PLACES

22nd March

2009

Programming Language Approaches to Concurrency and Communication-cEntric Software is a workshop at ETAPS 2009, York, England

PROGRAMME COMMITTEE

Alastair Beresford (chair), University of Cambridge

Simon Gay (chair), University of Glasgow

Kohei Honda, Queen Mary University of London

Greg Morrisett, Harvard University

Alan Mycroft, University of Cambridge

Vijay A. Saraswat, IBM Research

Vasco T. Vasconcelos, University of Lisbon

Jan Vitek, Purdue University

Nobuko Yoshida, Imperial College London

Timetable

Session 1

9.00	Stefan Marr, Michael Haupt, Stijn Timbermont, Bram Adams, Theo D'Hondt, Pascal Costanza and Wolfgang De Meuter. **Virtual Machine Support for Many-Core Architectures: Decoupling Abstract From Concrete Concurrency Models**
9.30	Vasco T. Vasconcelos, Francisco Martins and Tiago Cogumbreiro. **Type Inference for Deadlock Detection in a Multithreaded Typed Assembly Language**
10.00	Prodromos Gerakios, Nikolaos Papaspyrou and Konstantinos Sagonas. **A Concurrent Language with a Uniform Treatment of Regions and Locks**
10.30	*Break*

Session 2

11.00	Marco Giunti, Kohei Honda, Vasco Thudichum Vasconcelos and Nobuko Yoshida. **Session-based Type Discipline for Pi Calculus with Matching**
11.30	Andi Bejleri, Raymond Hu and Nobuko Yoshida. **Session-Based Programming for Parallel Algorithms**
12.00	Hugo Andres Lopez, Jorge A. Perez and Carlos Olarte. **Towards a Unified Framework for Declarative Structured Communications**
12.30	*Lunch*

Session 3

14.00	**Vivek Sarkar. Invited lecture and discussion**
15.30	*Break*

Session 4

16.00	Marco Carbone and Joshua Guttman. **Execution models for Choreographies and Cryptoprotocols**
16.30	Francisco Martins, Luis Lopes and Joao Barros. **Towards the Safe Programming of Wireless Sensor Networks**
17.00	Matthew Kehrt, Laura Effinger-Dean, Michael Schmitz and Dan Grossman. **Programming Idioms for Transactional Events**
17.30	Elena Giachino, Matthew Sackman, Sophia Drossopoulou and Susan Eisenbach. **Softly safely spoken: Role playing for Session Types**
18.00	*End*

Virtual Machine Support for Many-Core Architectures: Decoupling Abstract from Concrete Concurrency Models

Stefan Marr[1]*, Michael Haupt[2], Stijn Timbermont[1]*, Bram Adams[3]
Theo D'Hondt[1], Pascal Costanza[1], Wolfgang De Meuter[1]

[1]Programming Technology Lab,
Vrije Universiteit Brussel, Belgium

[2]Hasso Plattner Institute,
University of Potsdam, Germany

[3]Software Analysis and Intelligence Lab,
Queen's University, Canada

Abstract

The upcoming many-core architectures require software developers to exploit concurrency to utilize available computational power. We argue that today's virtual machines (VMs), which are a cornerstone of software development, do not provide sufficient abstraction for concurrency concepts. To overcome this shortcoming, we propose to integrate concurrency operations into VM instruction sets. Since there will not be a single instruction set for all kind of VMs, our goal is to develop a methodology to design instruction sets with concurrency support. Therefore, we also propose a list of tradeoffs that have to be investigated to advise the design of such instruction sets. As a first basic prototype, we implemented instructions for shared-memory concurrency. From our experimental results, we derived a list of requirements for a full-grown experimental environment for further research.

1 Motivation

With the arrival of many-core architectures, the variance of processors will be increased by another order of magnitude. This variance will also increase the need for high-level language virtual machines (VMs) to abstract away from variations introduced by differences among many-core architectures [13,21,29,28]. We are concerned with processors having multiple cores and using recent memory connection scheme, i. e., *not* shared memory. We refer to such designs as *many-core architectures*.

For software developers, VMs have to provide abstractions from hardware details like number of cores or memory connection scheme. We identify three basic *concrete concurrency models* which need to be supported by VMs. The most fundamental one is a single-core system accessing memory not shared with another processor. The second one is a shared memory approach for multi-core systems. The third model provides an explicit communication facility between cores and does not rely on shared memory. VMs' concurrency abstraction layers must enable efficient implementations on top of these concrete models.

As opposed to the former, *abstract concurrency models* used by software developers are defined by the language or library being used. Our claim is that the current abstract models' available language and library incarnations are not sufficient. Shared memory with locking is too complicated [16]. Software transactional memory (STM) [22] and Actors [1] are promising but not widely adopted. Ongoing efforts in building languages to handle the inherent concurrency of many-core systems will likely lead to domain-specific languages. In this regard, VMs like the *Java Virtual Machine* (JVM) and the *Common Language Runtime* are becoming more important as common execution platforms for multiple languages, since the ability to reuse the existing infrastructure surrounding the VM is an economical concern.

*Funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

Realistically, there will not be one single model for expressing concurrency. Thus, we argue that a VM has to provide support for a wide range of concurrency models at its core. Very likely, developers will have to deal with several models; e. g., in relation with legacy code requiring proper support. Furthermore, support for a wide range of models eases the work of language designers to implement new ideas or domain-specific solutions. VM developers can also benefit from richer concurrency semantics, as it would enable efficient implementations of different abstract concurrency models on top of the concrete models.

The remainder of this paper discusses our idea of an instruction set for concurrency and the research that has to be conducted for its design. We give a brief overview of our first experiment and present the conclusions for a full-grown experimental environment. We also discuss work contributing approaches and solutions related to VMs for many-cores.

2 VM Instruction Sets with Concurrency Support

Our proposal to achieve such an abstraction layer is to extend the VM instruction set by concurrency operations. Such an instruction set will decouple the concurrency models on the different levels of implementation in such a way that they can be varied independently. Fig. 1 visualizes this idea by showing three different abstract concurrency models mapped to an instruction set with explicit concurrency support implemented on-top of three different concrete concurrency models.

Expressing concurrency in the instruction set instead of using libraries has two major advantages.

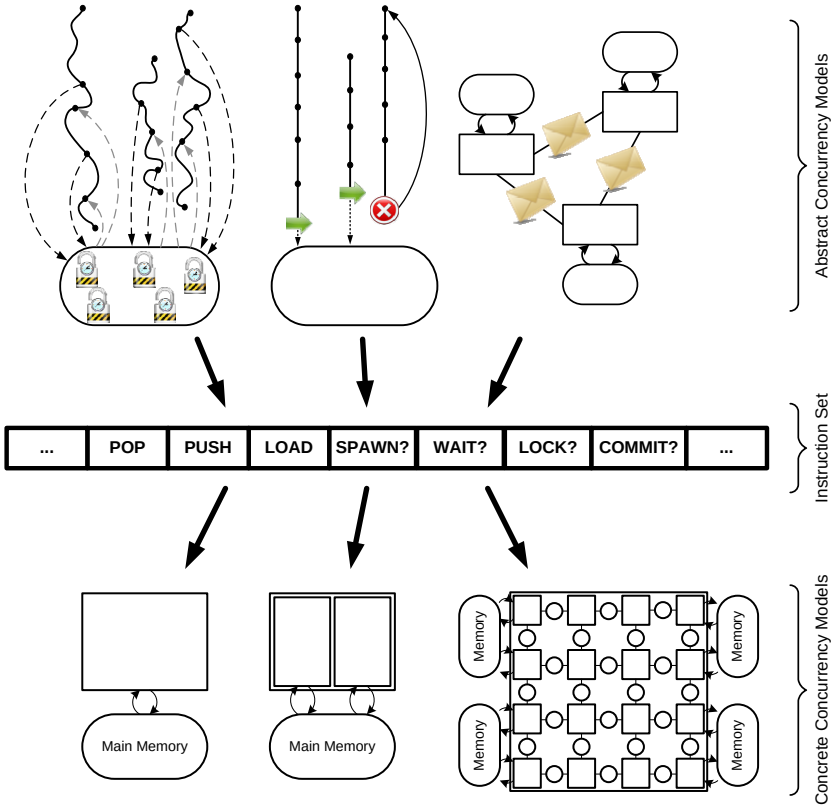

Figure 1: A VM instruction set as abstraction layer between abstract and concrete concurrency.

First, it will be possible to compile concurrency-related language constructs directly to these instructions, avoiding dependencies between languages and libraries on top of the VM. Second, this choice leads to a larger optimization potential at the VM level, e. g., for just-in-time (JIT) compilation, which benefits from the instruction set's precise semantics. Since there will not be a single instruction set matching all possible requirements, we will work on one instruction set representing a very generic set of requirements, and investigate the design tradeoffs to derive design advice for more concrete requirements as well. Thus, we plan to devise a methodology to develop VM instruction sets with inherent concurrency support, enabling VM designers to build a concurrency abstraction layer optimized for their particular requirements.

The methodology will describe how to decouple abstract and concrete concurrency. Language designers will be provided with a strategy to map abstract concurrency models to instruction sets, and VM implementers will be enabled to implement instruction sets efficiently on top of concrete concurrency models. Below, we discuss our approach in more detail.

2.1 Approach to Synthesize the Instruction Set

To devise a broadly applicable methodology, we decided to adopt a step-wise approach to designing a general instruction set and discovering the important design tradeoffs. The three currently most important concurrency models are significantly different in how they represent and realize concurrency: shared-memory with locking, STM, and actors. For each of these, we will survey different incarnations in languages or libraries to find each model's set of primitives relevant for an instruction set.

Potential candidates for examination are, to name just a few, Java [8], C# [12], Smalltalk [7], Cilk/J-Cilk [2, 6], and frameworks like Fork/Join for Java [15]. Furthermore, constructs like monitors and semaphores are considered as well [25]. In the field of STM, we currently consider the work of Shavit and Touitou [22], Ziarek et al. [32], Saha et al. [20], and Marathe et al. [17]. In the world of actor models the work of Agha [1], Erlang [26], Scala [10], and Kilim [24] are considered starting points.

2.2 Ideas to Combine Abstract Concurrency Models

One of the major research challenges will be to find appropriate combinations of the different abstract concurrency models. The idea is not to build an instruction set which is a simple enumeration of primitives for the different models, but instead of an elaborated combination thereof. Thus, the interaction between different models has to be completely understood and defined, too.

Our ideas for model combinations are based on the following work. Volos et al. [27] and Blundell et al. [3] have described possible solutions for combining locking based code with STM. A combination of locking based code and actors is described by Van Cutsem et al. [5]. STM has many similarities with common transaction processing systems; thus, we will investigate the application of transaction processing monitors [9] as used in distributed settings to use STM in conjunction with actors.

2.3 Tradeoffs to be Investigated

For the methodology, the discussion of the following design tradeoffs will be an important part.

Model Combination: Different solutions will be considered, and their benefits and drawbacks investigated. This will reveal critical details like incompatibilities and the possible degree of concurrency.

Condensed vs. Bloated Instruction Set: Only few instructions should be added to avoid exceeding the limited number of instructions in a typical bytecode set. However, additional semantics in the instruction set could reduce the complexity of implementing an abstract concurrency model on top

of it. It also can be beneficial for an efficient mapping to a concrete concurrency model. Since language and VM implementations should be reasonably manageable, these conflicting interests have to be investigated.

Bytecode vs. High-level Representation: Currently, bytecode sets are the most common representation for VM instruction sets. With respect to communication centric many-core architectures, we will investigate the potential of abstract syntax tree-like high-level representations of interpretable code in terms of reducing the implementation effort for new instructions and JIT compilers.

3 Preliminary Prototype

We have developed a first prototype VM incorporating a basic instruction set extension for shared-memory concurrency. We used CSOM, a simple Smalltalk VM developed for teaching purposes. Originally, it has a very small instruction set (16 instructions) and features a straightforward bytecode interpreter. Its overall simplicity makes CSOM a viable platform for doing experiments with minimal effort.

We added the following five instructions to support concurrency at the instruction set level: `spawn`, `lock`, `unlock`, `wait`, `notify`. They are designed with the simplicity of CSOM in mind. They operate on the top element of the execution stack. For `spawn`, the top element has to be a block which is then executed in a new thread. As a result, `spawn` pushes a new thread object onto the stack. The other four operate on an arbitrary object on the top of the stack, which is not affected.

We relied on an existing implementation of shared-memory concurrency using the pthreads library. Thus, the largest part of the work was refactoring the existing implementation from primitives, i. e., native functions for the Smalltalk thread library to bytecode instructions. Subsequently, the CSOM compiler was adapted to emit the new bytecodes on special messages.

From the experiment, we conclude four requirements for a full-grown experimental environment fit to demonstrate the advantages of an instruction set supporting a wide range of concurrency models:

- The VM has to be portable to platforms like TILE64 [29], or Cell BE [13] to be able to evaluate the benefits in mapping from an extended instruction set to different concrete concurrency models.
- Implementations of considered abstract concurrency models which use a compilation to the VM instruction set as implementation strategy should be available.
- The VM instruction set should provide space (i. e., unused bytecode instructions) for experiments.
- The VM should provide an easy to adapt JIT compiler.

Based on these requirements we consider CacaoVM[1] a possible candidate, since it has been ported to the Cell BE [23]. Maxine[2] could be a viable platform, too, as its excellent debugging facilities could ease the implementation. However, it still needs to be ported to the intended platforms. Our focus on Java VMs is founded in the availability of suitable STM [11] and Actors [24] implementations.

[1] `http://www.cacaovm.org/`

[2] `https://maxine.dev.java.net/`

4 Related Work

Support for concurrency in VM instruction sets is currently limited. The Erlang VM's BEAM instruction set[3] is a notable exception, providing dedicated support for its efficient light-weight process implementation. It includes instructions for asynchronous message sends, reading from the process' mailbox, waiting and timeouts. It is an example of how one particular model can be supported at the core of the VM. Still, we argue that support of a single concurrency model is not sufficient. VMs have to support many different programming models. Thus, they have to provide the basic means for a wide range of concurrency models in the same way as they act as execution platforms for different languages.

In the broader field of instruction set design, there are ongoing efforts to extend the capability of the JVM to act as a platform for different programming languages by introducing the `INVOKEDYNAMIC` instruction[4]. More general work on improving instruction sets with semantic extensions [14,19] has been done for the hardware level, but the concepts for, e. g., compiler adaption can be applied to VMs as well.

For the Cell BE, VM applicability has been evaluated. Besides porting and designing JVMs for this platform [18,23], some optimizations have been considered to utilize available computation power [4,30].

Distributing a VM over several computational elements bears additional challenges. Some of them have been addressed for VMs distributed on cluster setups; e. g., class loading, strategies for distributed method invocation, data access on the VM level [33], or thread migration [31].

5 Summary

We proposed to decouple abstract and concrete concurrency models to be able to cope with the variability of upcoming many-core architectures and their different memory connection schemes. We argue that this step is necessary to be able to provide support for several kinds of languages and their abstract concurrency models on top of a VM. Furthermore, the benefits of a semantically rich concurrency abstraction layer will allow more efficient VM implementations on the various different hardware platforms.

The goal of our ongoing research is to design a comprehensive methodology to design VM instruction sets combining several concurrency models to provide this abstraction. The methodology will address design tradeoffs. Our preliminary prototype enabled us to refine our initial requirements for an experimental environment and provided us with the necessary insights to be able to proceed with our research on a suitable platform.

References

[1] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[3] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report TR-CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, April 2006.

[4] C.-Y. Cher and M. Gschwind. Cell gc: Using the cell synergistic processor as a garbage collection coprocessor. In *Proc. VEE '08*, pages 141–150, New York, NY, USA, 2008. ACM.

[5] T. V. Cutsem, S. Mostinckx, and W. D. Meuter. Linguistic symbiosis between actors and threads. In *Proc. ICDL '07*, pages 222–248, New York, NY, USA, 2007. ACM.

[6] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson. Programming with exceptions in jcilk. *Sci. Comput. Program.*, 63(2):147–171, 2006.

[3] `http://erlangdotnet.net/2007/09/inside-beam-erlang-virtual-machine.html`

[4] `http://jcp.org/en/jsr/detail?id=292`

[7] Y. Gao and C. K. Yuen. A survey of implementations of concurrent, parallel and distributed smalltalk. *SIGPLAN Not.*, 28(9):29–35, 1993.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[10] P. Haller and M. Odersky. Actors that unify threads and events. *Coordination Models and Languages*, 2007.

[11] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proc. OOPSLA '06*, pages 253–262, New York, NY, USA, 2006. ACM.

[12] E. International. *Standard ECMA-334 - C# Language Specification*. 4 edition, June 2006.

[13] C. R. Johns and D. A. Brokenshire. Introduction to the cell broadband engine architecture. *IBM Journal of Research and Development*, 51(5):503–519, 2007.

[14] U. Kastens, D. K. Le, A. Slowik, and M. Thies. Feedback driven instruction-set extension. *SIGPLAN Not.*, 39(7):126–135, 2004.

[15] D. Lea. A java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, NY, USA, 2000. ACM.

[16] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[17] V. J. Marathe, W. N. Scherer, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *Proc. LCR '04*, pages 1–7, New York, NY, USA, 2004. ACM.

[18] A. G. A. Noll and M. Franz. Cellvm: A homogeneous virtual machine runtime system for a heterogeneous single-chip multiprocessor. In *Workshop on Cell Systems and Applications*, Beijing, China, June 2008.

[19] A. Peymandoust, L. Pozzi, P. Ienne, and G. D. Micheli. Automatic instruction set extension and utilization for embedded processors. In *Proc. ASAP '03*, pages 108–118, 2003.

[20] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In *Proc. PPoPP '06*. ACM, 2006.

[21] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[22] N. Shavit and D. Touitou. Software transactional memory. In *Proc. PODC '95*. ACM, 1995.

[23] G. Sorst. Java on cell b.e. Diploma thesis, Fachhochschule Aachen, September 2007.

[24] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *Proc. ECOOP 2008*, 2008.

[25] J. A. Trono and W. E. Taylor. Further comments on "a correct and unrestrictive implementation of general semaphores". *SIGOPS Oper. Syst. Rev.*, 34(3):5–10, 2000.

[26] R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall PTR, 2 edition, 1996.

[27] H. Volos, N. Goyal, and M. M. Swift. Pathological interaction of locks with transactional memory. Technical Report CS-TR-2008-1631, University of Wisconsin–Madison, February 2008.

[28] X. Wang, G. Gan, J. Manzano, D. Fan, and S. Guo. A quantitative study of the on-chip network and memory hierarchy design for many-core processor. In *Proc. ICPADS '08*. IEEE Computer Society, 2008.

[29] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[30] K. Williams, A. Noll, A. Gal, and D. Gregg. Optimization strategies for a java virtual machine interpreter on the cell broadband engine. In *Proc. CF '08*, pages 189–198. ACM, 2008.

[31] W. Zhu, C.-L. Wang, and F. C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. *cluster*, 00:381, 2002.

[32] L. Ziarek, A. Welc, A.-R. Adl-Tabatabai, V. Menon, T. Shpeisman, and S. Jagannathan. A uniform transactional execution environment for java. *Proc. ECOOP 2008*, pages 129–154, 2008.

[33] J. Zigman and R. Sankaranarayana. Designing a distributed jvm on a cluster. In *Proceedings of the 17th European Simulation Multiconference*, Nottingham, United Kingdom, June 2003.

Type Inference for Deadlock Detection in a Multithreaded Polymorphic Typed Assembly Language

Vasco T. Vasconcelos Francisco Martins Tiago Cogumbreiro

Abstract

We previously developed a type system and a type checker for a multithreaded lock-based polymorphic typed assembly language (MIL) that ensures that well-typed programs do not have race conditions. This abstract extends such work by taking into consideration deadlocks. The extended type system verifies that locks are acquired in the proper order. Towards this end we require a language with annotations that specify the locking order. Rather than asking the programmer (or the compiler's backend) to specifically annotate each newly introduced lock, we present an algorithm to infer the annotations. The result is a type checker whose input language is non-decorated as before, but that further checks that programs are exempt from deadlocks.

Motivation Type systems for lock-based race and deadlock static detection try to contradict the idea put forward by some authors that "the association between locks and data is established mostly by convention" [13]. Despite all the pathologies usually associated with locks (in the aforementioned article and others), and specially at system's level, locks are here to stay [7].

While deadlock detection should be addressed at the appropriate level of abstraction (for, in general, compiled code that does not deadlock allows us to conclude nothing of the source code), the problem remains valid at the assembly level, fitting quite nicely in the philosophy of typed assembly languages [12], providing for certified compilers that produce code to be used in systems with untrusted or malicious components, and where code must be checked for safety before execution.

Our language targets a shared-memory machine featuring an array of processors and a run pool common to all processors [8, 14]. The run pool holds threads for which no processor is available, a scheduler chooses a thread from this pool should a processor become idle. Threads voluntary release processors—our model is cooperative multi-threading. For increased flexibility (and unlike many other models, including [10]) we allow forking threads that hold locks, hence we allow the suspension of processes while in critical regions. The code in Figure 1 presents a typical example of a potential deadlock comprising a cycle of threads where each thread requests a lock hold by the next thread. Imagine the code running in a two-processors machine: after main completes its execution, each philosopher embarks on a busy-waiting loop, only that two of them will be running in processors, while the third is (and will indefinitely remain) in the run-pool. Situations of deadlocks comprising suspended code are known to be difficult to deal with [11]. Our notion of deadlocked state takes into account running and suspended threads.

Another source of difficulties in characterizing deadlock states derives from the low-level nature of our language that decouples the action of lock acquisition from that of entering a critical section, and that features non-blocking instructions only. As such the meaning of "entering a critical section" cannot be syntactic.

