf{RE} \\
\mathit{8}: \mathsf{R} \xrightarrow{0.1\times T} \mathsf{RT} & \mathit{16}: \mathsf{RE} \xrightarrow{0.1\times T} \mathsf{RTE} & D = \mathit{50},\ T = \mathit{500}
\end{array}
$$

Figure 3: The GTPase chemical reaction network and their rates as in Goryachev and Pokhilko (2006); Cardelli et al. (2009); Kahramanoğulları and Lynch (2013).

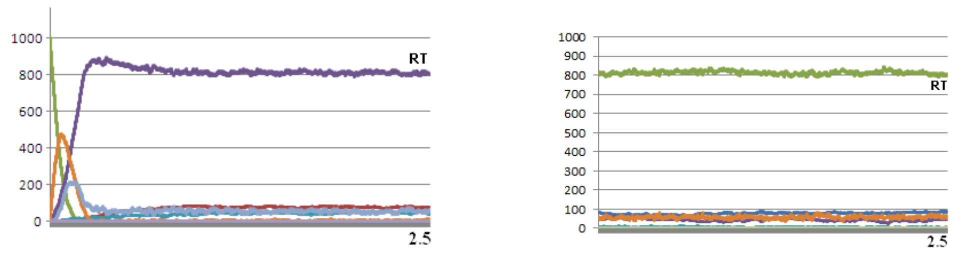


Figure 4: **Left:** Example simulation plot of the network in Figure 3. The initial numbers of the species are $\mathsf{R}_0 = 1000$, $\mathsf{E}_0 = 776$, and $\mathsf{A}_0 = 1$. **Right:** Example simulation plot of the network obtained by reducing the network in Figure 3 with respect to the dominant fluxes. This network consists of the reactions 3, 5, 6, 11, 13, 16, 20, and 21. The initial numbers are set to the steady-state values of the left-hand-side simulation with the complete network.

in Kahramanoğulları and Lynch (2013). When we run stochastic simulations at this regime we obtain time-series plots as on the left-hand-side of Figure 4.

The stochastic flux analysis can be applied on any arbitrary time interval that can be a transient period as well as steady state. However, in accordance with the analysis in Goryachev and Pokhilko (2006), we analyzed the steady state fluxes of this model for the time interval between 2.0 and 2.5 as this interval provides a sufficient number of events in accordance with the convergence time of the simulation. As with time-series analysis, flux analysis in stochastic simulations needs to be repeated on multiple simulations in order to increase the confidence levels. While some systems require a greater number of simulations, others converge quickly to their steady state as it is the case for the network here.

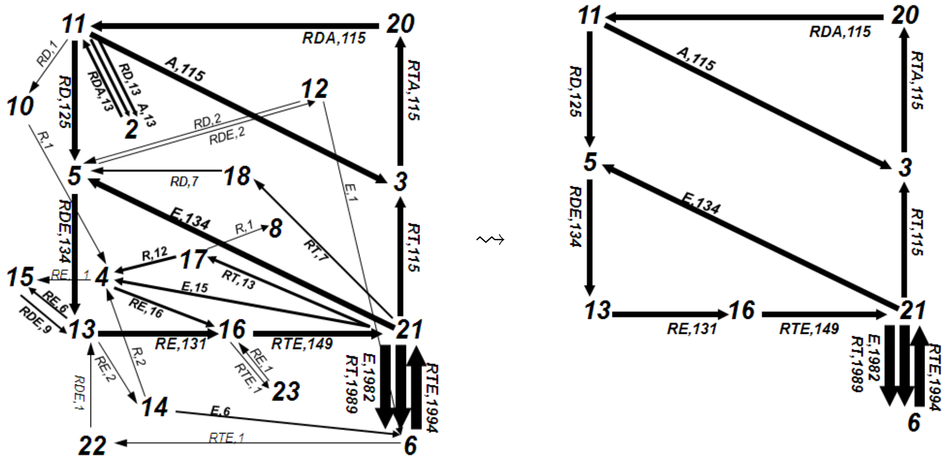


Figure 5: The flux configuration for the time interval from 2 to 2.5 and the structure obtained from it by filtering the fluxes that are weaker than 10% of the average flux, i.e., the flux configuration after cut-off at 0.1.

Nevertheless, due to the observations being made on stochastic simulations, to perform a statistical analysis on a small sample, we have repeated our analysis on a set of 25 simulations to verify our results, where we repeated the observations discussed below. This statistical analysis is discussed in Kahramanoğulları and Lynch (2013).

A representative flux configuration with this network is depicted on the left-hand-side of Figure 5. On the set of 25 simulations, we reduce the flux configurations to dominant fluxes that account for most of the dynamical behavior. For this purpose, we apply a cut-off value that is computed in terms of the average flux of the system at this interval.

Definition 1. *Let $\mathcal{F}[t,t']$ denote a flux configuration for a time interval between t and t', with the edges $\langle j_1, j_1', s_1, n_1\rangle, \ldots, \langle j_\ell, j_\ell', s_\ell, n_\ell\rangle$, where, for each i, we have that j_i, j_i' are nodes, s_i is a species name, and n_i is the count of that species on the edge. The* average flux *is $\sigma = (\sum_{k=1}^{\ell} n_k)/\ell$. For an $x \in \mathbb{R}^+$, the* flux after cut-off at x*, denoted by $\mathcal{F}[t,t'](x)$, is the restriction of $\mathcal{F}[t,t']$ to those edges $\langle j, j', s, n\rangle$ satisfying $n > x\sigma$.*

The flux configuration displays a quantification of the flow of species instances between reactions during a steady state interval of a simulation. As these flows determine the dynamic behavior of the simulation, the stronger fluxes have a greater influence in determining the emerging behavior in comparison to the weaker ones. By applying the definition above to the flux configuration and removing the weaker fluxes from the flux configuration, we obtain a picture of the dominant behavior of the system that is in accordance with the applied cut off value. This is because only a subset of the fluxes of the original network is significant, while the remaining fluxes can have negligible values in delivering the behavior that is, for example, observed at the time-series plot. In this setting, obtaining a convergence with a smaller cut-off value can be considered more reliable in terms of singling out the dominant behavior. The flux configuration obtained by applying the cut-off value of 0.1 on the set of 25 simulations is depicted on the right-hand-side of Figure 5.

The fluxes on the right-hand-side of Figure 5 are those that play a dominant role in tuning the behavior of the network during simulation. This is because these fluxes have a greater weight in comparison to the others, and they thus shift the simulation resources, thereby causing a shift in the time series of the simulation. In order to observe this, we reduce the network in Figure 3 to a network that consists of the reactions that participate in the flux configuration on the right-hand-side of Figure 5. These are the reactions 3, 5, 6, 11, 13, 16, 20, and 21. The simulations with the reduced system do not only agree in terms of their flux configurations, but also their time series behaviors are in agreement as depicted in Figure 4.

As illustrated in the network above, we relate different models according to a comparison of their flux configurations in terms of graph isomorphisms, whereby we impose the condition that the same cut-off value is applied to the compared flux configurations. This way, the cut-off value employed is used as a metric that quantifies the similarity between the compared models.

Definition 2. *Given two flux configurations, $\mathcal{F}[t, t']$ and $\mathcal{F}'[t'', t''']$, we say that they are flux equivalent at x, denoted with $\mathcal{F}[t, t'] \approx_x \mathcal{F}'[t'', t''']$, if and only if $\mathcal{F}[t, t'](x)$ and $\mathcal{F}'[t'', t'''](x)$ are isomorphic graphs.*

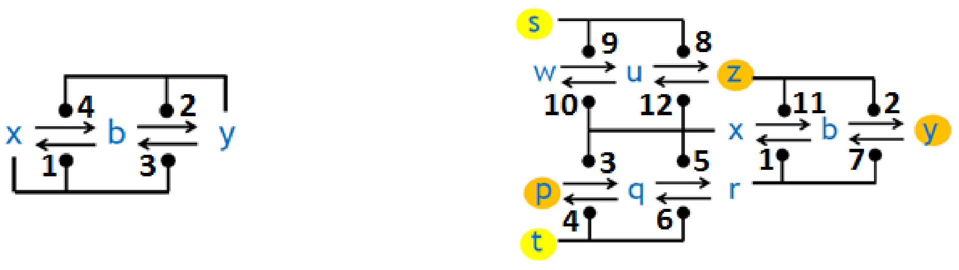
Proposition 3. *Flux equivalence is an equivalence relation.*

We define our metric in terms of cut-off values, which provide quantifications of the similarity of the models with respect to their flux configurations.

Definition 4. *The* distance *between two configurations $\mathcal{F}[t, t']$ and $\mathcal{F}'[t'', t''']$ is the smallest r such that $\mathcal{F}[t, t'] \approx_r \mathcal{F}'[t'', t''']$.*

Equivalence by Filters and Maps

The network that we have used above illustrates how flux analysis can be used to identify dominant reactions of a network. This way, we can identify a subnetwork of the original network that is capable of producing a similar behavior



Approximate Majority	Cell Cycle	
$1 : B + X \to X + X$	$1 : B + R \to X + R$	$9 : W + S \to U + S$
$2 : B + Y \to Y + Y$	$2 : B + Z \to Y + Z$	$7 : Y + R \to B + R$
$3 : X + Y \to X + B$	$3 : P + X \to Q + X$	$8 : U + S \to Z + S$
$4 : X + Y \to B + Y$	$4 : Q + T \to P + T$	$10 : U + X \to W + X$
	$5 : Q + X \to R + X$	$11 : X + Z \to B + Z$
	$6 : R + T \to Q + T$	$12 : Z + X \to U + X$

Figure 6: Graphical representation of the approximate majority network (left) and the cell cycle network (right), and their lists of reactions.

as the original one. We now show that we can compare complex chemical reaction networks with different structures according to their capability to emulate each other. For this purpose, we use the two chemical reaction networks in Figure 6, that is, the approximate majority network (AM) and cell cycle network (CC). These two systems (and intermediate systems) were previously compared in Cardelli and Csikász-Nagy (2012) based on stochastic and deterministic simulations, and probabilistic model checking with the conclusion that they emulate each other. Another approach that uses morphisms, however based on the static structures of the models, is proposed in Cardelli (2014).

Here, we use the ideas above together with maps that collapse larger flux configurations into smaller ones. In these systems X and Y compete against each other for domination. As an example, we compare the networks AM and CC to show the emulation of the more complex CC network by the simpler AM network. Example simulation plots with these networks that illustrate their time series behavior for X, Y, and B are depicted in Figure 7, where X dominates. For the comparison of these networks, we use two different approaches, namely equivalence by filters and equivalence by maps.

Equivalence by Filters. We introduce a mechanism that filters out the flux of the enzymatic species instances in all the reactions in both networks. For

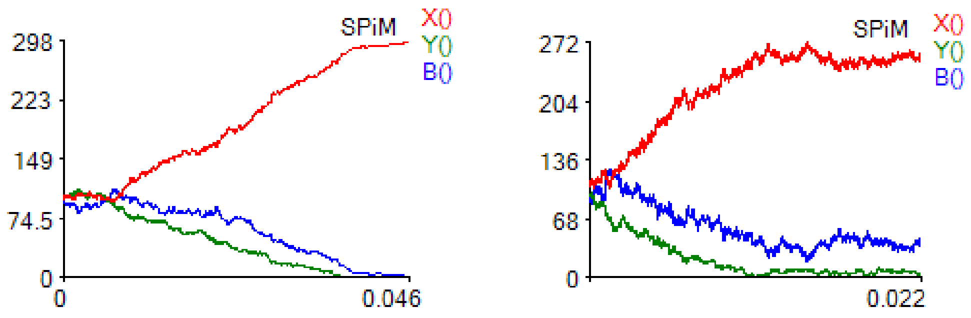


Figure 7: Example simulation plots with the approximate majority network (left) and the cell cycle network (right) for the cases, where X dominates.

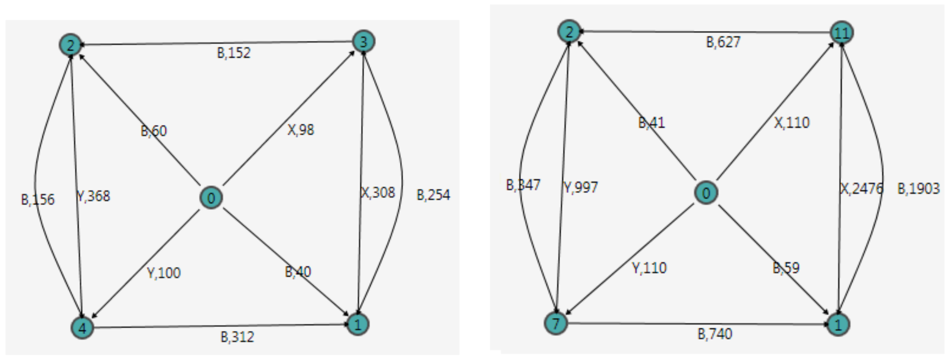


Figure 8: Example flux configuration with the approximate majority network (left) and the cell cycle network (right) for the cases, where X dominates. In these simulations, only non-enzymatic fluxes are observed. The nodes are the reactions of both networks, and 0 denotes the species at the initial state.

example, for the reaction 1 of the AM network, we only consider the reactant B for the analysis, while disregarding the reactant X and keeping track of only one of the X in the product. In other words, in the flux configurations, we only monitor the species instances that get transformed by the reactions. For the species X, Y, and B, these result in the flux-equivalent configurations depicted in Figure 8, where the left-hand-side graph is AM and the right-hand-side is CC.

Equivalence by Maps. We consider all the species in all reactions, and in order to compare AM and CC, we use maps on CC that merge reactions and aggregate network species. This is because the flux configuration graph of CC contains 12 reactions and 11 species in contrast to 4 reactions and 3 species in AM. A flux configuration of CC is depicted in the Appendix in Figure 10. A flux configuration of AM is depicted as the large graph in Figure 9, where the nodes A,

B, C, and D denote the reactions 2, 3, 4, and 1 of the AM network, respectively. The black, red, and blue edges are the fluxes of X, Y, and B, respectively.

In order to relate the two networks, we thus employ a map that merges the fluxes of the reactions and aggregates the species of the CC in Figure 10. As a first step for this, we use the observation that both of the systems employ two non-linear positive feedback loops. In the cell cycle network, X contributes as an enzyme by reactions 10 and 12 to the inhibition of Z, which inhibits the transformation of X to B. Similarly, the reactions 3 and 5 contribute to the production of R, which contributes to the production of X by participating in the reactions 1 and 7. By relying on these observations, we are faced with multiple options for merging the reactions, however not all of these merges can provide a good match between the compared graphs. We thus use the information on the feedback loops together with the structure of the flux configuration, and merge the reactions 1, 3, and 5 into a single reaction; and we merge the reactions 7, 10, and 12 to another. At the second positive feedback loop, the reactions 4, 6, 8, 9, and 11 contribute to the production of Y. By merging these reactions, we obtain the node C in Figure 9. In the graph, the blue edges denote the B fluxes, whereas the black and red edges denote X and Y fluxes together with the fluxes of others that are involved in the nodes.

The resulting merged configuration is identical to the configuration of the AM network in Figure 9, that is, the large graph in Figure 9 is a flux graph of both AM and CC. This is because, as a result of merging the fluxes in Figure 10 as described above,

– the reaction nodes 7, 10 and 12 become node B in Figure 9;
– the reaction nodes 1, 3 and 5 become node D in Figure 9;
– the reaction nodes 4, 6, 8, 9 and 11 become node C in Figure 9.

All the species that are different from X,Y and B in Figure 10 are put in an X or a Y edge in Figure 9. This is because between two reactions, there is never an X and Y flux together, so all the fluxes that are different from X and Y in Figure 10 are considered auxiliary fluxes to X and Y. As a result of this reasoning, we map the components of the cell cycle network that participate in a complex mechanism to a simpler component in the AM network in a way that takes into account the feedback mechanisms that are shared by these two networks.

Discussion

Our notion of equivalence is based on flow of resources between reactions during simulations. The stochastic nature of the approach makes it plausible for the

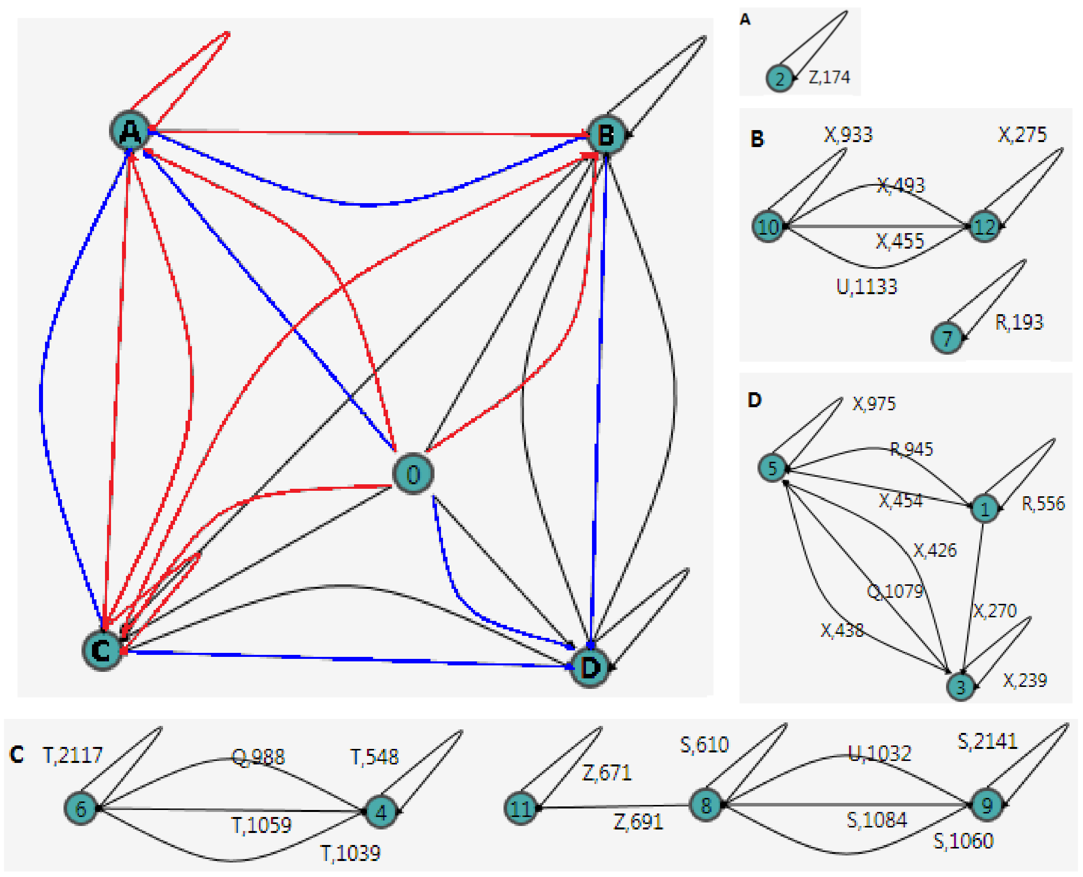


Figure 9: A complete flux configuration of the approximate majority network, which is equivalent to the configuration obtained by merging the reaction nodes and edges of the complete cell cycle flux configuration. A, B, C, and D are the graphs of the merged fluxes in the complete cell cycle network. 0 denotes the species at the initial state. Here, for example, the fluxes in box C are the ones due to the reactions 4, 6, 8, 9, and 11, which contribute to the conversion of X to B in a positive feed-back loop of the cell cycle network. These reactions are merged into a single node C in the main graph. As the node C of the main graph also denotes the reaction 4 of the approximate majority network that converts X to B, these maps imply that the reaction 4 of approximate majority network emulates the reactions 4, 6, 8, 9, and 11 in box C. See the text for further details.

models, where the quantities of certain species are arbitrarily small or the time intervals of interest are not necessarily steady state intervals.

The different graphs that are obtained prior to a flux configuration are the phases that correspond to the intermediate steps in obtaining event structures from transition system trajectories Kahramanoğulları (2009). In this respect, the event structures approach provides a quantitative means to observe the causality within the system dynamics. Moreover, the different kinds of graphs that we use while computing the flux configurations expose different aspects of the same simulation, and they can thus be of independent interest.

We have employed a cut-off function that is based on the average fluxes of the system. However, different notions of cut-off can be more appropriate for different systems, which remains a topic of future investigation. Other questions concern investigations of a statistical nature: as a simulation with a certain initial state has infinitely many different time-series, it has infinitely many simulation trajectories. Future research can provide estimates to reach a desired level of confidence.

References

L. Cardelli. Morphisms of reaction networks that couple structure to function. 2014.

L. Cardelli and A. Csikász-Nagy. The cell cycle switch computes approximate majority. *Scientific Reports, Nature Publishing Group*, 2(656), 2012.

L. Cardelli, E. Caron, P. Gardner, O. Kahramanoğulları, and A. Phillips. A process model of Rho GTP-binding proteins. *Theoretical Computer Science*, 410/33-34:3166–3185, 2009.

A. B. Goryachev and A. V. Pokhilko. Computational model explains high activity and rapid cycling of Rho GTPases within protein complexes. *PLOS Computational Biology*, 2:1511–1521, 2006.

O. Kahramanoğulları. On linear logic planning and concurrency. *Information and Computation*, 207:1229–1258, 2009.

O. Kahramanoğulları and J. Lynch. Stochastic flux equivalence. Technical Report at The Microsoft Research - University of Trento COSBI, PP-0179-2011, 2011.

O. Kahramanoğulları and J. Lynch. Stochastic flux analysis of chemical reaction networks. *BMC Systems Biology*, 7(133), 2013.

O. Kahramanoğulları. *Nondeterminism and Language Design in Deep Inference.* PhD thesis, TU Dresden, 2006.

Appendix A

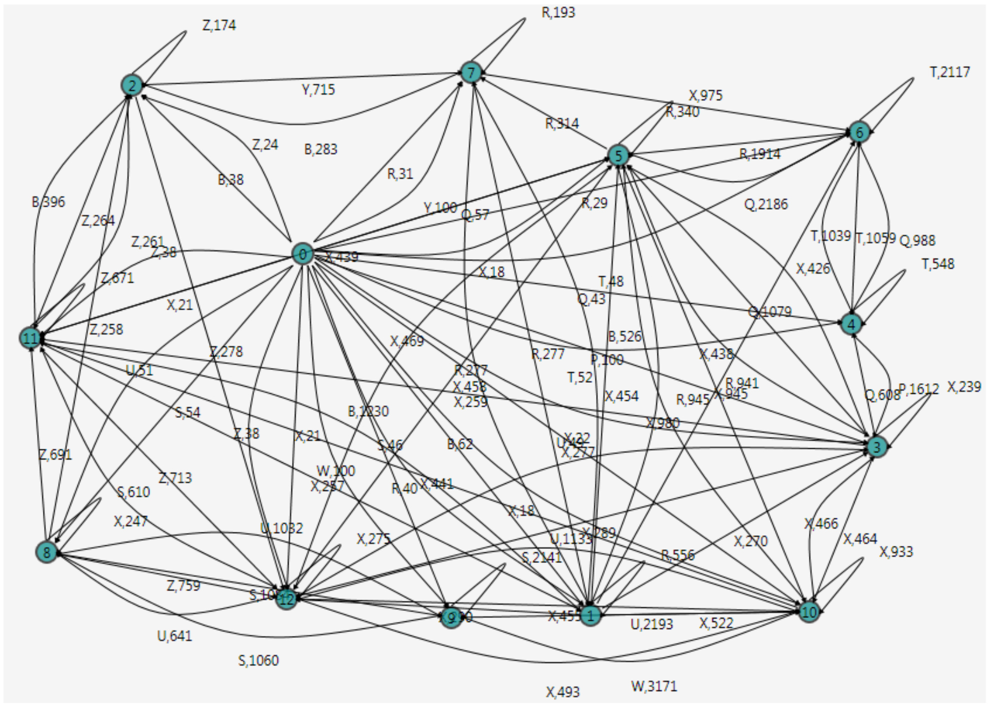


Figure 10: A flux configuration of the cell cycle model.

Challenges in automated verification and synthesis for molecular programming

Marta Kwiatkowska
Department of Computer Science, University of Oxford, UK
marta.kwiatkowska@cs.ox.ac.uk

Abstract

Molecular programming is concerned with building synthetic nanoscale devices from molecules, which can be programmed to autonomously perform a specific task. Several artifacts have been demonstrated experimentally, including DNA circuits that can compute a logic formula and molecular robots that can transport cargo. In view of their natural interface to biological components, many potential applications are envisaged, e.g. point-of-care diagnostics and targeted delivery of drugs. However, the inherent complexity of the resulting biochemical systems makes the manual process of designing such devices error-prone, requiring automated design support methodologies, analogous to design automation tools for digital systems. This paper gives an overview of the role that probabilistic modelling and verification techniques can play in designing, analysing, debugging and synthesising programmable molecular devices, and outlines the challenges in achieving automated verification and synthesis software technologies in this setting.

1 Introduction

Recently, significant advances have been made in the experimental design and engineering of synthetic, biomolecular systems, such as those built from DNA, RNA or enzymes. The interest in such devices stems from the fact that they are *autonomous* – they can interact with the biochemical environment, process information, make decisions and act on them – and *programmable*, that is, they can be systematically configured to perform specific computational or mechanical tasks. The computational power of such systems has been shown to be equivalent to Turing computability (Soloveichik et al. 2010), albeit the computation itself proceeds through molecules acting as inputs, interacting with each other and producing product molecules. Experimental advances are fast accelerating, with examples that have been demonstrated including diagnostic biosensors (Jung and Ellington 2014), logic circuits built from DNA (Seelig et al. 2006; Qian and Winfree 2011), DNA-only controllers (Chen et al. 2013) and molecular robots that can deliver cargo (Yurke et al. 2000; Yin et al. 2004). Since such systems can perform information processing within living cells, their use is envisaged in healthcare applications, where safety is paramount. The fast-growing field of *molecular programming* is concerned with developing techniques to design, analyse and realise the computational potential of such programmable molecular devices. In conjunction with the DNA self-assembly capabilities, which has enabled a wide range of structure-forming technologies at the nanoscale (Rothemund 2006), the future potential of these developments is tremendous, particularly for smart therapeutics, point-of-care diagnostics and synthetic biology.

Device design is supported by electronic design automation (EDA) environments, which provides methodologies and tools to *automate* the design, verification, testing and synthesis of electronic systems from a high-level description. The software level, at which design is applied, is separate from the hardware level, e.g. fabrication, and can involve multiple levels of abstractions. In the semiconductor industry, design automation has established itself as a key technology to tackle the complexity of the designs, improve design quality, and increase reuse. In the 1990s, VLSI design was revolutionised by *formal verification*, and in particular *automated* methods such as model checking, now a key component of modern EDA tools, which ensure device safety and reliability, and significantly reduce development costs.

Molecular programming aims to devise programming languages, techniques and software tools to achieve automatic compilation of a molecular system down to the set of components that can be implemented physically at the molecular level and executed. This is analogous to the motivation for hardware description languages, e.g. VHDL, which are refined automatically, through a series of intermediate abstractions, down to a detailed physical implementation in silicon. This paper puts forward the view that formal verification will play a similar role in design automation for molecular programming. However, the latter brings with it unique challenges: the necessity to consider inherent *stochasticity* of the underlying molecular interactions, the need to state requirements in *quantitative* form, and the importance to consider *control* of molecular systems, and not just programming in the conventional sense. Therefore, *probabilistic modelling* and *automated, quantitative verification* techniques (Kwiatkowska 2007; Kwiatkowska et al. 2007, 2011), such as those already developed for systems biology (Regev et al. 2001; Heath et al. 2008; Ciocchetta and Hillston 2009; Kwiatkowska et al. 2010) in addition to tools tailored to DNA computing (Phillips and Cardelli 2009;

Aubert et al. 2014), will form a key component of design automation for molecular programming.

The paper begins by giving a brief introduction to molecular programming, illustrated by a simple example of DNA biosensing, and then reviews the current status of formal modelling and verification technologies for molecular programming, outlining the research challenges. More detail about application of automated, quantitative verification in DNA computing can be found in the tutorial paper (Kwiatkowska and Thachuk 2014) and elsewhere (Lakin et al. 2012; Dannenberg et al. 2013b,a, 2014).

2 Molecular programming

The term *molecular programming* (Hagiya 2000; Winfree 2008) refers to the application of computational concepts and design methods to the field of nanotechnology, and specifically biochemical reaction systems. The idea is to design biochemical networks that can *process information* and are *programmable*, that is, can be configured to perform a given task, be it computation of a logic formula or transporting a cargo to a specified target. Chemical reaction networks (CRNs) provide a canonical notation for describing biochemical systems, based on well understood stochastic or kinetic laws, and the computational and nanorobotic mechanisms that they can implement. A *molecular program* is thus a series of reactions, for example $X + Y \to Z$, meaning that inputs (specially designed DNA strands) X and Y are transformed to produce strand Z. An example is the Approximate Majority program (Angluin et al. 2008), where, starting with given initial numbers of molecules X and Y placed in solution, with high probability the network will converge to a state that only contains the molecules that were initially in majority.

In order to implement molecular programs, DNA technologies have been developed, of which *DNA strand displacement (DSD)* (Zhang et al. 2007; Zhang and Seelig 2011) is particularly popular, since it uses only DNA molecules, is enzyme-free, and easy to synthesize chemically. Any CRN can be implemented using the limited set of DSD reactions (Soloveichik et al. 2010); in fact, the DSD realisation of Approximate Majority was experimentally demonstrated in Chen et al. (2013) and related to the cell cycle in Cardelli and Csikász-Nagy (2012). DSD can be used to implement logic gates, where inputs and outputs are (single) DNA strands. An example is the transducer gate designed by Cardelli (2010) and the diagnostic biosensors of Jung and Ellington (2014).

The promise of DNA systems is that they can interact with biological components in their local environment, including within living cells. An important application of such systems is therefore *biosensing*, a decision process that aims to

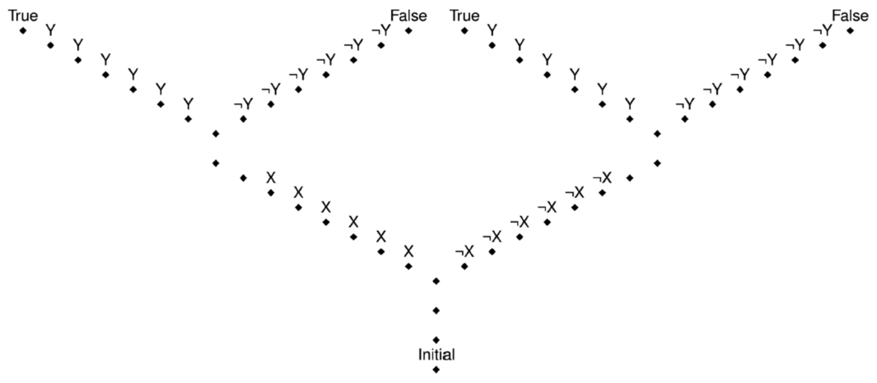


Figure 1: A DNA walker track that acts as a biosensor, shown here with six blockades.

detect various input biomarkers in an environment, such as strands of messenger RNA within a cell, and take action based on the detected input. We illustrate molecular programming applications with an example of a biosensor based on *DNA walker circuits*, which are realised using DNA strand displacement technology. DNA walkers (Wickham et al. 2011) can traverse tracks of DNA strands (*anchorages*) that are tethered to a surface, typically DNA origami tile (Rothemund 2006), taking directions at junctions that fork into two tracks, respectively labelled with X and $\neg X$. When the system is prepared, a self-consistent set of unblocking strands is added to unblock X or $\neg X$ but not both, ensuring that the walker is directed towards the target. Alternatively, the walker *senses* the strands that guide it towards the target, indicating detection and readiness to take action. The tracks can also join, and in general this type of DNA walker can be represented as a planar circuit that can compute an arbitrary Boolean function (Dannenberg et al. 2013b, 2014).

Example 1. *We consider a biosensor that detects the presence of DNA strands X and Y, and delivers cargo to the end node. Figure 1 shows the walker circuit, where the tracks labelled X and $\neg Y$ are unblocked, meaning Y is absent. At the junction, the walker selects or senses the unblocked track.*

The stepping process is illustrated in Figure 2. The walker strand carries a load (Q) that will quench fluorophores (F) that are attached to absorbing anchorages (1). It starts on the initial anchorage and, when a nicking enzyme (E) is present (2), traverses the circuit until it reaches an absorbing anchorage, which prevents further stepping. When the nicking enzyme is added to the solution, it binds to the walker-anchorage complex and cuts the anchorage into two strands. The strand formed from the tip is too short to remain hybridized to the walker, and melts away into solution. This exposes the top six nucleotides of the walker,

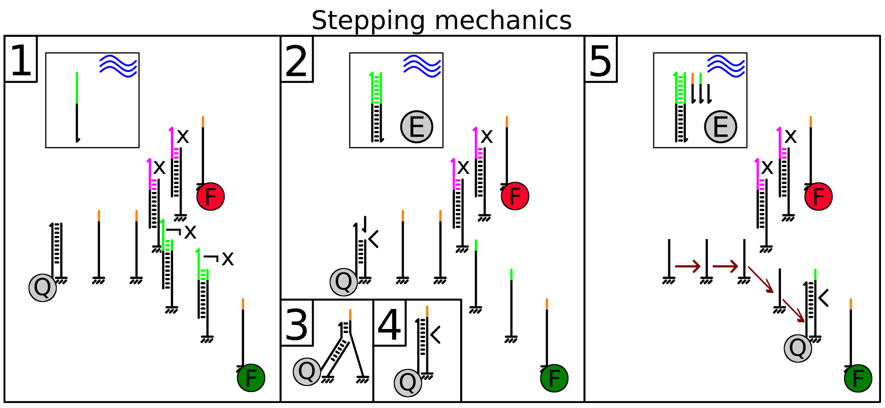


Figure 2: DNA walker circuit that can sense incoming strands (the domain coloured green).

which then attach to the next anchorage (3). In a displacement reaction, the walker becomes completely attached to the new anchorage (4). The reaction is energetically favourable, as the walker re-forms the six base-pairs with the intact anchorage that were lost after the nicking.

Repeating this process, the walker arrives at the junction. The walker continues down the track that was unblocked due to the presence of the strand being sensed (5), eventually quenching the fluorophore (F) on reaching the final node. The change in fluorescence is easily detectable, indicating that the presence of a certain configuration of molecules has been detected.