Deadlocked states A state $S = \langle H; T; P \rangle$ of our machine is composed of a heap H mapping labels l to data tuples $\langle v_1, \ldots, v_n \rangle^{\lambda}$ or code blocks $\forall[\vec{\omega}].(\Gamma \text{ requires } \Lambda)\{I\}$, a thread-pool T of threads waiting an available processor to execute, and an array P of processors. Tuples $\langle v_1, \ldots, v_n \rangle^{\lambda}$ store values $v_1, \ldots, v_n$ in the heap and are guarded by lock λ, meaning that a thread must hold permission λ in order to read from, or write to, the tuple. Polymorphic code blocks $\forall[\vec{\omega}].(\Gamma \text{ requires } \Lambda)\{I\}$ declare the types of the registers Γ (a map from registers to types) and the permissions Λ (a set of locks λ) required before the execution of instruction sequence I. Each processor p in P is of the form $\langle R; \Lambda; I \rangle$, where R maps the processor's registers to values, Λ is the set of locks currently held by the (thread executing in the) processor, and I is the sequence of instructions to be executed next by the processor.

```
main () {
  f1,r3 := newLock; f3,r5 := newLock; f2,r4 := newLock -- 3 forks
  r1 := r3; r2 := r4; fork liftLeftFork [f1,f2]  -- 1st philosopher
  r1 := r4; r2 := r5; fork liftLeftFork [f2,f3]  -- 2nd philosopher
  r1 := r5; r2 := r3; fork liftLeftFork [f3,f1]  -- 3rd philosopher
  done
}
liftLeftFork [l,m] (r1:⟨l⟩^l, r2:⟨m⟩^m) {
  r3 := testSetLock r1
  if r3 = 0 jump liftRightFork[l,m]
  jump liftLeftFork[l,m]
}
liftRightFork [l,m] (r1:⟨l⟩^l, r2:⟨m⟩^m) requires (l) {
  r3 := testSetLock r2
  if r3 = 0 jump eat[l,m]
  jump liftRightFork[l,m]
}
eat [l,m] (r1:⟨l⟩^l, r2:⟨m⟩^m) requires (l,m) {
  -- eat
  unlock r1 -- lay down the left fork
  unlock r2 -- lay down the right fork
  -- think
  jump liftLeftFork[l,m]
}
```

Figure 1: The dining philosophers written in MIL

A characteristic of our machine is the syntactic dissociation of the test-and-set-lock and the jump-to-critical operations, for which we provide two distinct instructions, as found in conventional instruction sets. Furthermore, there is no syntactic distinction between a conventional conditional jump and a (conditional) jump-to-critical instruction, and the two instructions can be separated by arbitrary assembly code. As far as the type system goes, the thread holds the lock only after the conditional jump, even though at run-time it may have been obtained long before.

We say that a *processor $P[i]$ in state S immediately tries to enter a critical section guarded by lock λ* if $P[i]$ is of the form $\langle R;\Lambda;(\text{if } r = \mathbf{0} \text{ jump } v;I)\rangle$ and $R(r) = \langle\_\rangle^{\lambda}$. We start by restricting reduction of a state S to that of a single processor: denote by $S \rightarrow_i S'$ a reduction step on processor i excluding rules for schedule, fork, and unlock. Then, we say that *the thread in processor $P[i]$ is trying to enter a critical region guarded by λ* if $S \rightarrow_i^* S'$ and $P[i]$ in S' immediately tries to enter a critical section guarded by λ. We also say that a *thread $\langle l[\vec{\tau}],R\rangle$ in the run pool T is trying to enter a critical region guarded by λ* if $H(l) = \forall[\vec{\omega}].(\_ \text{ requires } \Lambda)\{I\}$, and $S + P\{1: \langle R;\Lambda[\vec{\tau}/\vec{\omega}];I[\vec{\tau}/\vec{\omega}]\rangle\} \rightarrow_1^* S'$, where processor $P[1]$ in S' immediately requests lock λ.

Finally, we say that state S is deadlocked if there exists locks $\lambda_0,\ldots,\lambda_n$, with $\lambda_0 = \lambda_n$, and indices $d_0,\ldots,d_{n-1}$ ($n > 0$) such that for each $0 \leq i < n$, either processor p_{d_i} or suspended thread t_{d_i} holds lock λ_i and is trying to enter a critical region guarded by λ_{i+1}.

The annotated syntax for lock ordering Deadlocks are usually avoided by imposing a strict partial order on locks, and by respecting this order when acquiring locks [4, 10]. Our extended syntax introduces annotations that specify the locking order.

When creating a new lock, we declare the order between the newly introduced singleton lock type and the locks known in the program. If Λ is a set of locks, we use the notation $\lambda :: (\Lambda_1,\Lambda_2)$ to mean that lock type λ is greater than all lock types in Λ_1 and less than each lock type in Λ_2. We also use special

lock types $\bot$ and $\top$ to denote the smallest and the largest lock, respectively. The annotated syntax differs from the original syntax in four places (novelties in grey): *i*) at lock creation λ $::(\Lambda,\Lambda)$ $,r :=$ newLock; *ii*) when unpacking values λ $::(\Lambda,\Lambda)$ $,r :=$ unpack v; *iii*) in universal types $\forall[\omega$ $::K$ $].(\Gamma$ requires $\Lambda)$; and *iv*) in existential types $\exists\omega$ $::K$ $.\tau$, where K denotes a lock kind (Λ,Λ) or a conventional type variable kind TyVar.

Examples of annotations for the code in the figure are at lock creation in code block main:

f1 ::({ $\bot$},{$\top$}), r_3 := **newLock**
f3 ::({ f1 },{$\top$}), r_5 := **newLock**
f2 ::({ f1 },{f3 }), r_4 := **newLock**

and at the types for the three code blocks below.

liftLeftFork [l ::({ $\bot$},{$\top$})] [m::({l },{$\bot$})] ($r_1 : \langle$l$\rangle^l$, $r_2 : \langle$m$\rangle^m$)
liftRightFork [l ::({ $\bot$},{$\top$})] [m::({l },{$\bot$})] ($r_1 : \langle$l$\rangle^l$, $r_2 : \langle$m$\rangle^m$) **requires** (l)
eat [l ::({ $\bot$},{$\top$})] [m::({l },{$\bot$})] ($r_1 : \langle$l$\rangle^l$, $r_2 : \langle$m$\rangle^m$) **requires** (l,m)

Notice that the somewhat more compact notation used in the example $\forall[l_1 :: K_1, l_2 :: K_2].\tau$, short for $\forall[l_2 :: K_2](\forall[l_1 :: K_1].\tau)$, reverses the bound-variable order. Also, abstracting one lock at a time, as in the type just show, precludes declaring code blocks with non-strict partial orders on locks, such as $\forall[l :: (\{\bot\},\{m\}), m :: (\{l\},\{\top\}].\tau$, which anyway cannot be fulfilled by any conceivable sequence of instructions.

The type system for deadlock elimination The typing environment Ψ now associates kind (Λ_1,Λ_2) (rather than a plain Lock kind) to a lock λ.

The rule for lock creation below assigns a lock type $\langle\lambda\rangle^\lambda$ to the register. The new singleton lock type is recorded in Ψ, so that it may be used in the rest of the instructions I. The novelty, in grey, is that the annotation in the program is propagated to the environment.

$$\frac{\Psi,\lambda' :: (\Lambda_1,\Lambda_2);\Gamma\{r\colon \langle\lambda'\rangle^{\lambda'}\};\Lambda \vdash I[\lambda'/\lambda] \qquad \lambda' \notin \Psi,\Gamma,\Lambda,I}{\Psi;\Gamma;\Lambda \vdash \lambda :: (\Lambda_1,\Lambda_2), r :=\ \mathsf{newLock}; I}$$

The rule for jumping into a critical section, below, ensures that the current thread holds the exact number of locks required by the target code block and adds the lock under test to the set of locks of the thread. We make sure that the lock under test is strictly larger than all the locks held by the thread.

$$\frac{\Psi;\Gamma \vdash r\colon \lambda \quad \Psi;\Gamma \vdash v\colon \forall[].(\Gamma \text{ requires } (\Lambda \uplus \{\lambda\})) \quad \Psi;\Gamma;\Lambda \vdash I \quad \Psi \vdash \Lambda \prec \lambda}{\Psi;\Gamma;\Lambda \vdash \mathsf{if}\ r = \mathbf{0}\ \mathsf{jump}\ v; I}$$

Type application, used in the various **jump** instructions in the example, also suffers upgrading. The rule for type application below, specialised for locks, checks that the argument λ is within the interval required by the parameter λ'.

$$\frac{\Psi \vdash \lambda \quad \Psi;\Gamma \vdash v\colon \forall[\lambda' :: (\Lambda_1,\Lambda_2), \vec{\omega} :: \vec{K}].(\Gamma' \text{ requires } \Lambda) \quad \Psi \vdash \Lambda_1\sigma \prec \lambda \prec \Lambda_2\sigma \quad \sigma = [\lambda/\lambda']}{\Psi;\Gamma \vdash v[\lambda]\colon \forall[\vec{\omega} :: \vec{K}\sigma].(\Gamma'\sigma \text{ requires } \Lambda\sigma)}$$

As shown above, the type system now contains judgments such as $\Psi \vdash \Lambda \prec \lambda$ expressing that lock type λ is larger than each lock in Λ. The rules for such judgement, to be complemented with transitivity,

should be straightforward.

$$\frac{\Psi \vdash \lambda :: (\Lambda_1, \_) \quad \lambda_1 \in \Lambda_1}{\Psi \vdash \lambda_1 \prec \lambda} \qquad \frac{\Psi \vdash \lambda :: (\_, \Lambda_2) \quad \lambda_2 \in \Lambda_2}{\Psi \vdash \lambda \prec \lambda_2} \qquad \frac{\Psi \vdash \lambda}{\Psi \vdash \lambda \prec \top} \qquad \frac{\Psi \vdash \lambda}{\Psi \vdash \bot \prec \lambda}$$

$$\frac{\Psi \vdash \lambda \qquad \Psi \vdash \lambda_1 \prec \lambda \;\; \ldots \;\; \Psi \vdash \lambda_n \prec \lambda}{\Psi \vdash \{\lambda_1, \ldots, \lambda_n\} \prec \lambda} \qquad \frac{\Psi \vdash \lambda \qquad \Psi \vdash \lambda \prec \lambda_1 \;\; \ldots \;\; \Psi \vdash \lambda \prec \lambda_n}{\Psi \vdash \lambda \prec \{\lambda_1, \ldots, \lambda_n\}}$$

As expected, the example is not typable with the annotations introduced previously. The three **newLock** instructions place in Ψ three entries $f_1 :: (\{\bot\}, \{\top\})$, $f_2 :: (\{f_1\}, \{f_3\})$, $f_3 :: (\{f_1\}, \{\top\})$. Then the value (liftLeftFork [f2])[f1] (in the example: liftLeftFork [f1 ,f2]) in the first **fork** instruction issues goals $\Psi \vdash \{\bot\} \prec f_1 \prec \{\top\}$ and $\Psi \vdash \{f_1\} \prec f_2 \prec \{\top\}$, which are easy to guarantee given that Ψ contains an entry $f_2 :: (\{f_1\}, \{f_3\})$. Likewise, the second **fork** instruction, generates goals $\Psi \vdash \{\bot\} \prec f_2 \prec \{\top\}$ and $\Psi \vdash \{f_2\} \prec f_3 \prec \{\top\}$, which are again hold because of same entry. However, the last **fork** instruction requires $\Psi \vdash \{\bot\} \prec f_3 \prec \{\top\}$ and $\Psi \vdash \{f_3\} \prec f_2 \prec \{\top\}$, the second of which does not hold.

Notice however that each of the three **jump** instructions are typable per se. For example, in code block liftRightFork , instruction **if** r_3 = 0 **jump** eat[l,m] requires $\Psi \vdash \{l\} \prec m$, which holds because the signature for the code block includes the kinding annotation $m :: (\{l\}, \bot)$.

The main result of the type system, namely that typable annotated states do not deadlock (that is, if S is typable and $S \rightarrow^* S'$, then S' is not deadlocked), remains a conjecture at the time of this writing.

Type inference Annotating lock ordering on large assembly programs may not be an easy task. In our setting, programmers (compilers, more often) produce annotation free programs such as the one in the figure, and an inference algorithm provides the annotations. More precisely, given a program H, algorithm $\mathscr{W}$ produces a pair, comprising a typing environment Ψ and an annotated program $H^\star$, such that $\Psi \vdash H^\star$, hence stipulating the H does not deadlock, or fails, meaning that there is no possible labeling for H.

Algorithm $\mathscr{W}$ runs in two phases: the first, $\mathscr{A}$ produces a triple comprising a typing environment Ψ, an annotated program $H^\star$, and a collection of constraints C, all containing variables over permissions Λ (i.e., sets of locks, $\bot$, and $\top$). The set of constraints is then passed to a constraint solver, that either produces a substitution θ or fails. In the former case, the output of $\mathscr{W}$ is the pair $(\Psi\theta, H^\star\theta)$; in the latter $\mathscr{W}$ fails. In practice, we do not need to generate $H^\star$ or to perform the substitutions; our compiler accepts H if the produced collection of constraints is solvable, and rejects it otherwise.

The soundness of the algorithm, namely that if $\mathscr{W}(H) = (\Psi, H^\star)$ then $\Psi \vdash H^\star$, remains a conjecture at the time of this writing. Conversely, we believe that if $\Psi \vdash H^\star$, then $\mathscr{W}(\text{erase}(H^\star))$ does not fail, where erase is the obvious lock-order annotation erasure function. A stronger result would include a notion of principal solutions, on which we are working.

Generating constraints Algorithm $\mathscr{A}$ visits the program twice. On a first step it builds the initial type environment Ψ_0 collecting the types for all code blocks in the program, annotating the intervals for the locks with variables (denoted by ν and ρ); on a second visit it generates the constraints and the annotated syntax for instructions and values. The definition of $\mathscr{A}$ is as follows, where all permission-variables $\vec{\nu}_i, \vec{\rho}_i$ are fresh.

$$\mathscr{A}(\{l_i : \forall[\vec{\lambda}_i].(\Gamma_i \text{ requires } \Lambda_i)\{I_i\}\}_{i \in I}) = (\Psi_0, \{l_i : \forall[\vec{\lambda}_i :: (\vec{\nu}_i, \vec{\rho}_i)\,].(\Gamma_i \text{ requires } \Lambda_i)\{I_i^\star\}_{i \in I}, \cup_{i \in I} C_i)$$

$$\text{where } \Psi_0 = \{l_i : \forall[\vec{\lambda}_i :: (\vec{\nu}_i, \vec{\rho}_i)\,].(\Gamma_i \text{ requires } \Lambda_i)\}_{i \in I} \quad \text{and} \quad (I_i^\star, C_i) = \mathscr{I}(I_i, \Psi_0, \Gamma_i, \Lambda_i)$$

The algorithm for instructions, $\mathscr{I}$, generates further annotations for **newLock** (and **unpack**) instructions, where again variables ν, ρ are fresh:

$$\mathscr{I}((\lambda, r := \mathsf{newLock}; I), \Psi, \Gamma, \Lambda) = ((\lambda :: (\nu, \rho), r := \mathsf{newLock}; I'), C)$$
$$\text{where } (I', C) = \mathscr{I}(I, \Psi \cup \{\lambda :: (\nu, \rho)\}, \Gamma, \Lambda)$$

or further constraints, in the case of the jump-to-critical instruction:

$$\mathscr{I}((\mathsf{if}\ r = \mathbf{0}\ \mathsf{jump}\ v; I), \Psi, \Gamma, \Lambda) = (\mathsf{if}\ r = \mathbf{0}\ \mathsf{jump}\ v'; I'), C_1 \cup C_2 \cup \{\Lambda \prec \lambda\})$$
$$\text{where } (v', C_1) = \mathscr{V}(v, \Psi, \Gamma) \quad \text{and} \quad (I', C_2) = \mathscr{I}(I, \Psi, \Gamma, \Lambda) \quad \text{and} \quad \Gamma(r) = \lambda$$

The algorithm for values, $\mathscr{V}$, generates constraints in the case of type application (and **pack**):

$$\mathscr{V}(v[\lambda], \Gamma, \Psi) = (v'[\lambda], \forall[\vec{\omega} :: \vec{K}\sigma].(\Gamma\sigma\ \mathsf{requires}\ \Lambda\sigma), C \cup \{\nu \prec \lambda \prec \rho\})$$
$$\text{where } (v', \forall[\lambda' :: (\nu, \rho), \vec{\omega} :: \vec{K}].(\Gamma\ \mathsf{requires}\ \Lambda), C) = \mathscr{V}(v, \Psi, \Gamma) \quad \text{and} \quad \sigma = [\lambda/\lambda']$$

For the running example, we first rename all bound variables, so that the type of code block liftLeftFork mentions l1 and m1, that of liftLeftFork mentions l2 and m2, and m_1 and that of eat uses l3and m3. For example:

liftRightFork [l2 ::({⊥},{⊤})] [m2::({l2},{⊥})] ($r_1 : \langle \mathsf{l2} \rangle^{l2}$, $r_2 : \langle \mathsf{m2} \rangle^{m2}$) **requires** (l1)

Then, algorithm $\mathscr{A}$ creates an initial environment Ψ_0 by generating twelve variables (ρ_1 to ρ_{12}) to annotate the six locks (l_i and m_i) in the three code blocks that mention locks (liftLeftFork , liftRightFork , and eat). They are $l_1 :: (\rho_1, \rho_2), \ldots, m_3 :: (\rho_{11}, \rho_{12})$.

In the second pass, while in code block main, algorithm $\mathscr{I}$ generates six more permission variables (ρ_{13} to ρ_{18}) to annotate the new lock variables f_1 to f_3 introduced with the **newLock** instructions. They are: $f_1 :: (\rho_{13}, \rho_{14}) \ldots f_3 :: (\rho_{17}, \rho_{18})$. The rest of the second pass generates new constraints in type application and in jump-to-critical instructions. For example, in code block liftRightFork , and for value eat[l2 ,m2], four constraints are generated: $\rho_9 \prec l_2 \prec \rho_{10}, \rho_{11} \prec m_2 \prec \rho_{12}$. Then, in the jump-to-critical instruction, **if** r_3 = 0 **jump** eat[l2,m2], and since the thread holds lock l_2 (as witnessed by its signature **requires** l2), a new constraint $\{l_2\} \prec m_2$ is generated.

Related work The literature on type systems for deadlock freedom in lock-based languages is vast; space restrictions prohibit a general survey. We however believe that the problem of type inference for deadlock freedom in lock-based languages has been given not so much attention in high-level languages, let alone low-level (assembly) languages.

Our work is inspired by Flanagan and Abadi [10], but their language is functional and threads physically block. A lot of attention has been devoted to type-based deadlock freedom in object-oriented languages. For example, Boyapati *et al.* [5] use a variant of ownership types for prevent deadlocks in Java, performing partial inference of annotations, but not those related to lock order.

Another thread of research, orthogonal to ours, checks deadlocks at run-time. Cunningham *et al.* infer locks for atomicity in an object-oriented language, but use a runtime mechanism to detect when a thread's lock acquisition would cause a deadlock [9]. Java PathFinder [6] and Driver Verifier [3] identify violations of lock discipline during test runtime. Agarwal *et al.* present an algorithm that detects potential deadlocks involving any number of threads [1, 2].

Concluding remarks Capturing the notion of deadlocked state in an assembly language where no instruction blocks the processor is quite challenging. Our notion of deadlocked states captures only busy-wait lock-acquisition strategies, while taking into account suspended threads (cf. code block liftLeftFork in the figure). Sleep-lock strategies (see MIL code in reference [14]), where each thread trying to acquire the lock cooperatively fork a new copy of itself and releases the processor, are more difficult to characterise .

We have implemented the first phase of the algorithm, the part that generates an annotated program and a collection of constraints. We are working on having the constraints solved by adequate constraint solving.

References

[1] Rahul Agarwal and Scott D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *PADTAD'06: Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 51–60, New York, NY, USA, 2006. ACM.

[2] Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. In *In Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) Track of the 2005 IBM Verification Conference*, pages 191–207. Springer, 2005.

[3] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, 2006.

[4] Andrew Birrell. An introduction to programming with threads. Technical Report 35, Digital Systems Research Center, 1989.

[5] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of OOPSLA'02*, pages 211–230. ACM, 2002.

[6] Guillaume Brat, Klaus Havelund, SeungJoon Park, and Willem Visser. Java pathfinder — second generation of a java model checker. In *In Proceedings of the Workshop on Advances in Verification*, 2000.

[7] Bryan Cantrill and Jeff Bonwick. Real-world concurrency. *Queue*, 6(5):16–25, 2008.

[8] Tiago Cogumbreiro, Francisco Martins, and Vasco T. Vasconcelos. Compiling the pi-calculus into a multithreaded typed assembly language. In *PLACES 2008 — 1st International Workshop in Programming Language Approaches to Concurrency and Communication-cEntric Software*, ENTCS. Elsevier Science Publishers, 2009.

[9] David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Lock Inference Proven Correct. In *Proceedings of FTfJP'08*, 2008.

[10] Cormac Flanagan and Martín Abadi. Types for safe locking. In S. Doaitse Swierstra, editor, *Proceedings of ESOP'99*, volume 1576 of *LNCS*, pages 91–108. Springer, 1999.

[11] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. Technical report, University of Rochester, Rochester, NY, USA, 1994.

[12] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programing Language and Systems*, 21(3):527–568, 1999.

[13] Nir Shavit. Technical perspective transactions are tomorrow's loads and stores. *Commun. ACM*, 51(8):90–90, 2008.

[14] Vasco T. Vasconcelos and Francisco Martins. A multithreaded typed assembly language. In Ganesh Gopalakrishnan and John O'Leary, editors, *Proceedings of TV'06*, pages 133–141, 2006.

A Concurrent Language with a Uniform Treatment of Regions and Locks

Prodromos Gerakios Nikolaos Papaspyrou Konstantinos Sagonas
School of Electrical and Computer Engineering, National Technical University of Athens, Greece
{pgerakios,nickie,kostis}@softlab.ntua.gr

Abstract

A challenge for programming language research is to design and implement multi-threaded low-level languages providing static guarantees for memory safety and freedom from data races. Towards this goal, we sketch a concurrent language employing safe region-based memory management and hierarchical locking of regions. Both regions and locks are treated uniformly, and the language supports ownership transfer, early deallocation of regions and early release of locks in a safe manner.

1 Introduction

Writing safe and robust code is a hard task; writing safe and robust multi-threaded low-level code is even harder. In this paper we sketch a minimal, low-level concurrent language with advanced region-based memory management and hierarchical lock-based synchronization primitives.

Region-based memory management achieves efficiency by bulk allocation and deallocation of objects in segments of memory called regions. Similar to other approaches, our regions are organized in a hierarchical manner such that each region is physically allocated within a single parent region and may contain multiple children regions. This hierarchical structure imposes an ownership relation as well as lifetime constraints over regions. Unlike other languages employing hierarchical regions, our language allows early subtree deallocation in the presence of region sharing between threads. In addition, no memory leaks are possible as each thread is obliged to release each region it owns by the end of its scope.