The biosensing example illustrates well the functionality that design automation for molecular programming must support. Firstly, we need *modelling frameworks* that can provide rigorous foundations to the mechanisms described by populations of molecules interacting through biochemical reactions, both in well-mixed solution, as well as localised, e.g., tethered to a surface. These must be able to capture molecular motion and structure-forming, in addition to information processing and decision making. Secondly, we need *programming languages* tailored to molecular programming and *programming abstractions*, to provide the layers through which molecular programs are transformed into the actual physical realisations. Thirdly, in view of the stochasticity, *quantitative specifications* are needed, which the designed molecular programs must meet, such as ensuring that “the probability of incorrect detection is tolerably low”, and “the expected time for the walker to reach the target node is sufficiently fast”. Finally, a broad range of *analysis* and *synthesis* methods is necessary, where the former should focus on predictability, correctness and resource usage, and the latter will need to cover automated synthesis of both the mechanism as well as the layout, as in DNA walker circuits.

3 Design automation for molecular programming

Modelling frameworks. A computational process is typically formalised by defining a transition system comprising a set of system states and a formal set of rules that govern the evolution of the system over time. A molecular program combines *discrete, continuous and stochastic* dynamics. The class of models that naturally captures such behaviours is known as stochastic hybrid systems. In view of their complexity and intractability, the modelling frameworks for stochastic hybrid systems resort to discretisation or approximation, which, under suitably strong conditions, can reduce the system model to, e.g., a finite-state Markov chain variant or a coupled system of linear equations, which are all tractable. When in solution, molecular networks, such as DNA strand displacement networks above, induce discrete stochastic dynamics; if spatiality is included, however, the continuous dimension must also be integrated to model motion in the physical space. Another important requirement is the need to augment models with quantitative features specific to molecular programming, e.g., thermodynamics, kinetics and resource usage.

There are two established frameworks for modelling molecular reactions in solution, the *continuous deterministic* approach and the *discrete stochastic* approach (Kurtz 1972; Gillespie 1977; McAdams and Arkin 1997). In the deterministic approach, one approximates the number of molecules by a continuous function that represents the time-dependence of the molecular concentration and evolves according to differential equations (ODEs) based on mass action kinetics. The ODE approach is suitable for modelling averaged behaviour and assumes large numbers of molecules. The discrete stochastic framework, on the other hand, models the stochastic evolution of populations of molecules. Reactions are discrete, stochastic events governed by rates that are typically assumed to be dependent only on the number of molecules: the system is conveniently represented as a continuous-time Markov chain (CTMC). This approach is more accurate in cases where the number of molecules is small and the system behaviour becomes non-continuous. It is also appropriate when it is necessary to take account of abstract spatial information, such as track layout in the case of molecular walkers.

The modelling challenge: State-based abstractions of molecular programs have the potential to enable analysis of correctness of the computation performed by the molecular program, for example deadlock, presence or absence of a given strand, or termination. The key challenge is *scalability*, which can be improved by identifying suitable model reductions, for example based on bisimulation quotients, symmetry reductions or symbolic techniques, or compositional theories. Another difficulty is the need to *integrate* the discrete, continuous and stochastic dynamics within a tractable modelling framework, in order to model molecular robotic systems and origami folding.

Languages for molecular programming. A number of programming languages exist that are specifically tailored to DNA computation, for example Visual DSD (Phillips and Cardelli 2009), from which CTMC models are automatically generated; Caltech's Seesaw Compiler, which accepts descriptions of logic circuits and outputs DNA sequences; and DACCAD (DNA Artificial Circuits Computer-Assisted Design) (Aubert et al. 2014), which outputs reaction networks in ODE semantics. The tools also output SBML format for further processing. One advantage of textual design descriptions is their flexibility and ease of modification, with the view to enable design reusability. Another advantage is that, as for conventional circuit designs, a range of analysis techniques are available to check for correctness of the designs, in addition to specialised properties of DNA systems such as sequencing thermodynamic properties, e.g. NUPACK, and structural descriptions for origami designs, as supported by CadNano. More generally, a variety of stochastic process algebras supported by software tools, such as stochastic pi-calculus (Regev et al. 2001) and BioPEPA (Ciocchetta and Hillston 2009), are capable of modelling molecular networks. Systems biology tools such as COPASI (Hoops et al. 2006) are also often applicable.

The language challenge: Though existing process-algebraic languages have proved useful for describing complex molecular programs, much more effort is needed to design high-level languages for emerging *experimental phenomena*, such as localised hybridization, origami design and molecular motors, to capture aspects such as spatiality, geometry and mobility, together with associated rigorous semantics, equivalence/refinement notions, and compositional behavioural theories.

Programming abstractions for design automation. Design automation tools implement design flows that compile high-level system descriptions, via intermediate languages, down to the physical design. Typically, at the top level, designs will be given in the form of a Boolean function or component-based designs, and compiled into *intermediate notations*, such as the CRN level and the sequence level, before they can be physically realised through nanofabrication. An example intermediate language is Cardelli's Strand Algebra (Cardelli 2010), a stochastic process algebra supported by the DSD tool (Phillips and Cardelli 2009). Designs can be analysed using a variety of techniques at the intermediate level, in technology neutral fashion. In common with digital designs, molecular programming languages are *modular*, and can be built from appropriate bio-components, which are themselves abstractions of certain biological mechanisms, for example molecular motors, local hybridization or self-assembly. Compositionality at the design level is therefore a desirable feature of the abstractions, both at the level of syntax, as well as semantics, which is harder to achieve in presence of stochasticity and hybrid dynamics.

The abstraction challenge: Despite progress made towards modeling well-mixed molecular systems, for example using stochastic pi-calculus, there is an urgent need to develop *quantitative theories* for the different levels of abstraction hierarchy, in order to support the design of *predictable* molecular systems. In particular, we need to develop compiler technology for molecular programming, including design and implementation of intermediate language abstractions, and efficient algorithms for computing approximate abstractions to a specified level of precision.

Specification notations. Since molecular programs are naturally characterised using quantities, for example kinetics, thermodynamics and stochasticity, the specifications that these programs must meet have to reflect these characteristics. For example, a biosensor must detect the given molecule with sufficiently high probability, and a molecular walker will need to guarantee delivery in a specified time interval, while tolerating a predefined failure rate. Stochastic and real-time temporal logics, for example CSL (Baier et al. 2003; Kwiatkowska et al. 2007), can be used to express many such properties for CTMC models, and logics such as MTL and STL for hybrid models. Conventional temporal logic notations also have their uses; for example, we may wish to require that a molecular program reaches some final state. Furthermore, characteristics typical for device engineering, such as safety, reliability and performance, also apply here. For example, for the DNA walker, which may fail to step correctly on to the next anchorage and instead jump to the following one, we may express the probability of finishing correctly at time T by the CSL formula $P_{=?}[\, F^{[T,T]}\, \text{finished-correct}\,]$.

The specification challenge: Designs must meet specifications that are set in advance. There has been little work concerning *quantitative* specification formalisms that capture aspects specific to the molecular programming setting, such as kinetic energy and thermodynamics, as well as behavioural specifications for out-of-equilibrium systems, e.g. oscillations. Devising suitable logic formalisms to support these more expressive specifications is desirable.

Analysis methods. A broad range of analysis methods exist to exercise models of molecular programs, which are dependent on the modelling framework used. They include techniques similar to those for digital systems, for example *equivalence checking* and *substitutivity* for components, and must extend to capture specialised features, to mention support for DNA self-assembly and structure forming, where thermodynamics and sequence design play a part. For ODE or hybrid models of molecular programs, *analytical* methods or *numerical simulation* can be used to plot average quantities, such as population sizes, over time.

For discrete stochastic models, *stochastic simulation*, for example Gillespie's algorithm, generates individual trajectories by applying Monte Carlo techniques. In contrast, *automated verification* via model checking is able to establish whether a given temporal logic property – e.g., can the system reach a terminal (deadlock) state? – holds in the model. For discrete stochastic models, we apply *automated, quantitative verification* (also known as probabilistic or stochastic model checking) (Kwiatkowska et al. 2007), which accepts a model description and a property specified as a probabilistic temporal logic formula, and computes the probability that the property is satisfied in the model, or expected cost/reward. Compared with simulation, such methods are *exhaustive* and can offer *guarantees* on reliability or performance. The computation can be exact, involving numerical algorithms based on uniformisation (Baier et al. 2003) or fast-adaptive uniformisation (Dannenberg et al. 2013a) (for transient probability), or approximate, based on probability estimation of the proportion of simulated trajectories that satisfy the property (Younes et al. 2006) (referred to as *statistical* model checking). These techniques have been applied to analyse molecular signalling networks (Heath et al. 2008) and have since been adapted to molecular programs. For example, an undesirable deadlock state was automatically discovered in the Cardelli transducer gate design modelled in Visual DSD, and the probability of reaching deadlock obtained using the `PRISM` (Kwiatkowska et al. 2011) probabilistic model checker as a back-end (Lakin et al. 2012). In Dannenberg et al. (2013b,a, 2014), DNA walker systems were subjected to comprehensive analysis of their reliability and performance; a CTMC model was developed based on experimental data (Wickham et al. 2011) and analysed with `PRISM`. The results of the analysis by `PRISM` can be used by the designer to improve the design of the circuit.

Example 2. *We illustrate the role of* automated, quantitative verification *techniques in molecular programming through assessing the reliability and performance of a biosensor implemented using DNA walker circuits (Dannenberg et al. 2013b, 2014). Experiments (Wickham et al. 2011) demonstrate that the walker can traverse a track with one or more omitted anchorages, which shows that the walker is capable of stepping across distances that are double or triple the normal anchorage-to-anchorage distance. This has been incorporated in the model, resulting in the walker being able to jump over blockades, or even reverse direction, which can prevent it from reaching the intended absorbing anchorage and quenching the fluorophore. The walker may also deadlock before reaching an absorbing anchorage, which happens when no uncut anchorages are within reach. This affects the* safety *of biosensors implemented using DNA walkers.*

The trade-off between reliability and deadlock as a function of blockade length is depicted in Fig. 3, obtained by model checking the model in Fig. 1 against the

CSL property $P_{=?}[\,F^{[T,T]}$ end-node$\,]$ that queries the probability of the walker being either finished or deadlocked at time T, where T is 8 mins multiplied by circuit depth. Note that the probability that the walker arrives at the incorrect end-node drops off, while the probability of deadlock increases with the depth of the circuit.

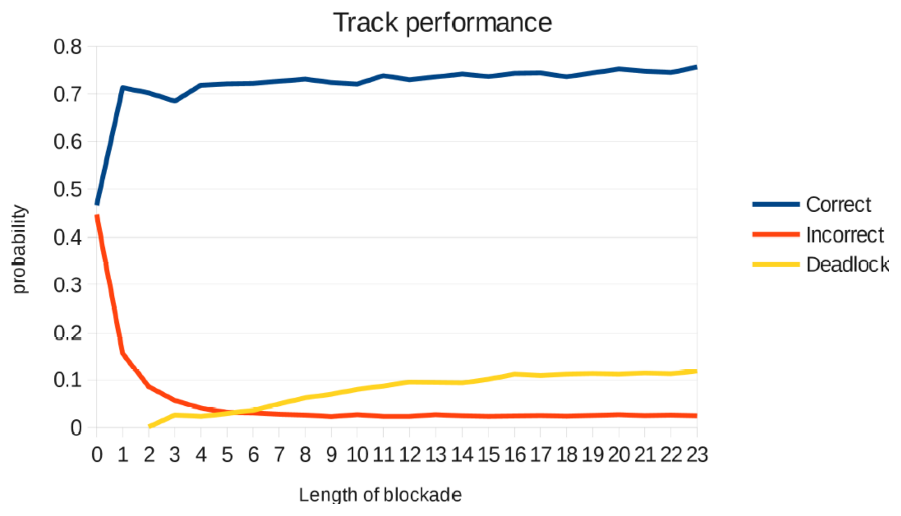


Figure 3: Probability of reaching an absorbing anchorage or deadlock by time T (8 mins times circuit depth) for the walker circuit in Fig. 1.

The analysis challenge: Stochastic simulation methods are known to be computationally intensive and their performance is poor for molecular systems that we wish to design and analyse today. We need much more *efficient* simulation techniques, for example those based on multi-scale simulation which have shown promise. Formal verification techniques, such as automated verification via model checking or interactive theorem proving, are able to establish, via a systematic exploration of the model or mathematical proof, the correctness of a molecular program. Their limitation is the size of the state-space of the model that can be handled, and therefore *scalable, quantitative verification* techniques are a major goal of this research. Promising techniques include SAT/SMT methods, abstraction-refinement schemes and compositional proof methods. Since the models of molecular programs typically include quantitative and possibly continuous aspects, for example stochasticity and energy, we ultimately need the analysis methods to extend to the full class of stochastic hybrid systems. To facilitate their analysis one must apply abstractions and (numerical) *approximations*. This raises the question of the level of precision, including accuracy and error bounds, that the analysis method can inherently guarantee, in turn impacting the predictability of the molecular program's behavior.

Synthesis methods. In addition to being able to perform verification of molecular programs against requirements, an important question is whether it is possible, given a specification, to automatically synthesize a program that is guaranteed to satisfy the specification. This approach would ensure *correct-by-construction* designs, and is in its infancy, particularly regarding *quantitative* synthesis. Techniques developed in systems biology to infer models from experimental data are naturally applicable here. For example, *parameter* synthesis (or estimation) can be used to fit the kinetic rates in a molecular program to observations (Hoops et al. 2006), or even find the optimal values of parameters to satisfy a given quantitative formula for stochastic models (Brim et al. 2013). One example based on the DNA walker circuits is finding the range of walker failure rate parameters so that a specified reliability of the design can be guaranteed. More generally, for a given (quantitative) specification, synthesis methods can be used to automatically configure a system from components; to synthesise a program (mechanism) that guarantees the satisfaction of the property; or even to *evolve* such as program, using techniques from evolutionary computation or genetic programming. For nanorobotic systems, a natural question is whether we can synthesise programmable *controllers* that ensure the safety of the molecular device. Similarly to digital systems, synthesis algorithms are also necessary to support and optimise the *structural* designs, including 2D/3D origami structures and the geometric layout of DNA walker circuits.

The synthesis challenge: This topic has been little researched in the context of quantitative or hybrid models, and its complexity and intractability pose a huge challenge. Promising technique might include template-based synthesis of mechanisms, and developing controller synthesis methods, including from multi-objective specifications.

Integration. A major challenge is to integrate all abstraction levels (from thermodynamics, to sequence, to CRNs), and to achieve fully automatic compilation from high-level specifications to physical structures, with analysis enabled at each level and connected across levels. This can be achieved by relying on modular designs and substitutivity of component specifications for their implementations. The compositional design methodologies and CAD tools that result from this effort will support effective processes to engineer systems from independently specified bio-components.

The integration challenge: Once the integrated CAD tools have been developed, their usefulness must be evaluated on real molecular programs and synthetic biology designs. Criteria for success include the rate of take up of the technologies, improvement in scale and complexity of the designs, the efficiency of software tools, the accuracy of predictions for quantitative aspects, and the quality of the synthesised designs.

4 Conclusions

This paper has given a brief overview of the emerging field of molecular programming, discussing existing techniques to support the design process and outlining future research challenges. Molecular programming has the potential to revolutionise personalised medicine and synthetic biology, with many applications envisaged, for example programmable intraveneous systems to deliver drugs that target specific molecules. Since safety is a paramount concern when deploying such devices, we have put forward the view that quantitative, automated verification techniques will constitute a key component of design automation for molecular programming. This can only be achieved through collaboration between experimental scientists, engineers and computer scientists.

Acknowledgments. The author is supported by the ERC Advanced Grant VERIWARE and would like to acknowledge participants of the NSF workshop "Advances in Molecular Programming and Computing: Toward Chemistry as a New Information Technology" held in 2013 in Copenhangen for helpful discussions.

References

D. Angluin, J. Aspnes, and D. Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, 2008.

N. Aubert, C. Mosca, T. Fujii, M. Hagiya, and Y. Rondelez. Computer-assisted design for scaling up systems based on dna reaction networks. *Journal of The Royal Society Interface*, 11(93):20131167, 2014.

C. Baier, B. Haverkort, H. Hermanns, and J. Katoen. Model-checking algorithms for continuous-time markov chains. *IEEE Transactions on Software Engineering*, 29:524–541, 2003.

L. Brim, M. Ceska, S. Drazan, and D. Safránek. Exploring parameter space of stochastic biochemical systems using quantitative model checking. In N. Sharygina and H. Veith, editors, *Proc. CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 2013.

L. Cardelli. Two-domain DNA strand displacement. *Developments in Computational Models*, 26:47–61, 2010.

L. Cardelli and A. Csikász-Nagy. The cell cycle switch computes approximate majority. *Nature Scientific Reports*, 2, 2012.

Y.-J. Chen, N. Dalchau, N. Srinivas, A. Phillips, L. Cardelli, D. Soloveichik, and G. Seelig. Programmable chemical controllers made from DNA. *Nature Nanotechnology*, 8(10):755–762, 2013.

F. Ciocchetta and J. Hillston. Bio-PEPA: A framework for the modelling and analysis of biological systems. *Theoretical Computer Science*, 410(33-34):3065–3084, 2009.

F. Dannenberg, E. M. Hahn, and M. Kwiatkowska. Computing cumulative rewards using fast adaptive uniformisation. In A. Gupta and T. Henzinger, editors, *Proc. 11th Conference on Computational Methods in Systems Biology (CMSB'13)*, volume 8130 of *LNCS*, pages 33–49. Springer, 2013a.

F. Dannenberg, M. Kwiatkowska, C. Thachuk, and A. Turberfield. DNA walker circuits: computational potential, design, and verification. In D. Soloveichik and B. Yurke, editors, *Proc. 19th International Conference on DNA Computing and Molecular Programming (DNA 19)*, volume 8141 of *LNCS*, pages 31–45. Springer, 2013b.

F. Dannenberg, M. Kwiatkowska, C. Thachuk, and A. Turberfield. DNA walker circuits: computational potential, design, and verification. *Natural Computing*, 2014. To appear.

D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

M. Hagiya. From molecular computing to molecular programming. In A. Condon and G. Rozenberg, editors, *DNA Computing*, volume 2054 of *Lecture Notes in Computer Science*, pages 89–102. Springer, 2000.

J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn. Probabilistic model checking of complex biological pathways. *Theoretical Computer Science*, 319(3):239–257, 2008.

S. Hoops, S. Sahle, R. Gauges, C. Lee, J. Pahle, N. Simus, M. Singhal, L. Xu, P. Mendes, and U. Kummer. COPASI - a COmplex PAthway SImulator. *Bioinformatics*, 22(24):3067–3074, 2006.

C. Jung and A. D. Ellington. Diagnostic applications of nucleic acid circuits. *Accounts of Chemical Research*, 2014. To appear.

T. G. Kurtz. The relationship between stochastic and deterministic models for chemical reactions. *The Journal of Chemical Physics*, 57:2976, 1972.

M. Kwiatkowska. Quantitative verification: Models, techniques and tools. In *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 449–458. ACM Press, September 2007.

M. Kwiatkowska and C. Thachuk. Probabilistic model checking for biology. In *Software Safety and Security*, NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2014. To appear.

M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270. Springer, 2007.

M. Kwiatkowska, G. Norman, and D. Parker. *Symbolic Systems Biology*, chapter Probabilistic Model Checking for Systems Biology, pages 31–59. Jones and Bartlett, 2010.

M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. CAV'11*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

M. Lakin, D. Parker, L. Cardelli, M. Kwiatkowska, and A. Phillips. Design and analysis of DNA strand displacement devices using probabilistic model checking. *Journal of the Royal Society Interface*, 9(72):1470–1485, 2012.

H. H. McAdams and A. Arkin. Stochastic mechanisms in gene expression. *Proceedings of the National Academy of Sciences*, 94:814–9, 1997.

A. Phillips and L. Cardelli. A programming language for composable DNA circuits. *Journal of the Royal Society Interface*, 2009.

L. Qian and E. Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332:1196–1201, 2011.

A. Regev, W. Silverman, and E. Y. Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Pacific Symposium on Biocomputing*, pages 459–470, 2001.

P. Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440:297–302, 2006.

G. Seelig, D. Soloveichik, D. Zhang, and E. Winfree. Enzyme-free nucleic acid logic circuits. *Science*, 314:1585–1588, 2006.

D. Soloveichik, G. Seelig, and E. Winfree. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Science*, 107(12): 5393–5398, 2010.

S. F. J. Wickham, M. Endo, Y. Katsuda, K. Hidaka, J. Bath, H. Sugiyama, and A. J. Turberfield. Direct observation of stepwise movement of a synthetic molecular transporter. *Nature Nanotechnology*, 6:166–9, 2011.

E. Winfree. Toward molecular programming with DNA. *SIGPLAN Not.*, 43(3): 1–1, Mar. 2008. URL `http://doi.acm.org/10.1145/1353536.1346282`.

P. Yin, H. Yan, X. G. Daniell, A. J. Turberfield, and J. H. Reif. A unidirectional DNA walker that moves autonomously along a track. *Angewandte Chemie International Edition*, 43:4906–4911, 2004.

H. Younes, M. Kwiatkowska, G. Norman, and D. Parker. Numerical vs. statistical probabilistic model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(3):216–228, 2006.

B. Yurke, A. Turberfield, A. Mills, F. Simmel, and J. Neumann. A DNA-fuelled molecular machine made of DNA. *Nature*, 406(6796):605–8, 2000.

D. Zhang and G. Seelig. Dynamic DNA nanotechnology using strand displacement reactions. *Nature Chemistry*, 3:103–113, 2011.

D. Y. Zhang, A. J. Turberfield, B. Yurke, and E. Winfree. Engineering entropy-driven reactions and networks catalyzed by DNA. *Science*, 318(5853):1121, 2007.

Temporal Logic: The Lesser of Three Evils

Leslie Lamport

Microsoft Research

In the Beginning

Amir Pnueli introduced temporal logic to computer science in 1977 in a paper presented at FOCS (Pnueli 1977). That paper inspired Susan Owicki to organize an informal seminar on the subject at Stanford during the 1977–78 academic year. Temporal logic sounded to me like yet another of the useless formalisms that computer scientists seemed fond of, but I decided to attend anyway. That was one of the best decisions I ever made.

Owicki (together with David Gries) and I had independently developed what is now known as the Owicki-Gries method for proving invariance properties of concurrent programs (Lamport 1977; Owicki and Gries 1976). I had also devised a method for proving liveness properties of the form $P \leadsto Q$, read P *leads to* Q, which asserts that if P is true then Q will eventually become true. Written informally, my proofs were reasonable. However, their formalization was ugly and complicated.

Owicki and I soon realized that Pnueli's temporal logic was ideal for formalizing liveness proofs. The logic was simple, based on the single operator $\Box$, read *always* or *henceforth*, where $\Box P$ asserts that P is true from now on. Its dual $\Diamond P$, defined to equal $\neg\Diamond\neg P$, asserts that P is eventually true. My $\leadsto$ operator could be defined by

$$P \leadsto Q \;\equiv\; \Box(P \Rightarrow \Diamond Q)$$

Temporal logic added to my liveness proofs the ability to directly use invariance properties. The invariance of a formula I means that $\Box I$ is true. We could use this fact to prove liveness properties by applying the proof rule

$$\frac{(I \land P) \leadsto Q}{\Box I \Rightarrow (P \leadsto Q)}$$

This rule was the key to the elegant formalization of liveness proofs. It was enshrined in the boxes of the proof lattices Owicki and I introduced (Owicki and Lamport 1982).

Inadequate and Evil

In the late 1970s and early 1980s, I and many of my colleagues started going beyond the realm of proving that programs satisfied particular properties to trying to write and verify complete specifications. Temporal logic seemed to be wonderful for the task of specifying a system. A specification would simply be the conjunction of temporal-logic formulas that asserted properties the system must satisfy. I believe that the first publication advocating this approach was by Richard Schwartz and Michael Melliar-Smith (1981).

By the time of that paper's publication, I had realized that temporal logic was not all that wonderful. In fact, I was originally an author but had my name removed because I had become disillusioned with the method. Watching Schwarts and Melliar-Smith, along with Fritz Vogt, spend days trying to specify a FIFO queue (a very trivial example) convinced me that the method would never work on any real example.

Others also realized that there was a problem. Most thought that the source of the problem lay in the simplicity of Pnueli's temporal logic. So, they developed a multitude of new, more complicated logics based on more expressive and more complicated temporal operators. I was not immune to that temptation (Lamport 1981). However, I eventually realized that the fundamental problem lay in trying to specify something by a list of properties.

Years of experience have taught me that human beings cannot understand the consequences of a conjunction of separate properties. As one of many pieces of evidence, consider multiprocessor memory models. Engineers have often specified them by a list of properties—for example, in the specifications of the DEC/Compaq Alpha (Alpha Architecture Committee 1998) and the Intel Itanium (Intel 2002) memories. Even the people who wrote the specifications did not understand them. Jim Saxe discovered that the published Alpha memory specification permitted causal loops, in which a write stores a completely arbitrary value, and that value is justified by a later read. Using a formal specification that we wrote, my colleagues and I discovered errors in the (very simple) examples in an early version of the Itanium memory specification.

This experience revealed the inadequacy of temporal logic for writing specifications. However, inadequacy is not evil. I discovered that temporal logic is evil in the late 1980s when it led my colleague Martín Abadi and me to believe a false result for several days. Those who know me will not be surprised that I made such an error, but those who know Martín Abadi will realize that, if he could be confused by temporal logic, then anyone can be.

Temporal logic is evil because it does not satisfy the deduction principle. In ordinary mathematics, we prove the implication $P \Rightarrow Q$ (P implies Q) by assuming P is true and proving Q is true. This reasoning is expressed by the following proof rule, which is called the deduction principle.

$$\frac{\begin{array}{c} P \\ \hline Q \end{array}}{P \Rightarrow Q}$$

The deduction principle is not valid for temporal logic and other modal logics. For example, a basic axiom of temporal logic is $\frac{P}{\Box P}$, which asserts that, if P is true, then it is always true. The deduction principle would allow us to deduce from this the truth of $P \Rightarrow \Box P$, a formula asserting that, if P is true initially, then it is always true—which is not valid in general.

The Greater Evils

The source of temporal logic's evil is that its formulas have an implicit variable representing time. The truth of a temporal formula asserts that the formula is true for all values of this variable. Calling the variable t, a proof rule $\frac{P}{Q}$ asserts that $\forall\, t : P$ implies $\forall\, t : Q$, while the truth of $P \Rightarrow Q$ means $\forall\, t : (P \Rightarrow Q)$. The deduction principle is invalid for temporal logic because we cannot deduce $\forall\, t : (P \Rightarrow Q)$ from $(\forall\, t : P) \Rightarrow (\forall\, t : Q)$.

One way to eliminate this problem is to make the time variable t explicit. Every atomic formula becomes an explicit function of t, so $P \leadsto Q$ is written $\forall\, t : (P(t) \Rightarrow \exists\, s \geq t : Q(s))$. This is exactly what Nissim Francez did in his thesis (Francez 1976). The messiness of representing even so simple a formula as $P \leadsto Q$ indicates why this is a bad idea. In fact, Francez was Pnueli's student, and I believe it was his thesis that inspired Pnueli to use temporal logic. Temporal logic is a lesser evil than the complexity introduced by an explicit time variable.

Another way people have tried to avoid the evil of temporal logic is to use some form of program logic in its place. Logic is a branch of mathematics, and one of the most basic operations of mathematics is substituting an expression for a variable. Substitution is fundamental to computing because it lies at the heart of refinement, which is also called implementation. In a volume commemorating the retirement of Willem-Paul de Roever, I illustrated refinement as substitution by showing how to derive an important hardware protocol from a simple specification. The main step essentially consisted of substituting $(p + c) \bmod 2$ for the variable x in the assignment statement $x := x + 1$ (Lamport 2010).

Although evil, temporal logic is still mathematics. One can therefore derive a temporal-logic description of the protocol from its temporal-logic specification by substituting $(p + c)$ mod 2 for x. However, literally substituting for x in the statement $x := x + 1$ makes no sense. One cannot substitute an expression for a variable in a program logic with assignment statements. Indeed, I know of no program logic in which such substitution is possible. A "logic" that does not permit substitution is a greater evil than temporal logic.

A Necessary and Useful Evil

Although evil, temporal logic is necessary. It is the best way we know to reason about liveness. Moreover, its ability to describe reactive systems, even if only in principle, helps us to understand them. The traditional first step in creating a science is to introduce mathematics. Temporal logic is the natural mathematics of reactive systems.

We cannot remove the evil from temporal logic, but we can overcome its inadequacy for writing specifications. This doesn't require new temporal operators; the $\Box$ operator that Pnueli introduced in 1977 is (approximately) enough. The trick is to extend the base formulas from state predicates to *actions*, which are predicates on pairs of states (Lamport 1994). The result is a logic that confines the evil of temporal logic mainly to the domain for which it is both necessary and useful: liveness.

References

Alpha Architecture Committee. *Alpha Architecture Reference Manual.* Digital Press, Boston, third edition, 1998.

N. Francez. *The Specification and Verification of Cyclic (Sequential and Concurrent) Programs.* PhD thesis, Weizmann Institute of Science, Rehovot, Israel, June 1976.

Intel. A formal specification of Intel Itanium processor family memory ordering. Application Note. `http://download.intel.com/design/Itanium/Downloads/25142901.pdf`, Oct. 2002.

L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, SE-3(2):125–143, Mar. 1977.

L. Lamport. Timesets—a new method for temporal reasoning about programs. In D. Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer*

Science, pages 177–196, Berlin, Heidelberg, New York, May 1981. Springer-Verlag.

L. Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16 (3):872–923, May 1994.

L. Lamport. Computer science and state machines. In D. Dams, U. Hannemann, and M. Steffen, editors, *Concurrency, Compositionality, and Correctness (Essays in Honor of Willem-Paul de Roever)*, volume 5930 of *Lecture Notes in Computer Science*, pages 60–65. Springer, 2010.

S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–284, May 1976.

S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Prog. Lang. Syst.*, 4(3):455–495, July 1982.

A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, Nov. 1977.

R. L. Schwartz and P. M. Melliar-Smith. Temporal logic specification of distributed systems. In *Proceedings of the 2nd International Conference on Distributed Computing Systems*, pages 446–454. IEEE Computer Society Press, Apr. 1981.

Simple proofs of simple programs in Why3

Jean-Jacques Lévy
State Key Laboratory for Computer Science,
Institute of Software, Chinese Academy of Sciences
& Inria

Abstract

We want simple proofs for proving correctness of simple programs. We want these proofs to be checked by computer. We also want to use current interactive or automatic provers and not build new ones. Finally Hoare logic is a good framework to avoid complex definitions of the operational semantics of programming languages. In this short note, we take the example of merge-sort as expressed in Sedgewick's book about Algorithms and demonstrate how to prove its correctness in Why3, a system developed at Université de Paris-Sud, Cnrs and Inria. There are various computer systems which integrate proofs and programs, e.g. VCC, Spec#, F*, Frama-C, etc. Why3 integrates a small imperative language (Why ML) and an extension of Hoare logic with recursive data types and inductive predicates. It is interfaced with interactive systems (Coq, Isabelle/HOL, PVS) and automatic provers (Alt-Ergo, Z3, CVC3, CVC4, E-prover, Gappa, Simplify, Spass, Yices, etc). Therefore Why3 can also be considered as a fantastic back-end for other programming environments.

1 Introduction

Formal proofs of program safety are always a big challenge. Usually they comprise a large number of cases, which make them intractable on paper. Fortunately there are a few proof-assistants which guarantee the exactness of formal proofs, but less many mixing programs and proofs. Moreover we believe that these computer-checked proofs should be readable and simple when we have to prove simple programs. In this note, we take Hoare logic as the basic formal setting and we consider a simple Pascal-like imperative programming language (Why ML) with an ML-like syntax. This logic is first-order with several extensions to allow recursive data and inductively defined predicates. We hope that both syntax of the programs and logic are self-explainable. If not, the interested reader is referred to the Why3 web site. We illustrate here the use of the Why3 system through a proof of the merge-sort program. We consider two versions of it: on lists and on arrays. The version on arrays is far more complex to prove correct.

2 Mergesort on lists

In Why ML, merge-sort on lists of integers is expressed in figure 1. Its correctness proof is easy. Pre and post-conditions follow keywords **requires** and **ensures** in the headers of functions. In post-conditions, `result` represents the value returned by the function; `sorted`, `permut`, `(++)` are predicates and functions defined in the theory of polymorphic lists in the Why3 standard library (located at URL `http://why3.lri.fr/stdlib-0.83`). Part of this theory is visible in figure 2. The verification conditions generated by Why3 can be proved automatically with Alt-Ergo, CVC3 and Eprover 1-6. The longest proof is the one for the post-conditions of `merge` (5.02 sec by CVC3) and `split` (2.05 sec by Eprover). These timings are obtained on a regular dual-core notebook. The choice of the provers and of the transformations to apply to goals (splitting conjunctions, inlining function definitions) is manual, but easy to perform thanks to the Why3 graphic interface.

Moreover the assertions are natural and look minimal. Maybe the most mysterious part is the post-condition of `merge`, which states that the result is a permutation of the concatenation ℓ_1 `++` ℓ_2 of the parameters ℓ_1 and ℓ_2 of `merge`. This property is needed to prove the `sorted` post-condition since `x1` (or `x2`) should be ranked with respect to the result of the recursive calls of `merge`. In fact it is sufficient to know that that `result` only contains elements of the lists ℓ_1 (or ℓ_2). Now the proof of `permut` in the post-condition of `merge` is totally orthogonal and is resumed in the second part of figure 1.

3 Mergesort on arrays

We now consider the program as written in Sedgewick and Wayne (2011). In this version of `mergesort` there is a trick in organizing the area to merge as a bitonic sequence, increasing first and decreasing afterwards (although not expressed in that way in the book). It thus avoids multiple loops or tests as halting conditions of loops. See the code on figure 3 in Why ML language where the second half of `a` is copied into second half of `b` in reverse order. In pre and post-conditions, new predicates or functions are used: `sorted_sub` and `permut_sub` mean `sorted` and `permut` on subarrays between bounds `lo` (included) and `hi` (excluded); `(old a)` means the array `a` before calling the function.

```
let rec split (l : list int)                              (* first part *)
 = match l with
 | Nil -> (Nil, Nil)
 | Cons x Nil -> ((Cons x Nil) , Nil)
 | Cons x (Cons y l') -> let (xs,ys) = split l' in
                                  ((Cons x xs), (Cons y ys))
 end

let rec merge l1 l2
  requires { sorted l1 /\ sorted l2}
  ensures { sorted result /\ permut result (l1 ++ l2) }
 = match l1, l2 with
   | Nil, _ -> l2
   | _, Nil -> l1
   | Cons x1 r1, Cons x2 r2 ->
      if x1 <= x2 then Cons x1 (merge r1 l2)
                  else Cons x2 (merge l1 r2)
   end

let rec mergesort l
  ensures { sorted result }
   = match l with
   | Nil | Cons _ Nil -> l
   | _ -> let l1, l2 = split l in merge (mergesort l1) (mergesort l2)
   end

                                                          (* second part *)
let rec split (l : list int)
 ensures { let (l1, l2) = result in permut l (l1 ++ l2)} = ...

let rec merge l1 l2
  ensures { permut result (l1 ++ l2)}  = ...

let rec mergesort l
  ensures { permut result l}  = ...
```

Figure 1: Mergesort on lists

The proof of `sorted` in post-condition of `mergesort` needs several add-ons to the theory of arrays in the Why3 standard library as shown in figure 4. An array is represented by a record with two fields: an integer `length` and a total map `elts` from integers to values. The functions `get` and `set` reads and writes a value from or into an element of an array (or map). Thus we define the predicates `array_eq_sub_rev_offset`, `dsorted_sub` and `bitonic_sub` both on arrays and maps. (Why3 translates predicates over arrays into predicates over maps, where the solvers are mainly acting). The first predicate is a technical abbrevia-

```
function (++) (l1 l2: list 'a) : list 'a = match l1 with
     | Nil -> l2
     | Cons x1 r1 -> Cons x1 (r1 ++ l2)
end

inductive sorted (l: list t) =
     | Sorted_Nil:
         sorted Nil
     | Sorted_One:
         forall x: t. sorted (Cons x Nil)
     | Sorted_Two:
         forall x y: t, l: list t.
         le x y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))

function num_occ (x: 'a) (l: list 'a) : int =
    match l with
     | Nil      -> 0
     | Cons y r -> (if x = y then 1 else 0) + num_occ x r
   end

predicate permut (l1: list 'a) (l2: list 'a) =
    forall x: 'a. num_occ x l1 = num_occ x l2
```

Figure 2: Theory of lists

tion to test for equality after reversing and adding an offset to a subarray. The two last predicates mean down-sorted or bitonic on sub-arrays (and sub-maps). We finally add two lemmas about weakening the interval of a bitonic subarray.

To prove the first part (`sorted_sub`) of the post-condition of `mergesort1`, we add several assertions and invariants in its body as shown on figure 5. We use a new operator (`at a 'L`) in formulas meaning the value of array `a` at label `L`. The proof of the 161 verification conditions is fully automatic with the Alt-Ergo prover and quasi online. But this happened after many attempts and reformulations of assertions and invariants and use other provers. Retries are favourised by the incremental dependency analysis (stored in a so-called `why3session.xml` file) of Why3 which only recomputes the modified goals.

To summarize the logic of this function, there are two recursive calls on both halves of array `a`; the first half is copied into the first half of array `b`; the second half is copied in reverse order into the second half of array `b`; finally the merge of the two halves of `b` is returned in `a`. Notice that during the merge phase, the index `j` can go over the half `m` of the array `b`. Therefore the assertion `m` <= !`j` is not true since the index `j` can go up to `lo` when all elements are equal in `b`.

The second part (`permut_sub`) of the post-condition of `mergesort1` follows the same lines and is exhibited at URL `http://jeanjacqueslevy.net/why3/sorting/`.

Permutations on arrays are defined by counting the number of occurrences for each value. Therefore the proof demands several properties of occurrences of values in sub-arrays. The proof is very natural except for a couple of redundant assertions, which ease the behaviours of automatic provers. In fact, many provers are involved in that second part, namely Alt-Ergo, Yices, CVC4 and Z3, thoroughly chosen thanks to the graphical interface of Why3. Moreover several manual transformations were needed, such as inlining and splitting of conjunctions.

The two lemmas about weakening the interval of `map_bitonic` are proved by 30 lines of easy and readable Coq (with *ss-reflect* package). It needs 4 extra lemmas (each with 7-line long Coq proof) `sorted_sub_weakening`, `dsorted_sub_weakening`, `sorted_sub_empty` and `dsorted_sub_empty` which states the weakening of intervals for (d)sorted subarrays and the (d)sorted status of empty subarrays. In fact these lemmas could also be proved by automatic provers, but there is a trade-off between expressing abstract properties and a detailed computable presentation. For instance, `bitonic` is defined with an existential connector, which is quasi equivalent to the end of automatic first-order provers. A more precise presentation with parameterizing the index at peak of the bitonic sequence would have reactivated the automatic methods. In fact, this trade-off is a big advantage of Why3. For instance, a verification condition can also be first attempted in Coq, and later proved automatically after simplifications.

4 Conclusion

The mergesort example demonstrates the versatility of the Why3 system. One can nicely mix automatic and interactive proofs. The multiplicity of solvers (SMT solvers and theorem provers) give high confidence before attacking an interactive proof, which is then reserved for conceptual parts. It is even possible to call back automatic provers from the interactive parts (not in the mergesort example, but it did happen in a version of quicksort to avoid a long Coq proof with numerous cases), but it requires to solve several technical typing subtleties. The system demands some training since it is a bit complex to manipulate numerous solvers and interactive proof-assistants. The WhyML memory model is rather naive since variables and arrays only allow single assignments. New variables or new arrays are created after every modification of their contents. Moreover arrays are immediately expanded to maps. Therefore it would be interesting to understand how far one can go with this memory model. It did not prevent from already building a gallery of small verified programs existing in the Why3 public release. The Frama-C project uses Why3 among other analyzers to build a verification environment for C programs written for small run-times or embedded systems.

```
let rec mergesort1 (a b: array int) (lo hi: int) =
  requires {Array.length a = Array.length b /\
        0 <= lo <= (Array.length a) /\ 0 <= hi <= (Array.length a) }
    ensures { sorted_sub a lo hi /\ permut_sub (old a) a lo hi }
  if lo + 1 < hi then
  let m = div (lo+hi) 2 in
  mergesort1 a b lo m;
  mergesort1 a b m hi;
  for i = lo to m-1 do
    b[i] <- a[i]
    done;
  for j = m to hi-1 do
    b[j] <- a[m + hi - 1 - j]
    done;
  let i = ref lo in
  let j = ref hi in
  for k = lo to hi-1 do
    if b[!i] < b[!j - 1] then
      begin a[k] <- b[!i]; i := !i + 1 end
    else
      begin j := !j - 1; a[k] <- b[!j] end
  done

let mergesort (a: array int) =
  ensures { sorted a /\ permut (old a) a }
let n = Array.length a in
let b = Array.make n 0 in
  mergesort1 a b 0 n
```

Figure 3: Mergesort on arrays

Finally it is interesting to notice how robust and intuitive is Hoare logic.

5 Acknowledgements

This short note is dedicated to Luca Cardelli for his 60th anniversary. Luca has always loved mixing theoretical topics and real computing systems. I hope this short note satisfies that criterion. I also thank Chen Ran for her help in multiple Why3/Coq proofs.

References

C. Barrett and C. Tinelli. CVC4, the smt solver. New-York University - University of Iowa. URL http://cvc4.cs.nyu.edu.

```
use map.Map as M
clone map.MapSorted as N with type elt = int, predicate le = (<=)

predicate map_eq_sub_rev_offset (a1 a2: M.map int int) (l u: int)
  (offset: int) =
  forall i: int. l <= i < u ->
               M.get a1 i = M.get a2 (offset + l + u - 1 - i)

predicate array_eq_sub_rev_offset (a1 a2: array int) (l u: int)
                                                    (offset: int) =
  map_eq_sub_rev_offset a1.elts a2.elts l u offset

predicate map_dsorted_sub (a: M.map int int) (l u: int) =
  forall i1 i2 : int. l <= i1 <= i2 < u -> M.get a i2 <= M.get a i1

predicate dsorted_sub (a: array int) (l u: int) =
  map_dsorted_sub a.elts l u

predicate map_bitonic_sub (a: M.map int int) (l u: int) =  l < u ->
  exists i: int. l <= i <= u /\ N.sorted_sub a l i /\
                              map_dsorted_sub a i u

predicate bitonic_sub (a: array int) (l u: int) =
  map_bitonic_sub a.elts l u

lemma map_bitonic_incr : forall a: M.map int int, l u: int.
  map_bitonic_sub a l u -> map_bitonic_sub a (l+1) u

lemma map_bitonic_decr : forall a: M.map int int, l u: int.
  map_bitonic_sub a l u -> map_bitonic_sub a l (u-1)
```

Figure 4: Theory add-ons for mergesort on arrays

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The alt-ergo automated theorem prover, 2008. URL `http://alt-ergo.lri.fr/`.

F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. URL `http://proval.lri.fr/publications/boogie11final.pdf`.

L. de Moura and N. Björner. Z3, an efficient smt solver. Microsoft Research. URL `http://z3.codeplex.com`.

B. Dutertre and L. de Moura. The Yices SMT Solver. SRI. URL `http://yices.csl.sri.com`.

J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.

G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008. URL `http://hal.inria.fr/inria-00258384`.

R. Sedgewick and K. Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

A. Tafat and C. Marché. Binary heaps formally verified in Why3. Research Report 7780, INRIA, Oct. 2011. `http://hal.inria.fr/inria-00636083/en/`.

```
let rec mergesort1 (a b: array int) (lo hi: int) =
    requires {Array.length a = Array.length b /\
        0 <= lo <= (Array.length a) /\ 0 <= hi <= (Array.length a) }
    ensures { sorted_sub a lo hi /\ modified_inside (old a) a lo hi }
  if lo + 1 < hi then
  let m = div (lo+hi) 2 in
    assert{ lo < m < hi};
    mergesort1 a b lo m;
'L2: mergesort1 a b m hi;
    assert { array_eq_sub (at a 'L2) a lo m};
    for i = lo to m-1 do
      invariant { array_eq_sub b a lo i}
      b[i] <- a[i]
      done;
    assert{ array_eq_sub a b lo m};
    assert{ sorted_sub b lo m};
    for j = m to hi-1 do
      invariant { array_eq_sub_rev_offset b a m j (hi - j)}
      invariant { array_eq_sub a b lo m}
      b[j] <- a[m + hi - 1 - j]
      done;
    assert{ array_eq_sub a b lo m};
    assert{ sorted_sub b lo m};
    assert{ array_eq_sub_rev_offset b a m hi 0};
    assert{ dsorted_sub b m hi};
'L4: let i = ref lo in
    let j = ref hi in
    for k = lo to hi-1 do
      invariant{ lo <= !i < hi /\ lo <= !j <= hi}
      invariant{ k = !i + hi - !j}
      invariant{ sorted_sub a lo k }
      invariant{ forall k1 k2: int. lo <= k1 < k ->
                                    !i <= k2 < !j -> a[k1] <= b[k2] }
      invariant{ bitonic b !i !j }
      invariant{ modified_inside a (at a 'L4) lo hi }
      assert { !i < !j };
      if b[!i] < b[!j - 1] then
        begin a[k] <- b[!i]; i := !i + 1 end
      else
        begin j := !j - 1; a[k] <- b[!j] end
    done
```

Figure 5: Proof of mergesort on arrays

Introduction to New Perspectives in Biology

Giuseppe Longo[*] Mael Montévil[†]

Abstract

This note introduces recent work in Theoretical Biology by borrowing from the Introduction (chapter 1) of the book by the authors: "Perspectives on Organisms: Biological Time, Symmetries and Singularities", Springer, 2014. The idea is to work towards a Theory of Organisms analogue and along the Theory of Evolution, where ontogenesis could be considered as part of phylogenesis. As a matter of fact, the latter is made out of "segments" of the first: phylogenesis is the "sum" of ontogenetic paths and they should be made intelligible by similar principles. To this aim, we look at ontogenesis from different perspectives. By this, we shed light on the unity of the organism from different points of view, yet constantly keeping that unity as a core invariant. The analysis of *invariance*, as the result of theoretical symmetries, and of *symmetry changes*, is a key theme of the approach in the book and in the discussion in this note.

1 From physics towards biology

Current biology is largely an experimental discipline, that is most, and actually almost all, research activities are — highly dextrous — experimentations. For a natural science, this situation may not seem to be an issue. However, this is mostly associated to a belief that experiments and theoretical thinking could be decoupled, and that experiments could actually be performed independently from theories. Yet, "concrete" experimentations cannot be conceived as autonomous with respect to theoretical considerations, which may have abstract means but also have very practical implications. In the field of

[*]Centre Cavaillès, République des Savoirs, CNRS, Collège de France et Ecole Normale Supérieure, Paris, and Department of Integrative Physiology and Pathobiology, Tufts University School of Medicine, Boston. http://www.di.ens.fr/users/longo/

[†]IHPST, CNRS and université Paris I, Paris. M. Montévil's Work is supported by région Île-de-France, DIM ISC

molecular biology, for example, research is related to the finding of hypothesized molecules and molecular manipulations that would allow to understand biological phenomena and solve medical or other socially relevant problems. This experimental work can be carried on almost forever as biological molecular diversity is abundant. However, the understanding of the actual phenomena, beyond the differences induced by local molecular transformations is limited, precisely because such an understanding requires a theory, relating, in this case, the molecular level to the phenotype and the organism. In some cases, the argued theoretical frame is provided by the reference to an unspecified "information theoretical encoding", used as a metaphor more than as an actual scientific notion (Fox Keller 1995; Longo et al. 2012a). This metaphor is used to legitimate observed correlations between molecular differential manipulations and phenotype changes, but it does so by putting aside considerable aspects of the phenomena under study. For example, there is a gap between a gene that is experimentally necessary to obtain a given shape in a strain and actually entailing this shape. In order to justify this "entailment", genes are argued to correspond to "code", that is a one-dimensional discrete structure, meanwhile shapes are the result of a constitutive history in space and time: the explanatory and conceptual gap between the two is enormous. In our opinion, the absence or even the avoidance of theoretical thinking leads to the acceptance of the naive or common sense theory, possibly based on unspecified metaphors, which is generally insufficient for satisfactory explanations or even false — when it is well defined enough as to be proven false.

We can then informally describe the reasons for the need of new theoretical perspectives in biology as follows. First, there are empirical, theoretical and conceptual *instabilities* in current biological knowledge. This can be exemplified by the notion of the gene and its various and changing meanings (Fox Keller 2002), or the unstable historical dynamics of research fields in molecular biology (Lazebnik 2002). In both cases, the reliability and the meaning of research results is at risk. Another issue is that the molecular level does not accommodate phenomena that occur typically at other *levels of organization*. We propose many examples in Longo and Montévil (2014), but let's quote as for now the work on microtubules (Karsenti 2008), on cancer at the level of tissues (Sonnenschein and Soto 2000), or on cardiac functions at its different levels (Noble 2010). Some authors also emphasize the historical and conceptual shifts that have led to the current methodological and theoretical situation of molecular biology, which is, therefore, subject to ever changing interpretations (Amzallag 2002; Stewart 2004). In general, when considering the molecular level, the problem of the composition, that is the putting together, of a great variety of molecular phenomena arises. Single molecule phenomena may be biologically irrelevant *per se*: they need to be related to other levels of organization (tissue, organ, organism, ...) in order to understand their possible biological significance.

In no way do we mean to negate that DNA and the molecular cascades related to it play a fundamental role, yet their investigations are far from *complete* regarding the description of life phenomena. Indeed, these cascades may causally depend on activities and organization at different level of analysis, which interact with them and in particular shape them and deserve proper insights.

Thus, it seems that, with respect to explicit theoretical frames in biology, the situation is not particularly satisfying, and this can be explained by the complexity of the phenomena of life. Theoretical approaches in biology are numerous and extremely diverse in comparison, say, with the situation in theoretical physics. In the latter discipline, theorizing has a deep methodological unity, even when there exists no unified theory to understand different classes of phenomena — typically, the Relativistic and Quantum Fields are not (yet) unified (Weinberg 1995; Bailly and Longo 2011). A key component of this methodological unity, in physics, is given by the role of "symmetries", which we will extensively stress. Biological theories instead range from conceptual frameworks to highly mathematized physical approaches, the latter mostly dealing with *local* properties of biological systems (e. g. organ shape). The most prominent conceptual theories are Darwin's approach to evolution — its principles, "descent with modification" and "selection", shed a major light on the dynamics of phylogenesis, the theory of common descent — all current organisms are the descendants of one or a few simple organisms, and cell theory — all organisms have a single cell life stage and are cells, or are composed of cells. It would be too long to quote work in the second and third group: they mostly deal with the dynamics of forms of organs (morphogenesis), cellular networks of all sorts, dynamics of populations . . . when needed, we will refer to specific analyses. Very often, this relevant mathematical work is identified as "theoretical biology", while we care for a distinction, in biology, between "theory" and "mathematics" analogous to the one in physics between theoretical physics and mathematical physics: the latter mostly or more completely formalizes and technically solves problems (equations, typically), as set up within or by theoretical proposals or directly derived from empirical data.

In our view, there is currently no satisfactory *theory* of biological organization as such, and in particular, in spite of many attempts, there is no theory of the organism. Darwin's theory, and even more so neo-Darwinian approaches, basically avoid as much as possible the problem raised by the organism. Darwin uses the duality between life and death as natural selection to understand why, between given biological forms, some are observed and others are not. That is, he gave us a remarkable theoretical frame for phylogenesis, without confronting the issue of what a theory of organisms could be. In the modern synthesis, since Fisher (1930), the properties of organisms and phenotypes, fitness in particular, are predetermined and defined, in principle, by genetics (hints to this view may be found already in Spencer's approach to evolution (Stiegler 2001)). In modern terms, "(potential) fitness is already encoded in genes". Thus, the "structure of determination" of organisms is assumed to be theoretically unnecessary and is not approached[1].

In physiology or developmental biology the question of the structure of determination of the system is often approached on qualitative grounds and the mathematical descriptions are usually limited to specific aspects of organs or tissues. Major examples are provided by the well established and relevant work in morphogenesis, since Turing, Thom and many others (see Jean (1994) for phillotaxis and Fleury (2009) for recent work on organogenesis), in a biophysical perspective. In cellular biology, the equivalent situation leads to (bio-)physical approaches to specific biological structures such as membranes, microtubules, ..., as hinted above. On the contrary, the tentative, possibly mathematical, approaches that aim to understand the proper structure of determination of organisms as a whole, are mostly based on ideas such as autonomy and autopoiesis, see for example Rosen (2005); Varela (1979); Moreno and Mossio (2013). These ideas are philosophically very relevant and help to understand the structure of the organization of biological entities. However, they usually do not have a clear connection with experimental biology, and some of them mostly focus on the question of the definition of life and, possibly, of its origin, which is not our aim. Moreover, their relationship with the aforementioned biophysical and mathematical approaches is generally not made explicit. In a sense, our specific “perspectives" on the organism as a whole (time, criticality, anti-entropy, the main themes of our book (Longo and Montévil 2014)) may be used to fill the gap, as on one side we try to ground them on some empirical work, on the other they may provide a theoretical frame relating the global analysis of organisms as autopoietic entities and the local analysis developed in biophysics.

In this context, physiology and developmental biology (and the study of related pathological aspects) are in a particularly interesting situation. These fields are directly confronted with empirical work and with the complexity of biological phenomena; recent methodological changes have been proposed and are usually described as “systems biology”. These changes consist, briefly, in focusing on the systemic properties of biological objects instead of trying to reconstruct macroscopic properties from their components, see Noble (2006, 2011); Sonnenschein and Soto (1999) and, in particular, Noble (2008). In the latter, it is acknowledged that, as for theories in systems biology:

> There are many more to be discovered; a genuine “theory of biology” does not yet exist. *(Noble 2008)*

Systems biology has been recently and extensively developed, but it also corresponds to a long tradition. The aim of our book (Longo and Montévil 2014)

[1]By the general notion of structure of determination we refer to the theoretical determination provided by a conceptual frame, in more or less formalized terms. In physics, this determination is generally expressed by systems of equations or by functions describing the dynamics.

can be understood as a theoretical contribution to this research program. That is, we aim at a preliminary, yet possibly general theory of biological objects and their dynamics, by focusing on “perspectives” that shed some light on the unity of organisms from a specific point of view.

In this project, there are numerous pitfalls that should be avoided. In particular, the relation with the powerful physical theories is a recurring issue. In order to clarify the relationships between physics, mathematics and biology, a critical approach to the very foundations of physical theories and, more generally, to the relation between mathematized theories and natural phenomena is most helpful and we think even necessary. This analysis is at the core of Bailly and Longo (2011) and, in the rest of this text, we just review some of the key points in our approach. By this, we provide below a brief account of the philosophical background and of the methodology that we follow in Longo and Montévil (2014). We also discuss some elements of comparison with other theoretical approaches and then summarize some of the key ideas of our approach.

Physical theorizing guides our attempts in biology, without reductions to the “objects” of physics, but by a permanent reference, even by local reductions, to the *methodology* of physics. We are aware of the historical contingency of this method, yet by making explicit its working principles, we aim at its strongest possible conceptual stability and *adaptability*: “perturbing” our principles and even our methods may allow further progress in knowledge construction.

Our “perspectives” on organisms complement Luca Cardelli’s contributions, largely based on molecular analyses. Yet, links may be established with his more “systemic” approaches, as beautifully developed in the Brane Calculi, Stochastic Gene Networks and Process Algebra Models.

2 Objectivization and Theories

As already stressed, theories are conceptual and — in physics — largely mathematized frameworks that frame the intelligibility of natural phenomena.

One of the most difficult theoretical tasks in biology is to insert the autonomy of the organism in the unavoidable ecosystem, both internal and external: life is variability *and* constraints, and neither make sense without the other. In this sense, the recent exploration in Moreno and Mossio (2013); Montévil and Mossio (2014) relates constraints and autonomy in an original way and complements our effort. Both this “perspective" and ours are only possible when accessing living organisms in their unity and by taking this “wholeness” as a “condition of possibility" for the construction of biological knowledge. However, we do not discuss here this unity *per se*, nor directly analyze its auto-organizing structural stability. In this sense, these two complementary approaches may enrich each other and produce, by future work, a novel integrated framework.

As for the interplay with physics, our approach particularly s the *praxis* underlying scientific theorizing, including mathematical reasoning, as well as the cognitive resources mobilized and refined in the process of knowledge construction. From this perspective, mathematics and mathematized theories, in particular, are the result of human activities, in our historical space of humanity (Husserl 1970). Yet, they are the most stable and conceptually invariant knowledge constructions we have ever produced. This singles them out from the other forms of knowledge. In particular, they are grounded on the constituted *invariants of our action*, gestures and language, and on the *transformations* that preserve them: the concept of number is an invariant of counting and ordering; symmetries are fundamental cognitive invariants and transformations of action and vision — made concepts by language, through history (Dehaene 1997; Longo and Viarouge 2010). More precisely, both ordering (the result of an action in space) and symmetries may be viewed as "principles of conceptual construction" and result from core cognitive activities, shared by all humans, well before language, yet spelled out in language. Thus, jointly to the "principles of (formal) proof", that is to (formalized) deductive methods, the principles of construction ground mathematics at the conjunction of action and language. And this is so beginning with the constructions by rotations and translations in Euclid's geometry (which are symmetries) and the axiomatic-deductive structure of Euclid's proofs (with their proof principles).

This distinction, construction principles vs. proof principles, is at the core of the analysis in Bailly and Longo (2011), which begins by comparing the situation in mathematics with the foundations of physics. The observation is that mathematics and physics share the same construction principles, which were largely co-constituted, at least since Galileo and Newton up to Noether and Weyl, in the XXth century[2]. One may formalize the role of symmetries and orders by the key notion of group. Mathematical groups correspond to symmetries, while semi-groups correspond to various forms of ordering. Groups and semi-groups provide, by this, the mathematical counterpart of some fundamental cognitive grounds for our conceptual constructions, shared by mathematics and physics: the active gestures which organize the world in space and time, by symmetries and orders.

Yet, mathematics and physics differ as for the principles of proof: these are the (possibly formalized) principles of deduction in mathematics, while proofs need to be grounded on experiments and empirical verification, in physics. What can we say as for biology? On one side, "empirical evidence" is at the

[2]Archimedes should be quoted as well: why is a balance with equal weights at equilibrium? for symmetry reasons, says he. This is how physicists still argue now: why does that particle exist? for symmetry reasons — see the case of anti-matter and the negative solution of Dirac's equations (Dirac 1928).

core of its proofs, as in any science of nature, yet mathematical invariance and its transformations do not seem to be sufficiently robust and general as to construct biological knowledge, at least not at the level of organisms and their dynamics, where variability is one of the major "invariants". So, biology and physics share the principles of proofs, in a broad sense, while we claim that the principles of conceptual constructions cannot be transferred as such. The aim of Longo and Montévil (2014) is to highlight and apply some cases where this can be done, by some major changes though, and other cases where one needs radically different insights, from those proper to the so beautifully and extensively mathematized theories of the inert.

It should be clear by now, that our foundational perspective concerns as a priority the methodology (and the practice) that allows the establishment of scientific objectivity in our theories of nature. As a matter of fact, in our views, the constitution of theoretical thinking is at the same time a process of objectivization. That is, this very process co-constitutes the object of study, jointly to the empirical evidence, in a way that simultaneously allows its intelligibility. The case of quantum mechanics is paradigmatic for us, as a quanton (and even its reference system) is the result of active measurement and its practical and theoretical preparation. In this perspective, then, the objects are defined by measuring and theorizing that simultaneously give their intelligibility, while the validity of the theory (the proofs, in a sense) is given by further experiments. Thus, in quantum physics, measurement has a particular status, since it is not only the access to an object that would be there beyond and before measurement, but it contributes to the constitution of the very object measured. More generally, in natural sciences, measurement deals with the questions: where to look, how to measure, where to set boundaries to objects and phenomena, which correlations to check and even propose .... This co-constitution can be intrinsic to some theories such as quantum mechanics, but a discussion seems crucial to us also in biology, see Montévil (2014).

Following this line of reasoning, the research program we follow towards a theory of organisms aims at finding ways to constitute theoretically biological objects and objectivize their behavior. Differences and analogies, by conceptual continuities or dualities with physics will be at the core of our method (as for dualities, see, for example, our understanding of "genericity vs. specificity" in physics vs. biology in Longo and Montévil (2011, 2014)), while the correlations with other theories can, perhaps, be understood later[3]. In this context, thus, a certain number of problems in the philosophy of biology are not methodological barriers; on the contrary, they may provide new links between remote theorizing such as physical and social ones, which would not be based on the transfer of already constructed mathematical models.

[3]The "adjacent" fields are, following Bailly (1991), physical theories in one direction and social sciences in another. The underlying notion of "extended criticality", may prove to be useful in economics, since we seem to be always in a permanent, extended, crisis or critical transition, very far from economic equilibria.

3 A short synthesis of our approach to biological phenomena

A methodological point that we first want to emphasize is that we focus on "current" organisms, as a result of the process of biological evolution. Indeed, the question of the origin of life is a very active field of research. In this field, most of these analyses use physical or almost physical theories as such, that is they try to analyze how, from a mix of (existing) physical theories, one can obtain "organic" or evolutive systems. We will not work at the (interesting, *per se*) problem of the origin of life, as the transition from the inert to the living state of matter, but we will work at the transition from *theories* of the inert to *theories* of living objects. In a sense this may contribute also to the "origin" problem, as a sound theory of organisms, if any, may help to specify what the transition from the inert leads to, and therefore what it requires.

More precisely, the method of mathematical biology and biophysical modeling quoted above is usually the transformation of *a part* of an organism (more generally, of a living system) into a physical system, in general separated from the organism and from the biological context it belongs to. This methodology often allows an understanding of some biological phenomena, from morphogenesis (phyllotaxis, formation of some organs . . .) to cellular networks and more, see above. For example, the modeling of microtubules allows to approach their self-organization properties (Karsenti 2008), but it corresponds to a theoretical (and experimental) *in vitro* situation, and their relation with the cell is not understood by the physical approach alone. The understanding of the system in the cell requires an approach external to the structure of determination at play in the purely physical modeling. Thus, to this technically difficult work ranging from morphogenesis and phyllotaxis to cellular networks, one should add an insufficiently analyzed issue: these organs or nets, whose shape and dynamics are investigated by physical tools, are generally part of an organism. That is, they are regulated and integrated in and by the organism and never develop like isolated or generic (completely defined by invariant rules) crystals or physical forms. It is instead this integration and regulation in the coherent structure of an organism that contributes in making the biologically relevant situations, which is often non-generic in the physical sense (Lesne and Victor 2006).

The general strategy we use for our investigations in theoretical biology, is to approach the biological phenomena from different perspectives, each of them focusing on different *aspects* of biological organization, not on different *parts* such as organs or cellular nets in tissues . . . . The aim is to propose a basis for a partially mathematized theoretical understanding. This strategy allows us to obtain relatively autonomous progresses on the corresponding aspects of living systems. An essential difficulty is that, *in fine*, these concepts are fully meaningful only in the interaction with each other, that is to say in a

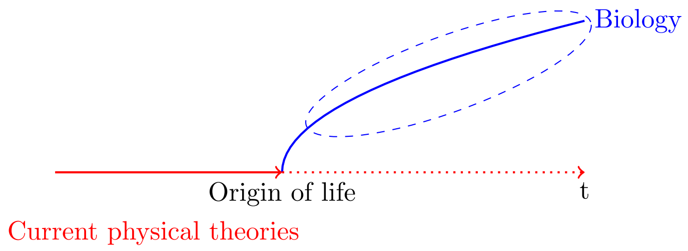


Figure 1: *A scheme of the relation between physics and biology, from a diachronic point of view.* Theoretical approaches that focus on the origin of life usually follow the physical line (stay within existing physical theories) and try to approach the "bifurcation" point. The latter is not well defined since there is no proper theory for the biological entities that are assumed to emerge. Usually, the necessary ingredients for Darwinian evolution are used as goals to be obtained from physical systems. From our perspective, a proper understanding of biological phenomena needs to focus directly, at least as a first (huge) step, on the properly biological domain, where the Darwinian tools soundly apply, but also where organisms are constituted. It may then be easier to fill the gap.

unified framework that we are contributing to establish. In this sense, then, we are making progresses by revolving around this not yet existing framework, proposing and browsing these different perspectives in the process. However, this allows a stronger relation to empirical work, in contrast to theories of biological autonomy, without losing the sense of the biological unity of an organism.

The method we follow in order to progress in each of these specific aspects of life phenomena can mostly be understood as taking different points of view on organisms: we look at them from the point of view of time and rhythms, of the interplay of global stability vs. instability, of the formation and maintenance of organization through changes .... As a result, we combine in Longo and Montévil (2014) a few of these theoretical perspectives, for which the principal common organizing concepts are biological time, on one side, and extended criticality on the other. More specifically, the main conceptual frames that we either follow directly or that make recurrent appearance in this text are the following:

Biological temporal organization The idea is that, more than space or especially energy, biological time is a at the center of biological organization. This does not mean that energy is irrelevant, but both time and energy have a different role from the one they play in physics. The reasons for this are explained throughout Longo and Montévil (2014). The approach in terms of symmetry changes that we develop provides a radical argument for this point of view. Intuitively, the idea is that what matters in biological theorizing is the notion of "organization" and the way it is constructed along and, we dare to say, *by* time, since biological time will be an operator for us, in a precise mathematical sense. In contrast to this, the energetic level (say, between mammals of different sizes) is relatively contingent, as supported by the allometric relations,reviewed in the second chapter of Longo and Montévil (2014), where energy or mass appear as a parameter. Some preliminary arguments from physics are provided by the role of time (entropy production) in dissipative structures (Nicolis and Prigogine 1977) and by the non-ergodicity of the molecular phase space, discussed in Kauffman (2002); Longo et al. (2012b).

Extended critical transitions A large part of our work uses the notion of extended critical transition (Bailly 1991; Bailly and Longo 2008, 2011; Longo and Montévil 2011) to understand biological systems. This notion is relatively complex, in particular because of its physical prerequisites. It is discussed at length, with these prerequisites, in Longo and Montévil (2014). Note that it provides a precise meaning to the idea of the physical singularity of life phenomena in the sense that the biological is approached as a limit case of a physical situation.

Enablement Biologists working on evolution often refer to a contingent state of the ecosystem as "enabling" a given form of life. A niche, typically, enables a, possibly new, organism; yet, a niche may be also constructed by an organism Pocheville (2010). In Longo et al. (2012b) and Longo and Montévil (2013) an attempt is made to frame this informal notion in a rigorous context. We borrow here from that work to link enablement to the role of symmetry changes and we provide by this a further conceptual transition from physics to biology.

Anti-entropy This notion aims to quantify the "amount of biological organization" of an organism (Bailly and Longo 2009; Longo and Montévil 2012) as a non-reducible opposite of entropy. It also determines some temporal aspects of biological organization. This aspect of our investigation gives a major role to randomness. The notion of randomness is related to entropy and to the irreversibility of time in thermodynamics and statistical mechanics. As a result, we consider a proper notion of biological randomness as related to anti-entropy, to be added on top of the many (at least three) forms of randomness present in physical theories (classical, thermodynamical, quantum).

Various physical theories (classical, relativistic, quantum, thermodynamic) make the inert intelligible in a remarkable way. Significant incompatibilities exist (the relativistic and quantum fields are not unified; they are in fact incompatible). However, some major principles of conceptual construction confer a great unity to contemporary theoretical physics. The geodesic principle and its accompaniment by "symmetries" (Weyl 1983; Van Fraassen 1989; Bailly and Longo 2011), enable to grasp, under a conceptually unitary perspective, a wide area of knowledge regarding the inert. Biology, having to date been less "theorized" and mathematized, can also progress in the construction of its theoretical frameworks by means of analogies, extensions and differentiations regarding physical theories, even by means of conceptual dualities. Regarding dualities, we recall here one that is, we believe, fundamental and that has been extensively addressed in Bailly and Longo (2011); Frezza and Longo (2010); Longo and Montévil (2011, 2014)): the *genericity* of physical objects (that is, their theoretical and experimental invariance) and the *specificity* of their trajectories (basically, their reconstruction by means of the geodesic principle or identification by mathematical techniques, by symmetries typically). In our perspective, this is inverted in biology, as it is transformed into the *specificity* (individuation and history) of the living object and the *genericity* of trajectories (evolutionary, ontogenetic: they are just "possibilities" within spaces — ecosystems — in co-constitution). As a result, the work of theorization differs strongly between biology and physics.

References

G. N. Amzallag. *La raison malmenée. De l'origine des idées reçues en biologie moderne.* CNRS édition, 2002.

F. Bailly. L'anneau des disciplines. *Revue Internationale de Systémique*, 5(3), 1991.

F. Bailly and G. Longo. Extended critical situations: the physical singularity of life phenomena. *Journal of Biological Systems*, 16(2):309, 2008. doi: 10.1142/S0218339008002514.

F. Bailly and G. Longo. Biological organization and anti-entropy. *Journal of Biological Systems*, 17(1):63–96, 2009. doi: 10.1142/S0218339009002715.

F. Bailly and G. Longo. *Mathematics and the natural sciences; The Physical Singularity of Life.* Imperial College Press, London, 2011. Preliminary version in French: Hermann, Vision des sciences, 2006.