Multi-threaded programs that interact through shared memory generate random execution interleavings. A data race occurs in a multi-threaded program when there exists an interleaving such that some thread accesses a memory location while some other thread attempts to write to it. So far, type systems and analyses that guarantee race freedom [5] have mainly focused on lexically-scoped constructs. The key idea in those systems is to statically track or infer the lockset held at each program point. In the language presented in this paper, implicit re-entrant locks are used to protect regions from data races. Our locking primitives are non-lexically scoped. Locks also follow the hierarchical structure of regions so that each region is protected by its own lock as well as the locks of all its ancestors, in contrast with ownership types [2].

Furthermore, our language allows regions and locks to be safely aliased, escape the lexical scope when passed to a new thread, or become logically separated from the remaining hierarchy. These features are invaluable for expressing numerous idioms of multi-threaded programming such as *sharing*, *region ownership* or *lock ownership transfers*, *thread-local regions* and *region migration*.

2 Language Design

We briefly outline the main design goals for our language, as well as some of the main design decisions that we made to serve these goals.

Low-level and concurrent. Our language must efficiently support systems programming. As such, it should cater for memory management and concurrency. It also needs to be low-level: it is not intended to be used by programmers but as the target language of higher-level systems programming languages.

Static safety guarantees. We define safety in terms of *memory safety* and absence of *data races*. A static type system should guarantee that well-typed programs are safe, with minimal run-time overhead.

Safe region-based memory management. Similarly to other languages for safe systems programming (e.g. Cyclone) our language employs region-based memory management, which achieves efficiency by *bulk allocation* and *deallocation* of objects in segments of memory (*regions*). Statically typed regions [11, 12] guarantee the absence of dangling pointer dereferences, multiple release operations of the same memory area, and memory leaks. Traditional stack-based regions [11] are limiting as they cannot be deallocated early. Furthermore, the stack-based discipline fails to model region lifetimes in concurrent languages, where the lifetime of a shared region depends on the lifetime of the longest-lived thread accessing that region. In contrast, we want regions that can be *deallocated early* and that can safely be *shared* between concurrent threads.

We opt for a *hierarchical region* [7] organization: each region is physically allocated within a single parent region and may contain multiple children regions. Early region deallocation in our multi-level hierarchy automatically deallocates the immediate subtree of a region without having to deallocate each region of the subtree recursively. The hierarchical region structure imposes the constraint that a child region is *accessible* only when its ancestors are accessible. In order to allow a function to access a region without having to pass all its ancestors explicitly, we allow its ancestors to be abstracted (i.e., our language supports *hierarchy abstraction*) for the duration of the function call. To maintain the *accessibility* invariant we require that the abstracted parents are *accessible* before and after the call. Regions whose parent information has been abstracted cannot be passed to a new thread as this may be unsound.

Race freedom. To prevent data races we use *lock-based* mutual exclusion. Instead of having a separate mechanism for locks, we opt for a uniform treatment of locks and regions: locks are placed in the same hierarchy as regions and enjoy similar properties. Each region is protected by its own private lock and by the locks of its ancestors. The semantics of region locking is that the entire subtree of a region is *atomically locked* once the lock for that region has been acquired. Hierarchical locking can model complex synchronization strategies and lifts the burden of having to deal with explicit acquisition of multiple locks. Although deadlocks are possible, they can be *avoided* by acquiring a single lock for a group of regions rather than acquiring multiple locks for each region separately. Additionally, our language provides explicit locking primitives, which in turn allow a higher degree of concurrency than lexically-scoped locking, as some locks can be released early.

Region polymorphism and aliasing. Our language supports *region polymorphism*: it is possible to pass regions as parameters to functions or concurrent threads. This enables *region aliasing*: one actual region could be passed in the place of two distinct formal region parameters. In the presence of mutual exclusion and early region deallocation, aliasing is dangerous. Our language allows safe region aliasing with minimal restrictions. The mechanism that we employ for this purpose also allows us to encode numerous useful idioms of concurrent programming, such as *region migration*, *lock ownership transfers*, *region sharing*, and *thread-local regions*.

3 Language Description

The syntax of the language is illustrated in Figure 1. We only present the subset of the language that is related to regions and references. The language core comprises variables (x), constants (c), functions, and function application. Functions can be region polymorphic ($\Lambda\rho.\, f$) and region application is explicit ($e[\rho]$). Monomorphic functions ($\lambda x.\, e$) must be annotated with their type. The application of monomorphic functions is annotated with a *calling mode* (ξ), which is seq for normal (sequential) application and $\mathsf{par}(\epsilon)$ for spawning a new thread (parallel). Parallel application is annotated with the list of regions (ϵ) that migrate to the spawned thread. This annotation can be automatically inferred by the type checker. The constructs for manipulating references are standard. A newly allocated memory cell is returned by

Function	f	::=	$\lambda x.e\ \texttt{as}\ \tau \xrightarrow{\gamma\to\gamma} \tau \mid \Lambda\rho.f$
Expression	e	::=	$x \mid c \mid f \mid (e\ e)^{\xi} \mid e[\rho] \mid \texttt{new}\ e\ \texttt{at}\ e \mid e := e$
		|	$\texttt{deref}\ e \mid \texttt{newrgn}\ \rho, x\ \texttt{at}\ e\ \texttt{in}\ e \mid \texttt{cap}_{\eta}\ e$
Type	τ	::=	$b \mid \tau \xrightarrow{\gamma\to\gamma} \tau \mid \forall\rho.\tau \mid \texttt{ref}(\tau,\rho) \mid \texttt{rgn}(\rho)$
Effect	γ	::=	$\emptyset \mid \gamma, \rho^{\kappa} \triangleright \pi$

Calling mode	ξ	::=	$\mathsf{seq} \mid \mathsf{par}(\epsilon)$
Capability op	η	::=	$\psi+ \mid \psi-$
Capability kind	ψ	::=	$\mathsf{rg} \mid \mathsf{lk}$
Capability	κ	::=	$n,n \mid \overline{n,n}$
Region parent	π	::=	$\rho \mid \bot \mid ?$
Region list	ϵ	::=	$\emptyset \mid \epsilon, \rho$

Figure 1: Syntax.

$\texttt{new}\ e_1\ \texttt{at}\ e_2$, where e_1 is the value that will be placed in the cell and e_2 is a handle of the region in which the new cell will be allocated. Standard assignment and dereference operators complete the picture.

The construct $\texttt{newrgn}\ \rho, x\ \texttt{at}\ e_1\ \texttt{in}\ e_2$ allocates a new region ρ with a region *handle* bound to x. The new region resides in a *parent* region, whose handle is given in e_1. The scope of ρ and x is e_2, which is supposed to do something useful with the new region. The evaluation of e_2 is *obliged to consume* the new region by the end of its scope. A region can be consumed either by deallocation or by transferring its ownership to another thread. At any given program point, each region is associated with a *capability* (κ). Capabilities consist of two natural numbers, the *capability counts*: the *region* count, which denotes whether the region is live, and the *lock* count, which denotes whether the region has been locked to provide the current thread with exclusive access to its contents. When first allocated, a region starts with capability $(1,1)$, meaning that it is live and locked for providing exclusive access to the thread which allocated it (this our equivalent of a thread-local region).

By using the construct $\texttt{cap}_{\eta}\ e$, a thread can *increment* or *decrement* the capability counts of the region whose handle is specified in e. The capability operator η can be, e.g., $\mathsf{rg}+$ (meaning that the region count is to be incremented) or $\mathsf{lk}-$ (meaning that the lock count is to be decremented). If the region count reaches zero, then the region may be physically deallocated and no subsequent operation on it can be performed. If the lock count becomes zero, then the region is unlocked. Capability counts determine the validity of operations on regions and references. All operations require that the involved regions are live, i.e., the region count is greater than zero. Assignment and dereference can be performed only when the corresponding region is live and locked.

A capability of the form (n_1,n_2) is called a *pure* capability, whereas a capability of the form $(\overline{n_1,n_2})$ is called an *impure* capability. In both cases, it is implied that the current thread can decrement the region count n_1 times and the lock count n_2 times. Impure capabilities are obtained by splitting, in the same spirit as *fractional capabilities* [4], pure or other impure capabilities into several pieces, e.g., $(3,2) = (\overline{2,1}) + (\overline{1,1})$. These pieces are useful for region aliasing, when the same region is to be passed to a function in the place of two distinct region parameters. An impure capability implies that our knowledge of the region and lock count is inexact. The use of such capabilities must be restricted; e.g., an impure capability with a non-zero lock count cannot be passed to another thread, as it is unsound to allow two threads to simultaneously hold the same lock. Capability splitting takes place automatically with function application.

4 Static Semantics

In this section we discuss the most interesting parts of our type system. To enforce our safety invariants, we use a *type and effect system*. Effects are used to statically track the capability of each region. An effect (γ) is a list of elements of the form $\rho^{\kappa} \triangleright \pi$, denoting that region ρ is associated with capability κ and has parent π, which can be another region, $\bot$, or ?. Regions whose parents are $\bot$ or ? are considered as roots in our region hierarchy. We assume that there is an initial (physical) root region corresponding to

$$\frac{\Delta \vdash \tau \qquad \tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \qquad \Delta;\Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, (\gamma_1;\gamma_2)}{\Delta;\Gamma \vdash \lambda x.e \text{ as } \tau : \tau \,\&\, (\gamma;\gamma)} \;(T\text{-}F) \qquad \frac{\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \,\&\, (\gamma;\gamma') \qquad \Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma';\gamma'') \qquad \xi = \mathsf{par}(\epsilon) \Rightarrow \tau_2 = \langle\rangle \qquad \xi \vdash \gamma''' = \gamma_2 \oplus (\gamma'' \ominus \gamma_1)}{\Delta;\Gamma \vdash (e_1\ e_2)^{\xi} : \tau_2 \,\&\, (\gamma;\gamma''')} \;(T\text{-}AP)$$

$$\frac{\Delta;\Gamma \vdash e : \mathtt{ref}(\tau,\rho) \,\&\, (\gamma;\gamma') \qquad \rho \in \mathit{accessible}(\gamma')}{\Delta;\Gamma \vdash \mathtt{deref}\ e : \tau \,\&\, (\gamma;\gamma')} \;(T\text{-}D) \qquad \frac{\Delta;\Gamma \vdash e_1 : \mathtt{rgn}(\rho') \,\&\, (\gamma;\gamma') \qquad \rho' \in \mathit{dom}(\gamma') \qquad \Delta \vdash \tau \qquad \Delta,\rho;\Gamma, x : \mathtt{rgn}(\rho) \vdash e_2 : \tau \,\&\, (\gamma', \rho^{1,1} \triangleright \rho';\gamma'') \qquad \rho \notin \mathit{dom}(\gamma'')}{\Delta;\Gamma \vdash \mathtt{newrgn}\,\rho, x \ \mathtt{at}\ e_1 \ \mathtt{in}\ e_2 : \tau \,\&\, (\gamma;\gamma'')} \;(T\text{-}NG)$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau \,\&\, (\gamma;\gamma') \qquad \Delta;\Gamma \vdash e_2 : \mathtt{rgn}(\rho) \,\&\, (\gamma';\gamma'') \qquad \rho \in \mathit{dom}(\gamma'')}{\Delta;\Gamma \vdash \mathtt{new}\ e_1 \ \mathtt{at}\ e_2 : \mathtt{ref}(\tau,\rho) \,\&\, (\gamma;\gamma'')} \;(T\text{-}NR) \qquad \frac{\Delta;\Gamma \vdash e_1 : \mathtt{rgn}(\rho) \,\&\, (\gamma;\gamma', \rho^{\kappa} \triangleright \pi) \qquad \kappa' = [\![\eta]\!](\kappa) \qquad \gamma'' = \mathit{live}(\gamma', \rho^{\kappa'} \triangleright \pi)}{\Delta;\Gamma \vdash \mathtt{cap}_{\eta}\ e_1 : \langle\rangle \,\&\, (\gamma;\gamma'')} \;(T\text{-}CP)$$

Figure 2: Selected typing rules.

$$\frac{\xi \vdash \gamma = \gamma_1 \oplus \gamma_r \qquad \xi \vdash \gamma' = \gamma_2 \oplus \gamma_r \qquad \gamma'' = \mathit{live}(\gamma') \qquad \mathit{consistent}(\gamma,\gamma'') \qquad \xi = \mathsf{seq} \Rightarrow \mathit{abs\_par}(\gamma,\gamma_1) \subseteq \mathit{dom}(\gamma'') \qquad \xi = \mathsf{par}(\epsilon) \Rightarrow \gamma_2 = \emptyset \wedge \epsilon = \mathit{pure}(\gamma) \cap \mathit{dom}(\gamma_1)}{\xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)} \;(ESJ)$$

$$\frac{}{\xi \vdash \gamma = \emptyset \oplus \gamma} \;(ES\text{-}N) \qquad \frac{\pi' \in \{\pi, ?\} \qquad \xi = \mathsf{par}(\epsilon) \Rightarrow \pi' \neq ? \qquad \xi \vdash \kappa = \kappa_1 + \kappa_2 \qquad \xi \vdash \gamma = \gamma_1 \oplus \gamma_2}{\xi \vdash \gamma, r^{\kappa} \triangleright \pi = \gamma_1, r^{\kappa_1} \triangleright \pi' \oplus \gamma_2, r^{\kappa_2} \triangleright \pi} \;(ES\text{-}C)$$

$$\frac{rg(\kappa) = rg(\kappa_1) + rg(\kappa_2) \qquad lk(\kappa) = lk(\kappa_1) + lk(\kappa_2) \qquad rg(\kappa_1) > 0 \qquad \mathit{is\_pure}(\kappa_1) \Leftrightarrow \mathit{is\_pure}(\kappa_2) \qquad \mathit{is\_pure}(\kappa_1) \Rightarrow \kappa = \kappa_1 \qquad \xi \neq \mathsf{seq} \wedge \neg \mathit{is\_pure}(\kappa_1) \Rightarrow lk(\kappa_2) = 0}{\xi \vdash \kappa = \kappa_1 + \kappa_2} \;(CS)$$

Figure 3: Effect and capability splitting.

the entire heap, whose handle is available to the main program. The parent of the heap region is $\bot$. More (logical) root regions can be created using hierarchy abstraction. The abstract parent of a region that is passed to a function is denoted by ?.

The syntax of types in Figure 1 (on page 3) is more or less standard. A collection of base types b is assumed; the syntax of values belonging to these types and operations upon such values is omitted from this paper. We assume the existence of a *unit* base type, which we denote by $\langle\rangle$. Region handle types $\mathtt{rgn}(\rho)$ and reference types $\mathtt{ref}(\tau,\rho)$ are associated with a type-level region name ρ. Monomorphic function types carry an *input* and an *output effect*. A well-typed expression e has a type τ under an input effect γ and results in an output effect γ'. We denote this by $\Delta;\Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$. The typing relation (cf. Figure 2) uses two standard typing contexts: Δ, a set of region variables, and Γ, a mapping of term variables to types. The effects that appear in our typing relation must satisfy a *liveness invariant*: all regions that appear in the effect are *live*, i.e. their region counts and those of all their ancestors are non-zero. Thus, in order to check whether a region ρ is live in the current effect γ, we only need to check that $\rho \in \mathit{dom}(\gamma)$.

The typing rule for lambda abstraction (*T-F*) requires that the body e is well-typed with respect to the effects ascribed on its type. The typing rule for function application (*T-AP*) splits the output effect of e_2 (γ'') by subtracting the function's input effect (γ_1). It then joins the remaining effect with the function's output effect (γ_2). In the case of parallel application, rule *T-AP* also requires that the return type is unit. The splitting and joining of effects is controlled by the judgement $\xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$, which is defined in Figure 3 (the auxiliary functions and predicates are defined in Figures 4 and 5). Its definition enforces the following properties:

- the liveness invariant for γ'';
- the consistency of γ and γ'', i.e. regions cannot change parent and capabilities cannot switch from pure to impure or vice versa;

$$\frac{(\rho^{\kappa} \triangleright \pi) \in \gamma \qquad \pi \in \{\bot, ?\}}{\mathit{is\_live}(\gamma, \rho)} \qquad \frac{(\rho^{\kappa} \triangleright \rho') \in \gamma \qquad \mathit{is\_live}(\gamma, \rho')}{\mathit{is\_live}(\gamma, \rho)}$$

$$\frac{(\rho^{\kappa} \triangleright \pi) \in \gamma \qquad \mathit{lk}(\kappa) > 0}{\mathit{is\_accessible}(\gamma, \rho)} \qquad \frac{(\rho^{\kappa} \triangleright \rho') \in \gamma \qquad \mathit{is\_accessible}(\gamma, \rho')}{\mathit{is\_accessible}(\gamma, \rho)}$$

Figure 4: Auxiliary predicates: region liveness and accessibility.

$$\begin{array}{lcl}
\mathit{rg}(\kappa) & = & n_1 \quad \text{if } \kappa = n_1, n_2 \vee \kappa = \overline{n_1, n_2} \\
\mathit{lk}(\kappa) & = & n_2 \quad \text{if } \kappa = n_1, n_2 \vee \kappa = \overline{n_1, n_2} \\
\mathit{dom}(\gamma) & = & \{\rho \mid (\rho^{\kappa} \triangleright \pi) \in \gamma\} \\
\mathit{live}(\gamma) & = & \{\rho^{\kappa} \triangleright \pi \mid (\rho^{\kappa} \triangleright \pi) \in \gamma \wedge \mathit{is_live}(\gamma, \rho)\} \\
\mathit{accessible}(\gamma) & = & \{\rho \mid (\rho^{\kappa} \triangleright \pi) \in \gamma \wedge \mathit{is_accessible}(\gamma, \rho)\} \\
\mathit{is_pure}(\kappa) & = & \exists n_1.\, \exists n_2.\, \kappa = n_1, n_2 \\
\mathit{pure}(\gamma) & = & \{r^{\kappa} \triangleright \pi \mid (r^{\kappa} \triangleright \pi) \in \gamma \wedge \mathit{is_pure}(\kappa)\} \\
\mathit{consistent}(\gamma_1, \gamma_2) & = & \forall (\rho^{\kappa} \triangleright \pi) \in \gamma_1.\, \forall (\rho^{\kappa'} \triangleright \pi') \in \gamma_2.\, \pi = \pi' \wedge (\mathit{is_pure}(\kappa) \Leftrightarrow \mathit{is_pure}(\kappa')) \\
\mathit{abs_par}(\gamma_1, \gamma_2) & = & \left\{\rho \mid (\rho^{\kappa} \triangleright \rho') \in \gamma_1 \wedge (\rho^{\kappa'} \triangleright ?) \in \gamma_2\right\}
\end{array}$$

Figure 5: Auxiliary functions and predicates.

- in the case of sequential application, regions that pass under hierarchy abstraction must be live after the function returns;
- in the case of parallel application, the thread output effect is empty, the thread input effect does not contain impure capabilities with non-zero lock counts, no hierarchy abstraction is permitted, and regions that migrate are exactly those given by the annotation ϵ.

The typing rules for references are standard. Here we only show the rules for dereference (*T-D*) and reference allocation (*T-NR*). The former checks that the region ρ where the reference resides is *accessible*, i.e. ρ itself or one of its ancestors has a non-zero lock count in the current effect. The latter just checks that the region ρ is live. The rule for creating new regions (*T-NG*) first checks that e_1 is a handle for some live region ρ'. The body expression e_2 is type checked in a context extended with the fresh region ρ and its handle x. A new element $\rho^{1,1} \triangleright \rho'$ is added to the input effect of e_2, stating that the new region is thread-local (live, locked and not known to anybody else). The rule also checks that the type of the result τ and the output effect γ'' do not contain any occurrence of region variable ρ. This implies that e_2 must have consumed the new region by the end of its scope. The capability manipulation rule (*T-CP*) checks that the given expression is the handle of a live region ρ. It then modifies the capability count of the effect element corresponding to that region, as dictated by function $[\![\eta]\!]$ which increases or decreases the region or the lock count of its argument, according to the value of η. The dynamic semantics makes sure that evaluation returns only when the actual capability counts are consistent with the desired output effect. For instance, if the lock of region ρ is held by some other executing thread, evaluation of $\mathtt{cap}_{\mathsf{lk}+}$ must be suspended until the lock can be obtained. On the other hand, evaluation of $\mathtt{cap}_{\mathsf{rg}-}$ does not need to suspend but may not be able to physically deallocate a region, as it may be in use by other threads.

5 Related Work

The first statically checked stack-based region system was developed by Tofte and Talpin [11]. Since then, several memory-safe systems that enabled early region deallocation for a sequential language were proposed [1, 10, 13, 6]. Cyclone [9] and RC [7] were the first imperative languages to allow safe region-based management with explicit constructs. They both allowed early region deallocation and RC also

introduced the notion of multi-level region hierarchies. RC programs may throw region-related exceptions, whereas our approach is purely static. Both Cyclone and RC make no claims of memory safety or race freedom for concurrent programs. In the context of Cyclone, Grossman proposed a type system for safe multi-threading [8]. Race freedom is guaranteed by statically tracking locksets within lexically-scoped synchronization constructs. Grossman's proposal allows for fine-grained locking, but only deals with stack-based regions and does not enable early release of regions and locks. Furthermore, we offer hierarchical locking as opposed to just primitive locking. Bulk region deallocation is impossible in Grossman's system.

Statically checked region systems have also been proposed [3, 15, 14] for real-time Java to rule out dynamic checks imposed by its specification. Boyapati et al. [3] introduce hierarchical regions in ownership types but the approach suffers from the same disadvantages as Grossman's work. Additionally, their type system only allows sub-regions for *shared* regions, whereas we do not have this limitation. Boyapati also proposed an ownership-based type system that prevents deadlocks and data races [2]. Static region hierarchies (depth-wise) have been used by Zhao [15]. Their main advantage is that programs require fewer annotations compared to programs with explicit region constructs. In the same track, Zhao et al. [14] proposed implicit ownership annotations for regions. Thus, classes that have no explicit owner can be allocated in any static region. This is a form of *existential ownership*. In contrast, we allow a region to completely abstract its owner/ancestor information by using the *hierarchy abstraction* mechanism. None of the above approaches allow full ownership abstraction for region subtrees.

The main limitation of our work is that we require explicit annotations regarding ownership and region capabilities. Moreover, our locking system offers coarser-grained locking compared to [5] and related type systems. The use of hierarchical locking avoids some, though not all, deadlocks.

References

[1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, New York, NY, USA, June 1995. ACM Press.

[2] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, New York, NY, USA, Nov. 2002. ACM Press.

[3] C. Boyapati, A. Salcianu, W. S. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 324–337, New York, NY, USA, June 2003. ACM Press.

[4] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: Proceedings of the 10th International Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, June 2003.

[5] C. Flanagan and M. Abadi. Object types against races. In J. C. M. Baeten and S. Mauw, editors, *Concurrency Theory: Proceedings of the 10th International Conference*, volume 1664 of *LNCS*, pages 288–303. Springer, 1999.

[6] M. Fluet, G. Morrisett, and A. Ahmed. Linear regions are all you need. In P. Sestoft, editor, *Programming Language and Systems: Proceedings of the European Symposium on Programming*, volume 3924 of *LNCS*, pages 7–21. Springer, Mar. 2006.

[7] D. Gay and A. Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, New York, NY, USA, May 2001. ACM Press.

[8] D. Grossman. Type-safe multithreading in Cyclone. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 13–25, New York, NY, USA, Jan. 2003. ACM Press.

[9] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293, New York, NY, USA, June 2002. ACM Press.

[10] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 175–186, New York, NY, USA, 2001. ACM.

[11] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, New York, NY, USA, Jan. 1994. ACM Press.

[12] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Trans. Prog. Lang. Syst.*, 22(4):701–771, July 2000.

[13] D. Walker and K. Watkins. On regions and linear types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 181–192, New York, NY, USA, Oct. 2001. ACM Press.

[14] T. Zhao, J. Baker, J. Hunt, J. Noble, and J. Vitek. Implicit ownership types for memory management. *Sci. Comput. Program.*, 71(3):213–241, 2008.

[15] T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 241–251. IEEE Computer Society, 2004.

Session-based Type Discipline for Pi Calculus with Matching

Marco Giunti* Kohei Honda† Vasco T. Vasconcelos* Nobuko Yoshida‡

Introduction. In [7] we have introduced an extension of the first session typing system [10] that allows higher-order session communication. In the new system, the reduction rule for session passing

$$k![k'].P \mid k?(k').Q \rightarrow P \mid Q$$

does not allow the transmission of an arbitrary channel. In most situations a receiving process $k?(k'').Q$ can be alpha-converted ahead of communication so that the bound channel k'' syntactically matches the free channel k' in the object of the sending process [11]. The exception happens exactly when k' is free in Q: alpha-conversion becomes impossible (for it would capture the free variable k'), and communication cannot occur.

A more liberal rule allows the transmission of an arbitrary channel, implying a substitution on the client side.

$$k![k'].P \mid k?(x).Q \rightarrow P \mid Q[k'/x]$$

Unfortunately this rule breaks subject reduction, a technique that is commonly used to prove that typable processes do not reduce to errors. A counterexample is a process which, possessing one end of a channel, receives the second end. The process:

$$k![k'] \mid k?(k'').k''?(y).k'![1]$$

is typable in the system of [7] under typing $k\colon \bot, k'\colon \bot$, but reduces to process $k'?(x).k'![1]$ which is not typable under the *same typing* [4]. To tackle this problem and recover subject reduction, starting from Gay and Hole [6], many works resort to decorate channel-ends with polarities. In this paper we show that the polarity-free language of reference [7] extended with the more liberal session-passing rule above is type-safe, even though it does not enjoy subject reduction.

A second contribution of the paper is a simple generalization of the theory of session types that allows processes to both send a channel and still use it, under limited conditions. The idea is that processes can safely use channels both for communication and for non-communication operations, such as testing the identity of a session, or storing the channel in some data structure. The example below illustrates the idea.

Consider a system with long running sessions composed of different degrees of trust. The protocol relies on a database of malicious sessions that offers operations to add and to check for untrusted channels; the database runs in parallel with the web service. Untrusted channels are stored in a set implemented as a pi calculus process and are typed with end. By providing for channel matching, we can implement a process $\mathsf{if}\ y \in \mathsf{set}\ \mathsf{then}\ P\ \mathsf{else}\ Q$ that continues as P if channel y is stored in the set of untrusted channels, and continues as Q otherwise. The insertion of a new entry y in the set is done by the process $\mathsf{Store}[y].P$. The code for the database process is below; we annotate variables with the type at which they are used.

$$\mathsf{DB}(d) = \mathsf{accept}\ d(x).x \rhd \{\mathsf{contains} : x?(y^{\mathsf{end}}).\mathsf{if}\ y \in \mathsf{set}\ \mathsf{then}\ x \lhd \mathsf{no}.\mathsf{DB}(d) \mathsf{else}\ x \lhd \mathsf{yes}.\mathsf{DB}(d),$$
$$\mathsf{put} : x?(y^{\mathsf{end}}).\mathsf{Store}[y].\mathrm{DB}(d)\}$$

Instances of the service that need an high level of confidentiality, whenever they receive a channel at some type T, send the channel to the database at type end and wait for an ack ensuring that the session is trusted before using it. The fragment of the code of a client querying the database for the trust of a channel is below. Notice that after passing the session at type end the continuation uses it at type T.

$$x?(y).\mathsf{request}\ d(z).z \lhd \mathsf{contains}.z![y^{\mathsf{end}}].z \rhd \{\mathsf{yes} : P(y^T)\ [\!]\ \mathsf{no} : Q(y^T)\}$$

The service itself is in charge of signaling untrusted channels to the database. Such channels could still be used by instances of the service that do not require confidentiality, for instance when sessions are meant for exchanging public data. The code below describes an instance sending an untrusted channel y at type T to the context and signaling the channel at type end to the database.

$$x![y^T].\mathsf{request}\ d(z).z \lhd \mathsf{put}.z![y^{\mathsf{end}}].$$

*Department of Informatics, University of Lisbon
†Department of Computer Science, Queen Mary University of London
‡Department of Computing, Imperial College of London

$$
\begin{array}{rll}
P & ::= u![e].P \mid u?(x).P \mid P \mid P' \mid (\nu a)P \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P' \mid \mathbf{0} & \\
& \mid \mathsf{def}\ D\ \mathsf{in}\ P \mid X[\tilde{e}] & \text{recursion} \\
& \mid \mathsf{request}\ u(x).P \mid \mathsf{accept}\ u(x).P & \text{session request / acceptance} \\
& \mid u \lhd l.P \mid u \rhd \{l_i\colon P_i\}_{i\in I} & \text{label selection / branching} \\
e & ::= v \mid [u=v] \mid e\ \mathsf{and}\ e' \mid e\ \mathsf{or}\ e' \mid \mathsf{not}\ e & \text{expressions} \\
u,v & ::= a \mid x \mid \mathsf{true} \mid \mathsf{false} & \text{values} \\
D & ::= \{X_i(\tilde{x}_i) = P_i\}_{i\in I} & \text{declaration for recursion}
\end{array}
$$

Figure 1: The top level syntax of processes.

Pi calculus with sessions. We distinguish two languages: the *top-level language* and the *runtime* language. The former constitutes the language programmers program with; the latter includes constructs useful to describe the operational semantics, but that need not be accessible to programmers.

The top level language relies on a few base sets: *names*, ranged over by a, *variables* ranged over by x,y, *labels* ranged over by l, and *process variables* ranged over by X. The syntax is in Figure 1 and includes matching processes of the form $\mathsf{if}\ [u=v]\ \mathsf{then}\ P\ \mathsf{else}\ P'$.

The *runtime* language is the object of the operational semantics. It requires one more base set : *channels* ranged over by k. We indicate runtime processes with Q.

$$
\begin{array}{rll}
Q & ::= \langle \text{Fig. 1} \rangle \mid (\nu k)Q & \text{channel binder} \\
u,v & ::= \langle \text{Fig. 1} \rangle \mid k & \text{linear channel} \\
n & ::= a \mid k & \text{identifiers}
\end{array}
$$

The runtime language differs only in that (synchronization) identifiers include channels, so that, for example $k![\mathsf{true}].\mathbf{0}$ is a process (as opposed to $x![\mathsf{true}].\mathbf{0}$ in the base language). The bindings for the runtime language are processes $(\nu n)P$, which binds occurrences of the identifier n in P, i.e. $(\nu a)P$ or $(\nu k)P$, and $\mathsf{def}\ \{X_i(\tilde{x}_i) = P_i\}_{i\in I}\ \mathsf{in}\ P$, which binds occurrences of each process variable X_i in process P_i. The definition of *bound* and *free* names and channels is standard, and so is the capture-free *substitution* of a variable x with value v in a process P, denoted by $P[v/x]$. We implicitly assume that in all mathematical contexts all bound identifiers are pairwise disjoint and disjoint from the free identifiers. The operational semantics relies on structural congruence, for the syntactic rearrangement of processes, preparing these for the application of the rules in the reduction relation. *Structural congruence* is the smallest relation including the rules below:

$$
\begin{array}{c}
Q \mid \mathbf{0} \equiv Q \qquad Q \mid Q' \equiv Q' \mid Q \qquad (Q \mid Q') \mid Q'' \equiv Q \mid (Q' \mid Q'') \\
(\nu n)Q \mid Q' \equiv (\nu n)(Q \mid Q') \qquad (\nu n')(\nu n)Q \equiv (\nu n)(\nu n')Q \qquad (\nu n)\mathbf{0} \equiv \mathbf{0} \\
\mathsf{def}\ D\ \mathsf{in}\ \mathbf{0} \equiv \mathbf{0} \quad \mathsf{def}\ DD'\ \mathsf{in}\ Q \equiv \mathsf{def}\ D'D\ \mathsf{in}\ Q \quad \mathsf{def}\ D\ \mathsf{in}\ (\nu n)Q \equiv (\nu n)\mathsf{def}\ D\ \mathsf{in}\ Q \\
(\mathsf{def}\ D\ \mathsf{in}\ Q) \mid Q' \equiv \mathsf{def}\ D\ \mathsf{in}\ (Q \mid Q') \qquad \mathsf{def}\ D\ \mathsf{in}\ (\mathsf{def}\ D'\ \mathsf{in}\ Q) \equiv\ \mathsf{def}\ D\ D'\ \mathsf{in}\ Q
\end{array}
$$

Reduction also relies on a standard *evaluation* function $\downarrow$, reducing expressions to values. We assume $[u=u]\downarrow \mathsf{true}$, and $[u=v]\downarrow \mathsf{false}$ whenever $u \neq v$.

The reduction rules are in Figure 2. Sessions between two partners start when an accept process meets a request process, as described in rule [LINK]. The result of such interaction is the creation of a new channel k, that replaces both the bound variable x in the continuation Q of the accept process and the bound variable y in continuation Q' of request. In rule [COM], when a send process $k![e].Q$ meets a receive process $k?(x).Q$, the semantics start by evaluating expression e to a value v, which is passed from the send party to the receive party, replacing variable x in Q. In case the expression evaluates to a channel k, we have session passing.

Reduction may go wrong for a number of reasons. Here we are interested on *communication* errors, problems arising from a mismatch of the expectations of the partners involved in a particular interaction. Examples include the parallel composition of two partners both trying to output, as in $k![\mathsf{true}]. \mid k \lhd l.Q$, or even when three partners try to read/write on the same channel, as for example in $k![\mathsf{true}]. \mid k \lhd l.Q \mid k?(x).Q'$. The formal definition of what we mean by an error process is below [7].

$$\mathsf{accept}\ a(x).Q \mid \mathsf{request}\ a(y).Q' \ \rightarrow\ (\nu k)(Q[k/x] \mid Q'[k/y]) \qquad [\textsc{Link}]$$

$$e \downarrow v \ \Rightarrow\ k![e].Q \mid\ k?(x).Q' \ \rightarrow\ Q \mid Q'[v/x] \qquad [\textsc{Com}]$$

$$k \lhd l_j.Q \mid k \rhd \{l_i\colon Q_i\}_{i\in I} \ \rightarrow\ Q \mid\ Q_j \qquad (j \in I) \qquad [\textsc{Label}]$$

$$e \downarrow \mathsf{true} \ \Rightarrow\ \mathsf{if}\ e\ \mathsf{then}\ Q\ \mathsf{else}\ Q' \ \rightarrow\ Q \qquad [\textsc{IfT}]$$

$$e \downarrow \mathsf{false} \ \Rightarrow\ \mathsf{if}\ e\ \mathsf{then}\ Q\ \mathsf{else}\ Q' \ \rightarrow\ Q' \qquad [\textsc{IfF}]$$

$$\tilde{e} \downarrow \tilde{v} \Rightarrow \mathsf{def}\ X(\tilde{x}) = Q\ \mathsf{in}\ (X[\tilde{e}] \mid Q') \ \rightarrow\ \mathsf{def}\ X(\tilde{x}) = Q\ \mathsf{in}\ (Q[\tilde{v}/\tilde{x}] \mid Q') \qquad [\textsc{Def}]$$

$$Q \rightarrow Q' \ \Rightarrow\ (\nu n)Q \rightarrow (\nu n)Q' \qquad [\textsc{Scop}]$$

$$Q \rightarrow Q' \ \Rightarrow\ Q \mid Q'' \ \rightarrow\ Q'' \mid Q' \qquad [\textsc{Par}]$$

$$Q \rightarrow Q' \ \Rightarrow\ \mathsf{def}\ D\ \mathsf{in}\ Q \ \rightarrow\ \mathsf{def}\ D\ \mathsf{in}\ Q' \qquad [\textsc{Defin}]$$

$$Q' \equiv Q_1 \text{ and } Q_1 \rightarrow Q_2 \text{ and } Q_2 \equiv Q'' \ \Rightarrow\ Q' \rightarrow Q'' \qquad [\textsc{Str}]$$

Figure 2: Reduction

Definition 1 (Error Process)**.** *A* k-process *is a process prefixed by channel* k*; that is:* $k![e].Q$*,* $k?(x).Q$*,* $k \lhd l.Q$*, or* $k \rhd \{l_i\colon Q_i\}_{i\in I}$*. A* k-redex *is the parallel composition of two* k*-processes, either of form* $(k![e].Q \mid k?(x).Q')$ *or* $(k \lhd l.Q \mid k \rhd \{l_i\colon Q_i\}_{i\in I})$*. Then* Q *is an* error *if* $Q \equiv (\nu\tilde{n})(\mathsf{def}\ D\ \mathsf{in}\ (Q' \mid Q''))$ *where* Q' *is, for some* k*, the parallel composition of* either *two* k*-processes that do not form a* k*-redex,* or *three or more* k*-processes.*

In the following, we provide for a typing system for top level processes able to filter out all errors that can arise during the computation.

Type Assignment for the Top Level Language. We borrow from [7] the distinction between *sorts*, ranged over by S, and *types*, ranged over by T. Sorts are assigned to values: a boolean value has type bool, a name has a type $\langle T\rangle$ describing the interactive sessions it may engage upon. The interactive session, in turn, is described by a type T with the following grammar:

$$T \ ::= ?[S].T \ \mid ?[T].T \ \mid\ \&\{l_i\colon T_i\}_{i\in I} \ \mid\ \mathsf{end} \ \mid ![S].T \ \mid ![T].T \ \mid\ \oplus\{l_i\colon T_i\}_{i\in I} \ \mid\ t \ \mid\ \mu t.T$$

Types $![S].T$ and $?[S].T$ describe channels willing to send or to receive a value of sort S (that is a boolean value or a name) and then continue its interaction as prescribed by T. Types $![T].T$ and $?[T].T$ are similar, only that this time the value exchanged is a linear value (a channel) described by a type, rather than a shared value described by a sort. Differently from [7], in our framework the type end can also be used to type, e.g., conditional expressions. Types $\oplus\{l_i\colon T_i\}_{i\in I}$ and $\&\{l_i\colon T_i\}_{i\in I}$ represent channels ready to select (to send) a label or to branch on an incoming label. The two last type constructors allow for recursive type structures.

Duality is a central concept in the theory of session types. The function $\overline{T}$ yields the canonical dual of a session type T by exchanging ! with ?, and & with $\oplus$. The formal definition is below.

$$\overline{?[\alpha].T} = ![\alpha].\overline{T} \qquad \overline{\oplus\{l_i\colon T_i\}_{i\in I}} = \&\{l_i\colon \overline{T_i}\}_{i\in I} \qquad \overline{\mathsf{end}} = \mathsf{end}$$

$$\overline{![\alpha].T} = ?[\alpha].\overline{T} \qquad \overline{\&\{l_i\colon T_i\}_{i\in I}} = \oplus\{l_i\colon \overline{T_i}\}_{i\in I} \qquad \overline{\mu X.T} = \mu X.\overline{T} \qquad \overline{X} = X$$

The type system distinguishes the sorts assigned to names and boolean values, from the types assigned to channels. Types are treated linearly, a map from channels and variables into types T is denoted by the *type environment* Δ. Sorts are treated classically, a map from names and variables into sorts S is denoted by the *sort environment* Γ. Syntax $\Delta \cdot x\colon T$ denotes the disjoint map union of Δ and $(x\colon T)$. The notation is extended straightforwardly to sort and type environments.

We let the merge, noted $\otimes$, be the smallest commutative binary relation over types satisfying $T \otimes \mathsf{end} = T$. We extend the operation to type environments and let $\Delta \otimes (k : T) = \Delta \cdot k : T$ whenever $k \notin \mathrm{dom}(\Delta)$, otherwise if $\Delta(k) \otimes T$ is defined we let $\Delta \otimes k : T = \Delta'$ with Δ' differing from Δ only in $\Delta'(k) = \Delta(k) \otimes T$.

The typing rules for matching expressions are below. Notice that in rule for compare values taken from the type environment the types at which the values are known may be different. Particularly, to test the identity of a value it is sufficient that the value is known at type end.

$$\frac{\Gamma \vdash u\colon S, v\colon S}{\Gamma \vdash [u = v]\colon \mathsf{bool}} \qquad \frac{\Delta \vdash u\colon T, v\colon T'}{\Delta \vdash [u = v]\colon \mathsf{bool}}$$

$$\frac{\Gamma \vdash u\colon \langle T\rangle \qquad \Gamma \vdash P \triangleright \Delta \cdot x\colon T}{\Gamma \vdash \mathsf{accept}\ u(x).P \triangleright \Delta} \qquad \frac{\Gamma \vdash u\colon \langle T\rangle \qquad \Gamma \vdash P \triangleright \Delta \cdot x\colon \overline{T}}{\Gamma \vdash \mathsf{request}\ u(x).P \triangleright \Delta} \quad [\textsc{Acc}], [\textsc{Req}]$$

$$\frac{\Gamma \vdash e\colon S \qquad \Gamma \vdash P \triangleright \Delta \cdot u\colon T}{\Gamma \vdash u![e].P \triangleright \Delta \cdot u\colon\ ![S].T} \qquad \frac{\Gamma \cdot x\colon S \vdash P \triangleright \Delta \cdot u\colon T}{\Gamma \vdash u?(x).P \triangleright \Delta \cdot u\colon\ ?[S].T} \quad [\textsc{Send}], [\textsc{Rcv}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot u\colon U \cdot v\colon T_2 \quad T = T_1 \otimes T_2}{\Gamma \vdash u![v].P \triangleright \Delta \cdot u\colon\ ![T].U \cdot v : T_1} \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot u\colon U \cdot x\colon T}{\Gamma \vdash u?(x).P \triangleright \Delta \cdot u\colon\ ?[T].U} \quad [\textsc{Thr}],[\textsc{Cat}]$$

$$\frac{\Gamma \vdash P_i \triangleright \Delta \cdot u\colon T_i \qquad \forall i \in I}{\Gamma \vdash\ u \triangleright \{l_i\colon P_i\}_{i\in I} \triangleright \Delta \cdot u\colon\ \&\{l_i\colon T_i\}_{i\in I}} \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot u\colon T_j \qquad j \in I}{\Gamma \vdash u \triangleleft l_j.P \triangleright \Delta \cdot u\colon\ \oplus\{l_i\colon T_i\}_{i\in I}} \quad [\textsc{Br}], [\textsc{Sel}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash P' \triangleright \Delta'}{\Gamma \vdash P \mid P' \triangleright \Delta \otimes \Delta'} \qquad \frac{\Gamma \cdot a\colon S \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a)P \triangleright \Delta} \quad [\textsc{Conc}],[\textsc{NRes}]$$

$$\frac{\Gamma \vdash e\colon \mathsf{bool} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash P' \triangleright \Delta}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P' \triangleright \Delta} \qquad \frac{\Delta \vdash e\colon \mathsf{bool} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash P' \triangleright \Delta}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P' \triangleright \Delta} \quad [\textsc{IfN}],[\textsc{IfC}]$$

$$\frac{}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \qquad \frac{\Gamma \vdash \tilde{e}\colon \tilde{S}}{\Gamma \cdot X\colon \tilde{S}\tilde{T} \vdash X[\tilde{e}\tilde{u}] \triangleright \Delta \cdot \tilde{u}\colon \tilde{T}} \quad [\textsc{Inact}], [\textsc{Var}]$$

$$\frac{\Gamma \cdot X\colon \tilde{S}\tilde{T} \cdot \tilde{x}\colon \tilde{S} \vdash P \triangleright \tilde{y}\colon \tilde{T} \qquad \Gamma \cdot X\colon \tilde{S}\tilde{T} \vdash P' \triangleright \Delta}{\Gamma \vdash \mathsf{def}\ X(\tilde{x}\tilde{y}) = P\ \mathsf{in}\ P' \triangleright \Delta} \quad [\textsc{Def}]$$

Figure 3: Type system for top level syntax

In rule [THR], process $u![v].P$ is typed with channel environment $\Delta \cdot u\colon\ ![T].U \cdot v\colon T_1$ describing the fact that channel variable v of type T_1 is sent on channel variable u of type $![T].U$. Similarly to [3], in passing channels we split the capabilities and let the continuation process P use the passed channel at type T if is sent at type end, or at type end if it is sent at type T (in such case the value can be used only for comparing it or sending it for comparison), as witnessed by knowing v at type $T = T_1 \otimes T_2$ in the typing environment of the hypothesis. In rule [CONC], the channel environment of $P \mid Q$ is the merge of the environments Δ and Δ' for the two processes, provided the operation is defined: if a value is both in the domain of Δ and Δ', then at least one of the typings is end. Notice that, differently from many works on session and linear types, in rules [INACT] and [VAR] we do not require the type environment to contain only depleted resources (of type end). We discuss this point further in the conclusions. The remaining rules are as in [7] .

We are now in a position to state the main result of the paper.

Theorem 2 (Type Safety)**.** *Let $\Gamma \vdash P \triangleright \Delta$ with P a top-level process. If $P \to^* Q$, then Q is not an error.*

Notice that the type system we are considering is meant to type top level processes only. The straightforward extension of the type system to the runtime syntax, which amounts to allow runtime channels k as opposed to channel variables x both in linear environments and in processes, does not satisfy subject reduction, as the following example shows (notice that $a\colon \langle ?[\mathsf{bool}].\mathsf{end}\rangle \vdash P \triangleright \emptyset$):

$$P = \mathsf{request}\ a(x).x![\mathsf{true}].\mathbf{0} \mid \mathsf{accept}\ a(y).y?(z).\mathbf{0} \to (\nu k)(k![\mathsf{true}].\mathbf{0} \mid k?(z).\mathbf{0}),$$

The parallel composition on the right-hand side is not typable, for both operands will have k on the channel environment at a type different from end, making the merge of the environments in rule [CONC] not defined.