S. Dehaene. *The number sense.* Oxford University Press, 1997.

P. A. M. Dirac. The quantum theory of the electron. *Proceedings of the Royal Society of London. Series A*, 117(778):610–624, 1928. doi: 10.1098/rspa.1928.0023. URL `http://rspa.royalsocietypublishing.org/content/117/778/610.short`.

R. Fisher. *The Genetical Theory of Natural Selection.* Clarendon, 1930.

V. Fleury. Clarifying tetrapod embryogenesis, a physicist's point of view. *The European Physical Journal Applied Physics*, 45, 2 2009. ISSN 1286-0050. doi: 10.1051/epjap/2009033. URL `http://www.epjap.org/article_S1286004209000330`.

E. Fox Keller. *Refiguring Life: Metaphors of Twentieth-Century Biology.* Columbia University Press, 1995.

E. Fox Keller. *The century of the gene.* Harvard University Press, 2002.

G. Frezza and G. Longo. Variations on the theme of invariants: conceptual and mathematical dualities in physics vs biology. *Human Evolution*, 25(3-4): 167–172, 2010.

E. Husserl. *The crisis of european sciences and transcendental phenomenology: an introduction to phenomenological philosophy*, chapter Origin of geometry. Northwestern University Press, Evanston, Illinois, 1970.

R. Jean. *Phyllotaxis: A Systemic Study in Plant Morphogenesis.* Cambridge Studies in Mathematics, 1994.

E. Karsenti. Self-organization in cell biology: a brief history. *Nature Reviews Molecular Cell Biology*, 9(3):255–262, 2008. doi: 10.1038/nrm2357.

S. Kauffman. *Investigations.* Oxford University Press, USA, 2002.

Y. Lazebnik. Can a biologist fix a radio? or, what i learned while studying apoptosis. *Cancer Cell*, 2(3):179 – 182, 2002. doi: 10.1007/s10541-005-0088-1.

A. Lesne and J.-M. Victor. Chromatin fiber functional organization: Some plausible models. *Eur Phys J E Soft Matter*, 19(3):279–290, 2006. doi: 10.1140/epje/i2005-10050-6.

G. Longo and M. Montévil. From physics to biology by extending criticality and symmetry breakings. *Progress in Biophysics and Molecular Biology*, 106 (2):340 – 347, 2011. ISSN 0079-6107. doi: 10.1016/j.pbiomolbio.2011.03.005. Invited paper, special issue: Systems Biology and Cancer.

G. Longo and M. Montévil. Randomness increases order in biological evolution. In M. Dinneen, B. Khoussainov, and A. Nies, editors, *Computation, Physics and Beyond*, volume 7160 of *Lecture Notes in Computer Science*, pages 289 – 308. Springer Berlin / Heidelberg, 2012. ISBN 978-3-642-27653-8. doi: 10.1007/978-3-642-27654-5_22. Invited paper, Auckland, New Zealand, February 21-24, 2012.

G. Longo and M. Montévil. Extended criticality, phase spaces and enablement in biology. *Chaos, Solitons & Fractals*, 55(0):64 – 79, 2013. ISSN 0960-0779.

doi: 10.1016/j.chaos.2013.03.008. URL `http://www.sciencedirect.com/science/article/pii/S0960077913000489`. Invited Paper, Special Issue.

G. Longo and M. Montévil. *Perspectives on Organisms: Biological time, symmetries and singularities.* Lecture Notes in Morphogenesis. Springer, 2014. ISBN 978-3-642-35937-8. doi: 10.1007/978-3-642-35938-5.

G. Longo and A. Viarouge. Mathematical intuition and the cognitive roots of mathematical concepts. *Topoi*, 29(1):15–27, 2010. Special issue on Mathematical knowledge: Intuition, visualization, and understanding (Horsten L., Starikova I., eds).

G. Longo, P.-A. Miquel, C. Sonnenschein, and A. M. Soto. Is information a proper observable for biological organization? *Progress in Biophysics and Molecular biology*, 109(3):108 – 114, 2012a. ISSN 0079-6107. doi: 10.1016/j.pbiomolbio.2012.06.004.

G. Longo, M. Montévil, and S. Kauffman. No entailing laws, but enablement in the evolution of the biosphere. In *Genetic and Evolutionary Computation Conference*, New York, NY, USA, July 7-11 2012b. GECCO'12, ACM. doi: 10.1145/2330784.2330946. Invited Paper.

M. Montévil. Biological measurement and systems biology. *to be submitted*, 2014.

M. Montévil and M. Mossio. Closure of constraints in biological organisation. 2014. To be submitted.

A. Moreno and M. Mossio. *Biological autonomy. A Philosophical and Theoretical Enquiry.* Springer, Dordrecht, 2013.

G. Nicolis and I. Prigogine. *Self-organization in non-equilibrium systems.* Wiley, New York, 1977.

D. Noble. *The music of life.* Oxford U. P., Oxford, 2006.

D. Noble. Claude bernard, the first systems biologist, and the future of physiology. *Experimental Physiology*, 93(1):16–26, 2008. doi: 10.1113/expphysiol.2007.038695. URL `http://ep.physoc.org/content/93/1/16.abstract`.

D. Noble. Biophysics and systems biology. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 368 (1914):1125, 2010. doi: 10.1098/rsta.2009.0245.

D. Noble. *The music of life: sourcebook.* Oxford, 2011.

A. Pocheville. *What niche construction is (not).* 2010. URL `http://hal.upmc.fr/tel-00715471/`.

R. Rosen. *Life itself: a comprehensive inquiry into the nature, origin, and fabrication of life.* Columbia U. P., 2005.

C. Sonnenschein and A. Soto. *The society of cells: cancer and control of cell proliferation.* Springer Verlag, New York, 1999.

C. Sonnenschein and A. Soto. Somatic mutation theory of carcinogenesis: why it should be dropped and replaced. *Molecular Carcinogenesis*, 29(4): 205–211, 2000. doi: 10.1002/1098-2744(200012)29:4<205::AID-MC1002>3.0.CO;2-W.

J. Stewart. *La vie existe-t-elle ?* Editions Vuibert, 2004.

B. Stiegler. *Nietzsche et la biologie.* Puf, Paris, 2001.

B. Van Fraassen. *Laws and symmetry.* Oxford University Press, USA, 1989.

F. Varela. *Principles of biological autonomy.* North Holland New York, 1979.

S. Weinberg. *The Quantum Theory of Fields.* Cambridge University Press, 1995.

H. Weyl. *Symmetry.* Princeton Univ Pr, 1983.

Luca Cardelli and the Early Evolution of ML

David MacQueen

Abstract

Luca Cardelli has made an enormous range of contributions, but the focus of this paper is the beginning of his career and, in particular, his role in the early development of ML. He saw the potential of ML as a general purpose language and was the first to implement a free-standing compiler for ML. He also made some important innovations in the ML language design which had a major influence on the design of Standard ML, in which he was an active participant. My goal is to examine this early work in some detail and explain its impact on Standard ML.

1 Introduction

My goal here is to tell the story of the early days of ML as it emerged from the LCF system via Luca Cardelli's efforts to create a general purpose version of ML, called VAX ML. Starting in 1983, the new ideas developed in VAX ML and the HOPE functional language inspired Robin Milner to begin a new language design project, Standard ML, and for the next two years Luca was an essential contributor to that effort.

2 VAX ML

We will start by giving a brief history of Luca's ML compiler, before considering the language innovations it introduced and how the language evolved in Section 2.1 below.[1]

Luca began working on his own ML compiler sometime in 1980. The compiler was developed on the Edinburgh Department of Computer Science VAX/VMS system, so Luca called it "VAX ML" to distinguish from "DEC-10 ML", the

[1]Note that Luca was building his VAX ML at the same time as he was doing research for his PhD thesis (Cardelli 1982b). He also developed his own text formatting software, inspired by Scribe, that he used to produce his thesis and the compiler documentation, and a simulation of the solar system!

LCF version of ML[2] which ran under Stanford Lisp on the DEC-10/TOPS-10 mainframe.[3] Luca's preferred working language at the time was Pascal, so both the compiler and the runtime system were written in Pascal, using Pascal's unsafe union type to do coercions for low-level programming (e.g. for the garbage collector).[4] The compiler generated VAX machine code, and was much faster than the DEC-10 (LCF) version, which used the host Lisp interpreter to execute translated ML code. The VAX ML compiler was working and had begun to be distributed to users by the summer of 1981 (version 12-6-81, i.e. 12 June 81), although a working garbage collector was not added until version 13-10-81.

The earliest surviving documents relating to the compiler date to late 1980: "The ML Abstract Machine", a description of the abstract machine AM (Cardelli 1980a), (which would develop into the FAM (Cardelli 1983a)), and "A Module Exchange Format", a description of an external string format for exporting ML runtime data structures (Cardelli 1980b). There is a README file titled "Edinburgh ML" from March, 1982 that describes how to install and run the system (Cardelli 1982a), and a partial manual titled "ML under VMS" providing a tutorial introduction to the language (Cardelli 1982d), corresponding to Section 2.1 of "Edinburgh ML" (Gordon et al. 1979).

In early 1982, Nobuo Saito, then a postdoc at CMU, ported VAX ML to Unix, using Berkeley Pascal (Saito 1982). In April, 1982, Luca completed his PhD at Edinburgh (Cardelli 1982b) and moved to the Computing Science Research Center (the home of Unix) at Bell Labs, and immediately began his own Unix port, which was available for distribution in August, 1982. The runtime system for the Unix port was rewritten in C, but most of the compiler itself remained in Pascal. The first edition of the Polymorphism newsletter (Volume I, Number 0) contained a list of known distribution sites (Cardelli 1982c) in November 1982; at that time, there were at least 23 sites spread around the world, several using the new Unix port. The Unix port had three releases during 1982 (13-8-82, 24-8-82, and 5-11-82), accompanied with some shifts in language design and system features, notably a new type checker for ref types and an early version of file I/O primitives.

The next major milestone was the first Standard ML meeting in Edinburgh in April, 1983 (See Section 3). Luca agreed to a request from Robin Milner to suspend work on his VAX ML manual pending developments around Robin's initial proposal for Standard ML (Milner 1983a). Following the meeting Luca began to change his compiler to include new features of the emerging Standard

[2]Commonly called LCF/ML

[3]Luca referred to these implementations as two varieties of "Edinburgh ML".

[4]Since the entire system was written in Pascal, there was no sharp distinction between the compiler and the runtime, which was simply the part of the system responsible for executing the abstract machine instructions (FAM code).

ML design, resulting in Pose[5] 2 (August 1983), Pose 3 (November 1983), and finally Pose 4 (April 1984). This last version is described in the paper "Compiling a Functional Language" in the 1984 Lisp and Functional Programming conference (Cardelli 1984a).

2.1 Language Innovations

The first description of the language of VAX ML was a file mlchanges.doc (Cardelli 1981) that was part of the system distribution. This file describes the language by listing the changes made relative to DEC-10 ML (i.e. LCF/ML). The changes include a number of minor notational shifts. For instance, LCF/ML used "`.`" for list cons, while VAX ML initially used "`_`", later shifting to the "`::`" used in POP-2 (Burstall and Popplestone 1968). The trivial type (called "`unit`" in Standard ML) was denoted by "`.`" in LCF/ML and by "`triv`" in VAX ML. A number of features of LCF/ML were omitted from VAX ML, e.g. the "`do`" operator, "sections", and the "`!`" and "`!!`" looping failure traps (Gordon et al. 1979, Chapter 2).

But the really interesting changes in VAX ML involved (1) new labelled record and union types, (2) the ref type for mutable values, (3) declaration combinators for building compound declarations, and (4) modules. We will describe these in the following subsections.

2.1.1 Labelled records and unions

Luca was of course familiar with the conventional record construct provided in languages like Pascal (Jensen and Wirth 1978, Chapter 7). But inspired by Gordon Plotkin's lectures on domain theory Luca looked for a purer and more abstract notion, where records and discriminated union types were an expression of pure structure, representing themselves without the need of being declared and named. The notation for records used *decorated parentheses*:

```
(|a1 = e1; ... ; an = en|)  :  (|a1: t1; ... ; an: tn|)
```

where `ei` is an expression of type `ti`. The order of the labelled fields in a record type did not matter – any permutation represented the same type.

Accessing the value of a field of a record was done using the conventional dot notation: `r.a`, where `r` is a record and `a` is a label. Records could also be deconstructed in declarations and function arguments by pattern-matching with a record *varstruct* (pattern), as in the declaration:

```
let (|a=x; b=y|) = r
```

[5]Luca called his compiler versions "Poses" adopting a terminology from dance.

From the beginning, Luca included an abbreviation feature for record patterns where a field name could double as a default field variable, so

```
let (|a; b|) = r
```

would introduce and bind variables named `a` and `b` to respective field values in `r`. All these features of the VAX ML record construct eventually carried over to Standard ML, with just a change in the bracket notation to use `{...}`.

Labelled unions were expressed as follows, using decorated square brackets:

```
[|a1 = e1|] : [|a1: t1; ... ; an: tn|]
```

The union type to which a given variant expression belonged had to be determined by the context or given explicitly by a type ascription. Variant varstructs could be used in varstructs for declaration and argument bindings, with their own defaulting abbreviation where a `[|a|]` stood for `[|a = ()|]`, both in varstructs and expressions, which supported an enumeration type style.[6] A case expression based on variant varstructs was used to discriminate on and deconstruct variant values, with the syntax

```
case e
  of [| a1 = v1 . e1;
        ...
        an = vn . en
     |]
```

where `e` is of type `[|a1: t1; ... ; an: tn|]`.

2.1.2 The ref type

In LCF/ML, mutable variables could be declared using the `letref` declaration keyword. In VAX ML, Luca replaced this with the `ref` type operator with its interface of operations `ref`, `:=`, and `!`. This approach was carried over unchanged into Standard ML, though the issue of how `ref` behaved relative to polymorphism took many years to resolve. In 1979, Mike Gordon wrote a brief note "Locations as first class objects in ML" (Gordon 1980) proposing the ref type for ML, with a restricted type discipline using *weak polymorphism* and *weak type variables*. Gordon suggested this as a research topic to Luis Damas, who eventually addressed the problem in his PhD thesis (Damas 1985, Chapter 3) using a rather complex method where typing judgements were decorated by sets of types involved in refs. Luca got the idea to use the ref type either from Gordon's note or via discussions with Damas, with whom he shared an office for a time. At the end of Chapter

[6]Initially, the record and variant abbreviation conventions were also applied to types, but this was not found useful and was quickly dropped.

3 of his thesis, Damas returns to a simpler approach to the problem of refs and polymorphism similar to the weak polymorphism suggested by Gordon, and he mentions that both he and Luca implemented this approach. However, the issue is not mentioned in the various versions of Luca's ML manuals (Cardelli 1982d, 1983c, 1984b), and the `ref` operator is described as having a normal polymorphic type.[7]

2.1.3 Declaration combinators

Another innovation in VAX ML was a set of (more or less) independent and orthogonal declaration combinators for building compound declarations. These are

- `enc`, for sequential composition, equivalent to nesting `lets`: "`d1 enc d2`" yields the bindings of `d1` augmented by or overridden by the bindings of `d2`.
- `and`, for simultaneous or parallel composition, usually used with recursion.
- `ins`, for localized declarations: "`d1 ins d2`" yields the bindings of `d2`, which are evaluated in an environment containing the bindings of `d1`.
- `with`, a kind of hybrid providing the effect of `enc` for type bindings and `ins` for value bindings; usually used with the special "`<=>`" type declaration to implement abstract types.
- `rec`, for recursion

There were also reverse forms of `enc` and `ins` called `ext` and `own` for use in `where` expressions, thus "`let d1 enc d2 in e`" is equivalent to "`e where d2 ext d1`".

This "algebra" of declarations (possibly inspired by ideas in Robert Milne's PhD thesis (Milne 1974)) was very interesting, but in programming the combinators would normally be used in a few limited patterns that would not take advantage of the generality of the idea. Indeed, certain combinations seemed redundant or problematical, such as "`rec rec d`", or "`rec d1 ins d2`" (`rec` syntactically binds weaker than the infix combinators).

Luca factored the `abstype` and `absrectype` of LCF/ML using `with` (and possibly `rec`) in combination of a special type declaration `tname <=> texp` that produced an opaque type binding[8] of `tname` to the type expression `texp` together with value bindings of two isomorphism functions:

[7]The notes at the end of (Cardelli 1982c), mention that the 5-11-82 Unix version has a "New typechecker for ref types." It is not known whether this typechecker used weak polymorphism.

[8]Meaning that `tname` was not equivalent to `texp`.

```
abstname : texp -> tname
reptname : tname -> texp
```

A compound declaration `tname <=> texp with decl` would compose the type binding of `tname` with `decl` while localizing the bindings of `abstname` and `reptname` to `decl`. Thus `with` acted like `enc` at the type level and `ins` at the value level. This in principle was more general than `abstype/absrectype` in LCF/ML, in that the declaration `d1` in `d1 with d2` was arbitrary and not restricted to a possibly recursive simultaneous set of isomorphism (`<=>`) type bindings, but it was not clear whether this greater generality would be exploited.

In the end, the `and` combinator was used in Standard ML, but at the level of value and type bindings, not declarations (just as it was used in LCF/ML), the `ins` combinator became the "`local d in e end`" declaration form, the `rec` combinator was adopted, but at the level of bindings (and still with too general a syntax!), and the `<=>`, `with` combination were replaced by a variant of the old `abstype` declaration, but using the datatype form for the type part and restricting the scope of the associated data constructors.

In later versions, starting with ML under Unix, Pose 2 (Cardelli 1983c), another declaration form using the `export` keyword was added. A declaration of the form

```
export exportlist from decl end
```

produced the bindings of *`decl`*, but restricted to the type and value names listed in the *`exportlist`*. Exported type names could be specified as abstract (in ML under Unix, Pose 4) meaning that constructors associated with the type were not exported. Thus both `local` and `abstype` declarations could be translated into export declarations.

2.1.4 Modules

LCF/ML had no module system; the closest approximation was the `section` directive that could delimit scope in the interactive top-level. Since VAX ML aimed to support general purpose programming, Luca provided a basic module system. A module declaration was a named collection of declarations. Modules were independent of the environment in the interactive system, and required explicit import declarations to access the contents of other modules (other than the standard library of primitive types and operations, which was always accessible); Luca called this *module hierarchy*. Compiling a module definition produced an external file that could be loaded into the interactive system or accessed by other modules using the import declaration. Importing (loading) a module multiple times would only create one copy, so two modules `B` and `C` that both imported a module `A` would share a single copy of `A`. The export declaration was commonly used to restrict the interface provided by a module. There was no way to separately specify interfaces (*signatures* in Standard ML).

3 The Standard ML Design

LCF/ML had great potential as a language, but its scope was severely limited by its context as a tool embedded in the LCF proof assistant. Luca realized its wider potential, which motivated him to create VAX ML. By 1982, both LCF and Luca's VAX ML had created a lot of interest, as had HOPE (Burstall et al. 1980), another functional language developed at Edinburgh by Rod Burstall, Don Sannella and myself. In November, 1982 a meeting was convened at the Rutherford Appleton Laboratory (RAL) to discuss the future of these three systems (Witty and Wadsworth 1982). The meeting was attended by 20 people, including Robin Milner, Rod Burstall, and several others from Edinburgh, John Darlington from Imperial College, and Bernard Sufrin from Oxford. The topics discussed included how the ML and HOPE languages could be supported and further developed (e.g. by creating new ports, etc.). Sometime during or after this meeting, Bernard Suffrin urged Robin to think about creating a new ML design that would consolidate what appeared successful about LCF/ML, VAX ML, and HOPE.

3.1 Meetings

By early April, 1983, Robin had created a hand-written first draft of "A Proposal for Standard ML" (Milner 1983a). By coincidence, both Luca and I were in Edinburgh at that time, as were Robin and Rod Burstall and their research groups (in particular, Kevin Mitchell, Alan Mycroft, and John Scott), and some other visitors (including, I believe, Guy Cousineau from INRIA)[9]. So this was a serendipitous opportunity for many of those most directly interested to immediately get together to discuss Robin's proposal. Many of the discussions took place in Robin's living room at Garscube Terrace. In the period immediately following these meetings, Robin summarized some suggested changes (Milner 1983b) and by June produced a second draft of the proposal (Milner 1983c). We collectively decided on several parallel efforts: Robin would continue to work on the Core language proposal, I would work on a module system,[10] and Luca would develop a proposal for stream Input/Output, based on his stream I/O interface in VAX ML. Luca also undertook to modify his VAX ML compiler by adding Standard ML design features and removing some features (records, labelled unions, type

[9]Unfortunately, the records of this first gathering in Edinburgh are spotty, and don't include a list of all the people involved

[10]I had already been working on a module system for HOPE (MacQueen 1981).

identity abbreviations) that did not get included in the Standard ML proposal. Discussion by correspondence and exchange of design proposals continued for the rest of 1983 and into 1984, including another draft of the Core language proposal (Milner 1983d), and Luca's proposal for stream I/O (Cardelli 1983b).

The next year, in early June, a second, more formally organized meeting was held to continue work on the design (MacQueen and Milner 1985). Luca was not able to attend this meeting, but I acted as his representative and he was active in providing comments. This meeting lead to the publication of the fourth draft of the Core proposal and my first Modules proposal in the Lisp and Functional Programming Conference that August (Milner 1984; MacQueen 1984). Luca's paper "Compiling a Functional Language", describing the ML under Unix (Pose 4) compiler, also appeared in that conference (Cardelli 1984a). The meeting report by Robin and myself appeared in December (MacQueen and Milner 1985).

The third meeting, which was called the ML Workshop, took place in May, 1985, and Luca was again able to participate in person. There are extensive reports and design proposals that were presented at the workshop or followed in its aftermath, including a meeting report by Bob Harper (Harper 1985a). There were also two further proposals for the stream I/O library, one by Robin and Kevin Mitchell (Mitchell and Milner 1985), and another by Bob Harper (Harper 1985b). Another revision of the Modules design was published in Polymorphism in October (MacQueen 1985).

From 1985 on, Luca's involvement in Standard ML tapered off because of other interests (see Section 4 below). Meanwhile, between 1985 and 1989, the refinement of the Standard ML design proceeded and work on its formal definition was completed. Counting design documents and versions of the formal definition, there were a total of (at least) 21 design or definition versions produced between April 1983 and the end of 1989, not counting modules and I/O.

3.2 Edinburgh ML

In 1982, while Luca was working on the Unix port of VAX ML at Bell Labs, back in Edinburgh Kevin Mitchell began a project to rewrite the VAX ML compiler in ML (the VAX ML dialect), replacing the Pascal code and thus allowing the compiler to bootstrap itself. John Scott and Alan Mycroft soon joined Kevin on the project. The runtime system of Edinburgh ML was rewritten in VAX assembly language, and then later rewritten in C on Unix based on a FAM bytecode interpreter, which saved significant code space. John Scott rewrote the parser, using an elaborate version of Vaughan Pratt's top-down precedence parser technique

(Pratt 1973)[11], and Alan Mycroft rewrote the type checker. One of the technical challenges of the project was that it was hoped the compiler could be ported to run on a locally developed workstation that had limited memory and no virtual memory.

The resulting new compiler was called Edinburgh ML, and initially it did not modify the language from VAX ML. But after the first Standard ML meeting in April, 1983 Edinburgh ML began to serve as a prototyping testbed for the Standard ML design along side VAX ML. Alan Mycroft left for Cambridge in the fall of 1984, and later Nick Rothwell and Bob Harper (in 1985) joined the effort, and K. V. S. Prasad was also involved at some point. Bob Harper rewrote the type checker again and implemented the module system as it stood at that point. This made Edinburgh a quite close approximation to Standard ML, although internally much of the compiler code remained in the VAX ML dialect, so the compiler had a general abstract syntax into which both VAX ML and Standard ML could be translated. Unfortunately this meant that the module system could not be exploited to organize the structure of the compiler itself, so the code was rather difficult to work with. In March, 1986, when Andrew Appel arrived in Princeton, he and I used the Edinburgh ML compiler to begin developing a new compiler, Standard ML of New Jersey, from scratch in Standard ML. We bootstrapped our compiler in the spring of 1987 just before exhausting certain size/space limitations in the Edinburgh compiler.

4 Beside and Beyond ML

During 1984 and 1985, Luca's effort on Standard ML was diluted as he began to work on other projects, such as the design and implementation of his own language, Amber (Cardelli 1986a,b). He also found time to write one of the most influential papers on type theory of the time, "A semantics of multiple inheritance" (Cardelli 1984c), which introduced the concept of record (width) subtyping.

I should also mention his very valuable expository efforts, which helped to propagate and popularize important ideas in type systems research. These include particularly the papers "Basic Polymorphic Typechecking" (Cardelli 1985), "On Understanding Types, Data Abstraction, and Polymorphism" (Cardelli and

[11]Vaughan Pratt had spent the summer of 1975 in Edinburgh, and had taught everyone his clever, quick hack for writing parsers. Both the LCF/ML parser (Malcolm Newey) and the HOPE parser (MacQueen) were written using his technique, partly because none of us knew about parser generators at that time, and it seemed much easier than writing a recursive descent parser. The drawback was that it was hard to use symbols for more than one purpose, hence the ".", ";", ";;" separators!

Wegner 1985), and "Typeful Programming" (Cardelli 1989).

In the fall of 1985 Luca moved from Bell Labs to DEC SRC in California, where his language research did not slow down at all! During the next decade he developed Quest, Fsub, and Obliq, and collaborated on the design of Modula 3. He also became a major player in research on type systems for object-oriented languages, and participated in the ML2000 discussions. But I will leave the story of those accomplishments to other contributors to this celebration.

5 Summary

Luca had the vision to see ML as a successful general purpose language, and he had the energy and talent to pursue this vision by designing and implementing the first free-standing version of ML. He followed that accomplishment up by becoming a key participant in the collective process of designing and prototyping Standard ML.

6 Acknowledgements

I want to thank Luca Cardelli, Bob Harper, Kevin Mitchell, Alan Mycroft, Mike Gordon, and Larry Paulson for useful recollections, helpful answers to my many questions and some bits of useful documentation.

References

R. M. Burstall and R. J. Popplestone. Pop-2 reference manual. In E. Dale and D. Mitchie, editors, *Machine Intelligence 2*, pages 207–46. University of Edinburgh Press, 1968.

R. M. Burstall, D. B. MacQueen, and D. Sannella. HOPE: an experimental applicative language. In *Conference Record of the 1980 Lisp Conference*, pages 136–143, August 1980.

L. Cardelli. The ML abstract machine. Early description of the ML abstract machine (AM), precursor to the FAM., November 1980a.

L. Cardelli. A module exchange format. Description of a textual format for exporting internal ML data structures., December 1980b.

L. Cardelli. Differences between VAX and DEC-10 ML. file mlchanges.pdf, 1981.

L. Cardelli. Edinburgh ML. README file for VAX ML (ML under VMS) distribution., March 1982a.

L. Cardelli. *An Algebraic Approach to Hardware Description and Verification.* PhD thesis, University of Edinburgh, April 1982b.

L. Cardelli. Known vax-ml system locations. *Polymorphism: The ML/LCF/Hope Newsletter*, 1(0), November 1982c.

L. Cardelli. *ML under VMS.* Department of Computer Science, Univ of Edinburgh, 1982d.

L. Cardelli. The Functional Abstract Machine. *Polymorphism: The ML/LCF/Hope Newsletter*, 1(1), January 1983a.

L. Cardelli. Stream input/output. *Polymorphism: The ML/LCF/Hope Newsletter*, 1(3), December 1983b.

L. Cardelli. *ML under Unix.* Bell Labs, August 1983c. Pose 2.

L. Cardelli. Compiling a functional language. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 208–217, New York, NY, USA, 1984a. ACM.

L. Cardelli. *ML under Unix.* Bell Labs, April 1984b. Manual for ML under Unix, Pose 4.

L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *International Symposium on Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 479–504, June 1984c.

L. Cardelli. Basic polymorphic typechecking. *Polymorphism: The ML/LCF/Hope Newsletter*, 2(1), January 1985.

L. Cardelli. Amber. In *Proceedings of the Thirteenth Spring School of the LITP on Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 21–47, 1986a.

L. Cardelli. The amber machine. In *Proceedings of the Thirteenth Spring School of the LITP on Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 48–70, 1986b.

L. Cardelli. Typeful programming. Technical Report 45, Digital Systems Research Center, May 1989.

L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

L. Damas. *Type Assignment in Programming Languages*. PhD thesis, Department of Computer Science, University of Edinburgh, 1985.

M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *LNCS*. Springer-Verlag, New York, 1979.

M. J. C. Gordon. Locations as first class objects in ML. Note to Luis Damas proposing a research topic., 1980.

R. Harper. Report on the Standard ML Meeting, Edinburgh, May 23-25, 1985 (DRAFT). Draft minutes of the May, 1985 Standard ML meeting., 1985a.

R. W. Harper. Standard ML input/output. *Polymorphism: The ML/LCF/Hope Newsletter*, 2(1), January 1985b.

K. Jensen and N. Wirth. *Pascal User Manual and Report*. Springer-Verlag, 2nd edition, 1978.

D. MacQueen. Modules for Standard ML. *Polymorphism: The ML/LCF/Hope Newsletter*, 2(2), October 1985.

D. MacQueen and R. Milner. Report on the Standard ML Meeting, Edinburgh, 6-8 June 1984. *Polymorphism: The ML/LCF/Hope Newsletter*, 2(1), January 1985.

D. B. MacQueen. Structure and parameterization in a typed functional language. In *Proc. 1981 Symposium on Functional Languages and Computer Architecture*, pages 524–538, June 1981. Gothenburg, Sweden.

D. B. MacQueen. Modules for Standard ML. In *Proceedings 1984 ACM Symposium on LISP and Functional Programming*, pages 198–207, August 1984.

R. Milne. *The Formal Semantics of Computer Languages and their Implementations*. PhD thesis, Oxford University, 1974.

R. Milner. A Proposal for Standard ML (TENTATIVE). first manuscript draft of the Standard ML Proposal, April 1983a.

R. Milner. Changes to Proposal for Standard ML. first manuscript draft of the Standard ML proposal, May 1983b.

R. Milner. A Proposal for Standard ML (second draft). second manuscript draft of the Standard ML proposal, June 1983c.

R. Milner. A Proposal for Standard ML. Technical Report CSR-157-83, Dept of Computer Science, Univ of Edinburgh, December 1983d.

R. Milner. A Proposal for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 184–197, New York, NY, USA, 1984. ACM.

K. Mitchell and R. Milner. Proposal for I/O in Standard ML. draft of an I/O proposal, February 1985.

V. R. Pratt. Top down operator precedence. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Langauges*, pages 41–51, 1973.

N. Saito. ML System on Vax Unix. README for Saito's Unix port of VAX ML, March 1982.

R. W. Witty and C. P. Wadsworth. ML, LCF, and HOPE. Record of a meeting about the future of ML, LCF, and HOPE at Rutherford Appleton Laboratory, November 1982.

Tiny Bang: Type Inference and Pattern Matching on Steroids

Pottayil Harisanker Menon   Zachary Palmer
Alexander Rozenshteyn   Scott Smith
The Johns Hopkins University
{pharisa2, zachary.palmer, arozens1, scott}@jhu.edu

Abstract

The ML language and its descendants have proven the power of type inference combined with pattern matching. But the concepts can be taken only so far in those language designs: some declared types are needed, and the systems are not compatible with subtyping and thus object-based styles of programming are challenging.

In this paper we show how an extreme approach may lead to a better language design. We describe Tiny Bang, a core language where all data destruction is expressed via pattern matching. We show how a subtype inference methodology can be used which never requires program annotations but still supports programmer-declared types as interface specifications. These programmer-declared typings are also defined in terms of pattern matching via an extension of pattern matching to higher-order functions.

1 Introduction

Pattern matching and type inference are well-known to be useful programming language constructs. In this paper we show how an extreme application of these concepts may lead to a better language design: we *require* all types be inferred and combine patterns with application to give a universal data destructor. In particular, we describe TinyBang, a core language with full type inference and flexible pattern matching, and show some benefits of this flexibility.

TinyBang's type system is grounded in subtype constraint type theory (Aiken et al. 1994), with a series of improvements to both expression syntax and typing. We briefly outline these features and some of their benefits.

Type-indexed records supporting asymmetric concatenation TinyBang uses type-indexed records: records for which content can be projected based on its type (Shields and Meijer 2001). For example, consider the type-indexed record `{foo = 45; bar = 22; 13}`: the untagged element `13` is implicitly tagged with type `int`, and projecting `int` from this record would yield `13`. Since records are type-indexed, we do not need to distinguish records from non-records; `22`, for example, is a type-indexed record of one (integer) field. Variants are also just a special case of 1-ary records of labeled data, so `‘Some 3` expresses the ML `Some(3)`. Type-indexed records are thus a universal data type and lend themselves to flexible programming patterns in the same spirit as Lisp lists and Smalltalk objects.

TinyBang records support asymmetric concatenation via the `&` operator; informally, `{foo = 45; bar = 22; 13} & {baz = 45; bar = 10; 99}` results in `{foo = 45; bar = 22; baz = 45; 13}` since the left side is given priority for the overlap. Asymmetric concatenation is key for supporting flexible object concatenation, as well as for standard notions of inheritance. We term the `&` operation *onioning*.

Dependently typed first-class cases TinyBang's first-class functions are written "*pattern* `->` *expression*". In this way, first-class functions are also first-class *case clauses*. We permit the concatenation of these clauses via `&` to give multiple dispatch possibilities. TinyBang's first-class functions generalize the first-class cases of Blume et al. (2006).

In standard type systems, all case branches are constrained to have the same result type, losing the dependency between the variant input and the output. This problem is solved in TinyBang by giving compound functions dependent types: application of a compound function can return a different type based on the variant constructor of its argument. Additionally, we define a novel notion of *slice* which allows the type of bindings in a case arm to be refined based on which pattern was matched. Dependently typed first-class cases are critical for typing our object encodings, a topic we discuss later in this section.

Object-oriented programming in TinyBang Objects are commonly encoded as records of functions, and method invocation is achieved by invoking a function in the record. While such an encoding could be made to work in TinyBang, it is more complex and so we opt to use a *variant-based* encoding: objects are message-processing functions which case on the form of message, and method invocation is a simple function call. This dual encoding can be challenging to type since the `case` branches (*i.e.*, the different methods) may return different types; the TinyBang type system is however flexible enough to type such a `case`. Additionally, TinyBang's first-class case clauses support object concatenation and thus mixins and subclassing.

A key idea of Bono and Fisher (1998) is a *sealing* transformation where an object template turns into a messageable (but non-extensible) object. Our encoding of objects extends that idea by adding a *resealing* operation: the type of `self` is captured in the closure of a message dispatch function that can be *overridden* due to the asymmetric nature of onioning in TinyBang.