Outline of the Proof of Type Safety for Top Level Processes, Theorem 2. We define a mapping $[\![\cdot]\!]$ from a different runtime language to the base runtime language, and we prove that typable top-level processes do not reduce to errors by showing operational and error correspondence results .

The syntax for the *double binder runtime language* is obtained by replacing the binder $(\nu k)P$ of the runtime language with a *double binder* $(\nu cd)R$ [], where c,d are distinct identifiers [5]. The syntax of double binder processes R includes processes P of Figure 1; we define a reduction relation $R \hookrightarrow R'$ and a typing system for double binder processes. The encoding $[\![\cdot]\!]$ maps double binder processes $(\nu cd)R$ in runtime processes $(\nu k)Q$ such that both channel-ends c,d related by the double binder are mapped into a single runtime channel k bound in Q.

Let P be a typed top level process, and assume $P \to^* Q$. We need to show that Q is not an error. The proof proceeds with the following steps.

1. *Typing Correspondence*. We show that typed top level processes are typed double binder processes.
2. *Operational Correspondence* We prove that the mapping $[\![\cdot]\!]$ is sound:
$$P \rightarrow^* Q \implies \exists R \text{ such that } P \hookrightarrow^* R \text{ and } [\![R]\!] \equiv Q$$
3. *Type Safety of Double Binder Language*. We provide for the subject reduction of the double binder runtime language and in turn we prove that typable double-binder processes do not reduce to errors. We apply this result to (1) and infer that
$$R \text{ not an error}$$
4. *Error Correspondence*. We prove that
$$R \text{ not an error} \implies [\![R]\!] \text{ not an error}$$
5. *Error Congruence*. We glue (2) and (4) and show that
$$([\![R]\!] \equiv Q \text{ and } [\![R]\!] \text{ not an error}) \implies Q \text{ not an error}$$

The remainder of paper details the various steps.

The double binder runtime language. As mentioned above we introduce a *double binder* $(\nu cd)R$ construct, that must not be confused with notation $(\nu c,d)R$ often used in the literature to indicate the process $(\nu c)(\nu d)R$. We underline that the top level language is a sub language of both runtime languages.

The semantics of the new language is defined over *configurations* of the form $\sigma \diamond R$ where σ is an irreflexive, symmetric and functional binary relation over channels, which we call *channel-end connection*. We let $\mathrm{dom}(\sigma)$ be the domain of σ. We write $\sigma \cdot (c,d)$ to indicate the union $\sigma \cup (c,d)$, whenever $\{c,d\} \cap \mathrm{dom}(\sigma) = \emptyset$. We let $\equiv_{\mathsf{db}}$ be the structural congruence relation for double binder processes, which is obtained straightforwardly from the structural rules for runtime processes e.g. $(\nu cd)R \mid R' \equiv_{\mathsf{db}} (\nu cd)(R \mid R')$.

The interesting rules for the reduction are below; the remaining rules are obtained by extending the rules in Figure 2 to configurations.

$$\sigma \diamond (\mathsf{accept}\ a(x).R \mid \mathsf{request}\ a(y).R) \rightarrow \sigma \diamond ((\nu cd)(R[c/x] \mid R'[d/y])) \qquad [\textsc{LinkD}]$$
$$c\,\sigma\,d \wedge e \downarrow v \implies \sigma \diamond (c![e].R \mid d?(y).R') \rightarrow \sigma \diamond (R \mid R'[v/y]) \qquad [\textsc{ComD}]$$
$$c\,\sigma\,d \wedge j \in I \implies \sigma \diamond (c \lhd l_j.R \mid d \rhd \{l_i\colon R_i\}_{i\in I}) \rightarrow \sigma \diamond (R \mid R_j) \qquad [\textsc{LabelD}]$$
$$\sigma \cdot (c,d) \diamond R \rightarrow \sigma \cdot (c,d) \diamond R' \implies \sigma \diamond (\nu cd)R \rightarrow \sigma \diamond (\nu cd)R' \qquad [\textsc{ScopD}]$$

The new rule [LinkD] creates, for two processes $\mathsf{accept}\ n(x).R$ and $\mathsf{request}\ n(y).R'$ ready to engage in a session, not one but two channel-ends c and d, one for each partner. The connection between channel-ends c and d is made explicit in the binding (νcd). Rule [ComD] performs value passing from the sending party $c![e].R$ to the receiving partner $d?(y).R'$, provided c and d are related by σ.

The types for the double binder runtime language are those of the base language. The type system now has judgments on configurations of the form $\Gamma \vdash \sigma \diamond R \triangleright \Delta$ with $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\sigma)$.

We add a single new rule w.r.t. the type system for top-level processes of Figure 3:

$$\frac{\Gamma \vdash \sigma \cdot (c,d) \diamond R \triangleright \Delta \cdot c\colon T \cdot d\colon \overline{T}}{\Gamma \vdash \sigma \diamond (\nu cd)R \triangleright \Delta} \qquad [\textsc{CResD}]$$

In rule [CResD], the double binding (νcd) gives raise to a (c,d) entry in σ, in line with the operational semantics. The rule types a channel double-binder, making sure that the two ends, c and d, of a channel have dual types.

Subject reduction and type-safety hold only for balanced environments [11].

Definition 3. *Let σ be an channel-end connection and Δ be a typing such that $\mathrm{fc}(\Delta) \subseteq \mathrm{dom}(\sigma)$. We say that Δ is* balanced *by σ whenever $c\,\sigma\,d$ and $\{c,d\} \subseteq \mathrm{dom}(\Delta)$ implies that or (i) $\Delta(c) = \overline{\Delta(d)}$ or (ii) $\Delta(c) \otimes \Delta(d) \downarrow$.*

$$
\begin{array}{rcl}
[\![\sigma \diamond \mathsf{accept}\ u(x).R]\!]_\Sigma & = & \mathsf{accept}\ [\![u]\!]_\Sigma(x).[\![\sigma \diamond R]\!]_\Sigma \\
[\![\sigma \diamond \mathsf{request}\ u(x).R]\!]_\Sigma & = & \mathsf{request}\ [\![u]\!]_\Sigma(x).[\![\sigma \diamond R]\!]_\Sigma \\
[\![\sigma \diamond u?(x).R]\!]_\Sigma & = & [\![u]\!]_\Sigma?(x).[\![\sigma \diamond R]\!]_\Sigma \\
[\![\sigma \diamond u![e].R]\!]_\Sigma & = & [\![u]\!]_\Sigma![[\![e]\!]_\Sigma].[\![\sigma \diamond R]\!]_\Sigma \\
[\![\sigma \diamond (\nu a)R]]\!]_\Sigma & = & (\nu a)[\![\sigma \diamond R]\!]_\Sigma \\
[\![\sigma \diamond (\nu cd)R]]\!]_\Sigma & = & (\nu k)[\![\sigma \cdot (c,d) \diamond R]\!]_{\Sigma\cdot(c\to k, d\to k)} \\
[\![\sigma \diamond u \lhd l_j.R]\!]_\Sigma & = & [\![u]\!]_\Sigma \lhd l_j.[\![\sigma \diamond R]\!]_\Sigma \\
[\![\sigma \diamond u \rhd \{l_i\colon R_i\}_{i\in I}]\!]_\Sigma & = & [\![u]\!]_\Sigma \rhd \{l_i\colon [\![\sigma \diamond R_i]\!]_\Sigma\}_{i\in I} \\
[\![\sigma \diamond \mathsf{if}\ e\ \mathsf{then}\ R\ \mathsf{else}\ R']\!]_\Sigma & = & \mathsf{if}\ [\![e]\!]_\Sigma\ \mathsf{then}\ [\![R]\!]_\Sigma\ \mathsf{else}\ [\![R]\!]'_\Sigma \\
[\![\sigma \diamond R \mid R']\!]_\Sigma & = & [\![\sigma \diamond R]\!]_\Sigma \mid [\![\sigma \diamond R']\!]_\Sigma \\
[\![\sigma \diamond X[\tilde{e}]]\!]_\Sigma & = & X[[\![\tilde{e}]\!]_\Sigma] \\
[\![\sigma \diamond \mathsf{def}\ \{X_i(\tilde{x}_i) = R_i\}_{i\in I}\ \mathsf{in}\ R]\!]_\Sigma & = & \mathsf{def}\ \{X_i(\tilde{x}_i) = [\![\sigma \diamond R_i]\!]_\Sigma\}_{i\in I}\ \mathsf{in}\ [\![\sigma \diamond R]\!]_\Sigma
\end{array}
$$

Figure 4: The encoding of double binder configurations into runtime processes.

The proof of the subject reduction is involved, because of session passing and of the rule for typing composition that merges possibly overlapping encodings.

Theorem 4 (Subject Reduction for the Double Binder Language)**.** *Let* $\Gamma \vdash \sigma \diamond R \triangleright \Delta$ *with* Δ *balanced by* σ*. If* $\sigma \diamond R \to \sigma \diamond R'$*, then there is* Δ' *balanced by* σ *s.t.* $\Gamma \vdash \sigma \diamond R' \triangleright \Delta'$*.*

Theorem 5 (Type Safety for the Double Binder Language)**.** *Let* $\Gamma \vdash \sigma \diamond R \triangleright \Delta$ *with* Δ *balanced by* σ*. If* $\sigma \diamond R \to^* \sigma \diamond R'$*, then* $\sigma \diamond R'$ *is not an error.*

Proof. Standard, by using Subject Reduction (Theorem 4). □

From the Double Binder Runtime to the Base Runtime Language. In order to state the correspondence between runtime processes, we define a mapping from the double binder language into the base language. A *forget-distinction function* over a end-channel configuration σ is an injective partial function from channel-ends c,d to runtime channels k such that $c\,\sigma\,d$ and $\Sigma(c) = k$ implies that $\Sigma(d) = k$.

The mapping $[\![\cdot]\!]_\Sigma$ from double binder configurations into runtime processes is defined in Figure 4. We let $[\![e]\!]_\Sigma = \Sigma(e)$ whenever e is a channel, and e otherwise. The compilation of a double binder configuration $\sigma \diamond (\nu cd)R$ with parameter Σ in a runtime process Q involves the generation of a new channel k bound in the encoding of $\sigma \diamond R$ with parameter Σ', where Σ' is obtained by extending Σ with the entries $\Sigma'(c) = k$ and $\Sigma'(d) = k$, so that the free occurrences of c,d in the continuation R will be translated to the channel k.

First, we need a lemma to export values outside the encoding.

Lemma 6. $[\![\sigma \diamond R[v/x]]]\!]_\Sigma = [\![\sigma \diamond R]\!]_\Sigma[[\![v]\!]_\Sigma/x]$.

Proof. By induction on the structure of $[\![\sigma \diamond R]\!]_\Sigma$. □

The core of the proof of operational correspondence is summarized the following lemma.

Lemma 7. *If* $[\![\sigma \diamond R]\!]_\Sigma \to Q$*, then there is a double binder process* R' *such that* $\sigma \diamond R \to \sigma \diamond R'$ *with* $[\![\sigma \diamond R']\!]_\Sigma \equiv Q$*.*

Proof. By induction on the length of the inference $[\![\sigma \diamond R]\!]_\Sigma \to Q$. The link and communication cases build on lemma 6. □

The next theorem says that structural congruence is preserved by $[\![\cdot]\!]$.

Theorem 8 (Congruence Correspondence)**.** *If $R' \equiv_{\mathsf{db}} R$ then $[\![\sigma \diamond R]\!]_\Sigma \equiv [\![\sigma \diamond R']\!]_\Sigma$.*

Proof. By case analysis on the rules for $\equiv_{\mathsf{db}}$. □

The next theorem establishes the soundness of the encoding.

Theorem 9 (Operational Correspondence)**.** *Let P be a top-level process of Figure 1. If there is a runtime process Q such that $P \to^n Q$, for $n \geq 0$, then there is a double binder process R such that $\emptyset \diamond P \to^n \emptyset \diamond R$ with $[\![\emptyset \diamond R]\!]_\emptyset \equiv Q$.*

Proof. By induction on the length n of reduction. When $n = 0$, we have that $Q = P$ and we are done since $[\![\emptyset \diamond P]\!]_\emptyset = P$. The induction step is proved by using Theorems 7 and 8. □

The following lemma says that the mapping $[\![\cdot]\!]_\Sigma$ preserve prefixes.

Lemma 10. *Let $Q = [\![\sigma \diamond R]\!]_\Sigma$. We have that Q is a k-processes if and only if R is a c-process and $\Sigma(c) = k$.*

The close the proof of our main result, we need to show that $[\![\cdot]\!]_\Sigma$ maps correct processes in correct processes.

Theorem 11 (Error Correspondence)**.** *Let $\sigma \diamond R$ be a configuration and assume Σ is a forget-distinction function over σ. If $\sigma \diamond R$ is not an error, then $[\![\sigma \diamond R]\!]_\Sigma$ is not an error.*

Proof. Without loss of generality, let

$$\begin{array}{rcl} R & \equiv & (\nu \tilde{a})(\nu \widetilde{cd})\mathsf{def}\ \{X_i(\tilde{x}_i) = R_i\}_{i\in I}\ \mathsf{in}\ R_1 \mid \cdots \mid R_n \\ [\![\sigma \diamond R]\!]_\Sigma & \equiv & (\nu \tilde{a})(\nu \widetilde{k})\mathsf{def}\ \{X_i(\tilde{x}_i) = Q_i\}_{i\in I}\ \mathsf{in}\ Q_1 \mid \cdots \mid Q_n \end{array}$$

where $\sigma' = \sigma \cdot (\tilde{c}, \tilde{d})$, $\Sigma' = \Sigma \cdot (\widetilde{cd} \to \tilde{k})$ and $Q_i = [\![\sigma' \diamond R_i]\!]_{\Sigma'}$, for $j \in 1, \ldots, n$.

We proceed by induction on n and show that if there are distinct $i, j \in 1, \ldots, n$ such that Q_i, Q_j are k-processes then both (a) and (b) hold:

(a) $Q_i \mid Q_j$ is a k-redex

(b) if there exists $r \in 1, \ldots, n$, $r \neq i, j$, such that Q_r is a k'-process, then $k' \neq k$.

From this and congruence correspondence (Theorem 8) we easily obtain that $[\![\sigma \diamond R]\!]_\Sigma$ is not an error. □

Discussion. In [7] we have introduced an extension of the first session typing system [10] that allows higher-order session communication. The calculus and typing system we present here extends these works by admitting session passing or delegation by using the standard communication rule of pi calculus and by providing for the comparison of channels. Besides the pragmatical interest of comparing values, we believe our result makes the theory of session types more general by relaxing the rule for sending channels, and allowing type more processes. Caires and Vieira [3] present a more flexible merge operation on types; we intend to pursue further investigation along these lines.

Starting from [6], many works on session types use polarized channels in order to achieve subject reduction in the presence of session passing or to avoid the runtime checking of free names (e.g., [1, 2, 8, 11]). In [6] the authors considered a typed pi calculus with branching and selection where channel-ends are decorated with polarities; communication and branching/selection occurs on channels with opposite polarities. While the type theory is polarized, polarities of processes can be inferred by a typechecking algorithm, provided the type environments in the rule for parallel composition are disjoint. In contrast, we avoid to use polarities in the type theory and admit overlapping environments in typing compositions; the connection between channel ends are relegated to the proof of type safety and are completely hidden to the programmer. This seems to us more suitable for synchronous implementations; in contrast asynchronous systems should use polarized channels for abstracting low level buffers [5]. We conjecture that our proof should be valid for most session-based calculi with accept and request primitives.

Differently from our framework, many systems for session [6, 7, 11] and linear [9] types require the inert process to be typed under an environment containing depleted resources (of type end). We think that the very technical reason is to let the structural rule $P \mid \mathbf{0} \equiv P$ preserve typing: adding non-empty capabilities to the type environment of P can break typability. For instance for $P = k?()$ in [7] we can have $\emptyset \vdash \mathbf{0} \triangleright k :![T]$ and $\emptyset \vdash P \triangleright k :?[T]$ and in turn $\emptyset \vdash P \mid \mathbf{0} \triangleright k : \bot$, while $\emptyset \not\vdash P \triangleright k : \bot$. By contrast, in our system non-empty capabilities can be added in compositions only to empty capabilities, because of the merge operation; the counter-example above is not typable since P is using the input capability. We believe that our general rule for the inert process should be sound for most systems typing a parallel composition with similar constraints on capabilities, for instance systems where the composed environments are disjoint.

Acknowledgments. This work has been partially sponsored by the project SENSORIA, IST–2005–016004.

References

[1] Roberto Bruni and Leonardo Gaetano Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines. In *AMAST*, pages 100–115, 2008.

[2] Roberto Bruni, Rocco De Nicola, Michele Loreti, and Leonardo Gaetano Mezzina. Provably correct implementations of services. In *Trustworthy Global Computing*, 2008.

[3] L. Caires and H. T. Vieira. Conversation types. In *ESOP'09*, LNCS. Springer-Verlag, 2009.

[4] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *ECOOP*, LNCS, pages 328–352. Springer-Verlag, 2006.

[5] Simon Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. Submitted, 2008.

[6] Simon J. Gay and Malcolm J. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2–3):191–225, 2005.

[7] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.

[8] Marija Kolundzija. Security types for sessions and pipelines. In *Web-Services and Formal Methods*, 2008.

[9] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[10] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.

[11] Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In *SecReT*, volume 171(4) of *ENTCS*, pages 73–93, 2007.

Session-Based Programming for Parallel Algorithms

Expressiveness and Performance

Andi Bejleri
Imperial College London
ab406@doc.ic.ac.uk

Raymond Hu
Imperial College London
rhu@doc.ic.ac.uk

Nobuko Yoshida
Imperial College London
yoshida@doc.ic.ac.uk

1 Introduction

At PLACES'08, we discussed the need to investigate benchmark examples of session types [10, 6] to compare productivity, safety and performance with other communications programming languages. As a starting point into the investigation of these issues, we examine SJ [3], the first full object-oriented language to incorporate session types for type-safe concurrent and distributed programming. The SJ language extends Java with syntax for declaring session types (protocols), and a set of core operations (session initiation, send/receive) and high-level constructs (branching, iteration, recursion) for implementing the interactions that comprise the sessions. The SJ compiler statically verifies session implementations against their declared types. Together with runtime compatibility validation between peers at session initiation, SJ guarantees communication safety in terms of message types and the structure of interaction. SJ has been shown to perform competitively with widely-used communication APIs such as network sockets, in certain cases out-performing RMI [8].

This abstract reports our on-going work on implementing parallel algorithms in SJ, with focus on the aforementioned aspects: ***productivity*** (including code readability and writability), ***safety*** (freedom from type and communication errors [10, 6]), and ***performance*** (optimisations enabled by SJ, and comparison against other communication systems). Parallel algorithms is a prominent topic in algorithmic research due to the increase of hardware resources such as multicore machines and clusters. The session-based programming methodology and expressiveness of SJ are demonstrated through implementations of: (1) a Monte Carlo approximation of π, (2) the Jacobi solution of the Discrete Poisson Equation, and (3) a simulation of the *n*-Body problem. These algorithms were selected to evaluate the SJ representation of, amongst other features, typical *task and data decomposition* patterns [9] (as featured in 1 and 2), a technique for exchanging *ghost points* [5] (in 2), and an intricate communication pattern over a circular pipeline structure (3). SJ is an evolving framework, and recent extensions to the SJ language (e.g. multicast output operations and advanced iteration structures) and the SJ Runtime (e.g. improved extensibility through the Abstract Transport) play an important part in the implementation of these algorithms.

Using these programs, which feature complex and representative interaction structures, we contribute new benchmark results for analysis to supplement the existing benchmarks for SJ. In particular, benchmark comparisons between SJ and MPJ Express [1], a reference Java messaging system based on the MPI [4] standard, for (1) and (2) yield further promising peformance results for SJ. We also show how SJ *noalias* types can greatly optimise performance, such as for the shared memory communication of the ghost points in (2).

We then compare the SJ implementations of the above algorithms with their MPI counterparts from programming perspectives. Despite rich libraries and functionality, MPI remains a low-level API, and can suffer from such commonly perceived disadvantages of explicit message passing as unexpected message structures and deadlocks due to incorrect protocol implementations. From our experiences implementing the above algorithms, we found high-level session programming to be easier than the basic MPI functions, which often require manipulating numerical process identifiers and array indexes (e.g. for message lengths in (3)) in tricky ways. SJ is able to exploit session types to compensate for, or eliminate, many of the MPI problems: session types themselves are inherently deadlock free, for example.

In conclusion, we observe that high-level session abstraction has significant impact on program structure, improving readability and reliability, and session type-safety can greatly facilitate the task of communications programming whilst retaining competitive performance. We also argue that extending SJ with full multiparty session types would allow richer topologies such as the ring and 2D-mesh to be expressed more naturally, and enable performance improvements through massive parallelism.

2 Monte Carlo π Approximation

A simple Monte Carlo simulation for approximating the value of π is amenable to parallelisation. We use this example to (1) introduce the basic and some new SJ constructs; (2) show their use in the description of a simple task decomposition pattern [9]; and (3) demonstrate the effect of parallelisation for performance gain in SJ (§ 5).

A unit square inscribes a circle of area $\pi/4$; hence, $\pi = 4t$, where t is the ratio of the circle area to the square. t can be determined by selecting a random set of points within the square ((x,y) where $x,y \in [-1,1]$), and checking how many fall inside the inscribed circle ($x^2+y^2 <= 1$). A Master process (or thread) can instruct Workers to

independently generate and check multiple sets of points in parallel, calculating the final value by combining the results from each Worker. The simple session type, from the Worker side, for the communications involved is:

```
protocol workerToMaster { sbegin.?(int).!<int> }
```

Each Worker service (`sbegin`) is told how many points to test by the Master (`?(int)`) and sends back the number that fall inside the circle (`!<int>`). The code for a basic SJ implementation looks like

```
// Workers run the simulation.
int trials = s_wm.receive(); // ?(int)
for (int i = 0; i < trials; i++)
    if (hit()) hits++;
s_wm.send(hits); // !<int>
```

```
// Master controls the Workers.
<s_mw1, s_mw2, ...>.send(trials); // Multicast.
int totalHits = // Collect the results.
    s_mw1.receive()
      + s_mw2.receive() + ...;
```

where `s_mw1` is the Master's session socket to Worker1, etc.; `s_wm` a Worker's session with the Master; and `hit` returns the boolean from testing a generated point. The Master can then calculate t by `totalHits / (trials * n)`, where `n` is the number of Workers. The SJ compiler statically verifies correctness by checking each session implementation against its declared type (e.g. `s_wm` against `workerToMaster`).

3 Jacobi Solution of the Discrete Poisson Equation

The implementation of this algorithm demonstrates (1) expressiveness of SJ due to multicast session-iteration operation; (2) guaranteed type and communication safety in SJ; (3) a type-directed optimisation (for exchanging ghost points) through the new SJ *noalias* type; and (4) the *transport-independence* of SJ programs, due to the design of the SJ language-runtime Framework. Poisson's Equation is a partial differential equation with applications in, for example, heat flow, electrostatics, gravity and climate computations. The discrete two-dimensional Poisson equation $(\nabla^2 u)_{ij}$ for a $m \times n$ grid can be written as in (a)

$$\text{(a)}\ \ u_{ij} = \tfrac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - dx^2 g_{i,j}) \qquad \text{(b)}\ \ u_{ij}^{k+1} = \tfrac{1}{4}(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k})$$

where $2 \le i \le m-1$, $2 \le j \le n-1$, and $dx = 1/(n+1)$. Jacobi's Method converges on a solution by repeatedly replacing each element of the matrix u by an average of its four neighbouring values and $dx^2 g_{i,j}$; for this example, we set g to 0. Then from the k-th approximation of u, the next iteration performs the calculation in (b) above. Termination may be on reaching a target convergence threshold or completing a certain number of iterations. Parallelization exploits the fact that each element can be updated independently (within one step): the grid can be divided up and the algorithm performed on each subgrid in separate processes or threads. The key is that neighbouring processes must exchange their subgrid boundary values as they are updated.

We illustrate a one-dimensional decomposition of a square grid into three non-overlapping subgrids for three separate processes. Two Workers are allocated the end subgrids; the Master has the central subgrid, and controls the termination condition for all three processes. In addition to their allocated subgrid, each process maintains a copy of the boundary values (*ghost points*) of its neighbours; the new values are communicated after each iteration. This scheme allows the original grid to be divided in subgrids of any size. The session type between the Master and two Workers from the side of the former is:

```
protocol masterToWorker {
  cbegin.                     // Request the Worker service.
  !<int>.                     // Send the size of the matrix.
  ![                          // Enter scope of main algorithm iteration (check term. cond.).
    !<double[]>.?(double[]).  // Send our boundary values; get Worker's updated ghost points.
    ?(double).?(double)       // Receive the convergence data for Worker's subgrid.
  ]*.                         // After the last iteration...
  ?(double[][])               // ...get the final results.
}
```

To control all the Workers simultaneously, the implementation of Master uses the SJ session constructs for multicasting output operations such as message-send and session-iteration. For example,

```
// Master controls iteration condition.
<mw1, mw2>.outwhile( // ![...
    !accurateEnough(...) && iters < MAX_ITERS) {
  ... // Main body of the algorithm.
}                    // ...]*
```

```
// Workers obey the Master.
<wm>.inwhile() { // ?[...
  ... /* Main body
         of the algorithm. */
}                // ...]*
```

Inter-thread communication of large messages, such as arrays, can be optimised using SJ `noalias` types. A `noalias` variable on the RHS of an assignment or as a method argument — such as to the `send` operation — becomes `null`. Combined with static type checking that precludes any potential assignment of aliased values to `noalias` targets, a `noalias` variable is guaranteed the sole reference to the pointed object at all times, permitting zero-copy message passing of `noalias` messages over compatible shared memory transports. In the present example, the `noalias` optimisation can be used to communicate the ghost point data; for example, the Worker implementations contain the following code extract.

```
// Array containing our boundary values (ghost points for the Master) declared as noalias.
noalias double[] ghostPoints = ...; // Update and prepare our boundary values for sending.
s_wm.send(ghostPoints); // Zero-copy send, as directed by types: !<noalias double[]>.
... // ghostPoints variable becomes null.
```

Transports that do not support this feature (e.g. TCP) can fall back to copy-on-send; the overall semantics of the program remains unchanged. This illustrates the *transport-independent* nature of SJ programs: the virtualisation of communication due to the SJ Runtime allows programs to make the best use of the whichever transports are available, *without* requiring any modification to the programs themselves. If the Master and Worker processes are run on separate machines, then the SJ Runtime can arrange, e.g. a TCP-based session; for the same programs, run as co-located threads, shared memory will be used. This SJ feature is further demonstrated for the next algorithm.

4 The *n*-Body Problem

The *n*-Body Problem involves finding the motion, according to classical mechanics, of a system of bodies given their masses and initial position and velocities. This advanced example demonstrates (1) the expressiveness of SJ and the extensions for complex iteration structures, by implementing an intricate circular communication pipeline; (2) SJ transport-independence (see § 5); and (3) the benefits of high-level message types (see § 6). Parallelism is achieved by dividing the particle set, and hence the calculations to determine the resultant force exerted on each body, amongst a collection of parallel processes. We use the approach where the processes, maintaining only the current state of their individual particle sets, are deployed to form a circular pipeline (ring topology). Firstly, the number of processes in the pipeline, p, is dynamically determined by sending a token around the ring. Then each step of the simulation involves $p-1$ iterations. In the first iteration, each process sends their particle data to their neighbour on the right and calculates the partial resultant forces exerted within their own particle set. In the n-th iteration, each process forwards on the particle data received in the previous iteration (line (*i*) in the listing on the next page), adds this data to the running force calculation (*ii*), and receives the next data set (*iii*). The particle data from the right neighbour is received by the end of the final iteration: each data set has now been seen by all processors in the pipeline, allowing the final results for the current simulation step to be calculated.

The SJ implementation of the above algorithm has each process, i.e. each Worker unit in the pipeline, open a session server socket to accept a connection from its left neighbour, and create the connection to its right neighbour using a session client socket. The session type for the interaction in this algorithm, from the server side of each unit, is:

```
protocol serverSide {   // Interaction with the left neighbour.
  sbegin.               // Accept connection from left neighbour.
  !<int>.               // Forward on the ring initialisation token.
  ?[                    // Main simulation loop (iteration flag received from the left).
    ?[                  // Inner iterations within each simulation step.
      ?(Particle[])     // Particle data forwarded through pipeline.
    ]*
  ]*
}
```

Configuration	SJ (ms)	MPJ (ms)
Sequential (1 Worker)	16362	
1 Master & 1 Worker	8698	8314
1 Master & 4 Workers	3604	3377

(a)

Matrix Size	"Ordinary" (ms)	`noalias` (ms)
100	1270	992
300	24436	19448

(b)

Figure 1: (a) Monte Carlo π for a varying number of Workers; (b) Jacobi: "ordinary" vs. `noalias` versions.

The session type for the corresponding client side of each unit is simply the direct dual of `serverSide`: `protocol clientSide { cbegin.?(int).![![!<Particle[]>]*]* }`, given by inverting the input (`?`) and output (`!`) symbols. For this client-server architecture, the ring topology is bootstrapped by designating two neighbouring processes to be the "first" and "last" pipeline units.

As for the Jacobi example, SJ `noalias` can be used to optimise the communication of particle data as zero-copy shared memory transfers in multi-threaded implementations. Following is the SJ code for the main simulation loop.

```
s_r.outwhile(s_l.inwhile()) {    // Synch. with our two n'bours for each sim. step. s_l: ?[..
  noalias Particle[] current = ...;  // Prepare our own particle data for sending.
  s_r.outwhile(s_l.inwhile()) {      // Inner iters. within each simulation step. s_l: ?[..
    s_r.send(current);               // (i) Forward the current data set. s_r: !<Particle[]>.
    ...                              // (ii) Add the current data to the running calculation.
    current = (Particle[]) s_l.receive(); // (iii) Receive next data set. s_l: ?(Particle[]).
  }                                  // s_l: ..]*
  ...  // Calculate the final results for this sim. step and update our own particle data.
}                                    // s_l: ..]*
```

Note the assignment in (*iii*) is permitted because the received message is implicitly `noalias`. This algorithm works independently of the pipeline length ($p \geq 2$). Moreover, this SJ implementation is transport-independent: the p pipeline units can be distributed or co-machine processes or co-process threads in any combination.

5 Performance Benchmarks

This section presents performance measurements for the three parallel algorithms described above. The first two benchmarks show that the SJ Runtime, although still at an early implementation version with much scope for further optimisation, can perform competitively with MPJ Express [1]. Unlike Java MPI implementations built around JNI wrappers to C functions, MPJ Express adopts a pure Java approach which makes for an interesting comparison with SJ.

The same machines in the same network environment were used for all the following benchmark experiments. Each machine is a dual-core Intel Core 2 Duo (Conroe B2) at 2.13GHz with 2MB cache, 2GB main memory, running Ubuntu Linux 4.2.3 (kernel 2.6.24); the machines were connected via gigabit Ethernet, and the latency between two machines was measured using ping (64 Bytes) to be on average 0.10ms. The benchmark applications were compiled and executed using the standard Sun Java SE compiler and runtime versions 1.6.0. For each experiment, the results from 100 executions for each parameter configuration were recorded; here, we give the mean values. The full source code for the benchmark applications and the complete results can be found at [2].

Monte Carlo π approximation. The first benchmark uses the SJ implementation of this algorithm to (1) verify the performance gain from increased parallelism, and (2) to compare the performance of the SJ Runtime against MPJ Express. Each process (Master, Workers and Client) was run on a separate machine, communicating via TCP. The results (Figure 1a), comparing both sequential and parallel versions of the algorithm, show that for a constant sample size (total number of test points), increasing the number of Workers indeed reduces the time to complete the algorithm proportionally. The results for the MPJ Express implementation are around 5–6% faster than the SJ implementation.

Jacobi Poisson solution. The second benchmark, through the SJ implementation of the Jacobi iteration algorithm, demonstrates (1) the effectiveness of `noalias` types for zero-copy message transfer in a shared memory environment, and (2) again compares SJ performance to MPJ Express. Firstly, "Ordinary" (i.e. without `noalias`)

Matrix size	SJ (ms)	MPJ (ms)
100	3713	4460
300	19501	19834

(a)

Particles	Distributed (ms)	Localhost (ms)	Threads (ms)
100	496	452	326
300	1194	1144	865
1000	7702	7497	6785

(b)

Figure 2: (a) Jacobi: SJ vs. MPJ Express; (b) *n*-Body simulation: Distributed vs. Localhost vs. Threads versions.

and `noalias` versions of the Master and two Workers were run as co-VM threads on a single machine; the Client is connected to the Master from a separate machine via a TCP-session. We measured the time to complete the algorithm for square matrices of size (i.e. the length of one side of the matrix) 100 and 300. In both cases, the `noalias` version is approximately 20% faster than the ordinary one (Figure 1b). Secondly, the distributed SJ implementation of Jacobi (the Client, Master and Workers run on separate machines connected via TCP) performs better than the MPJ Express implementation by 6% on average (Figure 2a).

***n*-Body simulation.** The third benchmark uses the *n*-Body simulation to demonstrate the important improvement in productivity enabled by SJ transport-independence: this single SJ implementation was run in the different communication environments (locally concurrent, distributed), making the best use of the available transports (TCP, shared memory, etc.), without *any* changes to the source code for the Workers (although the shared memory version required a few lines of external code to bootstrap the Workers as Java threads). The benchmark was executed using two pipeline Worker units (not using `noalias`) in three different configurations: the two Workers on separate machines using TCP (Distributed), as separate processes on the same machine using TCP (Localhost), and as co-VM threads using shared memory (Threads). We recorded the results for simulations involving 100, 300 and 1000 particles, distributed equally between the Workers.

As expected, the results (Figure 2b) show the Threads version is faster than Localhost: around 27% for 100 particles, 24% for 300, and 10% for 1000. The Distributed version is in turn slightly slower (latency is very low) than Localhost: 10% for 100 particles, 4% for 300, and 3% for 1000. The relative performance gain between each version decreases for larger particle sets because the local computation costs begin to dominate the communication costs for this fixed number of Workers. Naturally, performance can be improved for simulations involving many particles by increasing the degree of parallelism, i.e. using more Workers.

6 SJ and MPI Comparison and Future Work

This section compares SJ against MPI in terms of language support for communications programming, with reference to MPI implementations of the above algorithms [5]. Since MPI has an extensive library of functions developed over 15 years, many of these are not yet directly supported in SJ, e.g. MPI Jacobi makes use of a virtual topology (`MPI_Cart_Create`) and collective data movement operations (`MPI_Bcast` and `MPI_Allreduce`, for broadcasting the matrix size and distributing the termination condition in (2)). However, many of these features can be encoded into a session type, as shown above. Furthermore, we observed the following benefits of SJ against MPI.

Type and communication safety from session types. MPI is designed as a portable API specification to be implemented for varying host languages. Coupled to the low-level nature of many MPI functions, the design of accompanying MPI program verification techniques for a host language can be difficult. Common MPI errors recognized by the community include:

- **Doing things before** `MPI_Init` **and after** `MPI_Finalize`. The execution of such MPI operations can lead to runtime errors such as broken invariants, messages not broadcasted, and incorrect collective operations.
- **Unmatched** `MPI_Send` **and** `MPI_Recv`. Such errors can lead to a mismatch between the sent and expected message type/structure, or a variety of deadlock situations depending on the communication mode. For example, two processes deadlock if each is waiting for a message before sending the message expected by the other. In the standard (buffer-blocking) mode, the converse situation (both processes attempting to send before receiving) can also deadlock: if both message sizes are bigger than the available space in the medium and opposing receive buffers, then the processes cannot complete their write operations. A related problem is matching a `MPI_Bcast` output with `MPI_Recv`. Standard usage is to receive a broadcast message using

the complementary `MPI_Bcast` input. `MPI_Recv` consumes the message; hence, the receiver must be able to determine which processes have not yet seen the message and manually re-broadcast it.

- **Concurrency issues**. Incorrect access of a shared communicator by separate threads can violate the intended message causalities between the sender(s) and the receivers. In addition, race conditions can arise due to modifying, or even just by accessing, messages that are in transit.

As illustrated in the previous sections, ***SJ programs are guaranteed free from all of the above errors*** by the semantics of session communication and static session type checking. The first two points are directly prevented by the properties of session types. For the third point, the SJ compiler disallows sharing of session socket objects (implicitly `noalias`), and message copying/linear transfer can be safely and explicitly controlled via `noalias` types.

High-level message types. In many parallel algorithms, messages are mainly communicated via arrays. For MPI, effort is required to manually track and communicate array indices, e.g. for message length or the number of messages. In contrast, the high-level type-abstraction for messages allows SJ programmers to treat both object and primitive array messages as regular Java array objects. For instance, the MPI version of the *n*-Body simulation broadcasts the number of particles managed by each process so the amount of data to be read from each particle set message can be determined; in SJ, the particle data is simply received as discrete array messages, avoiding manual handling of message sizes.

Transparent zero-copy message passing. SJ provides direct language support for zero-copy transfer in shared memory contexts through `noalias` types. This feature can enable significant performance increases for multi-threaded programs (see § 5). Moreover, the communication of `noalias` types retains consistent semantics in all transport contexts (see *transport-independence* in § 3).

Future Work. We demonstrated expressiveness, productivity and performance benefits of session-based programming in SJ through the presented parallel algorithm implementations. Although we have seen that the above algorithms were readily implemented in the current SJ, immediate future work includes expanding the set of SJ operations and constructs, e.g. with session typed equivalents of MPI functions and features that are not yet directly supported. For example, whilst the MPI *standard* mode (send and receive block on their respective buffers) corresponds to the session communication semantics in SJ, MPI has several additional modes: *synchronous* (send and receive operations synchronise), *ready* (programmer notifies the system that a receive has been posted), and *buffered* (user manually handles send buffers). We also wish to compare SJ to PGAS languages such as X10 [11] using parallel algorithm implementation as a basis.

We believe that extending SJ with full multiparty session types [7] would allow richer topologies such as the ring and 2D-mesh to be expressed more naturally in a type-safe manner. For example, the SJ *n*-Body implementation currently requires creating one intermediary session (for the final pipeline link) in each simulation step; with multiparty sessions, we would only need to open a single session for the complete simulation. Our prediction is that multiparty sessions will offer better support for massive parallelism than the current client-server based session sockets. We plan to identify design issues and possible overheads for global type-checking through further implementation of parallel algorithms with complex communication patterns.

Conclusion. SJ programs are guaranteed free from type and communication errors, and perform competitively against other Java communication runtimes. In certain cases, SJ programs can out-perform their counterparts implemented in communication-safe systems such as RMI [8] and also lower-level, non communication-safe message passing systems such as MPJ Express (§ 5).

7 Acknowledgments

We thank Kohei Honda and Vijay Saraswat for their comments on a first draft of this abstract. The work is partially supported by EPSRC GR/T03208, GR/T03215 and IST2005-015905 MOBIUS.

References

[1] MPJ Express homepage. `http://mpj-express.org/`.

[2] Session-based programming for parallel algorithms. `http://www.doc.ic.ac.uk/~ab406/parallel_algorithms.html`.

[3] SJ homepage. `http://www.doc.ic.ac.uk/~rhu/sessionj.html`.

[4] Using MPI: Example Programs. http://www-unix.mcs.anl.gov/mpi/usingmpi/examples/intermediate/main.htm.

[5] William Gropp, Ewing Lusk, and Anthony Skjellkum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.

[6] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[7] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *POPL*, pages 273–284. ACM, 2008.

[8] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In Jan Vitek, editor, *ECOOP*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.

[9] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[10] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

[11] X10 homepage. `http://x10.sf.net`.

Towards a Unified Framework for Declarative Structured Communications

Hugo A. López
IT University of Copenhagen
hual@itu.dk

Carlos Olarte
INRIA and LIX, École Polytechnique
colarte@lix.polytechnique.fr

Jorge A. Pérez
Università di Bologna
perez@cs.unibo.it

Abstract

We describe ongoing work on the definition of a formal framework for the declarative analysis of structured communication. By relying on a (timed) concurrent constraint programming language, we show that in addition to the usual behavioral techniques from process calculi, session analysis can elegantly exploit logic based reasoning techniques, all in a single framework. As a preliminary result, we report a concurrent constraint interpretation of a language for structured communication proposed by Honda, Vasconcelos and Kubo. Distinguishing features of our approach are: the possibility of including partial information (constraints) in the session model; the use of explicit time for reasoning about session duration and expiration; a tight correspondence with logic, which formally relates session execution and linear-time temporal logic formulas.

1 Introduction

Motivation. From the viewpoint of *reasoning techniques*, two main trends in modeling Service Oriented Computing (SOC) can be singled out. On the one hand, a *behavioral approach* focuses on how process interactions can lead to correct configurations. Typical representatives of this approach (as, e.g., [8, 1, 7, 14]) are based on process calculi and Petri nets, and count with behavioral equivalences and type disciplines as main analytic tools. On the other hand, in a *declarative approach* the focus is on the set of conditions components should fulfill in order to be considered correct, rather than on the complete specification of the control flows within process activities (e.g. [14]). Even if these two trends address similar concerns, we find that they have evolved rather independently from each other.

The quest for a unified approach in which behavioral and declarative techniques can harmoniously converge is therefore a legitimate research direction. In this paper we shall argue that Concurrent Constraint Programming (CCP) [17] can provide solid foundations for such an approach. Indeed, we find that the unified framework for behavioral and logic techniques that CCP provides can be fruitfully exploited for analysis in SOC, and could complement well other approaches, such as those based on type systems. Below we briefly introduce the CCP model and then elaborate on how it can shed light on a particular issue: the analysis of *declarative* sessions.

CCP [17] is a well-established model for concurrency where processes interact with each other by *telling* and *asking* for pieces of information (*constraints*) in a shared medium, the *store*. While the former operation simply adds a given piece of partial information to the store (thus making it available for other processes), the latter allows for rich, parameterizable forms of process synchronization. Interaction in CCP is thus inherently *asynchronous*, and can be related to a broadcast-like communication discipline, as opposed to the point-to-point communication in process calculi such as the π-calculus [15]. In CCP, the store grows monotonically, i.e., constraints cannot be removed. This condition is relaxed in *timed* extensions of CCP (e.g., [16, 11]), where processes evolve along a series of *discrete time intervals*. Each interval contains its own store and information is not automatically transferred from one interval to another. In this paper we shall use a CCP process language that is timed in this sense.

In addition to the traditional operational view of process calculi, CCP enjoys a *declarative* view that distinguishes it from other models of concurrency: CCP programs can be seen, at the same time, as computing agents and as logic formulas [17, 11, 12], i.e., they can be read and understood as logical specifications. Hence, CCP-based languages are suitable for *both* the specification and verification of programs. In the CCP language used in this paper processes can be interpreted as linear temporal logic formulas; we shall exploit this correspondence to verify properties of structured communications.

This Work. We describe ongoing work on the definition of a formal framework for the declarative analysis of sessions. We shall exploit utcc, a timed CCP process calculus [13], to give an alternative interpretation to the language defined by Honda, Vasconcelos and Kubo in [6] (henceforth referred to as HVK). This way, structured communications can be studied in a declarative framework in which time is explicit. We begin by proposing an encoding of the HVK language into utcc; such an encoding defines asynchronous session establishment and satisfies a rather standard operational correspondence property. We then move to the timed setting, and propose $\mathsf{HVK}^{\mathsf{T}}$, a timed extension of the HVK language. The extended language explicitly includes information on session duration, allows for declarative preconditions within session establishment constructs, and features a construct for session abortion. We then show that the encoding of HVK into utcc straightforwardly extends to $\mathsf{HVK}^{\mathsf{T}}$. Finally, we show the applicability of a declarative characterization of structured communications by relating processes specifications with linear temporal logic (FLTL).

An extended version of this work, including the appendices herewith referred, is available at [9].

A Compelling Example. We now give intuitions on how a declarative approach could be useful in the analysis of sessions. Consider the ATM example from [6, Sect. 4.1]. There, an ATM has established sessions with a user and his/her bank; it allows for `deposit`, `balance`, and `withdraw` operations. In the latter case, if there is no enough money to withdraw, then an *overdraft* message appears to the user. It would be interesting to verify what occurs when the example is extended with a malicious card reader that keeps the user's sensible information and uses it to continue withdrawing money without his/her authorization. A greedy card reader could even repeatedly withdraw until causing an overdraft.