Higher order pattern matching We also extend pattern matching in a new direction: *higher-order function pattern matching*. Function patterns can match a function based on its *behavior*: the pattern `int ~> char`, for instance, matches functions which will accept an `int` argument and return a `char`. This pattern matching check is performed at compile time by the type system, meaning that there is minimal runtime overhead. Note that our notion of higher-order function pattern is not simply matching against a declared or nominal type: it is resursively invoking the typechecker on the function to verify it has the declared type.

There are several potentially important uses of function patterns. First, they support a type assertion language which TinyBang otherwise lacks (the inferred types are constraint sets which are unreadable). This overlay is similar to how dynamic contracts (Findler and Felleisen 2002) are an independent layer of program specification; unlike contracts, function pattern matching is statically verified on all inputs. Second, higher-order function patterns support dynamic dispatch based on what kind of data an input function will be able to process.

2 Programming in TinyBang

This section gives an overview of the TinyBang language. The formal semantics are defined in Smith et al. (2014b).

2.1 Language Features for Flexible Objects

While there are many dimensions of flexibility of TinyBang, we initially focus on object-oriented programming since TinyBang is particularly well-suited to expressing the kinds of flexible object operations found in scripting languages. The TinyBang syntax used in this section appears in Figure 2.1. Operator precedence is as follows: labels (e.g. `‘Foo 0`) have highest parse precedence, onioning (e.g. `‘A 0 & ‘B 0`) has the next highest precedence, and function arrows (e.g. `x -> ‘A x`) have the least precedence.

Program types take the form of a set of subtype constraints (Aiken et al. 1994); for the purposes of this section we will be informal about type syntax. Our formal

$$
\begin{array}{rcll}
e & ::= & x \mid \texttt{()} \mid \mathbb{Z} \mid l\ e \mid \texttt{ref}\ e \mid \texttt{!}\ e \mid e\,\texttt{\&}\,e \mid & \textit{expressions} \\
 & & \phi\ \texttt{->}\ e \mid e \boxdot e \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid x \texttt{ := } e\ \texttt{in}\ e & \\
\phi & ::= & x \mid \texttt{()} \mid \texttt{int} \mid l\ \phi \mid \phi\ \texttt{\~{}>}\ \phi \mid \phi\,\texttt{\&}\,\phi \mid \rho & \textit{patterns} \\
\boxdot & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{==} \mid \texttt{<=} \mid \texttt{>=} & \textit{operators} \\
\rho & ::= & \texttt{'}\,\textit{(alphanumeric)} & \textit{opaque patterns} \\
l & ::= & \texttt{`}\,\textit{(alphanumeric)} & \textit{labels}
\end{array}
$$

Figure 2.1: TinyBang Syntax

type constraint syntax can be viewed as an A-normalized type grammar, and this grammar implicitly supports (positive) union types by giving a type variable multiple lower bounds: for example, `int` $\cup$ `bool` is equivalently expressed as a type α with constraints `int` $<: \alpha$ and `bool` $<: \alpha$. The details of the type system are presented in (Smith et al. 2014b).

Simple functions as methods We begin by considering the oversimplified case of an object with a single method and no fields or self-awareness. In the variant encoding, such an object is represented by a function which matches on a single case. We write functions as ϕ `->` e, with ϕ being a *pattern* to match against the function's argument. Combining pattern match with function definition is also possible in ML and Haskell, but we go further: there is no need for any `match` syntax in TinyBang since `match` can be encoded as a pattern and its application. We call these one-clause pattern-matching functions *simple functions*. Consider the following object and its invocation:

```
let obj = (`twice x -> x + x) in obj (`twice 4)
```

The syntax `` `twice 4 `` is a label constructor similar to an OCaml polymorphic variant. The simple function `` `twice x -> x + x `` takes a `` `twice `` label argument and binds its contents to the variable `x`. Note that the expression `` `twice 4 `` represents a *first-class message*; the object invocation is represented with its arguments as a variant.

Unlike a traditional `match` expression, a simple function is only capable of matching only one pattern. To express general match expressions, functions are concatenated via the higher-order *onion* operation `&` to give *compound functions*. We can thus write a dispatch on an object with two methods simply as:

```
let obj = (`twice x -> x + x) & (`isZero x -> x == 0) in
obj `twice 4
```

So, ML `match` expressions can be encoded using the `&` operator using one simple function for each case. Function conjunction generalizes the first-class cases of (Blume et al. 2006); that work does not support "override" of existing clauses or heterogeneously typed case branches.

Dependent pattern types The above shows how to encode an object with multiple methods as an onion of simple functions. But we must be careful not to type this encoding in the way that match/case expressions are traditionally typed. The analogous OCaml match/case expression

```
let obj m = (match m with
                | `twice x -> x + x
                | `isZero x -> x == 0) in ...
```

will not typecheck; OCaml match/case expressions must return the same type in all case branches.[1] Instead, we give the function a dependent pattern type that is informally (`` `twice int → int``) & (`` `isZero int → bool``). If the function is applied in the context where the type of message is known, the appropriate result type is inferred. Because of this dependent typing, `match` expressions encoded in TinyBang may be heterogeneous; that is, each case branch may have a different type in a meaningful way. These dependent pattern types extend the expressiveness of conditional constraint types (Aiken et al. 1994; Pottier 2000) in a dimension critical for typing objects.

Onions are records There is no record syntax in TinyBang; we only require the record concatenation operator `&` so we can append values into type-indexed records. We informally call these records *onions* to signify these properties. Here is an example of how objects can be encoded with multi-argument methods:

```
let obj = (`sum (`x x & `y y) -> x + y)
        & (`equal (`x x & `y y) -> x == y)
in obj (`sum (`x 3 & `y 2))
```

The `` `x 3 & `y 2`` is an onion of two labels and amounts to a two-label record. This `` `sum``-labeled onion is passed to the pattern `` `x x & `y y``. (We highlight the pattern `&` differently than the onioning `&` because the former is a *pattern conjunction* operator: the value must match both subpatterns.)

2.2 Self-Awareness and Resealable Objects

Up to this point objects have not been able to invoke their own methods, so the encoding is incomplete. To model self-reference we build on the work of (Bono and Fisher 1998), where an object exists in one of two states: as a prototype,

[1]The recent OCaml 4 GADT extension mitigates this difficulty but requires an explicit type declaration, type annotations, and only works under a closed world assumption.

which can be extended but not messaged, or as a "proper" object, which can be messaged but not extended. A prototype may be "sealed" to transform it into a proper object, at which point it may never again be extended.

Unlike the aforecited work, our encoding permits sealed objects to be extended and then *resealed*. This flexibility of TinyBang allows the sharp phase distinction between prototypes and proper objects to be relaxed. All object extension below will be performed on sealed objects. Object sealing in TinyBang requires no special metatheory; it is defined directly as a function `seal`, which takes an object `obj` and returns its sealed counterpart. We define `seal` as follows:

```
let fixpoint =
        f -> (g -> x -> g g x) (h -> y -> f (h h) y) in
let seal = fixpoint (seal -> obj ->
        (msg -> obj (msg & `self (seal obj))) & obj) in
let obj = (`twice x -> x + x) &
          (`quad x & `self self ->
                self (`twice x) + self (`twice x))
let sObj = seal obj in
let twenty = sObj `quad 5 in          // returns 20
```

The `seal` function operates by adding a message handler which captures *every* message sent to `obj`. The message handler adds a `` `self`` component to the right of the message and then passes it to the original object. We require `fixpoint` to ensure that this self-reference is also sealed. Thus, every message send to `sObj` will, in effect, be sent to `obj` with `` `self sObj`` attached to the right.

Extending previously sealed objects In the self binding function above, the value of `self` is onioned onto the *right* of the message; this gives any explicit value of `` `self`` in a message passed to a sealed object priority over the `` `self`` provided by the self binding function (onioning is asymmetric with left precedence). Consider the following continuation of the previous code:

```
let sixteen = sObj `quad 4 in         // returns 16
let obj2 = (`twice x -> x) & sObj in
let sObj2 = seal obj2 in
let eight = sObj2 `quad 4 in ...      // returns 8
```

We can extend `sObj` after messaging it, here overriding the `` `twice`` message; `sObj2` represents the (re-)sealed version of this new object. `sObj2` properly knows its "new" self due to the resealing, evidenced here by how `` `quad`` invokes the new `` `twice``. To see why this works let us trace the execution. Expanding the sealing of `sObj2`, `` sObj2 (`quad 4) `` has the same effect as `` obj2 (`quad 4 & `self sObj2) ``, which has the same effect as `` sObj (`quad 4 & `self sObj2) ``. Recall `sObj` is also a sealed object which adds a `` `self`` component to the *right*; thus this has the same effect as `` obj (`quad 4 & `self sObj2 & `self sObj) ``. Because the leftmost `‘self` has priority, the `‘self` is properly `sObj2` here. We see from the original definition of `obj` that it sends a `‘twice` message to the contents of `self` (here, `sObj2`), which then follows the same pattern as above until `obj (‘twice 4 & ‘self sObj2 & ‘self sObj)` is invoked (two times – once for each side of `+`).

Sealed and resealed objects obey the desired object subtyping laws because we “tie the knot” on `self` using `seal`, meaning there is no contravariant `self` parameter on object method calls to invalidate object subtyping. Additionally, our type system includes parametric polymorphism and so `sObj` and the re-sealed `sObj2` do not have to share the same `self` type, and the fact that `&` is a *functional* extension operation means that there will be no pollution between the two distinct `self` types. Key to the success of this encoding is the asymmetric nature of `&`: it allows us to override the default `‘self` parameter.

Resealing is not perfect: if information about the self type is lost due to e.g. the object being extended having been placed in a heterogenous collection, the self type may not be expressive enough to support resealing. There still is a possibility of extension without resealing in this case if no methods are overridden.

Onioning it all together Onions also provide a natural mechanism for including fields; we simply concatenate them to the functions that represent the methods. Consider the following object which stores and increments a counter:

```
let obj = seal (‘x (ref 0) &
               (‘inc _ & ‘self self ->
                   (‘x x -> x := !x + 1 in !x) self))
in obj ‘inc ()
```

Observe how `obj` is a heterogeneous “mash” of a record field (the `‘x`) and a function (the handler for `‘inc`). This is sound because onions are *type-indexed* (Shields and Meijer 2001), meaning that they use the types of the values themselves to identify data. For this particular example, invocation `obj ‘inc ()` (note `()` is an empty onion, a 0-ary conjunction) correctly increments in spite of the presence of the `‘x` label in `obj`.

2.3 Flexible Object Operations

Default arguments and overloading TinyBang can encode default arguments. For instance, consider:

```
let obj = seal ( (‘add (‘x x & ‘y y) -> x + y)
               & (‘sub (‘x x & ‘y y) -> x - y) ) in
let dflt = obj -> (‘add a -> obj (‘add (a & ‘x 1))) & obj in
let obj2 = dflt obj in
obj2 (‘add (‘y 3)) + obj2 (‘add (‘x 7 & ‘y 2))  // 4 + 9
```

Object `dflt` overrides `obj`'s `‘add` to make `1` the default value for `‘x`. Because the `‘x 1` is onioned onto the *right* of `a`, it will have no effect if an `‘x` is explicitly provided in the message.

The pattern-matching semantics of functions provides a simple mechanism for defining overloaded functions. We might originally define negation on the integers as

```
let neg = x & int -> 0 - x in ...
```

Later code could then extend the definition of negation to include boolean values:

```
let neg = (‘True _ -> ‘False ())
        & (‘False _ -> ‘True ()) & neg in ...
```

Mixins The following example shows how a simple two-dimensional `point` object can be combined with a `mixin` providing extra methods (we use sugar `o.x` for `(‘x x -> x) o`):

```
let point = seal (‘x (ref 0) & ‘y (ref 0)
    & (‘l1 _ & ‘self self -> self.x + self.y)
    & (‘isZero _ & ‘self self ->
            self.x == 0 and self.y == 0)) in
let mixin = ‘near _ & ‘self self -> self ‘l1 ()) < 4) in
let mixPt = seal (point & mixin) in mixPt ‘near ()
```

Here `mixin` is a function which invokes the value passed as `self`. Because an object's methods are just functions onioned together, onioning `mixin` into `point` is sufficient to produce a properly functioning `mixPt`.

The above example typechecks in TinyBang; parametric polymorphism is used to allow `point`, `mixin`, and `mixPt` to have different self-types. The `mixin` variable has the approximate type "`(‘near unit & ‘self` α`)` $\rightarrow$ `bool` where α is an object capable of receiving the `‘l1` message and producing an `int`".

3 Function Patterns as Interfaces

TinyBang also includes *function patterns*. Just as other patterns match data based on its shape, TinyBang's function patterns match higher-order functions based on their behavior: a function pattern ϕ_1 `~>` ϕ_2 matches an argument which (1) is a function, (2) takes any argument matching the pattern ϕ_1, and (3) returns a result matching the pattern ϕ_2. For instance, consider the following:

```
let callDefault = ((() ~> int) & f -> f ())
                & ((int ~> int) & f -> f 1) in
callDefault (n -> n + 3)
```

The purpose of `callDefault` is to take a function from either the empty record or an integer onto an integer; `callDefault` then invokes that function with some default argument of the correct type. Here, since the argument is a function from integers onto integers, the simple function on line 2 is invoked.

Note that, as per the function conjunction rules described above, the simple function on line 1 is skipped, because the argument does not match its pattern. There is no explicit pattern or type signature in `n -> n + 3` which dictates this behavior; the dispatch to the right side of `callDefault` is based on the type inferred for the argument.

To check if a function argument ϕ `->` e matches the pattern ϕ_1 `~>` ϕ_2, we simply verify that the function applied to an argument of the form of the pattern ϕ_1 gives a result matching ϕ_2. This amounts to a *subordinate* typecheck: if the application of the higher-order function argument to this fabricated type successfully typechecks and matches ϕ_2, then the function matches the pattern; otherwise, it does not.

Although the algorithm we present above is intuitive, we must take care to ensure that our definitions are well-founded. In particular it could become possible for the result of one function pattern match to influence the outcome of another. To avoid paradoxes, we incorporate an occurrence check and force all self-referential cases to fail typechecking.

3.1 Patterns as Type Signatures

One well-known problem with type systems using subtype constraint inference (such as TinyBang's) is that the inferred types are incredibly complex and therefore impractical for programmers to read, write, or understand. Often, this is because the types are "too good": they incorporate subtle subtyping relationships which are irrelevant to the programmer. Even with extensive simplification (Pottier 1999; Eifrig et al. 1995), the inferred types of programs can be of the same magnitude in size as the program itself! Fortunately, function patterns provide us with an alternative: because our pattern grammar is now sufficiently powerful, it can be used (at the programmer's option) to *encode* the typical use cases of type signatures in other languages. For instance,

```
let f = (x -> x + 1) : int ~> int
```

may be viewed as sugar for

```
let f = let g = ... in
        (a & (int ~> int) -> a) g
```

Here, `(a & (int ~> int) -> a)` is a restricted form of the identity function which only applies to functions from `int` onto `int`; thus, if `g` is not such a function, a type error will arise.

Traditional constraint type systems typically use the constrained types themselves to define type signatures but this problem may be undecidable in the face of subtyping (Henglein and Rehof 1998). TinyBang's pattern matching does not suffer this complexity because a simpler question is being asked since there is no subtyping on the input type of our function patterns.

While the underlying The TinyBang type system infers polymorphic types for functions, it would also be nice to allow such types in declarations. So, TinyBang supports parametric pattern polymorphism via *opaque pattern variables*, written `'a` to mimic ML syntax. An example of use appears below:

```
let id = x -> x : 'a ~> 'a in ...
```

As per the desugaring above, this will match `id` against the pattern `'a ~> 'a`. Externally, `'a` behaves like a type variable: the pattern matches only functions which return the same type they receive. But our algorithm works by fabricating an (opaque) type nonce and applying the function (in this case, `id`) to it.

4 Related Work

TinyBang shares some features and goals with CDuce (Castagna et al. 2014): both aim to be flexible typed languages built around constraint subtyping. CDuce uses explicit union/intersection types whereas we embed them in subtype constraints. We have complete type inference whereas CDuce is a declared type language with an overlay of local type inference. Our flexible case clause extension and record concatenation operations are not found in their theory. CDuce has no precise analogue of our function patterns (it is possible to match on a function type in CDuce, but it is matching against the type declared and not the function's behavior). CDuce takes a local approach to type inference; this has the important advantage of being modular (which we are not), but the disadvantage of not being provably complete.

TinyBang's object resealing is inspired by the Bono-Fisher object calculus (Bono and Fisher 1998). Objects in this calculus must be "sealed" before they are messaged; unlike our resealing, sealed objects cannot be extended. Some related works relax this restriction but add others Riecke and Stone (2002); Bettini et al. (2011) Unlike all previous works, in our approach (re-)sealing is expressible directly in the existing language syntax.

Typed multimethods (Millstein and Chambers 1999) perform a dispatch similar to TinyBang's dispatch on compound functions, but multimethod dispatch is nominally typed while compound function dispatch is structurally typed. First-class cases (Blume et al. 2006) allow composition of case branches much like TinyBang. In Blume et al. (2006), however, general case concatenation requires a phase distinction between constructing a case and matching with it. TinyBang has a form of dependent type which allows different case branches to return different types; this generalizes the expressiveness of conditional constraints (Aiken et al. 1994; Pottier 2000) and is related to support for typed first-class messages *a la* Nishimura (1998); Pottier (2000) – first-class messages are just labeled data in our encoding.

TinyBang's onions are a form of record supporting typed asymmetric concatenation. The bulk of work on typing record extension addresses symmetric concatenation only (Rémy 1994). Standard typed record-based encodings of inheritance (Bruce et al. 1999) avoid the problem of typing first-class concatenation by reconstructing records rather than extending them, but this requires the superclass to be fixed statically. A combination of conditional constraints and row types can be used to type record extension (Pottier 2000); TinyBang uses an approach that does not need row types.

To our knowledge there is no direct precedent for structural higher-order function pattern matching. Our interest in the topic was inspired by the power of runtime contracts (Findler and Felleisen 2002); function patterns are more static and declarative compared to contracts, because higher-order function contracts only enforce the contract at runtime and only on the values they were invoked on, whereas higher-order function patterns are statically enforced at all potential runtime values.

5 Conclusions

Here we have outlined TinyBang, a core language combining flexible scripting-style syntax with full static type inference. We illustrated how its flexible operations can be used to encode standard object-oriented paradigms. TinyBang also includes higher-order function pattern matching that match on the *behavior* of the function. Function pattern matching can be used for in-place type refinements and interface declarations for subtype constraint systems.

TinyBang without function patterns is proved type sound and decidable in (Smith et al. 2014b), and an implementation may be downloaded from (Smith et al. 2014a). We do not expect programmers to write in TinyBang. Instead, programmers would write in BigBang, a language we are developing which includes syntax for objects, classes, and so on, and which will de-sugar to TinyBang.

References

A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL 21*, pages 163–173, 1994.

L. Bettini, V. Bono, and B. Venneri. Delegation by object composition. *Science of Computer Programming*, 76:992–1014, 2011.

M. Blume, U. A. Acar, and W. Chae. Extensible programming with first-class cases. In *ICFP*, pages 239–250, 2006.

V. Bono and K. Fisher. An imperative, first-order calculus with object extension. In *ECOOP*, pages 462–497. Springer Verlag, 1998.

K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155(1-2):108–133, 1999.

G. Castagna, K. Nguyen, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions with set-theoretic types. part 1: Syntax, semantics, and evaluation. In *POPL*, 2014.

J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA Conference Proceedings*, volume 30(10), pages 169–184, 1995.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.

F. Henglein and J. Rehof. Constraint automata and the complexity of recursive subtype entailment. In *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, 1998.

T. D. Millstein and C. Chambers. Modular statically typed multimethods. In *ECOOP*, pages 279–303. Springer-Verlag, 1999.

S. Nishimura. Static typing for dynamic messages. In *POPL*, 1998.

F. Pottier. A framework for type inference with subtyping. In *ICFP*, 1999.

F. Pottier. A versatile constraint-based type inference system. *Nordic J. of Computing*, 7(4):312–347, 2000.

D. Rémy. Type inference for records in a natural extension of ML. In *Theoretical Aspects Of Object-Oriented Programming.* MIT Press, 1994.

J. G. Riecke and C. A. Stone. Privacy via subsumption. *Inf. Comput.*, 172(1):2–28, Feb. 2002.

M. Shields and E. Meijer. Type-indexed rows. In *POPL*, pages 261–275, 2001.

S. Smith, P. H. Menon, Z. Palmer, and A. Rozenshteyn. Tinybang implementation, Jan 2014a. `http://pl.cs.jhu.edu/big-bang/tiny-bang_2014-03-01.tgz`.

S. Smith, P. H. Menon, Z. Palmer, and A. Rozenshteyn. Types for flexible objects. Technical report, The Johns Hopkins University Programming Languages Laboratory, 2014b. `http://pl.cs.jhu.edu/big-bang/types-for-flexible-objects_2014-01-13.pdf`.

The Spider Calculus: Computing in Active Graphs

Dedicated to Luca Cardelli on his 60th Birthday

Benjamin C. Pierce
University of Pennsylvania

Alessandro Romanel
CIBIO University of Trento

Daniel Wagner
University of Pennsylvania

Abstract

We explore a new class of process calculi, collectively called *spider calculi*, in which processes inhabit the nodes of a directed graph, evolving and communicating by local structural mutations. We identify a *kernel spider calculus* that is both minimal and expressive. Processes in this kernel calculus can construct arbitrary finite graphs, encode common data structures, and implement the communication primitives of the π-calculus.

1 Introduction

The execution environment for modern software is fundamentally graph-structured. On both large and small scales—from routers and fiber to processors, memories, and buses—software components inhabit different physical or logical locations, and information must cross links when they cooperate. Ordinarily, this graph structure is hidden behind simpler abstractions such as point-to-point internet communication or shared memory on a multiprocessor. But there are also situations where we want to deal with it explicitly.

For example, *implementations* of these abstraction layers must themselves deal directly with the underlying graph structure. This includes internet protocols for routing, broadcast, name resolution, hardware cache coherence mechanisms, etc. There are also applications that work directly with the graph structure to increase efficiency. Content distribution networks such as Akamai (Maggs 2001) and Coral (Freedman et al. 2004) fall in this category, as do numerous distributed graph algorithms—e.g., algorithms for answering reachability queries on streaming graphs (Unel et al. 2009) and approximating shortest paths with incomplete knowledge of the graph (Henzinger et al. 1997).

In these applications, the graph structure is relatively fixed. In other cases, the graph may change as the program evolves—for example, in routing algorithms for ad hoc mobile networks. Indeed, there are cases where the program itself may alter the topology of the graph in which it is working. Peer-to-peer networks do just this, establishing logical graph structures for organizing communication over the location-transparent abstraction of internet routing. Also, some well-known parallel algorithms operate on virtual graphs: Delaunay mesh triangulation modifies graphs in which nodes represent physical locations (Kulkarni et al. 2007), and n-body simulations often use graphs in which each node represents a volume of the space being simulated (Barnes and Hut 1986).

In the theoretical literature, there has been a corresponding interest in foundational models that explicitly embody notions of locality and connectivity. Variants of the π-calculus such as Nomadic Pict (Wojciechowski and Sewell 2000) and the Distributed π-calculus (Hennessy 2007) model mobile computation in the internet, allowing direct communication only between processes that have migrated to the same location. Cardelli and Gordon's Ambient Calculus (Cardelli and Gordon 2000), the biologically-inspired Brane Calculus (Cardelli 2004), and their many variants (Bugliesi et al. 2001; Cardelli et al. 1999; Cardelli and Gordon 1999; Levi and Sangiorgi 2003, etc.) structure locations into a tree; process movement is restricted to paths in the tree, and processes can alter the tree structure in the immediate neighborhood of their current location.

We study here a class of systems, dubbed *Spider Calculi*, which generalize the ideas of the Ambient Calculus to arbitrary graphs. Our goal in this short paper is to take a first step into this new design space: to experiment with basic programming idioms, and to identify a *kernel spider calculus* that strikes an attractive balance between expressiveness and simplicity. Some omitted technical details and a discussion of alternative primitives can be found in a longer online version (Pierce et al. 2010).

2 Overview

We consider computing in edge-labeled, directed graphs—that is, the "universe" in which computation happens (called the *web*, naturally) is a graph with labeled edges and anonymous nodes.We call the edges *links*. Self-links are allowed, as are multiple links between a given pair of nodes and multiple links with the same label. Also, we prohibit "action at a distance." Each computational process ("spider") is associated with one node at any given time, and only the links incident to that node are visible to or modifiable. To observe or modify links elsewhere in the graph, the spider must first travel there.

To describe computations in graphs, we need three things. First, we need a notation for describing the graphs themselves. Second, we need a notation for local computations at the nodes, including data structures, conditionals, loops, communication, synchronization, and so forth. And third, we need ways for processes to navigate and observe the graph. In the interest of parsimony, we have combined these three as much as possible. In particular, there is no fundamental need for "local data"; all the data and control structures we need can be encoded in the structure of the web, just as in the λ-calculus and π-calculus, which encode local data as functions or processes.

One consequence of this choice is that there are really two distinct kinds of links: links in some "real graph" and links that are created and used for spiders' local computation. To avoid interference between the two kinds, and also to avoid interference between the local computations of distinct spiders, we assume that link names are *scoped*—in particular, there is a way to generate fresh link names that are only known to the spider that generates them—and that a spider can neither observe nor affect links whose names it does not know. Formally, we follow the π-calculus and its relatives by introducing a *restriction* operator ν for this purpose.

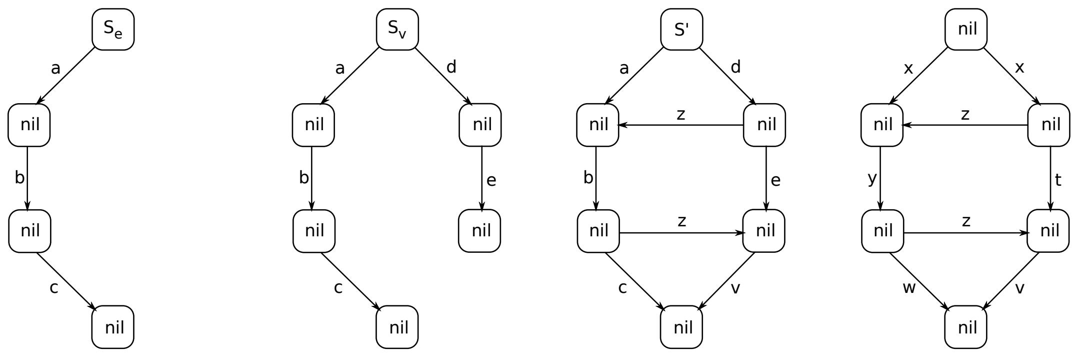
Spiders are written using a few generic combinators—an inert "null process," parallel composition, and replication—plus a small set of primitive *actions*. The actions express atomic steps that modify and navigate the graph.

We use two benchmarks to test our primitives: building arbitrarily shaped finite graphs and emulating the π-calculus. Our experience experimenting with spider programming suggests that any calculus that can do these two things is expressive enough to capture a broad range of computations in and on graphs.

3 Example

Any "graph computing calculus" should certainly be expressive enough to construct finite graphs of arbitrary shapes. For instance, consider the graph in Figure 1d. We'd like to write a spider program that builds this graph, beginning with a single node containing a single spider. Figures 1a, 1b, and 1c show intermediate webs; these will act as subgoals.

The first task is to build a new node with edges to it. We introduce three primitives for this purpose: create, copy, and reverse. The behavior of the create action is sketched in Figure 2a (a formal reduction semantics is given in the next section). The small dashed lines represent other links that may be incident to the node; these are not affected by the create operation. The figure leaves implicit the fact that there may be other spiders running at this node, which are also



(a) single path (b) spanning tree (c) graph shape (d) edge renaming

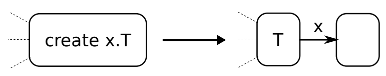
Figure 1: Building a finite graph

unaffected. This is an instance of a general design principle: spiders interact *only* by observing each others' effects on the graph.

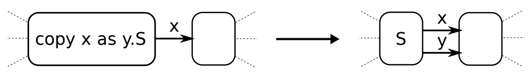
The create operation builds a single directed link from the creating node to the new one. In a moment, we will also need another edge pointing back from the new node. The copy and reverse actions, sketched in Figures 2b and 2c, make this possible. The copy primitive simply creates an additional link with the same source and target as another link, while reverse swaps the source and target of an existing link. The three can be combined—create x. copy x as x. reverse x. S'—to create a new node with forward and backward links named x to it. (The reverse primitive chooses nondeterministically between the two available x links, but since they are identical, the outcome is the same.) We write createboth x. S' as shorthand for this spider.

By creating new nodes in sequence, we can create a path with unique names for each edge on the path. (These names are temporary; the last step will be replacing them with the actual labels that we want on these edges. Using distinct names during construction avoids ambiguity in cases where some node in the final graph has two links with the same name, like the topmost node in Figure 1d.) After the execution of a single createboth x operation, we have a new node accessible by a link named x, but no spider at the new node. To put one there, we use the go action, depicted in Figure 2d. Together, the createboth and go actions are enough to construct any finite path; the spider S_b shown in Figure 3 demonstrates how to build the path in Figure 1a. The spider S_e in Figure 3 finishes building the spanning tree by adding another path, then continues as S_v.

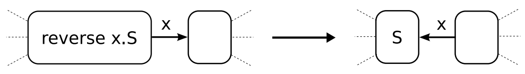
Now we introduce cycles. We can create a small cycle with copy x as y, then increase its size using the throw action depicted in Figure 2e. The spider S_v in Figure 3 demonstrates this process for building the v edge of our final graph, creating it initially at the top of the graph by copying a and then throwing each



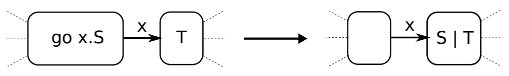
(a) the create primitive



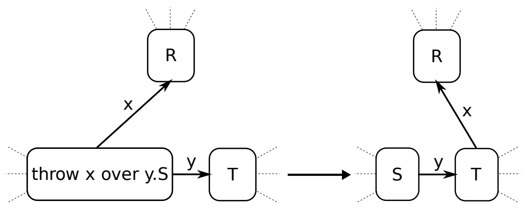
(b) the copy primitive



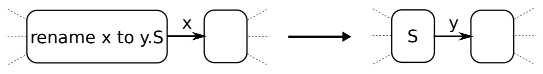
(c) the reverse primitive



(d) the go primitive



(e) the throw primitive



(f) the rename primitive

Figure 2: Spider Primitives

end down the spanning tree. The spider S_z (not shown) is similar, putting the two z edges into place, then continuing as S_r.

We now have a graph with the desired shape, but with the wrong names. To rectify this, we introduce the rename primitive, depicted in Figure 2f. The spider S_r shown in Figure 4 demonstrates the use of this primitive. This leaves us a graph with the right shape, but with some extra links named a, b, c, d, and e. We can remove them from the observable web by using the restriction operator ν to ensure that they cannot affect the behavior of other spiders. This gives us the final spider S shown in Figure 3.

4 Formal Definition

In the syntax of the spider calculus (Figure 5), we use V, W to range over webs and S, T to range over spiders. Both include syntactic forms for restriction and

$$S = \nu abcde.\ S_b$$

$$\begin{array}{rcl@{\qquad}rcl}
S_b & = & \mathsf{createboth}\ a.\ \mathsf{go}\ a. & S_e & = & \mathsf{createboth}\ d. \\
 & & \mathsf{createboth}\ b.\ \mathsf{go}\ b. & & & \mathsf{go}\ d. \\
 & & \mathsf{createboth}\ c.\ \mathsf{go}\ c. & & & \mathsf{createboth}\ e. \\
 & & \mathsf{go}\ c.\ \mathsf{go}\ b.\ \mathsf{go}\ a.S_e & & & \mathsf{go}\ e. \\
 & & & & & \mathsf{go}\ e.\ \mathsf{go}\ d.\ S_v
\end{array}$$

$$\begin{array}{rcl}
S_v & = & \mathsf{copy}\ d\ \mathsf{as}\ v. \\
 & & \mathsf{throw}\ v\ \mathsf{over}\ a.\ \mathsf{go}\ a. \\
 & & \mathsf{throw}\ v\ \mathsf{over}\ b.\ \mathsf{go}\ b. \\
 & & \mathsf{throw}\ v\ \mathsf{over}\ c.\ \mathsf{go}\ c. \\
 & & \mathsf{reverse}\ v.\ \mathsf{go}\ c.\ \mathsf{go}\ b.\ \mathsf{go}\ a.\ \mathsf{go}\ d. \\
 & & \mathsf{throw}\ v\ \mathsf{over}\ e.\ \mathsf{go}\ e. \\
 & & \mathsf{go}\ e.\ \mathsf{go}\ d.\ S_z
\end{array}$$

Figure 3: Parts of the spider that builds the graph in Figure 1d.

$$\begin{array}{rcl}
S_r & = & \mathsf{go}\ a.\ \mathsf{go}\ b.\ \mathsf{rename}\ c\ \mathsf{to}\ w.\ \mathsf{go}\ b.\ \mathsf{rename}\ b\ \mathsf{to}\ y.\ \mathsf{go}\ a. \\
 & & \mathsf{rename}\ a\ \mathsf{to}\ x.\ \mathsf{go}\ d.\ \mathsf{rename}\ e\ \mathsf{to}\ t.\ \mathsf{go}\ d.\ \mathsf{rename}\ d\ \mathsf{to}\ x.\ \mathsf{nil}
\end{array}$$

Figure 4: Renaming edges to match the desired graph.

$$\begin{array}{rcl}
V, W & ::= & \mathsf{Nil} \ \big|\ \nu x.\ W \ \big|\ V \mid W \ \big|\ [S]^i \ \big|\ i \xrightarrow{x} j \\
S, T & ::= & \mathsf{nil} \ \big|\ \nu x.\ S \ \big|\ S \mid T \ \big|\ !S \ \big|\ M.\ S \\
M & ::= & \mathsf{create}\ x \ \big|\ \mathsf{go}\ x \ \big|\ \mathsf{copy}\ x\ \mathsf{as}\ y \ \big| \\
 & & \mathsf{rename}\ x\ \mathsf{to}\ y \ \big|\ \mathsf{throw}\ x\ \mathsf{over}\ y \ \big|\ \mathsf{reverse}\ x
\end{array}$$

Figure 5: Syntax of the spider calculus

$$\begin{array}{lll}
V \mid \nu x.\ W \ \equiv\ \nu x.\ (V \mid W) \qquad \text{if } x \notin \mathsf{fn}(V) & & \text{(res-par)} \\
[\,\mathsf{nil}\,]^i \ \equiv\ \mathsf{Nil} & & \text{(trans-nil)} \\
[S \mid T]^i \ \equiv\ [S]^i \mid [T]^i & & \text{(trans-par)} \\
[\nu x.\ S]^i \ \equiv\ \nu x.\ [S]^i \qquad \text{if } x \neq i & & \text{(trans-res)}
\end{array}$$

Figure 6: Structural congruence laws (plus associative, commutative laws for $|$)

for binary and nullary parallel composition. The graph structure of the web is represented as a parallel composition of nodes and edges, with node identity determined by name. For example, we regard a web with one node labeled x

$$\frac{W \longrightarrow W'}{W \mid V \longrightarrow W' \mid V} \qquad \text{(red-par)}$$

$$\frac{W \longrightarrow W'}{\nu x.\, W \longrightarrow \nu x.\, W'} \qquad \text{(red-res)}$$

$$\frac{W \;\equiv\; W' \longrightarrow V' \;\equiv\; V}{W \longrightarrow V} \qquad \text{(red-struct)}$$

$$\begin{array}{ll}
[\,!S\,]^i \;\longrightarrow\; [\,S\,]^i \mid [\,!S\,]^i & \text{(red-repl)} \\
[\,\mathsf{throw}\ x\ \mathsf{over}\ y.\ S\,]^i \mid i \xrightarrow{x} j \mid i \xrightarrow{y} k \;\longrightarrow\; [\,S\,]^i \mid k \xrightarrow{x} j \mid i \xrightarrow{y} k & \text{(red-dthrow)} \\
[\,\mathsf{create}\ x.\ S\,]^i \;\longrightarrow\; [\,S\,]^i \mid \nu j.\ i \xrightarrow{x} j \qquad \text{where } j \notin \{i, x\} & \text{(red-dcreate)} \\
[\,\mathsf{rename}\ x\ \mathsf{to}\ y.\ S\,]^i \mid i \xrightarrow{x} j \;\longrightarrow\; [\,S\,]^i \mid i \xrightarrow{y} j & \text{(red-drename)} \\
[\,\mathsf{copy}\ x\ \mathsf{as}\ y.\ S\,]^i \mid i \xrightarrow{x} j \;\longrightarrow\; [\,S\,]^i \mid i \xrightarrow{x} j \mid i \xrightarrow{y} j & \text{(red-dcopy)} \\
[\,\mathsf{go}\ x.\ S\,]^i \mid i \xrightarrow{x} j \;\longrightarrow\; [\,S\,]^j \mid i \xrightarrow{x} j & \text{(red-dgo)}
\end{array}$$

Figure 7: Operational semantics

containing a spider $S|T$ as structurally equivalent to a web with two nodes labeled i, one containing S and one containing T.[1] The form $[\,S\,]^i$ denotes a spider S living at node i, and $i \xrightarrow{x} j$ denotes an edge from i to j labeled x.