In this simple scenario, the correspondence between utcc and linear temporal logic may come in handy to reason about the possible states for this specification. These can be used not only to describe the operational behavior of the compromised ATM above, but also to provide declarative arguments regarding its the evolution. For instance, assuming a global channel `out` and an overdraft message M_O, one could show that a utcc specification of the ATM example satisfies the FLTL formula $\Diamond\,\mathtt{out}(M_O)$, which intuitively means that in the presence of the malicious card reader the user's bank account will eventually go to overdraft.

Related Work. One approach to combine the declarativeness of constraints and process calculi techniques is represented by a number of works that have extended name-passing calculi with some form of partial information (see, e.g., [18, 4]). The crucial difference between such a strand of work and CCP-based calculi is that the latter offer a tight correspondence with logic, which greatly broadens the spectrum of reasoning techniques at one's disposal. Recent works similar to ours include CC-Pi [2] and the calculus for structured communication in [3]. Such languages feature elements that resemble much ideas underlying CCP (especially [2]). The main difference between such works and our approach is that the reasoning techniques they feature are different from logic-based ones. In [2], a language for Service-Level Agreement (SLA) is proposed, featuring constructs for name-passing, constraint retraction and soft constraints. There, the reasoning techniques are essentially operational. In [3] a language for sessions featuring constraints is proposed. There, the key for analysis is represented by a type system which provides consistency for session execution, much as in the original approach in [6].

2 Preliminaries

We begin by introducing HVK, the language for structured communication proposed in [6]. We assume the following notational conventions: *names* are ranged over by $a, b, \ldots$; *channels* are ranged over by k, k'; *variables* are ranged over by $x, y, \ldots$; *constants* (names, integers, booleans) are ranged over by

$c, c', \ldots$; *expressions* (including constants) are ranged over by $e, e', \ldots$; *labels* are ranged over by $l, l', \ldots$; *process variables* are ranged over by $X, Y, \ldots$. Finally, $u, u', \ldots$ denote names and channels.

Definition 1 (The HVK language [6])**.** *Processes in* HVK *are built from:*

P,Q	$::=$	**request** $a(k)$ **in** P	*Session Request*	$\mid$	**accept** $a(x)$ **in** P	*Session Acceptance*
	$\mid$	$k![\vec{e}]; P$	*Data Sending*	$\mid$	$k?(x)$ **in** P	*Data Reception*
	$\mid$	$k \lhd l; P$	*Label Selection*	$\mid$	$k \rhd \{l_1 : P_1 \parallel \cdots \parallel l_n : P_n\}$	*Label Branching*
	$\mid$	**throw** $k[k']; P$	*Channel Sending*	$\mid$	**catch** $k(k')$ **in** P	*Channel Reception*
	$\mid$	**if** e **then** P **else** Q	*Conditional Statement*	$\mid$	$P \mid Q$	*Parallel Composition*
	$\mid$	**inact**	*Inaction*	$\mid$	$(\nu u)P$	*Hiding*
	$\mid$	**def** D **in** P	*Recursion*	$\mid$	$X[\vec{e}\vec{k}]$	*Process Variables*
D	$::=$	$X_1(x_1k_1) = P_1$ **and** $\cdots$ **and** $X_n(x_nk_n) = P_n$				

Reduction for HVK processes, denoted by $\rightarrow$, is defined as the smallest relation generated by a set of rules which we omit for space reasons (see [6]). As usual, $\rightarrow^*$ is the reflexive, transitive closure of $\rightarrow$.

2.1 Timed Concurrent Constraint Programming

Timed concurrent constraint programming (`tcc`) [16] extends CCP for modeling reactive systems. In `tcc`, time is conceptually divided into *time units* or *time intervals*. In a particular time interval, a `tcc` process P gets an input c from the environment, it executes with this input as the initial *store*, and when it reaches its resting point, it *outputs* the resulting store d to the environment. The resting point determines also a residual process Q which is then executed in the next time interval. It is worth noticing that the final store is not automatically transferred to the next time unit.

The `utcc` calculus [13] extends `tcc` for mobile reactive systems. Here *mobility* is understood as the dynamic reconfiguration of system linkage through communication, much like in the π-calculus [15]. `utcc` generalizes `tcc` by considering a *parametric* ask operator of the form $(\textbf{abs}\ \vec{x}; c)\,P$, with the following intuitive meaning: process $P[\vec{t}/\vec{x}]$ is executed for every term $\vec{t}$ such that the current store entails $c[\vec{t}/\vec{x}]$. This process can be viewed as an *abstraction* of the process P on the variables $\vec{x}$ under the constraint (or with the *guard*) c.

`utcc` provides interesting reasoning techniques: First, `utcc` processes can be represented as partial closure operators (idempotent and extensive functions). Moreover, for a significant fragment of the calculus, the input-output behavior of a process P can be retrieved from the set of fixed points of its associated closure operator [12]. Second, `utcc` processes can be characterized as FLTL formulas [10]. This declarative view of the processes allows for the use of the well-established verification techniques from FLTL to reason about `utcc` processes.

Syntax. Processes in `utcc` are parametric in a *constraint system* [17] which specifies the basic constraints that agents can tell or ask during execution. It also defines an *entailment* relation "$\vdash$" specifying interdependencies among constraints. Intuitively, $c \vdash d$ means that the information in d can be deduced from that in c (as in, e.g., $x > 42 \vdash x > 0$). The syntax of the language is as follows:

$$P, Q := \quad \textbf{skip} \;\big|\; \textbf{tell}(c) \;\big|\; (\textbf{abs}\ \vec{x}; c)\,P \;\big|\; P \parallel Q \;\big|\; (\textbf{local}\,\vec{x}; c)\,P \;\big|\; \textbf{next}\,P \;\big|\; \textbf{unless}\ c\ \textbf{next}\,P \;\big|\; !P$$

with the variables in $\vec{x}$ being pairwise distinct.

A process **skip** does nothing; process $\textbf{tell}(c)$ adds c to the store in the current time interval. A process $Q = (\textbf{abs}\ \vec{x}; c)\,P$ binds the variables $\vec{x}$ in P and c. It executes $P[\vec{t}/\vec{x}]$ for every term $\vec{t}$ such that the current store entails $c[\vec{t}/\vec{x}]$. When the vector of variables $\vec{x}$ is empty, we retrieve the standard `tcc` ask operator **when** c **do** P. $P \parallel Q$ denotes P and Q running in parallel during the current time interval. A process $(\textbf{local}\,\vec{x}; c)\,P$ *binds* the variables $\vec{x}$ in P by declaring them private to P under a constraint c. The *unit-delay*

next P executes P in the next time interval. The *time-out* **unless** c **next** P is also a unit-delay, but P is executed in the next time unit iff c is not entailed by the final store at the current time interval. Finally, the *replication* $!P$ stands for $P \parallel \mathbf{next}P \parallel \mathbf{next}^2P \parallel \ldots$, i.e., unboundedly many copies of P but one at a time. We shall use $!_{[n]}P$ to denote $P \parallel \mathbf{next}P \parallel \ldots \parallel \mathbf{next}^{n-1}P$.

From a programming language perspective, $\vec{x}$ in $(\mathbf{abs}\ \vec{x}; c)\,P$ can be seen as the formal parameters of P. This way, *recursive definitions* of the form $X(\vec{x}) \stackrel{\mathtt{def}}{=} P$ can be encoded in `utcc` as

$$\mathscr{R}[\![X(\vec{x}) \stackrel{\mathtt{def}}{=} P]\!] = !\,(\mathbf{abs}\ \vec{x}; call_x(\vec{x}))\,\widehat{P} \tag{1}$$

where $call_x$ is an uninterpreted predicate (a constraint) of arity $|\vec{x}|$. Process $\widehat{P}$ is obtained from P by replacing recursive calls of the form $X(\vec{t})$ with $\mathbf{tell}(call_x(\vec{t}))$. Similarly, calls of the form $X(\vec{t})$ in other processes are replaced with $\mathbf{tell}(call_x(\vec{t}))$.

Operational Semantics. The structural operational semantics (SOS) of `utcc` is given by two transition relations (see [9, Appendix B]): The *internal* transition $\langle P, d\rangle \longrightarrow \langle P', d'\rangle$ informally means "P with store d reduces, in one internal step, to P' with store d' ". And the *observable transition* $P \stackrel{(c,d)}{\Longrightarrow} R$ meaning "P on input c, reduces in one *time unit* to R and outputs d". The latter is obtained from a finite sequence of internal transitions. They realize the operational intuitions given above (see [12] for details).

We shall write $P \Longrightarrow Q$ when $P \stackrel{(\mathtt{true},c)}{\Longrightarrow} Q$ when c is unimportant. Let $s = c_1.c_2....c_n$ be a sequence of constraints. If $P = P_1 \stackrel{(\mathtt{true},c_1)}{\Longrightarrow} P_2 \stackrel{(\mathtt{true},c_2)}{\Longrightarrow} \ldots P_n \stackrel{(\mathtt{true},c_n)}{\Longrightarrow} P_{n+1} = Q$ then we write $P \stackrel{s}{\Longrightarrow}{}^{*} Q$ ($P \Longrightarrow^{*} Q$, when s is unimportant). The *output behavior* of P is defined as $o(P) = \{s \mid P \stackrel{s}{\Longrightarrow}{}^{*}\}$. We shall write $P \sim^{o} Q$ if $o(P) = o(Q)$.

Logic correspondence. In the CCP spirit, `utcc` constructs admit a logic interpretation. The encoding below maps `utcc` processes into FLTL formulas. Theorem 1 relates this interpretation with the SOS.

Definition 2. *Let* $[\![\cdot]\!]$ *be a map from* `utcc` *processes to FLTL formulas given by:*

$$\begin{array}{lcllcllcl}
[\![\mathbf{skip}]\!] & = & \mathtt{true} & [\![\mathbf{tell}(c)]\!] & = & c & [\![P \parallel Q]\!] & = & [\![P]\!] \wedge [\![Q]\!] \\
[\![(\mathbf{abs}\ \vec{y}; c)\,P]\!] & = & \forall \vec{y}(c \Rightarrow [\![P]\!]) & [\![(\mathbf{local}\,\vec{x}; c)\,P]\!] & = & \exists \vec{x}(c \wedge [\![P]\!]) & [\![\mathbf{next}\,P]\!] & = & \circ[\![P]\!] \\
[\![\mathbf{unless}\ c\ \mathbf{next}\,P]\!] & = & c \vee \circ[\![P]\!] & [\![!\,P]\!] & = & \Box[\![P]\!] & & &
\end{array}$$

The modalities $\circ F$ and $\Box F$ mean that F holds *next* and *always*, respectively. We use the *eventual* modality $\Diamond F$ as an abbreviation of $\neg\Box\neg F$.

Theorem 1 (Logic correspondence [13])**.** *Let* $\mathscr{L}[\![\cdot]\!]$ *be as in Definition 2 and* $s = c_1.c_2.c_3....$ *such that* $P \stackrel{s}{\Longrightarrow}{}^{*}$. *For every constraint d, it holds that:* $\mathscr{L}[\![P]\!] \vdash \Diamond d$ *iff there exists* $i \geq 1$ *such that* $c_i \vdash d$.

Derived Constructs [9]. Let `out` be an uninterpreted predicate. One could attempt at representing the actions of sending and receiving as in a name-passing calculus (say, $\bar{k}\langle\vec{e}\rangle.P$ and $k(\vec{x}).P$, resp.) with the `utcc` processes $\mathbf{tell}(\mathtt{out}(k,\vec{e}))$ and $(\mathbf{abs}\ \vec{x}; \mathtt{out}(k,\vec{x}))\,P$, resp. Nevertheless, these processes are not automatically transferred from one time-unit to the next one. Thus, they will disappear right after the current time unit even if they did not interact. Consequently, we shall use the process $(\underline{\mathbf{wait}}\ \vec{x}; c)$ **do** P (written $(\underline{\mathbf{whenever}}\ c)$ **do** P when $|\vec{x}| = 0$) which transfers itself from one time-unit to the next one until for some $\vec{t}$, $c[\vec{t}/\vec{x}]$ is entailed by the current store —intuitively, after interacting with an output. When it happens, it outputs the constraint $\bar{c}[\vec{t}/\vec{x}]$ acknowledging the successful read of c. Similarly, we define $\underline{\mathbf{tell}}(c)$ for the persistent output of c until some process reads c.

3 A Declarative Interpretation for Sessions

Here we present a compositional encoding of HVK into `utcc`. The encoding, given in Table 3, is defined over *well-typed* HVK processes. Let us briefly provide intuitions on it. Consider HVK processes $P =$

$$
\begin{array}{rcl}
[\![\textbf{request}\ a(k)\ \textbf{in}\ P]\!] & = & (\textbf{local}\, k)\,(\underline{\textbf{tell}}(\texttt{req}(a,k)) \parallel \textbf{whenever}\ \texttt{acc}(a,k)\ \textbf{do next}\,[\![P]\!]) \\
[\![\textbf{accept}\ a(k)\ \textbf{in}\ P]\!] & = & (\underline{\textbf{wait}}\ k; \texttt{req}(a,k))\ \textbf{do}\ (\textbf{tell}(\texttt{acc}(a,k)) \parallel \textbf{next}\,([\![P]\!])) \\
 & & \\
[\![k![\vec{e}];P]\!] & = & \underline{\textbf{tell}}(\texttt{out}(k,\vec{e})) \parallel \textbf{whenever}\ \overline{\texttt{out}(k,\vec{e})}\ \textbf{do next}\,[\![P]\!] \\
[\![k?(\vec{x})\ \textbf{in}\ P]\!] & = & (\underline{\textbf{wait}}\ \vec{x}; \texttt{out}(k,\vec{x}))\ \textbf{do next}\,[\![P]\!] \\
 & & \\
[\![k \lhd l;P]\!] & = & \underline{\textbf{tell}}(\texttt{sel}(k,l)) \parallel \textbf{whenever}\ \overline{\texttt{sel}(k,l)}\ \textbf{do next}\,[\![P]\!] \\
[\![k \rhd \{l_1 : P_1 \parallel \ldots \parallel l_n : P_n\}]\!] & = & (\underline{\textbf{wait}}\ l; \texttt{sel}(k,l))\ \textbf{do} \prod_{1\le i\le n} \textbf{when}\ l = l_i\ \textbf{do next}\,[\![P_i]\!] \\
[\![\textbf{throw}\ k[k'];P]\!] & = & \underline{\textbf{tell}}(\texttt{out}(k,k')) \parallel \textbf{whenever}\ \overline{\texttt{out}(k,k')}\ \textbf{do next}\,[\![P]\!] \\
[\![\textbf{catch}\ k(k')\ \textbf{in}\ P]\!] & = & (\underline{\textbf{whenever}}\ \texttt{out}(k,k'))\ \textbf{do next}\,[\![P]\!] \\
 & & \\
[\![\textbf{if}\, e\ \textbf{then}\ P\ \textbf{else}\ Q\,]\!] & = & \textbf{when}\ e \downarrow \texttt{true}\ \textbf{do next}\,[\![P]\!] \parallel \textbf{when}\ e \downarrow \texttt{false}\ \textbf{do next}\,[\![Q]\!] \\
[\![P|Q]\!] & = & [\![P]\!] \parallel [\![Q]\!] \\
[\![\textbf{inact}]\!] & = & \textbf{skip} \\
[\![(\nu u)P]\!] & = & (\textbf{local}\, u)\,[\![P]\!] \\
[\![\textbf{def}\ D\ \textbf{in}\ P]\!] & = & \prod_{X_i(x_i k_i)\in D} \mathscr{R}[\![X_i(x_i k_i)]\!]\widehat{P}
\end{array}
$$

Table 3: An Encoding from HVK into `utcc`. $\mathscr{R}[\![\cdot]\!]$ and $\widehat{P}$ are defined in Equation 1.

request $a(k)$ **in** P' and $Q =$ **accept** $a(x)$ **in** Q'. The encoding of P declares a new variable session k and sends it through the channel a by posting the constraint $\texttt{req}(a,k)$. Once $[\![Q]\!]$ receives the session key (local variable) generated by $[\![P]\!]$, it adds the constraint $\texttt{acc}(a,k)$ to notify the acceptance of k. Then, $[\![P]\!]$ and $[\![Q]\!]$ synchronize using this constraint and they execute their continuation in the next time unit. Label selection and branching synchronize on the constraint $\texttt{sel}(k,l)$. We use the parallel composition $\prod_{1\le i\le n}$ **when** $l = l_i$ **do next** $[\![P_i]\!]$ to execute the selected choice. Notice that we do not require a non-deterministic choice since the constraints $l = l_i$ are mutually exclusive [11]. As in [6], in the encoding of **if** e **then** P **else** Q , we assume an evaluation function on expressions. Once e is evaluated, $\downarrow e$ is a *constant* boolean value. The encoding of **def** D **in** P exploits the scheme described in Section 2.1 (Eq.1).

In general, acceptance in HVK is not *persistent* since, e.g., given a single **accept** $a(k)$ **in** P and two or more processes **request** $a(k)$ **in** P', one of them will be discarded. It is interesting to consider a variant of HVK processes in which session acceptance is persistent over time; this is useful to represent services which are *always* available. For such a variant (that we shall call *persistent* HVK) the following holds:

Theorem 2 (Operational Correspondence)**.** *Suppose* $[\![\cdot]\!]$ *is the encoding in Table 3. Let* P, P' *be well-typed, persistent* HVK *processes, and let R,S be* `utcc` *processes. It holds:*
1) Soundness*: If* $P \to P'$ *then, for some* P'' *and R,* $P' \longrightarrow^* P''$ *and* $[\![P]\!] \Longrightarrow R \sim^o [\![P'']\!]$.
2) Completeness*: If* $[\![P]\!] \Longrightarrow^* S$ *then, for some* P', $P \to^* P'$ *and* $[\![P']\!] \sim^o S$.

3.1 An Extended Language

We now propose $\mathsf{HVK}^{\mathsf{T}}$, a timed extension to HVK. In $\mathsf{HVK}^{\mathsf{T}}$, constructs for session request and acceptance are refined with explicit time and a construct for session abortion is considered:

Definition 3 (A timed language for sessions)**.** $\mathsf{HVK}^{\mathsf{T}}$ *processes are given by the following syntax:*

$$
\begin{array}{rcll}
P & ::= & \textbf{request}\ a(k)\ \textbf{during}\ m\ \textbf{in}\ P & \textit{Timed Session Request} \\
 & | & \textbf{accept}\ a(k)\ \textbf{given}\ c\ \textbf{in}\ P & \textit{Declarative Session Acceptance} \\
 & | & \cdots & \{\ \textit{the other constructs, as in Def. 1}\ \} \\
 & | & \textbf{kill}\, c_k & \textit{Session Abortion}
\end{array}
$$

The intuition behind these three operators is the following: **request** $a(k)$ **during** m **in** P will request a session k over the service name a during m time units. Its dual construct is **accept** $a(k)$ **given** c **in** P, which will grant the session key k when requested over the service name a, provided a successful check over the constraint c. Notice that c stands for a precondition for agreement between session request and acceptance. In c, the duration m of the corresponding session key k can be referenced by means of the variable dur_k. In the encoding we syntactically replace it by the variable corresponding to m. Finally, **kill** c_k will remove c_k from the valid set of sessions.

$$
\begin{array}{rcl}
[\![\textbf{request } a(k) \textbf{ during } m \textbf{ in } P]\!] & = & (\textbf{local}\, k)\, \underline{\textbf{tell}}(\texttt{req}(a,k,m)) \parallel \\
 & & \qquad \textbf{whenever } \texttt{acc}(a,k) \textbf{ do next } (\textbf{tell}(\texttt{act}(k)) \parallel \mathscr{G}_{\texttt{act}(k)}([\![P]\!]) \parallel \\
 & & \qquad\qquad !_{[m]} \textbf{unless } \texttt{kill}(k) \textbf{ next tell}(\texttt{act}(k))) \\
[\![\textbf{accept } a(k) \textbf{ given } c \textbf{ in } P]\!] & = & (\underline{\textbf{wait}}\ k; \texttt{req}(a,k,m) \wedge c[m/dur_k])\ \textbf{do}\ (\textbf{tell}(\texttt{acc}(a,k)) \parallel \textbf{next}\, \mathscr{G}_{\texttt{act}(k)}([\![P]\!])) \\
[\![\textbf{kill}\, k]\!] & = & !\,\textbf{tell}(\texttt{kill}(k))
\end{array}
$$

Table 4: The Extended Encoding. $\mathscr{G}_d(P)$ is defined in [9, Appendix C]

Adapting the encoding in Table 3 to consider $\mathsf{HVK}^{\mathsf{T}}$ processes is remarkably simple. Indeed, modifications to the encoding of session request and acceptance are straightforward. The most evident change is the addition of the parameter m within the constraint $\texttt{req}(a,k,m)$. The duration of the requested session is suitably represented as a bounded replication of the process defining the activation of the session k represented as the constraint $\texttt{act}(k)$. The execution of the continuation $[\![P]\!]$ is guarded by the constraint $\texttt{act}(k)$ (i.e., P can be executed only when the session k is valid). In the encoding, we use function $\mathscr{G}_d(P)$ to stand for the process which behaves as P when the constraint d can be entailed from the current store, and that is precluded from execution otherwise; see [9, Appendix C] for details. In the side of session acceptance, the main novelty is the introduction of $c[m/dur_k]$. As explained before, we syntactically replace the variable dur_k by the corresponding duration of the session m. This is a generic way to represent the agreement that should exist between a service provider and a client; for instance, it could be the case that the client requests a session longer than what the service provider is willing to grant.

3.2 Case Study: Electronic booking

Here we present a more involved example in a electronic booking scenario which makes use of the constructs introduced in $\mathsf{HVK}^{\mathsf{T}}$.

On one side, consider a company AC which offers flights directly from its website. On the other side, there is a customer looking for the best offers. In a first stage, the customer establishes a timed session with AC and asks for a flight proposal given a set of constraints (dates allowed, destination, etc.). After receiving an offer from AC, the customer can refine the selection further (e.g. by checking that the prices are below a given threshold) and loops until he finds a suitable option, that he will accept by starting the contracting phase. One possible $\mathsf{HVK}^{\mathsf{T}}$ specification of this scenario is given in Table 5.

In a second stage, the customer uses an online broker to mediate between him and a set of airlines acting as service providers. First, the customer establishes a session with the broker for a given period m; later on, the customer asks for a flight by providing his request details to the broker. At the other side, the broker will start to look into his pool of trusted service providers for the ones which can supply flights fulfilling the customer needs. All possible offers are transmitted back to the customer, who will select one from the pool of options. After that, the broker will allow the interaction between the customer and the selected service by delegating the session key used to get the offer back to the customer, while doing some post-processing after the session is established. See [9] for an $\mathsf{HVK}^{\mathsf{T}}$ specification of this scenario.