We use the variable M to range over primitive actions and use lower-case letters for variable names. As usual, the spider $M.\,S$ blocks until M can be executed and then continues as S. The replication of spider S is written $!S$. (There is no replication form at the level of webs and no structural congruence rule for replication; instead, replication is implemented using a reduction rule.) We abbreviate $M.\,\mathsf{nil}$ as M and $\nu x_1.\cdots\nu x_n.\,W$ as $\nu x_1,\ldots,x_n.\,W$ or $\nu\tilde{x}.\,W$. We write $\mathsf{fn}(S)$ for the free names of S (ν is the only name binder). Figure 6 defines the structural congruence. Name restrictions via ν may be floated in and out of parallel compositions, node boundaries, and other restrictions provided this does not orphan any names or cause any clashes. Figure 7 gives the operational semantics, which simply formalizes the informal diagrams we have already seen.

[1]This may appear to be a significant difference from Ambients, where a solution with a single node is *not* equivalent to a solution with two nodes of the same name, but the appearance is deceptive. Ambient node names correspond to *edge* names in the spider calculus; node names in the spider calculus are just a means for representing graph structure in linear form; they are not directly accessible to programs.

5 Programming

We illustrate the expressiveness of the spider calculus by sketching encodings both for static data like numbers, strings, lists, and so on, and for the λ-calculus and π-calculus.

Passive encoding The simplest encoding strategy uses the graph structure of a web in a straightforward way. For example, a number n can be represented by a location with either a link named *succ* pointing to a web representing $n-1$ or a link named *zero*. Computing with numbers in this encoding is straightforward: just choose one of two spiders based on the existence of one of the above links. For example, suppose we have a number stored at the bidirectional link x. (By bidirectional, we mean that there are two links named x pointing in opposite directions.) Then the following snippet will compute the predecessor of x and making it available on a link named y:

$$\mathsf{go}\ x.\ (\mathsf{copy}\ \mathit{succ}\ \mathsf{as}\ y.\,\mathsf{throw}\ y\ \mathsf{over}\ x \mid \mathsf{rename}\ \mathit{zero}\ \mathsf{to}\ \mathit{zero}.\,\mathsf{go}\ x.\,\mathsf{copy}\ x\ \mathsf{as}\ y)$$

The first two actions in the parallel composition depend, respectively, on the existence of a link named *succ* or *zero*; if the location really is a number, exactly one of them will fire and one of the two continuations will be released.

Active encoding A different strategy—closer to familiar encodings in π-calculus—is to represent a number as a spider that repeatedly answers requests for its value. A number will wait for a link named *val* to appear, then go to that link, create either a *succ* or a *zero* link, and continue there as its predecessor (if it has one). Here is the 0 spider: $S_0 \ = !\nu t.\ \mathsf{rename}\ \mathit{val}\ \mathsf{to}\ t.\ \mathsf{go}\ t.\ \mathsf{create}\ \mathit{zero}.$ And if S_n is the spider representing the number n, the spider representing its successor is $S_{n+1} \ = !\nu t.\ \mathsf{rename}\ \mathit{val}\ \mathsf{to}\ t.\ \mathsf{go}\ t.\ \mathsf{create}\ \mathit{succ}.\ S_n$. Consuming a number in this representation is similar in spirit to the previous representation. For example, assuming there is a bidirectional link x pointing to a location that contains (only) a number, here is a spider that creates a link y pointing to a location containing that number's predecessor:

$$\begin{array}{l}\mathsf{go}\ x.\ \nu t.\ \mathsf{createboth}\ t.\ \mathsf{copy}\ t\ \mathsf{as}\ \mathit{val}.\ \mathsf{go}\ t.\\ \quad (\ \mathsf{rename}\ \mathit{succ}\ \mathsf{to}\ \mathit{succ}.\ \mathsf{go}\ t.\ \mathsf{rename}\ t\ \mathsf{to}\ y.\ \mathsf{throw}\ y\ \mathsf{over}\ x\\ \quad \mid \mathsf{rename}\ \mathit{zero}\ \mathsf{to}\ \mathit{zero}.\ \mathsf{go}\ t.\ \mathsf{go}\ x.\ \mathsf{copy}\ x\ \mathsf{as}\ y\ )\end{array}$$

Church encoding of the λ-calculus Figure 9 shows the complete encoding of the λ-calculus. In "λ-calculus webs," each node represents the encoding of a λ-term, with links pointing to other nodes encoding the values of its free variables. The evaluation strategy is "parallel reduction outside lambdas": λ-abstractions are inert until they receive an argument, whereas the function and argument parts

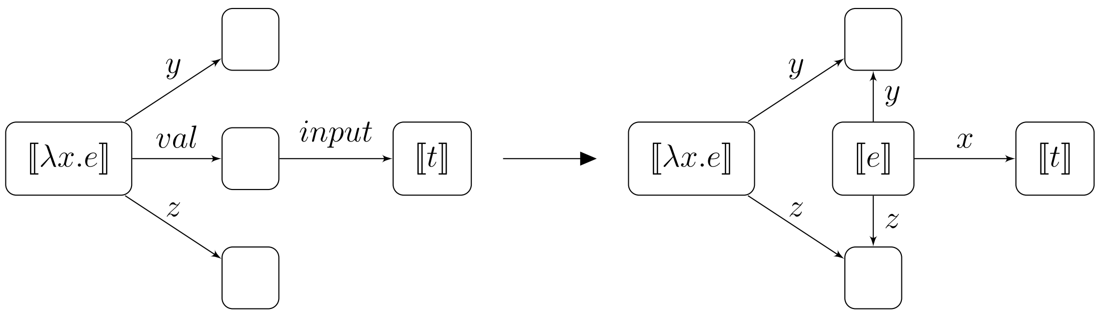


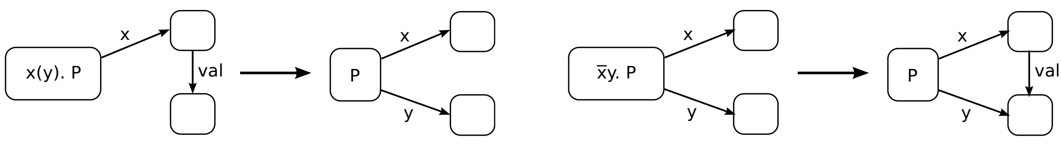
Figure 8: Emulating beta-reduction in the spider calculus, assuming $\mathsf{fn}(e) = \{x, y, z\}$

$$
\begin{aligned}
[\![\lambda x.e]\!] &= !\nu t.\ \mathsf{rename}\ val\ \mathsf{to}\ t. \\
&\quad \mathsf{closure}\ \mathsf{fn}(e) \setminus \{x\}\ \mathsf{at}\ t. \\
&\quad \nu x.\ \mathsf{rename}\ input\ \mathsf{to}\ x.\ [\![e]\!]
\end{aligned}
$$

$$
\begin{aligned}
[\![x]\!] &= !\mathsf{throw}\ val\ \mathsf{over}\ x \\
[\![e_1\ e_2]\!] &= \nu t_1, t_2.\ \mathsf{createboth}\ t_1.\ \mathsf{create}\ t_2. \\
&\quad (\ \ \mathsf{closure}\ \mathsf{fn}(e_1)\ \mathsf{at}\ t_1.\ [\![e_1]\!] \\
&\quad \mid\ \ \mathsf{closure}\ \mathsf{fn}(e_2)\ \mathsf{at}\ t_2.\ [\![e_2]\!] \\
&\quad \mid\ \ \mathsf{go}\ t_1.\ \mathsf{rename}\ t_1\ \mathsf{to}\ val \\
&\quad \mid\ \ \mathsf{copy}\ t_2\ \mathsf{as}\ input\ \ )
\end{aligned}
$$

Figure 9: Encoding the λ-calculus

of applications run at different nodes and can be evaluated in parallel. There are two different roles—the *function node* role and the *argument node* role—that a given location can play at different times. A λ-abstraction $\lambda x.\, e$ is translated to a function node containing a spider that waits for an argument to arrive in the form of a link named *val* appearing at its node. The node t at the other end of this link is an *argument node*, and it should have a link named *input* pointing to a function location. The λ-spider creates a running copy of its body e at the node t, after renaming *input* to x so that e can access it later. Figure 8 depicts this transformation, the heart of a beta-reduction step. The encoding of a function application $e_1\ e_2$ at some node t creates new nodes t_1 and t_2, places spiders encoding e_1 and e_2 at these nodes, and plumbs them together by turning the original node t into an argument node with a link from t_1 called *val* and a link to t_2 called *input*. A free variable in an encoded λ-term is represented as a link pointing to the location holding the value associated with that name. The translation of a variable simply forwards any *val* requests to the function location associated with that variable. We model closures as nodes with a link



(a) Input (b) Output

Figure 10: The execution of encoded π-calculus primitives

for each bound name. We often have to copy and transmit an entire closure, so we introduce an abbreviation: if $S = \{x_1, \ldots, x_n\}$ is a set of names, then closure S at t. T will stand for

copy x_1 as x_1. $\cdots$. copy x_n as x_n. throw x_1 over t. $\cdots$. throw x_n over t. go t. T.

This spider makes copies of some number of variable links (with names in S) at the local node, throws the copies across the link named t, and finally moves itself across t and continues as T.

Encoding the π-calculus We work with the standard (synchronous, choice-free) π-calculus. The idea is simple: all of the translations of π processes run in a single location (which we call *root*), and a π channel c is represented by a link named c to a location with zero or more links, all named *val*, each pointing to a message waiting to be sent along c. (Using a channel in the π-calculus requires knowing a name for it; using a channel in the spider calculus requires having an edge from *root* to the location for that channel.) Figures 10a and 10b sketch the high-level behavior of encoded input and output processes. The full definition of the encoding is given in Figure 11 (just the non-homomorphic cases). The top-level definition, $[\![P]\!]^{\pi}_{S}$, defines a web with "channel nodes" for each of the channel names in S and with the encoding of π-process P, written $[\![P]\!]^{\pi}$, running at *root*.

6 Related Work

Mobile ambients (Cardelli et al. 1999), a key inspiration for the spider calculus, structure processes into edge-labeled trees, much like the spider calculus's edge-labeled graphs. Many spider calculus actions are direct analogues of mobile ambient actions; in particular, the in x and out capabilities of mobile ambients are analogous to the go x action of spiders. An interesting difference is that, whereas open is a fundamental operation of ambients, the kernel spider calculus, which has no directly analogous operator, is still a quite expressive language. (One could consider adding a merge operation to identify two nodes in the web. This would raise headaches with possible code injection, etc., as open does in ambients.)

$$
\begin{aligned}
\llbracket P \rrbracket^{\pi}_{S} &= \llbracket S \rrbracket^{\pi} \mid [\llbracket P \rrbracket^{\pi}]^{root} \\
\llbracket \{x_1, \cdots, x_k\} \rrbracket^{\pi} &= (root \overset{x_1}{\to} j_1 \mid \cdots \mid root \overset{x_k}{\to} j_k) \\
&\quad \text{where } j_1 \ldots j_k \text{ are fresh names} \\
\llbracket (\nu x)P \rrbracket^{\pi} &= \nu x.\ (\textsf{create}\ x \mid \llbracket P \rrbracket^{\pi}) \\
\llbracket \overline{x}y \rrbracket^{\pi} &= \nu z, z'.\ S_1 \\
&\quad \text{where} \\
&\quad S_1 = \textsf{copy}\ x\ \textsf{as}\ z.\ S_2 \\
&\quad S_2 = \textsf{copy}\ y\ \textsf{as}\ z'.\ S_3 \\
&\quad S_3 = \textsf{throw}\ z'\ \textsf{over}\ z.\ S_4 \\
&\quad S_4 = \textsf{go}\ z.\ S_5 \\
&\quad S_5 = \textsf{rename}\ z'\ \textsf{to}\ val
\end{aligned}
\qquad
\begin{aligned}
\llbracket x(y).P \rrbracket^{\pi} &= \nu z, z', y.\ T_1 \\
&\quad \text{where} \\
&\quad T_1 = \textsf{copy}\ x\ \textsf{as}\ z.\ T_2 \\
&\quad T_2 = \textsf{copy}\ z\ \textsf{as}\ z'.\ T_3 \\
&\quad T_4 = \textsf{reverse}\ z'.\ T_5 \\
&\quad T_5 = \textsf{go}\ z.\ T_6 \\
&\quad T_6 = \textsf{rename}\ val\ \textsf{to}\ y.\ T_7 \\
&\quad T_7 = \textsf{throw}\ y\ \textsf{over}\ z'.\ T_8 \\
&\quad T_8 = \textsf{go}\ z'.\ \llbracket P \rrbracket^{\pi} \\
&\quad \text{provided } z, z' \notin \textsf{fn}(P)
\end{aligned}
$$

Figure 11: Encoding the π-calculus

Safe ambients (Levi and Sangiorgi 2003) and boxed ambients (Bugliesi et al. 2001) propose different restrictions that permit a more controlled style of programming than original ambients. Safe ambients introduce co-actions, allowing actions to fire only when they are properly paired with an identical co-action in the destination ambient. In essence, an action represents a request and a co-action represents permission. Boxed ambients replace open with special communication channels between a parent and its children. Processes stay within a single ambient, and only the communication topology changes. The spider calculus has no analog of safe ambients' co-actions or of boxed ambients' restricted mobility; indeed, the introduction of either feature seems to severely cripple the calculus. It would be interesting to look for similar restrictions with better properties.

The brane calculus (Cardelli 2004) shares many ideas with safe ambients, but it has two kinds of locations (the "membrane" and the "fluid"), which must alternate within the tree. This leads us to wonder whether the spider calculus could be generalized to allow processes at both nodes and links.

Besides the brane calculus, many other calculi have taken inspiration from Regev and Shapiro (2002), in which π-calculus was proposed as a tool to model the dynamics of biological systems. Some of them, like BioAmbients (Regev et al. 2004) and Beta Binders (Priami and Quaglia 2005), are attempts to adapt and extend Mobile Ambients to better capture biological phenomena. BioAmbients introduce several modications to ambients. They maintain the hierarchical structure of ambients but make them nameless; the primitive for opening is replaced by a merge primitive that fuses two ambients. Capabilities have corresponding co-capabilities and new primitives for communication between sibling and parent/child ambients are introduced. Beta Binders equip ambients with typed interfaces; processes in different ambients can communicate through corresponding ambient interfaces if some (domain-dependent and user-defined) affinity property between interface types is satisfied. Although this representation generalizes the hierarchical structure of ambients, processes can only partially affect ambient topology by locally modifying interfaces, while process migration and ambient restructuring is obtained by a set of user-defined functions that split and merge ambients based on their structure.

Another line of bio-inspired calculi, primarily intended to model phenomena like the assembly of complexes or polymers, enrich processes with alternative forms of location. The 3π calculus (Cardelli and Gardner 2012) uses a 3D coordinate system to represent locations. Similarly, in SpacePi (John et al. 2008) processes are embedded into a vector space and can move individually and communicate only if sufficiently close. The BioScape^L calculus (Compagnoni et al. 2013) builds upon 3π and SpacePi and provides a set of high level primitives designed to model bacteria-material interactions in space.

The distributed π-calculus (Hennessy 2007) and Nomadic Pict (Wojciechowski and Sewell 2000) both have explicit, named locations. Processes can go to any location whose name they know—that is, the connection topology for any given process is the complete graph on the set of known location names. The formal presentation of the spider calculus also has explicit locations, but processes themselves cannot refer to the location names in any way; only the local connectivity—which may not be complete—is known. The distributed π-calculus' notion of located channels is quite similar to the spider calculus's notion of located links.

The Chorus language (Lublinerman et al. 2009) proposes another approach—intuitively rather similar to that of the spider calculus—to computing in graphs. Chorus models synchronization with neighborhoods: variables in any particular neighborhood are synchronous, and the basic operations merge and split neighborhoods. Synchronization in the spider calculus is implicit: certain edges may be interpreted as locks. Nevertheless, Chorus and the spider calculus may both prove to be good models for the same sorts of sparse parallel algorithms: physical simulations, computing spanning trees, n-body problems, social networking simulations, and sparse matrix computations.

Bigraphs (Milner 2009) offer an abstract framework for a great variety of process calculi, including located ones. Bigraphs share some characteristics with spiders—in particular, nodes and named edges. However, the structure of nodes and edges differ significantly from webs: nodes are arranged in a tree structure, they have have prescribed *arities* telling how many edges must be incident to them, and edges can connect arbitrarily many nodes. The result is that the nodes and edges of bigraphs are used for significantly different purposes than the nodes and edges of a web. Nevertheless, it seems likely that the spider calculus could be presented as an instance of the bigraph framework.

The Distributed Join Calculus (Fournet and Gonthier 2002) extends the join calculus (a π-calculus variant) by adding locations and primitives for mobility. Although locations are arranged in a tree, processes may migrate directly to any location whose name they know. When a process migrates, it brings along any sublocations as well. In the spider calculus, by contrast, processes cannot refer directly to names of locations; only local migration along named edges is allowed. Also, the spider calculus separates process migration primitives from primitives that modify the topology.

Meld (Ashley-Rollman et al. 2009) is another language for distributed computing on networks with shifting connection topologies, but with some key differences. Like the spider calculus, there are nodes and links arranged in a graph structure, and computation occurs at the nodes, but in Meld, each node runs the same program. Meld's roots are in logic programming; each node generates base facts (representing local connectivity or observations of the node's environment, for example) and inferred facts (which may be inferred from either facts local to the current node or facts from logically connected nodes). Nodes themselves do not influence the connection topology, and migration does not make sense, since all nodes are executing the same program.

Unlike the spider calculus, where the goal is to model computation *in* a graph, Google's Pregel system (Malewicz et al. 2010) and its open source counterpart, Apache Giraph, are designed for computation *on* a graph. Consequently, the communication topology is divorced from the graph structure (any node may communicate with any other); also, unlike the asynchronous computation of process calculi, Pregel processes are synchronous: at computation step n, processes are expected to handle all messages from step $n-1$ and produce messages to be delivered at step $n+1$. Although processes are not directly mobile (each node runs a copy of the same code), nodes are permitted to store state of a user-defined type, so there are few restrictions on the computations that can be performed. These two modeling choices produce a system suitable for distributed algorithms whose input includes a graph, but less so for describing the network graph substrate supporting the distributed computation itself. A similar approach is implemented by the Graph Processing System (GPS) (Salihoglu and Widom 2013), a distributed system designed to run on clusters of machines that provides a domain specific language, Green-Marl (Hong et al. 2012) exposing high-level constructs for implementing parallel analysis algorithms on immutable graphs (e.g., computation of graph centrality measures).

Since the spider calculus is concerned with local transformations of graph structures, we should also mention *graph rewriting*. Connections between graph rewriting and process calculi have been studied in terms of semantics and behavioral theory (Gadducci 2007; König 2000; Gadducci and Montanari 2001). One interesting question is what forms of graph rewriting can be encoded in the spider calculus.

7 Future Work

The natural next step in this work is the development of a theory of contextual equivalence for the spider calculus (some preliminary definitions can be found in Pierce et al. 2010). This will not only provide us with a formal tool to prove the correctness of our programs and encodings, but will also allow us to more deeply understand the expressiveness of our kernel calculus and of its extension with different combinations of primitives.

More broadly, our survey of related work suggests several possible directions of interest, such as relating the spider calculus to graph rewriting or improving safety via typing, co-actions, or more restrictive primitives.

Acknowledgments Many thanks to Davide Sangiorgi, Giorgio Ghelli, Andrew Gordon, David Walker, Michael Kearns, Martin Hofmann, Martín Abadi, and Arnaud Sahuguet for insightful discussions and pointers to proof techniques and related work. Thanks also to the readers of the moca mailing list, who suggested many relevant papers.

References

M. Ashley-Rollman, P. Lee, S. Goldstein, P. Pillai, and J. Campbell. A Language for Large Ensembles of Independently Executing Nodes. In *Proceedings of the 25th International Conference on Logic Programming*, page 280. Springer, 2009.

J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *nature*, 324:4, 1986.

M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. *Lecture Notes in Computer Science*, pages 38–63, 2001.

L. Cardelli. Brane calculi. In *CMSB*, volume 4, pages 257–278. Springer, 2004.

L. Cardelli and P. Gardner. Processes in space. *Theoretical Computer Science*, 431(1):40–55, 2012.

L. Cardelli and A. Gordon. Types for mobile ambients. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 79–92. ACM New York, NY, USA, 1999.

L. Cardelli and A. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for mobile ambients. *Lecture Notes in Computer Science*, 1644:230–239, 1999.

A. Compagnoni, P. Giannini, C. Kim, M. Milideo, and V. Sharma. A calculus of located entities. In *Proceedings of Developments in Computational Models*, 2013.

C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming. *Lecture Notes in Computer Science*, 2395:268–332, 2002.

M. Freedman, E. Freudenthal, D. Mazires, and D. M. Eres. Democratizing content publication with coral. In *In NSDI*, pages 239–252, 2004.

F. Gadducci. Graph rewriting for the pi-calculus. *Mathematical Structures in Computer Science*, 17(3):407–437, 2007.

F. Gadducci and U. Montanari. A concurrent graph semantics for mobile ambients. *Electr. Notes Theor. Comput. Sci.*, 45, 2001.

M. Hennessy. *A distributed pi-calculus*. Cambridge Univ Pr, 2007.

M. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.

S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 349–362. ACM, 2012.

M. John, R. Ewald, and A. M. Uhrmacher. A spatial extension to the¡ i¿ π¡/i¿ calculus. *Electronic notes in theoretical computer science*, 194(3):133–148, 2008.

B. König. A graph rewriting semantics for the polyadic calculus. In *ICALP Satellite Workshops*, pages 451–458, 2000.

M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, page 222. ACM, 2007.

F. Levi and D. Sangiorgi. Mobile safe ambients. *ACM Transactions on Programming Languages and Systems*, 25(1):1–69, 2003.

R. Lublinerman, S. Chaudhuri, and P. Cerny. Parallel programming with object assemblies. In *ACM SIGPLAN Notices*, volume 44, pages 61–80. ACM, 2009.

B. Maggs. Global internet content delivery. In *Proc. 1st IEEE/ACM Int. Symposium on Cluster Computing and the Grid, CCGrid*, 2001.

G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

R. Milner. *Space and Motion of Communicating Agents*. Cambridge Univ Press, 2009.

B. C. Pierce, A. Romanel, and D. Wagner. The Spider Calculus: Computing in active graphs, 2010. Extended version, available from `http://www.cis.upenn.edu/~bcpierce/papers/spider_calculus.pdf`.

C. Priami and P. Quaglia. Beta binders for biological interactions. In *Computational Methods in Systems Biology*, pages 20–33. Springer, 2005.

A. Regev and E. Shapiro. Cellular abstractions: Cells as computation. *Nature*, 419(6905): 343–343, 2002.

A. Regev, E. M. Panina, W. Silverman, L. Cardelli, and E. Shapiro. Bioambients: an abstraction for biological compartments. *Theoretical Computer Science*, 325(1):141–167, 2004.

S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

G. Unel, F. Fischer, and B. Bishop. Answering reachability queries on streaming graphs. *Stream Reasoning*, 2009.

P. Wojciechowski and P. Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, 2000.

From Amber to Coercion Constraints

Didier Rémy
INRIA

Julien Cretin
TrustInSoft

Abstract

Subtyping is a common tool in the design of type systems that finds its roots in the η-expansion of arrow types and the notion of type containment obtained by closing System F by η-expansion. Although strongly related, subtyping and type containment still significantly differ from one another when put into practice. We introduce *coercion constraints* to relate and generalize subtyping and type containment as well as all variants of F-bounded quantification and instance-bounded quantification used for first-order type inference in the presence of second-order types. We obtain a type system with a clearer separation between computational and erasable parts of terms.

1 The different flavors of subtyping

Subtyping in Amber Nowadays, subtyping is a well-understood concept and tool in the design of type systems, but it has not always been so. The origin of subtyping goes back to the 60's when type conversions between base types or type classes were introduced in programming languages, but the first formalization of subtyping is by Reynolds (1980). Subtyping was introduced in the strongly typed functional language Amber by Cardelli (1984). By contrast with simple type conversions between base types, subtyping in Amber can be propagated through arrow types covariantly on the codomains of functions and contravariantly on their domains.

The language Amber had only subtyping but no parametric polymorphism: its typing rules are those of the simply typed λ-calculus extended with a subtyping rule Sub:

$$\frac{\Gamma \vdash a : \tau \qquad \tau \triangleright \sigma}{\Gamma \vdash a : \sigma}\ \textsc{Sub} \qquad \textsc{Bot}\ \bot \triangleright \tau \qquad \textsc{Top}\ \tau \triangleright \top \qquad \frac{\tau' \triangleright \tau \qquad \sigma \triangleright \sigma'}{\tau \to \sigma \triangleright \tau' \to \sigma'}\ \textsc{Arrow}$$

Through this paper we use the notation $\tau \triangleright \sigma$ to mean that τ is a subtype of σ (or, rather, τ *can be coerced to* σ) for homogeneity between different forms of coercions that we will encounter. The subtyping relation is defined by the three rules, Bot, Top, and Arrow. For the sake of brevity we only consider the bottom and top types and arrow types; the language Amber also had records and subtyping between them which played a crucial role in the encoding of objects (but this is not our focus here), as well as recursive types which we introduce later on.

Type containment The notion of type containment was introduced simultaneously by Mitchell (1984, 1988) who considered the closure of System F by η-expansion, called F_η: by definition, a closed term is in F_η if it has an η-expansion in System F. Interestingly, F_η has also a syntactic presentation that is closely related to the notion of subtyping. It extends the typing rules of System F with a subtyping rule similar to, but richer than, the one of Amber. Indeed, System F has polymorphic types, which we write $\forall(\alpha)\,\tau$, and therefore new subtyping rules are needed to describe how subtyping relates with polymorphism:

$$\textsc{All}\quad \frac{\tau \triangleright \sigma}{\forall(\alpha)\,\tau \triangleright \forall(\alpha)\,\sigma} \qquad \textsc{Trans}\quad \frac{\tau \triangleright \sigma \quad \sigma \triangleright \rho}{\tau \triangleright \rho} \qquad \textsc{InstGen}\quad \frac{\bar\beta \notin \mathsf{ftv}(\forall(\bar\alpha)\,\tau)}{\forall(\bar\alpha)\,\tau \triangleright \forall(\bar\beta)\,\tau[\bar\alpha \leftarrow \bar\sigma]} \qquad \textsc{Distrib-R}\quad \frac{\alpha \notin \mathsf{ftv}(\tau)}{\forall(\alpha)\,(\tau \to \sigma) \triangleright \tau \to \forall(\alpha)\,\sigma}$$

The congruence rule for polymorphic types All is unsurprising. Rule Trans is uninteresting but it is needed in this presentation as it does not follow from other rules. The two interesting rules are InstGen and Distrib-R. Rule InstGen allows implicit toplevel type instantiation, as in ML: it means that a polymorphic type can be freely used at any of its instances. By contrast with ML, this rule can also be applied under arrow types at the appropriate variance. We write $\bar\alpha$ for a sequence of type variables (and similarly for types) with the understanding that $\forall(\bar\alpha)\,\tau$ stands for a sequence of quantifiers. Free type variables of τ are written $\mathsf{ftv}(\tau)$. Notice, that although InstGen may generalize type variables that are introduced during type instantiation, it cannot generalize other type variables. Therefore, F_η still need a specific term typing rule for polymorphism introduction. Rule Distrib-R allows a quantifier to be pushed down on the right of arrow types when it does not appear on the left. The original rule allowed α to also occur in τ; then the quantifier must be pushed down on both sides of the arrow, *i.e.* the right-hand side of the coercion becomes $(\forall(\alpha)\,\tau) \to \forall(\alpha)\,\sigma$. This more general rule is however derivable from Distrib-R and the other rules.

The Rule Arrow can be easily explained in F_η by the η-expansion of arrow types. Since F_η is closed by η-expansion, whenever a term a has some type $\tau \to \sigma$ in some context Γ, any type of its η-expansion $\lambda x.\, a\, x$ should also be a type of a. In particular, given $\tau' \triangleright \tau$ and $\sigma \triangleright \sigma'$, we may type the η-expansion by giving the parameter x the type τ' and coerce the occurrence of x as an argument of a to the expected type τ which results in a value of type σ that can be coerced to σ'. Hence, the term a also has type $\tau' \to \sigma'$ in F_η.

What distinguishes type containment from traditional uses of subtyping is the inclusion in the subtyping relation of the implicit instantiation of quantifiers, which by congruence can be applied deeply inside terms. Unfortunately, this theoretical strength turns into a weakness for using F_η in practice since the type containment relation becomes undecidable.

Bounded quantification While F_η extends Amber with polymorphic types and can prove subtyping relations between polymorphic types, it does not allow to make subtyping assumptions about polymorphic type variables. Bounded quantification introduced by Cardelli and Longo (1991) in the language Quest and later in the language $\mathsf{F}_{<:}$ (Cardelli et al. 1994) solves this problem. Polymorphic type variables are introduced with an upper bound $\forall(\alpha <: \tau)\, \sigma$, whose instances are types of the form $\sigma[\alpha \leftarrow \rho]$ where ρ is a subtype of τ. The unbounded quantification of System F can be recovered as the special case where the bound is $\top$. Bounded polymorphism is quite expressive and has been used to model record subtyping and object oriented features.

The most interesting rule in $\mathsf{F}_{<:}$ is subtyping between bounded quantifications:

$$\frac{\Gamma \vdash \tau' \triangleright \tau \qquad \Gamma, \alpha \triangleright \tau' \vdash \sigma \triangleright \sigma'}{\Gamma \vdash \forall(\alpha <: \tau)\, \sigma \triangleright \forall(\alpha <: \tau')\, \sigma'}\ \textsc{Fsub}$$

Notice that subtyping holds only between two types having bounded quantification on both sides. That is, by contrast with F_η, type instantiation is not part of the subtyping relation and is made explicit at the level of terms. A good reason for this choice is to ease typechecking. Still, $\mathsf{F}_{<:}$ in its full generality is undecidable (Pierce 1994), but some restrictions of the rule, for instance where the bounds are identical on both sides, are decidable. Bounded polymorphism has also been extended to F-bounded polymorphism that allows the variable abstracted over to also appear in the bound, which is useful in object-oriented languages. The left-premise of Rule Fsub is then replaced by $\Gamma, \alpha \triangleright \tau' \vdash \alpha \triangleright \tau$.

Bounded polymorphism has been extensively used in many subsequent languages that combine polymorphism and subtyping, and in particular, to explore typechecking of objects (Abadi and Cardelli 1996).

Still, bounded polymorphism remains surprising in several ways. First, its formulation is asymmetric since type variables are introduced with an upper bound but no lower bound: dually, is there a use for lower bounds? Second, type variables have a unique bound: could the same variable have several bounds? Finally, while bounded polymorphism adds to F_η the ability to abstract over subtyping, it does not yet generalize type containment since subtyping holds only for types with the same polymorphic structure; in particular, the implicit instantiation $\forall(\alpha <: \top)\,\tau \triangleright \tau[\alpha \leftarrow \sigma]$ that is allowed in F_η is not permitted in $\mathsf{F}_{<:}$.

Instance-bounded quantification Surprisingly, the absence of lower bounds in $\mathsf{F}_{<:}$ seemed to have not been a practical limitation and the question remained mostly theoretical for more than a decade. The need for lower bounds finally appeared for performing first-order type inference in the presence of second-order types. Predictable, efficient type inference is usually built on a notion of principal types, *i.e.* the ability to capture as a simple type (or type constraint) the set of all possible types of an expression. Lower bounds allow more expressions to have principal types.

Full type inference for System F is known for not having principal types because guessing types of function parameters often leads to an infinite set of solutions. However, if we restrict our ambition to not guess polymorphic types, but just propagate them, which is the goal of $\mathsf{ML^F}$ (Le Botlan and Rémy 2009), there is still a problem to solve, namely the absence of principal solutions.

To illustrate this, consider the function `revapp` defined as $\lambda x.\,\lambda f.\,f\,x$ of type $\forall(\alpha)\,\forall(\beta)\,\alpha \to (\alpha \to \beta) \to \beta$ and its partial application to the identity function `id` of type $\forall(\gamma)\,\gamma^2$ where γ^2 stands for $\gamma \to \gamma$. What should be the type of `revapp id`? The type $\forall(\beta)\,((\forall(\gamma)\,\gamma^2) \to \beta) \to \beta$, say τ_1, obtained by keeping the identity polymorphic? Or the type $\forall(\gamma)\,\forall(\beta)\,(\gamma^2 \to \beta) \to \beta$, say τ_2, obtained by instantiating `id` to γ^2 before the application and generalizing the result of the whole application afterwards? Unfortunately, no other type of the application `revapp id` is an instance of both of these two types in System F. (Nor in F_η! but, conversely, both types have a common instance $\forall(\beta)\,(\forall(\gamma)\,(\gamma^2 \to \beta)) \to \beta$ in F_η.)

This dilemma is solved in $\mathsf{ML^F}$ by introducing lower-bounded (or, rather, instance-bounded) polymorphism $\forall(\alpha :> \tau)\,\sigma$, which reads "*for all α that is an instance of τ, σ.*" The application `revapp id` can then be assigned the type $\forall(\alpha :> \forall(\gamma)\,\gamma^2)\,\forall(\beta)\,(\alpha \to \beta) \to \beta$, which happens to be principal in $\mathsf{ML^F}$. In particular, τ_1 can be obtained by inlining the bound which is permitted by instantiation in $\mathsf{ML^F}$; and τ_2 can be obtained by applying instantiation under the lower bound, generalizing γ afterwards which gives $\forall(\gamma)\,\forall(\alpha :> \gamma^2)\,\forall(\beta)\,(\alpha \to \beta) \to \beta$, and finally inlining the bound.

While in this regard $\mathsf{ML^F}$ appears to be dual of $\mathsf{F}_{<:}$, it is still quite different: $\mathsf{ML^F}$ does not use contravariance of arrow types as both $\mathsf{F}_{<:}$ and F_η do; conversely, $\mathsf{ML^F}$ can freely instantiate polymorphic types, which $\mathsf{F}_{<:}$ cannot do: each of the three languages shares one feature with others but is still missing one of their key features.

$$\begin{array}{rcll}
a, b & ::= & x \mid \lambda x.\, a \mid a\, a & \text{Terms} \\
\tau, \sigma & ::= & \alpha \mid \tau \to \tau \mid \bot \mid \top \mid \forall(\alpha : \kappa)\, \tau \mid \langle\rangle \mid \langle \tau, \tau \rangle \mid \pi_{\mathrm{i}}\, \tau & \text{Types} \\
\kappa & ::= & \star \mid \{\alpha : \kappa \mid \mathrm{P}\} \mid 1 \mid \kappa \times \kappa & \text{Kinds} \\
\mathrm{P} & ::= & (\Delta \vdash \tau) \rhd \tau \mid \exists\, \kappa \mid \top \mid \mathrm{P} \wedge \mathrm{P} & \text{Propositions} \\
\Gamma & ::= & \varnothing \mid \Gamma, (\alpha : \kappa) \mid \Gamma, (x : \tau) & \text{Environments}
\end{array}$$

Figure 1: Syntax

Subtyping constraints The absence of multiple bounds in $\mathsf{F}_{<:}$ is somewhat surprising since ML extended with subtyping, say $\mathsf{ML}_{\leq}$, naturally comes with (and requires the use of) subtyping constrains that mix arbitrary upper and lower bounds (Odersky et al. 1999). It uses constrained type schemes of the form $\forall(\alpha|C)\, \tau$ where C is a set of arbitrary but coherent subtyping constraints between simple types. Can we extend subtyping constraint to first-class polymorphic types?

2 The language F_{cc}

We have seen several type systems with different combinations of subtyping features, but none of them supersedes all others. In particular, F_{η}, $\mathsf{F}_{<:}$, $\mathsf{ML^F}$, and $\mathsf{ML}_{\leq}$ are pairwise incomparable. We now describe an extension of System F with coercion constraints, called F_{cc}, that combines all features together so that each of the languages above becomes a (still interesting) subset of F_{cc}. Here, we present a small subset of the language just to carry the main ideas underlying its design. We refer the reader to (Cretin and Rémy 2014b) for technical details.

Terms The language F_{cc} is implicitly typed for reasons that will be explained later—but this happens to be an advantage for conciseness: terms are just those of the untyped λ-calculus, whose notations are reminded in Fig. 1.

Although we focus on the syntactic presentation of F_{cc} hereafter, we assume that its semantics is given by full β-reduction. This is to build its type system on solid ground, without taking advantage of the evaluation strategy: it prevents from sweeping errors under λ's just because their evaluation is delayed. Full β-reduction also models reduction of open terms. A practical language based on F_{cc} will eventually have a call-by-name or call-by-value evaluation strategy, which being a subset of full β-reduction, will remain sound.

Types, Kinds, and Propositions Types, defined on Figure 1, are simple types (type variables, arrow types, top, and bottom types) extended with con-

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{TermVar} \qquad \frac{\Gamma \vdash \tau : \star \qquad \Gamma, (x : \tau) \vdash a : \sigma}{\Gamma \vdash \lambda x.\, a : \tau \to \sigma}\ \textsc{TermLam} \qquad \frac{\Gamma \vdash a : \tau \to \sigma \qquad \Gamma \vdash b : \tau}{\Gamma \vdash a\, b : \sigma}\ \textsc{TermApp}$$

$$\frac{\Gamma, \Delta \vdash a : \tau \qquad \Gamma \vdash (\Delta \vdash \tau) \triangleright \sigma}{\Gamma \vdash a : \sigma}\ \textsc{TermCoer}$$

Figure 2: Term typing rules

strained polymorphic types of the form $\forall(\alpha : \kappa)\, \tau$ where κ is a kind that restricts the set of types in which α may range.

The kind $\star$ is that of ordinary types, *e.g.* to form arrow types. The interesting kind is $\{\alpha : \kappa \mid \mathrm{P}\}$—the kind of types α of kind κ that satisfy the proposition P.

The grammars of types and kinds also include type tuples[1] classified by tuple kinds. Namely, type tuples are constructed from the empty tuple $\langle\rangle$ of kind 1 and pairs of types $\langle\tau_1, \tau_2\rangle$ of kind $\kappa_1 \times \kappa_2$ when τ_i has kinds κ_i; and type projection $\pi_\mathrm{i}\, \tau$ to return the i's component of kind κ_i of a tuple type of kind $\kappa_1 \times \kappa_2$. Tuple types are useful for capturing multiple binders but are not technically difficult: the main technical change is to consider types equal modulo the projection of tuples and closing equality by equivalence and congruence for all syntactical constructs. Thus, we will ignore tuples (which are grayed in Figure 1) in the rest of the technical overview.

Propositions are used to restrict sets of types. The most useful proposition is the coercion proposition $(\Delta \vdash \tau) \triangleright \sigma$ which states that there exists a coercion from type τ in some context extended with Δ, to type σ. For now, one may just consider the particular case of *simple* coercions where Δ is empty, in which case the proposition is abbreviated as $\tau \triangleright \sigma$ and just means that type τ can be coerced to type σ. The general case will be explained together with the term typing rules.

The proposition $\exists\, \kappa$ asserts that the kind κ is coherent (intuitively, that it is inhabited, but this is relative to its typing context). It is interesting that coherence can be internalized as a proposition. The proposition $\top$ is the true proposition and the proposition $\mathrm{P}_1 \wedge \mathrm{P}_2$ is the conjunction of P_1 and P_2.

Finally, typing environments Γ bind type variables to kinds and program variables to types. The letter Δ is used to range over environments that only bind type variables.

Term typing judgment ($\Gamma \vdash a : \tau$) Typing rules, given in Figure 2 are almost as simple as in Amber: they contain the typing rules of the simply typed

[1]Not to be confused with the type of term tuples, which we do not include in this core version.

$$\frac{\Gamma, (\alpha : \kappa) \Vdash \mathrm{P} \qquad \Gamma \vdash \tau : \kappa \qquad \Gamma \vdash \mathrm{P}[\alpha \leftarrow \tau]}{\Gamma \vdash \tau : \{\alpha : \kappa \mid \mathrm{P}\}}\ \textsc{TypePack} \qquad \frac{\Gamma \vdash \tau : \{\alpha : \kappa \mid \mathrm{P}\}}{\Gamma \vdash \tau : \kappa}\ \textsc{TypeUnpack}$$

Figure 3: Type judgment relation (excerpt)

λ-calculus plus the coercion typing rule, which differs from the Amber Rule in two important ways. First, the coercion judgment depends on the context Γ, as in $\mathsf{F}_{<:}$, so that coercion assumptions in the context Γ can be used to prove a coercion judgment such as $\Gamma \vdash \tau \rhd \sigma$. More importantly, coercions are of the more general form $(\Delta \vdash \tau) \rhd \sigma$ rather than just $\tau \rhd \sigma$. As we can see in Rule TermCoer, the context Δ contains additional type variable bindings that are added to the context Γ of the premise under which the coerced term must have type τ. The typical use of this generalization is in the judgment $\Gamma \vdash (\alpha : \kappa \vdash \tau) \rhd \forall(\alpha : \kappa)\,\tau$, which holds whenever κ is coherent and $\forall(\alpha : \kappa)\,\tau$ is well-formed in Γ (see Rule CoerGen below). As a consequence, the rule for polymorphism introduction

$$\frac{\Gamma \vdash \exists \kappa \qquad \Gamma, (\alpha : \kappa) \vdash a : \tau}{\Gamma \vdash a : \forall(\alpha : \kappa)\,\tau}\ \textsc{TermGen}$$

is derivable by TermCoer and CoerGen. It is remarkable that polymorphism introduction (as well as all erasable features of the type system) can be handled purely as a coercion typing rule and need no counterpart in term typing rules.

Well-formedness rules ($\Gamma \Vdash \kappa$ and $\Gamma \Vdash \mathrm{P}$) The judgments $\Gamma \Vdash \kappa$ and $\Gamma \Vdash \mathrm{P}$ state well-formedness of kinds and propositions. Besides syntactical checks, they are recursively scanning their subexpressions for all occurrences of coercion propositions $(\Delta \vdash \tau) \rhd \sigma$ to ensure that Δ, τ, and σ are well-typed, as described by the following rule:

$$\frac{\Gamma \vdash \Delta \qquad \Gamma, \Delta \vdash \tau : \star \qquad \Gamma \vdash \sigma : \star}{\Gamma \Vdash (\Delta \vdash \tau) \rhd \sigma}$$

In particular, the auxiliary judgment $\Gamma \vdash \Delta$, which treats Δ as a telescope, means that each binding $\alpha : \kappa$ of Δ is well-formed in the typing context that precedes it, which also implies that the kind κ is coherent in any such binding relatively to its typing context.

Kinding judgment ($\Gamma \vdash \tau : \kappa$) An excerpt of kinding rules is given in Figure 3, but most rules have been omitted. Rule TypeUnpack states that whenever a type τ is known to be of a constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$, it is also of the kind κ, indeed.

$$\textsc{PropRes}\quad \frac{\Gamma \vdash \tau : \{\alpha : \kappa \mid \mathrm{P}\}}{\Gamma \vdash \mathrm{P}[\alpha \leftarrow \tau]} \qquad\qquad \textsc{PropExi}\quad \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \exists\, \kappa}$$

Figure 4: Proposition judgment relation (excerpt)

$$\textsc{CoerRefl}\quad \frac{\Gamma \vdash \tau : \star}{\Gamma \vdash \tau \triangleright \tau} \qquad \textsc{CoerTop}\quad \frac{\Gamma \vdash \tau : \star}{\Gamma \vdash \tau \triangleright \top} \qquad \textsc{CoerBot}\quad \frac{\Gamma \vdash \tau : \star}{\Gamma \vdash \bot \triangleright \tau}$$

$$\textsc{CoerTrans}\quad \frac{\Gamma, \Delta' \vdash (\Delta \vdash \tau) \triangleright \tau' \qquad \Gamma \vdash (\Delta' \vdash \tau') \triangleright \tau''}{\Gamma \vdash (\Delta', \Delta \vdash \tau) \triangleright \tau''}$$

$$\textsc{CoerWeak}\quad \frac{\Gamma \vdash (\Delta \vdash \tau) \triangleright \sigma}{\Gamma \vdash \tau \triangleright \sigma} \qquad \textsc{CoerArr}\quad \frac{\Gamma \vdash \tau : \star \qquad \Gamma, \Delta \vdash \tau \triangleright \tau' \qquad \Gamma \vdash (\Delta \vdash \sigma') \triangleright \sigma}{\Gamma \vdash (\Delta \vdash \tau' \to \sigma') \triangleright \tau \to \sigma}$$

$$\textsc{CoerGen}\quad \frac{\Gamma \vdash \exists\, \kappa \qquad \Gamma, (\alpha : \kappa) \vdash \tau : \star}{\Gamma \vdash (\alpha : \kappa \vdash \tau) \triangleright \forall(\alpha : \kappa)\, \tau} \qquad \textsc{CoerInst}\quad \frac{\Gamma, (\alpha : \kappa) \vdash \tau : \star \qquad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash \forall(\alpha : \kappa)\, \tau \triangleright \tau[\alpha \leftarrow \sigma]}$$

Figure 5: Coercion judgment relation

TypePack shows that, conversely, knowing that τ has kind κ, one must still prove that $\mathrm{P}[\alpha \leftarrow \tau]$ is satisfied to be able to consider that τ has the constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$.

Proposition judgment ($\Gamma \vdash \mathrm{P}$) The two most interesting rules for propositions are given in Figure 4. Rule PropRes means that a type of a constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$ satisfies the proposition P (where τ has been substituted for α). Rule PropExi means that one must exhibit a type τ of kind κ to ensure that the kind κ is coherent in its typing context Γ.

Coercion propositions ($\Gamma \vdash (\Delta \vdash \tau) \triangleright \sigma$) We have separated the rules for coercions in Figure 5 although coercions are just a particular case of propositions. Some rules are slightly obfuscated by the coercion typing context Δ that has to be added in some premises. These rules can first be read in the particular case where Δ is the empty context, so we have grayed Δ to help with this reading.

Rule CoerRefl, CoerTop, and CoerBot are obvious and can be skipped. Rule CoerTrans is also standard—the Δ's just need to be appropriately concatenated.

Rule CoerWeak implements a form of weakening: it tells that if any term of typing[2] $\Gamma, \Delta \vdash \tau$ has typing $\Gamma \vdash \sigma$, then any term of typing $\Gamma \vdash \tau$ also has typing $\Gamma \vdash \sigma$. Weakening is required as it would not be derivable from the other rules if

[2]We say that a term a has typing $\Gamma \vdash \tau$ if the typing judgment $\Gamma \vdash a : \tau$ holds.

we removed it from the definition.

Rule CoerArr is the usual contravariant rule for arrows when Δ is empty. Otherwise, the rule can be understood by looking at the η-expansion context $\lambda x.\,([]\,x)$ for the arrow type. Placing a term with typing $\Gamma, \Delta \vdash \tau' \to \sigma'$ in the hole, we may give $\lambda x.\,([]\,x)$ the typing $\Gamma \vdash \tau \to \sigma$ provided a coercion of type $\Gamma, \Delta \vdash \tau \triangleright \tau'$ is applied around x. The result of the application has typing $\Gamma, \Delta \vdash \sigma'$ which can in turn be coerced to $\Gamma \vdash \sigma$ if there exists a coercion of type $\Gamma \vdash (\Delta \vdash \sigma') \triangleright \sigma$. Thus, the η-expansion has typing $\Gamma \vdash \tau \to \sigma$.

Notice that CoerArr is the only η-expansion rule, since the arrow is the only computational type constructor in our core language. For each computational type constructor that we would add to the language, we would need a corresponding η-expansion coercion rule. Erasable type constructors need not have an η-expansion coercion rule, since these are derivable as their introduction and elimination rules are already coercions.

Rule CoerGen implements type generalization, but as a coercion rule. It says that $\Gamma \vdash (\alpha : \kappa \vdash \tau) \triangleright \forall(\alpha : \kappa)\,\tau$ is a valid coercion as long as kind κ is coherent in Γ and type τ has kind $\star$ in $\Gamma, (\alpha : \kappa)$. This coercion makes TermGen derivable as explained above.

Similarly, CoerInst is the counterpart of type instantiation coercion in F_η: it says that $\Gamma \vdash \forall(\alpha : \kappa)\,\tau \triangleright \tau[\alpha \leftarrow \sigma]$ is a valid coercion as long as τ has kind $\star$ in $\Gamma, (\alpha : \kappa)$ and σ has kind κ in Γ. Since CoerGen is a coercion rule, we do not need the more involved version of F_η that performs generalization afterwards.

Notice that there is no counterpart to Distrib-R since it is derivable by a combination of type generalization, type instantiation, and η-expansion.

Adding recursive types and coinduction

The full language F_{cc} also has recursive types. These are indeed present in the language Amber, $\mathsf{F}_{<:}$, *etc.* as they are unavoidable in a programming language.

We thus extend the grammar of types with the production $\mu\alpha\,\tau$, with the usual restriction on well-foundedness of recursive types that we will not detail here. We write $\alpha \mapsto \tau : \mathsf{wf}$ to mean that the function $\alpha \mapsto \tau$ is well-founded and can be used to form the recursive type $\mu\alpha\,\tau$.

Since types are implicit, we chose equi-recursive types, *i.e.* the equality of recursive types is witnessed by coercions, which are implicit. We add two coercions to witness folding and unfolding of recursive types:

$$\textsc{CoerUnfold}\quad \frac{\alpha \mapsto \tau : \mathsf{wf} \qquad \Gamma, (\alpha : \star) \vdash \tau : \star}{\Gamma \vdash \mu\alpha\,\tau \triangleright \tau[\alpha \leftarrow \mu\alpha\,\tau]} \qquad\qquad \textsc{CoerFold}\quad \frac{\alpha \mapsto \tau : \mathsf{wf} \qquad \Gamma, (\alpha : \star) \vdash \tau : \star}{\Gamma \vdash \tau[\alpha \leftarrow \mu\alpha\,\tau] \triangleright \mu\alpha\,\tau}$$

To reason with recursive types we also add coinduction in propositions. In order to do so, we change the judgment $\Gamma \vdash \mathrm{P}$ to $\Gamma; \Theta \vdash \mathrm{P}$ where Θ is used to collect coinductive hypotheses. We introduce two new rules PropFix and PropVar to allow reasoning by coinduction.

$$\frac{\Gamma \Vdash \mathrm{P} \qquad \Gamma; \Theta, \mathrm{P} \vdash \mathrm{P}}{\Gamma; \Theta \vdash \mathrm{P}}\ \textsc{PropFix} \qquad \frac{\mathrm{P}^{\dagger} \in \Theta}{\Gamma; \Theta \vdash \mathrm{P}}\ \textsc{PropVar} \qquad \frac{\Gamma, \Delta; \Theta^{\dagger} \vdash \tau \triangleright \tau' \qquad \Gamma; \Theta^{\dagger} \vdash (\Delta \vdash \sigma') \triangleright \sigma}{\Gamma; \Theta \vdash (\Delta \vdash \tau' \to \sigma') \triangleright \tau \to \sigma}\ \textsc{CoerArr}$$

Rule PropFix allows the judgment $\Gamma; \Theta \vdash \mathrm{P}$ to be proved under the additional coinductive hypothesis P (provided P is well-formed in Γ). Of course, coinductive hypotheses must be guarded before they can be used. This is implemented by tagging guarded propositions in Θ that are then ready for coinductive use with $\dagger$. Hence, rule PropVar can prove P only if P appears guarded, *i.e.* as $\mathrm{P}^{\dagger}$, in the coinductive context Θ.

Premises of the rule CoerArr use only subterms of the arrow type, so the rule acts as a guard for all coinductive assumptions, therefore all propositions of Θ are tagged in its premises. This is only the case of expansion rules for computational type constructors, which have a counterpart in terms. Currently, CoerArr is the only such rule. All other rules just transport Θ unchanged from the premise to the conclusion.

Reasoning by induction makes the following standard rules for equi-recursive types derivable (we write $\tau \triangleleft\!\triangleright \sigma$ for a pair of simple coercions $\tau \triangleright \sigma$ and $\sigma \triangleright \tau$):

$$\frac{\alpha \mapsto \sigma : \mathsf{wf} \qquad \Gamma; \Theta \vdash (\tau_i \triangleleft\!\triangleright \sigma[\alpha \leftarrow \tau_i])^{i \in \{1,2\}}}{\Gamma; \Theta \vdash \tau_1 \triangleright \tau_2}\ \textsc{CoerPeriod} \qquad \frac{\Gamma, (\alpha, \beta, \alpha \triangleright \beta); \Theta \vdash \tau \triangleright \sigma}{\Gamma; \Theta \vdash \mu\alpha\,\tau \triangleright \mu\beta\,\sigma}\ \textsc{CoerEtaMu}$$

Interestingly, the proof for CoerPeriod requires reinforcement of the coinduction hypothesis since we need $\tau_1 \triangleleft\!\triangleright \tau_2$ and not just $\tau_1 \triangleright \tau_2$ in the coinduction hypothesis.

3 Strength and weaknesses of F_{cc}

Soundness The type system of F_{cc} is sound for the full β-reduction semantics. Type soundness is not proved syntactically for reasons explained next, but semantically, by interpreting types as sets of terms. As a consequence of the presence of general recursive types (*i.e.* we do not restrict to positive recursion), we use a step-indexed technique. Unfortunately, the usual technique of Appel and McAllester (2001) does not apply to a full β-reduction setting. We propose a new technique where indexes that are traditionally outside terms are placed directly on terms and are transformed during reduction. See (Cretin and Rémy 2014a) for details.

Termination As a sanity check, the reduction of well-typed programs always terminates in the absence of recursive types and coinduction.

(Lack of a proof of) Subject reduction The reason not to do a syntactic proof is that we do not know how to prove subject reduction for F_{cc}. The problem is that the type system is either too expressive or too weak: it allows to type programs that are indeed safe, but with involved non local coercion constraints that cannot be easily traced during reduction.

Doing a syntactic proof would amount to having an explicitly typed version of the language and reduction rules for explicitly typed terms that preserve well-typedness; moreover, reduction of explicitly typed terms must be in bisimulation with the reduction of implicitly typed ones. The main obstacle is that, in the general case, abstract coercions may appear in the middle of a redex. Explaining why this is difficult in the general case is a bit tricky, as solving one issue immediately raises another one—see Cretin and Rémy (2012) for details. Quite interestingly, this configuration can never occur when we restrict to abstract coercions that are parametric in either their domain or their codomain, *i.e.* coercions of the form $\alpha \triangleright \tau$ or $\tau \triangleright \alpha$. Remarkably, these two subcases coincide with bounded quantification and instance-bounded quantification. Under this restriction, we can design an explicitly typed language that enjoys subject reduction (Cretin and Rémy 2012).

(Lack of a good) Surface language Since the type system is implicit, it is undecidable, of course. What is a good surface language for F_{cc} is still an open question. It is always possible to be fully explicit by introducing term syntax for describing typing derivations in source terms, hence turning type inference into an easy checking process. However, this would not only contain type annotations but also full coercion bodies (information which is typically inferred in languages such as $\mathsf{F}_{<:}$) and of coercion coherence proofs (information which is usually obtained by construction). Programming at this level of detail would be too cumbersome for the programmer. Notice however, that F_η already suffers from a similar problem, since its coercion relation is undecidable.

This issue can be addressed in two directions. Remaining within F_{cc}, we may apply partial type inference techniques and hope that sufficient type and coercion information can be reconstructed. It would also be interesting to look for subsystems of F_{cc} that compromise expressiveness for a smaller amount of annotations. The restriction to coercions that are parametric in either their domain or their codomain is one such solution. Are there other sweet spots?

Expressiveness As announced earlier, F_{cc} contains F_{η}, $\mathsf{F}_{<:}$, ML^{F}, and $\mathsf{ML}_{\leq}$ as sublanguages. This confirms that all their features are compatible and can safely be combined together.

Writing $\forall(\alpha \mid \mathrm{P})\,\tau$ for $\forall(\alpha : \{\alpha : \star \mid \mathrm{P}\})\,\tau$, bounded quantification and instance-bounded quantification can be encoded in F_{cc} as $\forall(\alpha \mid \alpha \triangleright \tau)\,\sigma$ and $\forall(\alpha \mid \tau \triangleright \alpha)\,\sigma$. It is then not difficult to see that the typing rules of $\mathsf{F}_{<:}$ (including F-Bounded quantification) and of ML^{F} are derivable.

The notation is genealized to bindings, writing $(\overline{\alpha} \mid \mathrm{P})$ for $(\alpha : \{\alpha : \star^n \mid \mathrm{P}\})$ where n is the size of $\bar{\alpha}$ and, by abuse of notation, let us write α_i for $\pi_{\mathrm{i}}\,\alpha$ when α_i is the i' component of a sequence $\bar{\alpha}$. Then, a constrained typing judgment $\Gamma \vdash e : \tau \mid C$ in $\mathsf{ML}_{\leq}$ can be seen as the F_{cc} judgment $(\bar{\alpha} \mid C), \Gamma \vdash e : \tau$ where $\overline{\alpha}$ are free variables of Γ, C, and τ.

We have only presented a core subset of F_{cc}. The full language (Cretin and Rémy 2014a) also contains products. Sum types are could be also be easily added. Existential types can be emulated by their CPS encoding. The language of propositions contains polymorphic propositions $\forall(\alpha : \kappa)\,\mathrm{P}$, and could also be enriched with other propositions.

Incoherent coercions In the subset of F_{cc} described above, coercions are always coherent relative to the typing context in which they are used. More precisely, when an expression a has type $\forall(\alpha : \kappa)\,\tau$ in Γ, it is always the case that $\Gamma \vdash \exists\,\kappa$. This is necessary because type abstraction does not have any counterpart in terms and, in particular, does not block the evaluation of a which may proceed immediately. Without coherence, one could abstract over the absurd kind constraint $\top \triangleright \bot$ and be able to type any program.

However, there are situations in which abstraction over incoherent coercions would be useful. First, the coercion may be coherent only for *some* instances of the typing context. This is typically the case in the presence of GADTs. Second, coherence at the abstraction point is often harder to prove than at instantiation points where types have been specialized. For these reasons, we have also extended F_{cc} with incoherent coercion abstractions. Of course, this abstraction must now block the evaluation and therefore have a counterpart in terms. Interestingly, this allows to model GADTs as incoherent coercions. Incoherent abstractions are indeed used in FC (Weirich et al. 2011), the intermediate language of Haskell. See (Cretin and Rémy 2014a) for further details.

Conclusions

We have given a tour of coercion constraints and shown how they can be used to explain several type systems that had been designed separated for different purposes, but all around some variations on the notion of subtyping. There are an amazingly large number of interesting works on subtyping, many of which actually initiated by Cardelli, and we could unfortunately just select a few citations among all the relevant ones. As for coercions, the idea is not at all new. There are in fact several notions of coercions and many works on type systems have already used coercions as a tool or studied them on their own. So, many more references could have been included here as well. We refer the reader to (Cretin and Rémy 2014a) for a more thorough treatment of related works.

Coercion constraints can factor many type features of programming languages. This allows sharing an important part of their meta-theoretical studies, such as type soundness. It also opens the door to new combinations of features. Besides, it helps separate the computational and erasable parts of type systems. We thus believe that they are an interesting framework for designing and studying type systems.

However, F_{cc} still lacks a good surface language for the programmer as well an explicitly-typed calculus to be used as an internal language in a compiler. While partial type inference techniques could be used for the surface language, finding an explicit version of coercion constraints that enjoys subject reduction without sacrificing expressiveness seems much more challenging.

References

M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996. ISBN 0387947752.

A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems.*, 23(5), Sept. 2001.

L. Cardelli. A semantics of multiple inheritance. In *Proc. Of the International Symposium on Semantics of Data Types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc. ISBN 3-540-13346-1. URL `http://dl.acm.org/citation.cfm?id=1096.1098`.

L. Cardelli and G. Longo. A semantic basis for quest. *J. Funct. Program.*, 1(4): 417–458, 1991.

L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of system f with subtyping. *Information and Computation*, 109(1/2):4–56, 1994.

J. Cretin and D. Rémy. On the power of coercion abstraction. In *Proceedings of the annual symposium on Principles Of Programming Languages*, 2012.

J. Cretin and D. Rémy. System F with Coercion Constraints. Rapport de recherche RR-8456, INRIA, Jan. 2014a. URL `http://hal.inria.fr/hal-00934408`.

J. Cretin and D. Rémy. System F with Coercion Constraints. In *Logic In Computer Science (LICS)*, July 2014b. To appear.

D. Le Botlan and D. Rémy. Recasting MLF. *Information and Computation*, 207 (6), 2009.

J. C. Mitchell. Type inference and type containment. In *Proc. Of the International Symposium on Semantics of Data Types*, pages 257–277, New York, NY, USA, 1984. Springer-Verlag New York, Inc. ISBN 3-540-13346-1. URL `http://dl.acm.org/citation.cfm?id=1096.1106`.

J. C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 2/3(76), 1988.

M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

B. C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994.

J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258, Berlin, 1980. Springer-Verlag.

S. Weirich, D. Vytiniotis, S. Peyton Jones, and S. Zdancewic. Generative type abstraction and type-level computation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'11, 2011.

Classical π

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

Maria Grazia Vigliotti
Adelard
3-11 Pine Street
London ECR1, UK

Abstract

In this paper we provide the a summary of the work carried out in collaboration with Luca on the relation between classical logic and the π-calculus, and outline a different direction the work has recently taken.

Introduction

Our collaboration with Luca Cardelli started when he was visiting professor at the Department of Computing at Imperial College. At that time the Theory Group organised weekly informal seminars, and after the presentation of our first results on the classical sequent calculus $\mathcal{X}$ (van Bakel and Lescanne 2008) at one of these seminars, Luca said "We need to talk." This started a very interesting collaboration which led to several philosophical and technical discussions on what computation and classical logic have in common and how calculi based on the latter can be modelled in the π-calculus.

The general idea of modelling classical logic in the π-calculus was studied before our collaboration with Luca started, but preceding research focussed mainly on linear logic. When $\mathcal{X}$ was introduced, it gave a means to directly study the computational content of Gentzen's sequent calculus LK (Gentzen 1935) and of proofs with cut-elimination. The clear presence of non-determinism and reduction through the exchange of names in $\mathcal{X}$ made us wonder if there could be a meaningful translation into π-calculus.

The main goal of our research was two-fold: on one side we wanted to better understand the computational content of proofs; on the other side, we wanted to explore the computational strength of the π-calculus. Was it possible that the normalised proofs of classical logic could be magically mapped into the π-calculus, a formal, abstract tool built to understand concurrent computation and name passing?

Proofs in the sequent calculus can be optimised (normalised) by a well-known procedure called *cut-elimination*, which eliminates an intermediate assumption. This normally comes at the cost that proofs become larger and slightly more complex. If we take the position of viewing the various steps of optimising a proof as computational steps, what kind of computation would we get? We could say that $\mathcal{X}$ itself of course is already a way of giving computational meaning to classical proofs, but to appreciate this the reader would need to be fluent in the reduction rules of $\mathcal{LK}$, which by themselves do not directly exhibit any familiar notion of computation. So, can we perhaps understand this better after mapping $\mathcal{X}$ into the π-calculus?

In order to answer that last question, we limited $\mathcal{X}$ to a variant called $\mathcal{LK}$ (i.e. $\mathcal{X}$ without activation, see (van Bakel and Lescanne 2008)). It took several attempts to define the 'correct' translation from $\mathcal{LK}$ to π; to solve several problems we realised that the most elegant way to proceed was to add pairing to the π-calculus. Once we fully understood the requirements of the translation, we were able to establish the properties that were needed; through a successful translation of $\mathcal{LK}$ into the π-calculus we have a different way of understanding the computational content of classical logic. To show that our translation preserved the logical aspect, we also had to define a novel type system for π: thus the idea of *Classical* π was born.

1 The calculus $\mathcal{LK}$

In this section, we first provide a brief discussion of the sequent calculus $\mathcal{LK}$ (a variant of $\mathcal{X}$), which is a term calculus that enjoys the Curry-Howard isomorphism for Gentzen's classical sequent calculus LK. $\mathcal{LK}$ features *two* separate categories of 'connectors', *plugs* and *sockets*, that act as output and input channels, respectively, and is defined without any notion of substitution or application. This is one of the greatest novelties of the calculus.

Definition 1 (Syntax) $\mathcal{LK}$-terms are defined by the following syntax, where Roman characters range over the infinite set of *sockets*, and Greek characters over the infinite set of *plugs*, collectively called *connectors*:

$$\begin{array}{rcccccccc} P,Q & ::= & \langle x{\cdot}\beta\rangle & \mid & \widehat{z}P\widehat{\alpha}{\cdot}\beta & \mid & P\widehat{\alpha}\,[x]\,\widehat{z}Q & \mid & P\widehat{\alpha}\dagger\widehat{z}Q \\ & & \textit{capsule} & & \textit{export} & & \textit{import} & & \textit{cut} \end{array}$$

The $\hat{\cdot}$ symbolises that the connector underneath is bound in the adjacent term.

These terms act as witnesses of the logical rules (Ax), $(\to R)$, $(\to L)$, and (cut), respectively. We now define a notion of type assignment for $\mathcal{LK}$:

Definition 2 (Type assignment for $\mathcal{LK}$) 1. *(Implicative) types* are defined by the grammar $A, B ::= \varphi \mid A{\to}B$, where φ is a basic type of which there are countably many. A *context of sockets* Γ is a mapping from sockets to types (written as a finite set of *statements* of the shape $x{:}A$). Contexts of *plugs* Δ are defined similarly.

2. *Type judgements* for $\mathcal{LK}$ are expressed via a ternary relation $P : \Gamma \vdash_{\text{LK}} \Delta$; we say that P is the *witness* of this judgement.

3. *Type assignment for* $\mathcal{LK}$ is defined by the following rules:

$$(cap) : \frac{}{\langle x{\cdot}\beta\rangle \mathrel{:\!\cdot} \Gamma, x{:}A \vdash \beta{:}A, \Delta} \qquad (cut) : \frac{P \mathrel{:\!\cdot} \Gamma \vdash \alpha{:}A, \Delta \quad Q \mathrel{:\!\cdot} \Gamma, z{:}A \vdash \Delta}{P\widehat{\alpha} \dagger \widehat{z}Q \mathrel{:\!\cdot} \Gamma \vdash \Delta}$$

$$(exp) : \frac{P \mathrel{:\!\cdot} \Gamma, z{:}A \vdash \alpha{:}B, \Delta}{\widehat{z}P\widehat{\alpha}{\cdot}\beta \mathrel{:\!\cdot} \Gamma \vdash \beta{:}A{\to}B, \Delta} \qquad (imp) : \frac{P \mathrel{:\!\cdot} \Gamma \vdash \alpha{:}A, \Delta \quad Q \mathrel{:\!\cdot} \Gamma, z{:}B \vdash \Delta}{P\widehat{\alpha}\,[x]\,\widehat{z}Q \mathrel{:\!\cdot} \Gamma, x{:}A{\to}B \vdash \Delta}$$

The reduction rules for $\mathcal{LK}$ are directly inspired by Gentzen's *cut*-elimination rules in LK. It is possible to define proof reduction in many ways; Gentzen decided to consider the simplest contractions, and considered only the last rule applied in the two sub-derivations of *cut*s. Following Gentzen's approach, $\mathcal{LK}$'s term rewriting rules explain in detail how *cuts* are propagated through terms to be eventually evaluated at the level of *capsules*, where renaming takes place. Reduction is defined by specifying both the interaction between well-connected basic syntactic structures, and how to deal with propagating nodes to points in the term where they can interact.

Reduction depends on the auxiliary notion of introduced connectors.

Definition 3 (Reduction on $\mathcal{LK}$) Introduction :

P introduces α : $P = \widehat{x}Q\widehat{\beta}{\cdot}\alpha$ and α not free in Q, or $P = \langle x{\cdot}\alpha\rangle$.

P introduces x : $P = Q\widehat{\beta}\,[x]\,\widehat{y}R$ with x not free in Q or R, or $P = \langle x{\cdot}\alpha\rangle$.

Logical Rules : (α and x are introduced).

$$\begin{array}{rrcl}
(cap) : & \langle y{\cdot}\alpha\rangle\widehat{\alpha} \dagger \widehat{x}\langle x{\cdot}\beta\rangle & \to & \langle y{\cdot}\beta\rangle \\
(exp) : & (\widehat{y}P\widehat{\beta}{\cdot}\alpha)\widehat{\alpha} \dagger \widehat{x}\langle x{\cdot}\gamma\rangle & \to & \widehat{y}P\widehat{\beta}{\cdot}\gamma \\
(imp) : & \langle y{\cdot}\alpha\rangle\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\beta}\,[x]\,\widehat{z}R) & \to & Q\widehat{\beta}\,[y]\,\widehat{z}R \\
(exp\text{-}imp) : & (\widehat{y}P\widehat{\beta}{\cdot}\alpha)\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R) & \to & \begin{cases} Q\widehat{\gamma} \dagger \widehat{y}(P\widehat{\beta} \dagger \widehat{z}R) \\ (Q\widehat{\gamma} \dagger \widehat{y}P)\widehat{\beta} \dagger \widehat{z}R \end{cases}
\end{array}$$

Left propagation : (α not introduced)

$$\begin{array}{rlcll}
(\mathit{cap}\dagger): & \langle y{\cdot}\beta\rangle\widehat{\alpha}\dagger\widehat{x}P & \to & \langle y{\cdot}\beta\rangle & (\beta\neq\alpha)\\
(\mathit{exp\text{-}out}\dagger): & (\widehat{y}Q\widehat{\beta}{\cdot}\alpha)\widehat{\alpha}\dagger\widehat{x}P & \to & (\widehat{y}(Q\widehat{\alpha}\dagger\widehat{x}P)\widehat{\beta}{\cdot}\gamma)\widehat{\gamma}\dagger\widehat{x}P & (\gamma\ \mathit{fresh})\\
(\mathit{exp\text{-}in}\dagger): & (\widehat{y}Q\widehat{\beta}{\cdot}\gamma)\widehat{\alpha}\dagger\widehat{x}P & \to & \widehat{y}(Q\widehat{\alpha}\dagger\widehat{x}P)\widehat{\beta}{\cdot}\gamma & (\gamma\neq\alpha)\\
(\mathit{imp}\dagger): & (Q\widehat{\beta}\,[z]\,\widehat{y}R)\widehat{\alpha}\dagger\widehat{x}P & \to & (Q\widehat{\alpha}\dagger\widehat{x}P)\widehat{\beta}\,[z]\,\widehat{y}(R\widehat{\alpha}\dagger\widehat{x}P) & \\
(\mathit{cut}\dagger): & (Q\widehat{\beta}\dagger\widehat{y}R)\widehat{\alpha}\dagger\widehat{x}P & \to & (Q\widehat{\alpha}\dagger\widehat{x}P)\widehat{\beta}\dagger\widehat{y}(R\widehat{\alpha}\dagger\widehat{x}P) &
\end{array}$$

Right propagation : (x not introduced).

$$\begin{array}{lllll}
(\dagger\mathit{cap}): & P\widehat{\alpha}\dagger\widehat{x}\langle y{\cdot}\beta\rangle & \to & \langle y{\cdot}\beta\rangle & (y\neq x)\\
(\dagger\mathit{exp}): & P\widehat{\alpha}\dagger\widehat{x}(\widehat{y}Q\widehat{\beta}{\cdot}\gamma) & \to & \widehat{y}(P\widehat{\alpha}\dagger\widehat{x}Q)\widehat{\beta}{\cdot}\gamma & \\
(\dagger\mathit{imp\text{-}out}): & P\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\beta}\,[x]\,\widehat{y}R) & \to & P\widehat{\alpha}\dagger\widehat{z}((P\widehat{\alpha}\dagger\widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha}\dagger\widehat{x}R)) & \\
 & & & & (z\ \mathit{fresh})\\
(\dagger\mathit{imp\text{-}in}): & P\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\beta}\,[z]\,\widehat{y}R) & \to & (P\widehat{\alpha}\dagger\widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha}\dagger\widehat{x}R) & (z\neq x)\\
(\dagger\mathit{cut}): & P\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R) & \to & (P\widehat{\alpha}\dagger\widehat{x}Q)\widehat{\beta}\dagger\widehat{y}(P\widehat{\alpha}\dagger\widehat{x}R) &
\end{array}$$

Contextual Rules :

$$P\to Q\ \Rightarrow\ \begin{cases}
\widehat{x}P\widehat{\alpha}{\cdot}\beta & \to\ \widehat{x}Q\widehat{\alpha}{\cdot}\beta\\
P\widehat{\alpha}\,[x]\,\widehat{y}R & \to\ Q\widehat{\alpha}\,[x]\,\widehat{y}R\\
R\widehat{\alpha}\,[x]\,\widehat{y}P & \to\ R\widehat{\alpha}\,[x]\,\widehat{y}Q\\
P\widehat{\alpha}\dagger\widehat{y}R & \to\ Q\widehat{\alpha}\dagger\widehat{y}R\\
R\widehat{\alpha}\dagger\widehat{y}P & \to\ R\widehat{\alpha}\dagger\widehat{y}Q
\end{cases}$$

We write $\to^*_{\mathcal{LK}}$ for the reduction relation defined as the smallest pre-order that includes the logical, propagation rules, and contextual rules.

The notion of *head* reduction, $\to_{\text{H}}$, is defined by excluding reductions in and toward *import*, via the *elimination* of the propagation rules that move into an *import* (*i.e.* (*imp*†), (†*imp-out*), and (†*imp-in*), as well as the second and third contextual rule).

It is well known that reduction in $\mathcal{LK}$ is highly non-confluent: it is easy to check that the *cut* $P\widehat{\alpha}\dagger\widehat{x}Q$ – with α not in P and x not in Q – in $\mathcal{LK}$ can run to both P and Q, so reducing it decreases the set of reachable normal forms. Also, *cut*-elimination is different from Gentzen's (implicit) definition: he considered only *innermost* reduction for his *Hauptsatz* result.

As we have introduced the calculus for the classical logic, we now introduce the π-calculus with pairing, which is based on the one defined in (Abadi and Gordon 1997).

2 The π-calculus with pairing

One of the challenges we had to face, during the development of our encoding, was to choose the right π-calculus. There are several versions of the π-calculus with both syntactic and semantics variations. We have chosen a π-calculus with pairs as it made the translation of the $\mathcal{LK}$, in which a term can be constructed binding *two connectors simultaneously*, more readable and results stated in a more natural, cleaner way.

We write either Greek characters or Roman characters for channel names; we use a, b, c, n for either a Greek or a Roman name.

Definition 4 ($\pi_{\langle\rangle}$: THE π-CALCULUS WITH PAIRING)

1. *Channel names* and *data* are defined by:

$$a, b, c, d \ ::= \ x \mid \alpha \quad \textit{names} \qquad\qquad p \ ::= \ a \mid \langle a,b\rangle \quad \textit{data}$$

2. *Processes* are defined by the grammar:

$$\mathsf{P}, \mathsf{Q} \ ::= \ \mathsf{0} \mid \mathsf{P} \mid \mathsf{Q} \mid !\mathsf{P} \mid (\nu a)\, \mathsf{P} \mid a(x).\mathsf{P} \mid \overline{a}\,p.\mathsf{P} \mid \textit{let}\ \langle x,y\rangle = p\ \textit{in}\ \mathsf{P}$$

We abbreviate $a(x).\textit{let}\ \langle y,z\rangle = x\ \textit{in}\ \mathsf{P}$ by $a(y,z).\mathsf{P}$, and $(\nu m)\,(\nu n)\,\mathsf{P}$ by $(\nu mn)\,\mathsf{P}$, and write $\overline{a}\,p$ rather than $\overline{a}\,p.\mathsf{0}$. We write $a \rightarrow\!\!\triangleright b$ for $a(w).\overline{b}\,w$.

3. *Structural congruence* '$\equiv$' is defined as normal, but extended with:

$$\textit{let}\ \langle x,y\rangle = \langle a,b\rangle\ \textit{in}\ \mathsf{P} \ \equiv \ \mathsf{P}[a/x, b/y]$$

4. The *reduction relation* $\rightarrow_\pi$ over the processes of the $\pi_{\langle\rangle}$-calculus is defined by following (elementary) rules:

$$\begin{array}{rcll}
\overline{a}\,p \mid a(x).\mathsf{Q} & \rightarrow & \mathsf{Q}[p/x], \ \textit{if well defined} & \textit{synchronisation} \\
\mathsf{P} \rightarrow \mathsf{Q} & \Rightarrow & (\nu n)\,\mathsf{P} \rightarrow (\nu n)\,\mathsf{Q} & \textit{binding} \\
\mathsf{P} \rightarrow \mathsf{Q} & \Rightarrow & \mathsf{P} \mid \mathsf{R} \rightarrow \mathsf{Q} \mid \mathsf{R} & \textit{composition} \\
\mathsf{P} \equiv \mathsf{Q} \ \& \ \mathsf{Q} \rightarrow \mathsf{Q}' \ \& \ \mathsf{Q}' \equiv \mathsf{P}' & \Rightarrow & \mathsf{P} \rightarrow \mathsf{P}' & \textit{congruence}
\end{array}$$

Notice that data occurs only in $\overline{a}\,p.\mathsf{P}$ and $\textit{let}\ \langle x,y\rangle = p\ \textit{in}\ \mathsf{P}$, and that then p is either a single name, or a pair of names. This implies that we do not allow $\langle a,b\rangle.\mathsf{P}$, nor $a(\langle b,c\rangle).\mathsf{P}$, nor $\overline{a}\,\langle\langle b,c\rangle,d\rangle$, nor $(\nu\langle a,b\rangle)\,\mathsf{P}$, nor $\textit{let}\ \langle\langle a,b\rangle,y\rangle = p\ \textit{in}\ \mathsf{P}$, etc.

There are several notion of equivalence defined for the π-calculus: the one we consider here, and will show is related to our encodings, is that of weak bisimilarity.

Definition 5 (WEAK BISIMILARITY) 1. We write $\mathsf{P}\downarrow\overline{n}$ (and say that P *outputs on n*) if $\mathsf{P} \equiv (\nu b_1 \ldots b_m)\,(\overline{n}\,p \mid \mathsf{Q})$ for some Q, where $n \neq b_1 \ldots b_m$. We write $\mathsf{P}\Downarrow\overline{n}$ (P *will output on n*) if there exists Q such that $\mathsf{P} \rightarrow_\pi^* \mathsf{Q}$ and $\mathsf{Q}\downarrow n$. $\mathsf{P}\downarrow n$ (P *inputs on n*) and $\mathsf{P}\Downarrow n$ (P *will input on n*) are defined similarly.

2. A *barbed bisimilarity* $\dot{\approx}$ is the largest symmetric relation such that $\mathsf{P} \dot{\approx} \mathsf{Q}$ satisfies the following clauses:
   - if for each name n: if $\mathsf{P}\downarrow\overline{n}$ then $\mathsf{Q}\Downarrow\overline{n}$, and if $\mathsf{P}\downarrow n$ then $\mathsf{Q}\Downarrow n$;
   - for all P', if $\mathsf{P} \to_{\pi}^{*} \mathsf{P}'$, then there exists Q' such that $\mathsf{Q} \to_{\pi}^{*} \mathsf{Q}'$ and $\mathsf{P}' \dot{\approx} \mathsf{Q}'$.
3. *Weak-bisimilarity* is the largest relation $\approx$ defined by: $\mathsf{P} \approx \mathsf{Q}$ if and only if $\mathsf{C}[\mathsf{P}] \dot{\approx} \mathsf{C}[\mathsf{Q}]$ for any context $\mathsf{C}[\cdot]$.
4. We write $\mathsf{P} \sqsubseteq_{\pi} \mathsf{Q}$ if and only if there exists an R such that $\mathsf{Q} \equiv \mathsf{P} \mid \mathsf{R}$, and write $\mathsf{P} \lesssim_{\pi} \mathsf{Q}$ if and only if there exists R such that $\mathsf{R} \approx \mathsf{Q}$ and $\mathsf{P} \sqsubseteq_{\pi} \mathsf{R}$.

Notice that $\mathsf{P} \lesssim_{\pi} \mathsf{Q}$ expresses that Q has more behaviour than P. This yields a clearer statement on the correctness of the encoding.

3 A natural encoding for $\mathcal{LK}$ into $\pi_{\langle\rangle}$

We now present the main body of our research, which is the translation of $[\![\cdot]\!]_{\mathrm{N}}$ of $\mathcal{LK}$ into $\pi_{\langle\rangle}$; it is called natural since it will create processes that output on names that are associated to plugs, and input on names that are associated to sockets, and tries as much as possible to encode the joining of connectors through substitution, following the syntactic structure of terms.

Definition 6 (Natural encoding of $\mathcal{LK}$ in $\pi_{\langle\rangle}$)

$$\begin{array}{rclrcl}
[\![\langle x{\cdot}\beta\rangle]\!]_{\mathrm{N}} &=& x(w).\overline{\beta}\,w & [\![P\widehat{\alpha}\,[x]\,\widehat{z}Q]\!]_{\mathrm{N}} &=& x(\alpha,z).(!\,[\![P]\!]_{\mathrm{N}} \mid !\,[\![Q]\!]_{\mathrm{N}}) \\
[\![\widehat{z}P\widehat{\alpha}{\cdot}\beta]\!]_{\mathrm{N}} &=& (\nu z\alpha)\,(!\,[\![P]\!]_{\mathrm{N}} \mid \overline{\beta}\langle z,\alpha\rangle) & [\![P\widehat{\alpha}\dagger\widehat{z}Q]\!]_{\mathrm{N}} &=& (\nu z)\,(!\,[\![P[z/\alpha]]\!]_{\mathrm{N}} \mid !\,[\![Q]\!]_{\mathrm{N}})
\end{array}$$

Since in this translation some sub-terms are placed under *input*, a full representation of reduction in $\mathcal{LK}$ cannot be achieved: it is not possible to reduce the (interpreted) terms that appear under an *input*. In view of the literature that exists on translations into the π-calculus, this is unfortunate but standard: the encoding forces a restriction on the modelled reduction rules. Here, it allows us to only model *head* reduction.

Observe that in the image of $\mathcal{LK}$ in $\pi_{\langle\rangle}$, being built without using 'choice', there is no notion of *erasure* of processes; this implies that, using reduction in the $\pi_{\langle\rangle}$-calculus, we cannot model $P\widehat{\alpha}\dagger\widehat{x}Q \to_{\mathcal{LK}} P$, assuming $\alpha \notin \mathit{fp}(P)$ and $x \notin \mathit{fs}(Q)$; we can at most show:

$$[\![P\widehat{\alpha}\dagger\widehat{x}Q]\!]_{\mathrm{N}} \triangleq (\nu x)\,(!\,[\![P[x/\alpha]]\!]_{\mathrm{N}} \mid !\,[\![Q]\!]_{\mathrm{N}}) \equiv\ !\,[\![P]\!]_{\mathrm{N}} \mid !\,[\![Q]\!]_{\mathrm{N}}$$

Now all reductions will take place in either $\llbracket P \rrbracket_{\rm N}$ or $\llbracket Q \rrbracket_{\rm N}$, and both parts will remain under reduction. This implies that, in this case, it is clear that the interpreted *cut* $\llbracket P\widehat{\alpha} \dagger \widehat{x}Q \rrbracket_{\rm N}$ must *contain* the behaviour of either its contractea, so, evidently, has more behaviour than both $\llbracket P \rrbracket_{\rm N}$ and $\llbracket Q \rrbracket_{\rm N}$ separately; this is natural for encodings of non-confluent calculi. We see this return in the formulation of the correctness result for the natural encoding, which is formulated through $\underset{\sim}{\sqsupset}_\pi$.

Theorem 7 (Operational Soundness of $\llbracket \cdot \rrbracket_{\rm N}$ with respect to $\rightarrow_{\rm H}$) *If $P \rightarrow_{\rm H}^{*} Q$, then there exists R such that $\llbracket P \rrbracket_{\rm N} \rightarrow_{\pi}^{*} R$ with $R \underset{\sim}{\sqsupset}_\pi \llbracket Q \rrbracket_{\rm N}$.*

It is easy to show that all reductions in the encoding are in fact in $\approx$, so the above result can be restated as:

Corollary 8 If $P \rightarrow_{\rm H}^{*} Q$, then $\llbracket P \rrbracket_{\rm N} \underset{\sim}{\sqsupset}_\pi \llbracket Q \rrbracket_{\rm N}$.

One of the main result of the encoding is that it led to the definition of the new type system for the $\pi_{\langle\rangle}$-calculus, $\vdash_\pi$ (van Bakel et al. 2008, 2014). As for $\mathcal{LK}$, typeability of a process is expressed as $P : \Gamma \vdash_\pi \Delta$, where the left context Γ contains types for all the *input* channels of P, and Δ for its *output* channels; since in P a channel can be used for both, it can appear in both contexts. Our system thereby gives an abstract functional encoding of processes by stating the connectability of a process via giving the names of the available (connectable) channels and their types.

Definition 9 (Implicative type assignment for $\pi_{\langle\rangle}$) Type assignment for $\pi_{\langle\rangle}$ is defined by the following sequent system:

$$(0):\ \frac{}{0\ :\ \vdash} \qquad (!):\ \frac{P : \Gamma \vdash \Delta}{!P : \Gamma \vdash \Delta} \qquad (out):\ \frac{}{\overline{a}\,b : b{:}A \vdash a{:}A, b{:}A}\ (a \neq b)$$

$$(\nu):\ \frac{P : \Gamma, a{:}A \vdash a{:}A, \Delta}{(\nu a)\,P : \Gamma \vdash \Delta} \qquad (pair\text{-}out):\ \frac{}{\overline{a}\langle b,c\rangle : b{:}A \vdash a{:}A{\to}B, c{:}B}\ (b \neq a, c)$$

$$(|):\ \frac{P_1 : \Gamma \vdash \Delta \ \cdots\ P_n : \Gamma \vdash \Delta}{P_1 \mid \cdots \mid P_n : \Gamma \vdash \Delta} \qquad (in):\ \frac{P : \Gamma, x{:}A \vdash x{:}A, \Delta}{a(x).P : \Gamma, a{:}A \vdash \Delta}$$

$$(W):\ \frac{P : \Gamma \vdash \Delta}{P : \Gamma' \vdash \Delta'}\ (\Gamma' \supseteq \Gamma, \Delta' \supseteq \Delta) \qquad (let):\ \frac{P : \Gamma, y{:}B \vdash x{:}A, \Delta}{let\ \langle x,y\rangle = z\ in\ P : \Gamma, z{:}A{\to}B \vdash \Delta}\ (*)$$

$(*)$ $y, z \notin \Delta; x \notin \Gamma$.

With this notion of type assignment for $\pi_{\langle\rangle}$, we show that the natural encoding $\llbracket \cdot \rrbracket_{\rm N}$ preserves assignable types:

Theorem 10 (Type preservation) *If* $P \mathrel{:\cdot} \Gamma \vdash \Delta$, *then* $\llbracket P \rrbracket_{\text{N}} : \Gamma \vdash_{\pi} \Delta$.

We have shown that, if P is a witness to a judgement (in $\vdash_{\text{LK}}$), then its encoding via $\llbracket \cdot \rrbracket_{\text{N}}$ is as well. Together with the preservation result (Theorem 7) this implies that we can not only interpret reduction in LK through synchronisation (similarly to what was done, for example, in (Milner 1992) for the lazy λ-calculus), but show that the processes we create through our interpretations accurately represent the *actual proofs*, so synchronisation correctly models *cut*-elimination, and transforms a proof into a proof.

4 New directions: Fully abstract output-based encoding of $\lambda\mu\mathbf{x}$

The idea of giving a computational meaning to classical logic exists for about twenty years. The pioneering work of Parigot (1992) created $\lambda\mu$, an extension of the λ-calculus with *names* and *context switches* that yields a variant of Gentzen's classical natural deduction calculus that focusses on confluence.

Here, we study a variant of $\lambda\mu$ with explicit substitution (Bloo and Rose 1995) we defined in (van Bakel and Vigliotti 2012, 2014). We define a notion of reduction that only ever replaces the head variable of a term; we will see that this is the notion of computation that the $\pi_{\langle\rangle}$-calculus naturally represents.

Definition 11 (Explicit head reduction for $\lambda\mu$) The syntax of the $\lambda\mu$ *calculus with explicit substitution*, $\lambda\mu\mathbf{x}$, is defined by:

$$M, N \ ::= \ x \mid \lambda x.M \mid MN \mid M \langle x := N \rangle \mid \mu\alpha.[\beta]M \mid M \langle \alpha := N{\cdot}\gamma \rangle$$

The notion of *head variable* $hv(M)$ is defined as usual, and *head name* $hn(M)$ are defined by $hn(\mu\alpha.[\beta]\boldsymbol{H}) = \beta$, $hn(M \langle x := N \rangle) = hn(M)$, and $hn(M \langle \alpha := N{\cdot}\gamma \rangle) = hn(M)(hn(M) \neq \alpha)$.

We call a term *pure* if it does not contain explicit substitution and write C for the pseudo-term $[\alpha]M$. The notion of *explicit head reduction* $\rightarrow_{\mathbf{xH}}$ on terms in $\lambda\mu\mathbf{x}$ is defined through the following rules:

Main reduction rules :

$$\begin{array}{rcll}
(\lambda x.M)N & \rightarrow & M \langle x := N \rangle & (\beta\textit{-rule}) \\
(\mu\alpha.M)N & \rightarrow & \mu\gamma.M \langle \alpha := N{\cdot}\gamma \rangle & (\gamma \textit{ fresh}) \\
\mu\alpha.[\alpha]M & \rightarrow & M & (\alpha \notin \mathit{fn}(M)) \\
\mu\alpha.[\beta]\mu\gamma.\mathsf{C} & \rightarrow & \mu\alpha.\mathsf{C}[\beta/\gamma] &
\end{array}$$

Term substitution rules :

$$\begin{array}{rcll}
x \langle x := N \rangle & \rightarrow & N & \\
M \langle x := N \rangle & \rightarrow & M & (x \notin \mathit{fv}(M)) \\
(\lambda y.M) \langle x := N \rangle & \rightarrow & \lambda y.(M \langle x := N \rangle) & (x = hv(M)) \\
(PQ) \langle x := N \rangle & \rightarrow & (P \langle x := N \rangle \, Q) \langle x := N \rangle & (x = hv(P)) \\
(\mu\alpha.[\beta]M) \langle x := N \rangle & \rightarrow & \mu\alpha.[\beta](M \langle x := N)\rangle & (x = hv(M))
\end{array}$$

Structural rules : $\begin{array}{rcll}(\mu\beta.[\alpha]M)\,\langle\alpha:=N{\cdot}\gamma\rangle & \to & \mu\beta.[\gamma](M\,\langle\alpha:=N{\cdot}\gamma\rangle)N & \\ M\,\langle\alpha:=N{\cdot}\gamma\rangle & \to & M & (\alpha\notin fn(M))\end{array}$

Contextual Rules :

$$M\to N \quad\Rightarrow\quad \begin{cases}\lambda x.M & \to\ \lambda x.N\\ ML & \to\ NL\\ \mu\alpha.[\beta]M & \to\ \mu\alpha.[\beta]N\\ M\,\langle x:=L\rangle & \to\ N\,\langle x:=L\rangle\\ M\,\langle\alpha:=L{\cdot}\gamma\rangle & \to\ N\,\langle\alpha:=L{\cdot}\gamma\rangle\end{cases}$$

Substitution rules :

$$\begin{array}{rcll}M\,\langle x:=N\rangle\,\langle y:=L\rangle & \to & M\,\langle y:=L\rangle\,\langle x:=N\rangle\,\langle y:=L\rangle & (y=hv(M))\\ M\,\langle\alpha:=N{\cdot}\beta\rangle\,\langle y:=L\rangle & \to & M\,\langle y:=L\rangle\,\langle\alpha:=N{\cdot}\beta\rangle\,\langle y:=L\rangle & (y=hv(M))\\ M\,\langle\alpha:=N{\cdot}\gamma\rangle\,\langle\beta:=L{\cdot}\delta\rangle & \to & M\,\langle\beta:=L{\cdot}\delta\rangle\,\langle\alpha:=N{\cdot}\gamma\rangle\,\langle\beta:=L{\cdot}\delta\rangle & (\beta=hn(M))\\ M\,\langle x:=N\rangle\,\langle\beta:=L{\cdot}\delta\rangle & \to & M\,\langle\beta:=L{\cdot}\delta\rangle\,\langle x:=N\rangle\,\langle\beta:=L{\cdot}\delta\rangle & (\beta=hn(M))\end{array}$$

Notice that we do not allow reduction inside the substitution nor inside the right-hand side of an application. Moreover, the substitution $\langle x:=N\rangle$ on PQ is postponed on Q until the variable x in Q becomes the head-variable.

The interpretation of $\lambda\mu\mathbf{x}$ terms into the $\pi_{\langle\rangle}$-calculus is defined by:

Definition 12 (LOGICAL INTERPRETATION OF $\lambda\mu\mathbf{x}$ TERMS)

$$\begin{array}{rcll}[\![x]\!]\,a & \triangleq & x(u).!u\text{-}\!\triangleright a & (u\ \mathit{fresh})\\ [\![\lambda x.M]\!]\,a & \triangleq & (\nu xb)\,([\![M]\!]\,b\mid\overline{a}\langle x,b\rangle) & (b\ \mathit{fresh})\\ [\![MN]\!]\,a & \triangleq & (\nu c)\,([\![M]\!]\,c\mid !c(v,d).([\![v:=N]\!]\mid !d\text{-}\!\triangleright a)) & (c,v,d\ \mathit{fresh})\\ [\![M\,\langle x:=N\rangle\,]\!]\,a & \triangleq & (\nu x)\,([\![M]\!]\,a\mid[\![x:=N]\!]) & \\ [\![x:=N]\!] & \triangleq & !\overline{x}(w).[\![N]\!]\,w & (w\ \mathit{fresh})\\ [\![\mu\gamma.[\beta]M]\!]\,a & \triangleq & [\![M]\!]\,\beta[a/\gamma] & \\ [\![M\,\langle\beta:=N{\cdot}\gamma\rangle\,]\!]\,a & \triangleq & (\nu\beta)\,([\![M]\!]\,a\mid[\![\beta:=N{\cdot}\gamma]\!]) & \\ [\![\alpha:=M{\cdot}\gamma]\!] & \triangleq & !\alpha(v,d).([\![v:=N]\!]\mid !d\text{-}\!\triangleright\gamma) & (v,d\ \mathit{fresh})\end{array}$$

It is called logical because the interpretation of application corresponds to the representation of natural deduction's *modus ponens* into LK.

Observe the similarity between

$$\begin{array}{rcl}[\![MN]\!]\,a & = & (\nu\beta)\,([\![M]\!]\,\beta\mid !\beta(v,d).([\![v:=N]\!]\mid !d\text{-}\!\triangleright a)) \quad\text{and}\\ [\![M\,\langle\beta:=N{\cdot}\gamma\rangle\,]\!]\,a & = & (\nu\beta)\,([\![M]\!]\,a\mid !\beta(v,d).([\![v:=N]\!]\mid !d\text{-}\!\triangleright\gamma))\end{array}$$

The first communicates N via the output channel β of M, whereas the second communicates with all the sub-terms that have β as output name. In other words, the encoding highlights that application is just a special case of explicit structural substitution, which corresponds to distributed application.

We can now show:

Theorem 13 (Soundness) *1. If $M \rightarrow^*_{\mathbf{x}\mathrm{H}} N$, and in this reduction the β-rule is applied, then $[\![M]\!]\,a \rightarrow^+_\pi P$ and $P \approx [\![N]\!]\,a$.*

2. If $M \rightarrow^*_{\mathbf{x}\mathrm{H}} N$ then $[\![M]\!]\,a \approx [\![N]\!]\,a$.

3. If $M \Uparrow_{\mathbf{x}\mathrm{H}}$ then $[\![M]\!]\,a\Uparrow$.

As for full abstraction, notice that since $\Delta\Delta$ and $\Omega\Omega$ (where $\Delta = \lambda x.xx$, $\Omega = \lambda y.yyy$) are closed terms that do not interact with any context, they are contextually equivalent; any well-defined interpretation of these terms into the $\pi_{\langle\rangle}$-calculus will therefore map those to processes that are weakly bisimilar to 0, and therefore to weakly bisimilar processes. Abstraction, on the other hand, enables interaction with a context, and therefore the interpretation of $\lambda z.\Delta\Delta$ will *not* be weakly bisimilar to 0. We therefore cannot hope to model normal $\beta\mu$-reduction in the $\pi_{\langle\rangle}$-calculus; rather, we need to consider a notion of reduction that considers *all* abstractions meaningful; therefore, the only kind of reduction on λ-calculi that can naturally be encoded into the $\pi_{\langle\rangle}$-calculus is *weak* (or *lazy*) reduction, $\rightarrow_{w\mathrm{H}}$. Here we just define weak head normal forms and weak equivalence.

Definition 14 (Weak equivalence for $\lambda\mu$) The $\lambda\mu$ *weak head-normal forms (*WHNF*)* are defined through the grammar:

$$\begin{array}{rcll} \boldsymbol{H}_w & ::= & \lambda x.M \mid xM_1\cdots M_n & (n \geq 0) \\ & \mid & \mu\gamma.[\delta]\boldsymbol{H}_w & (\gamma \neq \delta \text{ or } \gamma \in \boldsymbol{H}_w, \boldsymbol{H}_w \neq \mu\gamma'.[\delta']\boldsymbol{H}'_w) \end{array}$$

We say that M *has a* WHNF if there exists $\boldsymbol{H}_w$ such that $M \rightarrow^*_{w\mathrm{H}} \boldsymbol{H}_w$.

We define $\sim_{w\beta\mu}$ as the smallest congruence that contains:

$$\begin{array}{rcll} M, N \textit{ have no } \text{WHNF} & \Rightarrow & M \sim_{w\beta\mu} N & \\ (\lambda x.M)N & \sim_{w\beta\mu} & M[N/x] & \\ (\mu\alpha.\mathsf{C})N & \sim_{w\beta\mu} & \mu\gamma.\mathsf{C}[N{\cdot}\gamma/\alpha] & (\gamma \textit{ fresh}) \\ \mu\alpha.[\beta]\mu\gamma.\mathsf{C} & \sim_{w\beta\mu} & \mu\alpha.\mathsf{C}[\beta/\gamma] & \\ \mu\alpha.[\alpha]M & \sim_{w\beta\mu} & M & (\alpha \notin M) \end{array}$$

Through a number of bisimulations defined on $\lambda\mu$ and $\lambda\mu\mathbf{x}$, and a notion of approximation semantics, (van Bakel and Vigliotti 2014) goes on to show:

Theorem 15 (Full abstraction) *Let M, N be pure $\lambda\mu$-terms, then*

$$[\![M]\!]\,a \approx [\![N]\!]\,a \iff M \sim_{w\beta\mu} N$$

5 Conclusion

We have bridged the gap between classical *cut*-elimination and the semantics of concurrent calculi, by presenting a translation of Gentzen's classical sequent calculus LK to the π-calculus that preserves *cut*-elimination. This shows that the π-paradigm is truly classical in nature.

However, since $\mathcal{LK}$ is highly non-confluent, our result is expressed through $\underset{\sim}{\sqsupset}_{\pi}$. We therefore applied our techniques to a confluent classical calculus, $\lambda\mu$, and found that the π-calculus is again ideally suited: we can define an interpretation that respects single step explicit head reduction by weak bisimilarity $\approx$, and show that it is full abstract with respect to weak equivalence $\sim_{w\beta\mu}$.

So the π-calculus is perfectly fit to deal with calculi that have their basis in classical logic, both for sequent calculi as for (confluent) natural deduction.

References

M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *CCS'97*, pp. 36–47. ACM Press, 1997.

R. Bloo and K. Rose. Preservation of Strong Normalisation in Named Lambda Calculi with Explicit Substitution and Garbage Collection. In *CSN'95*, pages 62–72, 1995.

G. Gentzen. Investigations into logical deduction. In *The Collected Papers of Gerhard Gentzen.* Ed M. E. Szabo, North Holland, 68ff (1969), 1935.

R. Milner. Functions as Processes. *MSCS*, 2(2):269–310, 1992.

M. Parigot. An algorithmic interpretation of classical natural deduction. In *LPAR'92*, LNCS 624, pp. 190–201, 1992.

S. van Bakel and P. Lescanne. Computation with Classical Sequents. *MSCS* 18:555–609, 2008.

S. van Bakel and M. Vigliotti. An Output-Based Semantics of $\lambda\mu$ with Explicit Substitution in the π-calculus - Extended Abstract. In IFIP-TCS'12, LNCS 7604, pp. 372–387, 2012.

S. van Bakel and M. Vigliotti. A fully abstract semantics of $\lambda\mu$ in the π-calculus. CL&C'14, 2014.

S. van Bakel, L. Cardelli, and M. Vigliotti. From $\mathcal{X}$ to π; Representing the Classical Sequent Calculus in the π-calculus. In *CL&C'08*, 2008.

S. van Bakel, L. Cardelli, and M. Vigliotti. Classical Cut-elimination in the π-calculus. Submitted, 2014.