Customer	=	**request** $ob(k)$ **during** m **in** $(k![bookingdata];Select(k))$
Select(k)	=	$k?(offer)$ **in** (**if** $(offer.price \leq 1500)$ **then** $k \lhd Contract$; **else** $Select(k)$)
AC	=	**accept** $ob(k)$ **given** $m \leq MAX\_TIME$ **in** (
		$k?(bookingData)$ **in**
		$(\nu u)k![u];k \rhd \{Contract : \overline{Accept} \parallel Reject : \mathbf{kill}\, k\}$)

Table 5: Online booking example with two agents.

3.3 Exploiting the Logic Correspondence

To exploit the logic correspondence we can draw inspiration from the *constraint templates* put forward in [14], a set of LTL formulas that represent desirable/undesirable situations in service management. Such templates are divided in three types: *existence constraints*, that specify the number of executions of an activity; *relation constraints*, that define the relation between two activities to be present in the system; and *negation constraints*, which are essentially the negated versions of relation constraints. Appealing to Theorem 1, our framework allows for the verification of existence and relation constraints over $\mathsf{HVK}^{\mathsf{T}}$ programs. Assume a $\mathsf{HVK}^{\mathsf{T}}$ program P and let $F = \mathscr{L}[\![[\![P]\!]]\!]$ (i.e., the FLTL formula associated to the `utcc` representation of P). For existence constraints, assume that P defines a service accepting requests on channel a. If the service is eventually active, then it must be the case that $F \vdash \Diamond \exists_k(\mathtt{acc}(a,k))$ (recall that the encoding of **accept** adds the constraint $\mathtt{acc}(a,k)$ when the session k is accepted). A slight modification to the encoding of **accept** would allow us to take into account the number of accepted sessions and then support the verification of properties such as $F \vdash \Diamond(N_{sessions}(a) = N)$, informally meaning that the service a has accepted N sessions. This kind of formulas correspond to the existence constraints in [14, Figure 3.1.a–3.1.c]. Furthermore, making use of the guards associated to ask statements, we can verify relation constraints as eventual consequences over the system. Take for instance the specification in Table 5. Let $\overline{Accept}$ be a process that outputs "*ok*" through a session h. We may then verify the formula $F \vdash \exists_u(u.price < 1.500 \Rightarrow \mathtt{out}(h,ok))$. This is a responded existence constraint describing how the presence of an offer with price less or equal than 1.500 would lead to an acceptance state.

4 Concluding Remarks

We have argued for a timed CCP language as a suitable foundation for analyzing declarative sessions. Preliminary results presented here include an encoding of the language for structured communication in [6] into `utcc`, as well as an extension of such a language that considers explicitly elements of partial information and session duration. To the best of our knowledge, a unified framework where behavioral and declarative techniques converge for the analysis of sessions has not been proposed before.

Our work has not addressed the typed nature of the HVK language. Roughly speaking, the type discipline in [6] ensures a correct "pairing" between complementary components (e.g. session providers and requesters). Our encoding assumes processes to be well-typed with respect to such a discipline. This is because, in our view, declarative techniques should not conflict with operational techniques. Hence, we find it reasonable to assume that a `utcc`-based analysis of sessions takes places once the type system in [6] has ensured a correct pairing. In this initial effort, we have focused on exploring to what extent temporal logic can provide correctness guarantees at the session level. In a later stage, we expect to undertake a thorough study of the interplay between types, constraints, and temporal formulas in the unified framework CCP provides.

Ongoing work includes the development of a proof system —in the lines of [11]— to better support logic-based reasoning. Also, we plan to explore alternative formulations of our encodings. In particular, we would like to determine whether or not they can be expressed in the *monotonic* fragment of `utcc`,

i.e. the fragment without occurrences of **unless** processes. Such a fragment enjoys an extended set of reasoning techniques, including symbolic and denotational semantics [12], as well as static analysis techniques [5]. Hence, having encodings of HVK into such a fragment would further support our claims on the convenience of a CCP-based framework for declarative structured communications.

Acknowledgments. We are grateful to Thomas Hildebrandt and Marco Carbone for insightful discussions on the topics of this paper, and for giving useful comments on previous versions of this document. We are also grateful to the anonymous reviewers for their comments and remarks. The contribution of Olarte and Pérez was initiated during short research visits to the IT University of Copenhagen. They are most grateful to the IT University and to the FIRST PhD Graduate School for funding such visits.

References

[1] M. Boreale, R. Bruni, L. Caires, R. D. Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. T. Vasconcelos, and G. Zavattaro. Scc: A service centered calculus. In *Proc. of WS-FM*, volume 4184 of *LNCS*, pages 38–57. Springer, 2006.

[2] M. G. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. In *Proc. of ESOP*, volume 4421 of *LNCS*, pages 18–32. Springer, 2007.

[3] M. Coppo and M. Dezani-Ciancaglini. Structured Communications with Concurrent Constraints. In *Proc. of TGC'08*, LNCS. Springer. To Appear.

[4] J. F. Díaz, C. Rueda, and F. D. Valencia. Pi+- calculus: A calculus for concurrent processes with constraints. *CLEI Electron. J.*, 1(2), 1998.

[5] M. Falaschi, C. Olarte, and C. Palamidessi. A framework for abstract interpretation of universal timed concurrent constraint programs. Technical report, LIX, Ecole Polytechnique and Siena University, 2008.

[6] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. of ESOP*, volume 1381 of *LNCS*. Springer, 1998.

[7] I. Lanese, F. Martins, V. T. Vasconcelos, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *Proc. of SEFM*, pages 305–314. IEEE Computer Society, 2007.

[8] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Proc. of ESOP*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.

[9] H. A. López, C. Olarte, and J. A. Pérez. Towards a unified framework for declarative structured communications. Working Draft Available at `http://www.itu.dk/~hual/PersonalSite/Publications_files/sessions-extended.pdf`, February 2009.

[10] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1991.

[11] M. Nielsen, C. Palamidessi, and F. D. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nord. J. Comput.*, 9(1):145–188, 2002.

[12] C. Olarte and F. D. Valencia. The expressivity of universal timed ccp: undecidability of monadic fltl and closure operators for security. In *Proc. of PPDP*, pages 8–19. ACM, 2008.

[13] C. Olarte and F. D. Valencia. Universal concurrent constraint programing: symbolic semantics and applications to security. In *Proc. of SAC*, pages 145–150. ACM, 2008.

[14] M. Pesic and W. M. P. van der Aalst. A declarative approach for flexible business processes management. In *BPM'06 Workshops*, volume 4103 of *LNCS*, pages 169–180. Springer, 2006.

[15] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[16] V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of LICS*, pages 71–80. IEEE Computer Society, 1994.

[17] V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[18] B. Victor and J. Parrow. Concurrent constraints in the fusion calculus. In *Proc. of ICALP*, volume 1443 of *LNCS*, pages 455–469. Springer, 1998.

Execution Models for Choreographies and Cryptoprotocols

Marco Carbone*
IT University of Copenhagen
Copenhagen, Denmark
carbonem@itu.dk

Joshua Guttman
MITRE
Boston, MA, U.S.A.
guttman@mitre.org

Abstract

A choreography describes a transaction in which several principals interact. Since choreographies frequently describe business processes affecting substantial assets, we need a security infrastructure in order to implement them safely. As part of a line of work devoted to generating cryptoprotocols from choreographies, we focus here on the execution models suited to the two levels.

We give a strand-style semantics for choreographies, and propose a special execution model in which choreography-level messages are faithfully delivered exactly once. We adapt this model to handle multiparty protocols in which some participants may be compromised.

At level of cryptoprotocols, we use the standard Dolev-Yao execution model, with one alteration. Since many implementations use a "nonce cache" to discard multiply delivered messages, we provide a semantics for at-most-once delivery.

1 Introduction

Choreographies are global descriptions of transactions including business or financial transactions. They describe the intertwined behavior of several principals as they negotiate some agreement and—frequently—commit some state change. A key idea is *end-point projection* [2], which converts a global description into a set of descriptions that determine the local behavior of the individual participants in a choreography. Conversely, *global synthesis* of a choreography from local behaviors is also sometimes possible, i.e. meshing a set of local behaviors into a comprehensive global description [7].

Because these transactions may transfer sums of money and other objects of value, or may communicate sensitive information among the principals, they require a security infrastructure. It would be desirable to synthesize a cryptographic protocol directly from a choreography description, to control how adversaries can interfere with transactions among compliant principals. Corin et al. [4] have made a substantial start on this problem, with further advances described in [3]. However, many questions remain, for instance how to optimize the generated cryptographic protocols, how best to establish that they are always correct, and indeed how best to define their correctness.

This last question concerns how to state what control the protocol should provide, against adversaries trying to interfere with transactions. It is a substantial question because the execution model that choreographies use is quite distant from the execution model cryptographic protocols are designed to cope with. For instance, choreographies use an execution model—or communication model—in which messages are never received by any party other than the intended recipient, or if the formalism represents channels, they are received only over the channel. Moreover, messages are always delivered if the recipient is willing to receive the message. Messages are delivered only if they were sent, and specifically only if they were sent by the expected peer. Finally, they are delivered only once. These aspects of the model mean that confidentiality and integrity properties are built into the underlying assumptions. A security infrastructure is intended to justify exactly these assumptions, i.e. to provide a set of behaviors in which these assumptions are satisfied.

Naturally, these behaviors must be achieved within an underlying model in which the adversary is much stronger. In this model—typically called the *Dolev-Yao model*, after a paper [5] in which Dolev and Yao formalized ideas suggested by Needham and Schroeder [8]—all messages may be received by the adversary, so that confidentiality needs to be achieved by encryption. They may be delivered zero times,

*The author was supported by EPSRC grant EP/F002114

once, or repeatedly, and they may be misdelivered to the wrong participant. When delivered, a message may appear to come from a participant that did not send it. The adversary may alter messages in transit, including applying cryptographic operations using keys that he knows, or may obtain by manipulating the protocol.

Digital signatures may be used to notify a recipient reliably of the source of a message (and of the integrity of its contents). Symmetric encryption may also be used to ensure authenticity: a recipient knows that the encrypted message was prepared by a party that knew the secret key, and intended it for a peer that also knew the secret key. Nonces, which are simply randomly chosen bitstrings, may be used to ensure freshness. The principal P that chose a nonce knows, when receiving a message containing it, that the nonce was inserted after P chose it. Moreover, if P engages in many sessions and associates a different nonce with each, P can ensure that messages containing one nonce cannot be misdirected to a session using a different nonce.

In this paper, we begin the process of relating the Dolev-Yao model of execution to the choreography execution model. This is a key step in generating cryptographic protocols and proving them faithful to the intent of the choreography. In particular, we represent the two execution models using the strand space model [9, 6].

Strand Spaces. Strand spaces were developed as a simplest possible model for cryptographic protocol analysis, but are also applicable to other kinds of distributed systems.

A *strand* is a finite linear sequence of transmission and reception events. We use a strand to represent the communication events of a single principal in a single local run of a protocol. We also use a strand to represent an atomic action of the adversary, e.g. receiving a message and an encryption key, and transmitting the result of encrypting the given message with that key. We write $m \Rightarrow n$ when n is the *node* (i.e. event) immediately following m on the same strand.

A *bundle* is a causally well-founded graph—essentially, a Lamport diagram—built from strands and transmission edges. A transmission edge $m \rightarrow n$ is possible when m is a transmission node; n is a reception node; and the message a transmitted on m is the same as the message received on n. A finite, acyclic directed graph $\mathscr{B} = (\mathscr{N}, \mathscr{E})$ is a *bundle* if (1) $\mathscr{N}$ is a set of nodes such that if $n \in \mathscr{N}$ and $m \Rightarrow n$, then $m \in \mathscr{N}$; (2) $\mathscr{E}$ is the union of a set of transmission edges $\rightarrow_{\mathscr{B}}$ with the set of strand edges $\Rightarrow$ restricted to nodes in $\mathscr{N}$, which we will write as $\Rightarrow_{\mathscr{B}}$; and (3) for every reception node $n \in \mathscr{N}$, there is exactly one $m \in \mathscr{N}$ such that $m \rightarrow_{\mathscr{B}} n$.

We write $m \preceq_{\mathscr{B}} n$ if there is a path from m to n in $\mathscr{B}$ using arrows in $\Rightarrow_{\mathscr{B}} \cup \rightarrow_{\mathscr{B}}$. The relation $\preceq_{\mathscr{B}}$ is a well-founded partial order, meaning that the *bundle induction* principle holds, that every non-empty set of nodes of $\mathscr{B}$ contains $\preceq_{\mathscr{B}}$-minimal members.

The notions of strand and bundle, and the principle of bundle induction, are the essential ingredients in the strand space model. Choices—such as what operations the adversary strands offer, or what additional closure properties bundles may satisfy—can vary to model different problems concerning cryptographic protocols or distributed communication more generally.

Goals of this Paper. We provide a few definitions and an example to indicate how the strand space framework can relate choreographies to the cryptographic protocols that implement them.

In particular, we consider a very simple choreography language, and provide a semantics for it as a set of "abstract bundles." That is, each session of the protocol executes according to one of the bundles predicted by the semantics. Moreover, any collection of sessions that may have occurred takes the following form: its events partition into bundles that are obtained by instantiating the parameters in bundles given in the semantics. Moreover, if two nodes belong to different partition elements, there is no $\preceq$ ordering between them, unless the executions are generated as parts of some higher-level choreography

that might determine a causal ordering.

We call this an *abstract bundle semantics* because it builds in the assumptions of the choreography level: messages do not have explicit cryptographic operations, and the choreography-level communications assumptions are satisfied. Messages are always delivered exactly once; sender and recipient are never mismatched; no message is created by adversary operations. We must connect this idealized semantics with a more realistic semantics at the cryptographic level, in which the adversary may be active.

One peculiarity of our message datatype is that we allow "boxes." A box $[\tilde{M}]_{\rho,\rho'}$ is a message prepared on role ρ that can be opened only by a principal playing role ρ'. At the choreography level, this property is enforced by a type system. We use these boxes to make explicit the confidentiality and authentication requirements of a choreography in the case where some roles are played by compromised participants. However, in this note, we focus on the simplest case, in which no participants are compromised. That is, we will assume here, that any participant who is sent a box, will behave only as predicted by the choreography.

Our semantics at the *cryptographic level* is a standard strand space treatment, except for one ingredient. Namely, this semantics assumes that some kinds of messages are delivered at most once. These are session-initiating messages that contain a nonce, or in some protocols a freshly generated session key. Implementations now use a nonce-caching technique in which the nonces of previously executed sessions are retained in a cache. A new incoming message contains a nonce which is compared against the cache; if it is present, then with overwhelming probability there has been a replay attempt, and the message is discarded. Otherwise, the nonce is recorded and the session proceeds. So as not to need to retain nonces forever, implementations typically combine this with a timestamp, and assume that uncompromised principals are loosely synchronized. A message with too old a timestamp is discarded. Nonces may be dropped from the cache when their timestamps have expired. In this approach, the nonce and the timestamp must appear digitally signed in the incoming message to prevent manipulation by the adversary.

We define a cryptographic protocol to properly implement a choreography if, when abstracting its possible executions in this at-most-once semantics, we obtain exactly the possible executions of the abstract bundle semantics for the choreography.

We explore here a simple example in which the participants are well-known to each other from the start of the transaction. However, the ideas also apply when additional participants may be chosen during execution, and keys must be distributed as part of the message flow.

2 An Execution Model for Choreography

Syntax and Semantics. Let ρ range over the set of roles $\mathscr{R}$. The syntax of our choreography mini-language (based on the Global Calculus [2]) is given by the following grammar:

$$C ::= \quad \Sigma_i\, \rho_1 \to \rho_2 : \mathsf{op}_i\langle \tilde{M}_i\rangle.\, C_i \mid \quad \mathbf{0} \qquad\qquad M ::= \quad v \mid [\tilde{M}]_{\rho_1,\rho_2}$$

Above, the term $\Sigma_i\, \rho_1 \to \rho_2 : \mathsf{op}_i\langle \tilde{M}_i\rangle.\, C_i$ describes an interaction where a branch with label op_i is non-deterministically selected and a message $\tilde{M}_i$ is sent from role ρ_1 to role ρ_2. Each two roles in a choreography share a private channel hence it would be redundant to have them explicit in the syntax [1].

Term $\mathbf{0}$ denotes the inactive system. A message M can either be a value v or a box $[\tilde{M}]_{\rho_1\rho_2}$. The latter denotes a tuple of messages M_i from ρ_1 that can only be opened by ρ_2.

Our mini-language can be equipped with a standard trace semantics with configurations $C \xrightarrow{\mu} C'$ where μ contains the parameters of the interaction performed.

Buyer-Seller Example. We report a variant of the Buyer-Seller financial protocol [2]. A buyer Buyer asks a seller Seller for a quote about a product. If the quote is accepted, Buyer will send its credit

card `card` to Seller who will forward it to a bank Bank. The bank uses a trusted database Accounts containing all information about Buyer's account. Accounts will check if payment can be done and, if so, reply with a receipt `ROG` which will be forwarded to Buyer. In our mini-language:

1. $\mathsf{Buyer} \to \mathsf{Seller} : \mathsf{req}\langle \texttt{prod} \rangle.\ \mathsf{Seller} \to \mathsf{Buyer} : \mathsf{reply}\langle \texttt{quote} \rangle.$
2. $(\ \mathsf{Buyer} \to \mathsf{Seller} : \mathsf{OK}\langle [\texttt{card}]_{\mathsf{Buyer,Bank}} \rangle.\ \mathsf{Seller} \to \mathsf{Bank} : \mathsf{pay}\langle [\texttt{card}]_{\mathsf{Buyer,Bank}} \rangle.$
3. $\quad \mathsf{Bank} \to \mathsf{Accounts} : \mathsf{pay}\langle \texttt{card} \rangle.\ (\ \mathsf{Accounts} \to \mathsf{Bank} : \mathsf{OK}\langle \texttt{ROG} \rangle.\ \mathsf{Bank} \to \mathsf{Seller} : \mathsf{OK}\langle [\texttt{ROG}]_{\mathsf{Bank,Buyer}} \rangle.$
4. $\qquad\qquad\qquad\qquad\qquad \mathsf{Seller} \to \mathsf{Buyer} : \mathsf{OK}\langle [\texttt{ROG}]_{\mathsf{Bank,Buyer}} \rangle$
5. $\qquad\qquad\qquad\qquad\qquad\qquad +$
6. $\qquad\qquad\qquad\qquad\qquad \mathsf{Accounts} \to \mathsf{Bank} : \mathsf{NotOK}\langle\rangle.\ \mathsf{Bank} \to \mathsf{Seller} : \mathsf{NotOK}\langle\rangle.$
7. $\qquad\qquad\qquad\qquad\qquad \mathsf{Seller} \to \mathsf{Buyer} : \mathsf{NotOK}\langle\rangle\ )$
8. $\qquad\qquad\qquad\qquad +$
9. $\quad \mathsf{Buyer} \to \mathsf{Seller} : \mathsf{NotOK}\langle \texttt{reason} \rangle)$

Line 1. denotes the quote request and reply. Lines 2. and 9. are computational branches corresponding to acceptance and rejection of the quote respectively. If the quote is accepted, Buyer will send its credit card in the box $[\mathsf{card}]_{\mathsf{Buyer,Bank}}$ meaning that Seller cannot see it. The card number is then forwarded to Bank who opens the box and sends its content to Accounts (line 3.). If the transaction can be finalised a receipt is forwarded to Buyer. Otherwise, a NotOK message will be delivered. Bank boxes the receipt so that it cannot be seen or changed by Seller.

Abstract Bundle Semantics (ABS). We introduce an alternative semantics for choreography based on bundles defined as judgements of the form:

$$\models C \rhd \{(\mathscr{B}_1, \mathsf{who}_1), \ldots, (\mathscr{B}_i, \mathsf{who}_i)\}$$

where $(\mathscr{B}, \mathsf{who})$ is a *bundle environment*. Given a role ρ, $\mathsf{who}(\rho)$ denotes the strand in the bundle $\mathscr{B}$ associated to the behaviour of ρ. The abstract bundle semantics $[\![C]\!] = \{(\mathscr{B}_1, \mathsf{who}_1), \ldots, (\mathscr{B}_i, \mathsf{who}_i)\}$ if and only $\models C \rhd \{(\mathscr{B}_1, \mathsf{who}_1), \ldots, (\mathscr{B}_i, \mathsf{who}_i)\}$. The relation $\models$ is the minimum relation satisfying the following:

$$(\text{ABS-COM}) \quad \frac{\forall i.\ \models C_i \rhd \{(\mathscr{B}_{i1}, \mathsf{who}_{i1}), \ldots, (\mathscr{B}_{ij_i}, \mathsf{who}_{ij_i})\}}{\models \Sigma_i \rho_1 \to \rho_2 : \mathsf{op}_i\langle \tilde{M}_i \rangle.\ C_i \rhd (\ \bigcup_i \{(\mathscr{B}_{ij_i}, \mathsf{who}_{ij_i})\}_{j_i}[\rho_1, \rho_2, \mathsf{op}_i(\tilde{M}_i)]\ )}$$

$$(\text{ABS-ZERO}) \quad \frac{\mathsf{e}\ \text{fresh}}{\emptyset \models \mathbf{0} \rhd (\{\mathsf{e}^\rho\}_\rho, \lambda\rho.\ \mathsf{e}^\rho)}$$

In (ABS-COM), $(\mathscr{B}_{ij_i}, \mathsf{who}_{ij_i})[\rho_1, \rho_2, \mathsf{op}_i(\tilde{M}_i)]$ denotes a new bundle obtained from $\mathscr{B}_{ij_i}$ where the two strands $\mathsf{who}_{ij_i}(\rho_1)$ and $\mathsf{who}_{ij_i}(\rho_2)$ are prefixed with the events $+\mathsf{op}_i(\tilde{M}_i)$ and $-\mathsf{op}_i(\tilde{M}_i)$ respectively. The function who_{ij_i} is updated accordingly. This operation is applied to all those bundles obtained from the semantics of each branch and the result will be their union. In (ABS-ZERO), we augment the set A with fresh events $\{\mathsf{e}^\rho\} \in E$ in order to distinguish each strand.

ABS Example. The ABS for the Buyer-Seller protocol has three bundles corresponding to its possible executions, namely: (i) Buyer accepts the quote and Accounts successfully finalises the transaction sending back a receipt; (ii) Buyer accepts the quote but Accounts does not accept the payment (with reason `reason`); and (iii) Buyer does not accept the quote and the protocol terminates. The three corresponding bundles are reported in Fig. 1.

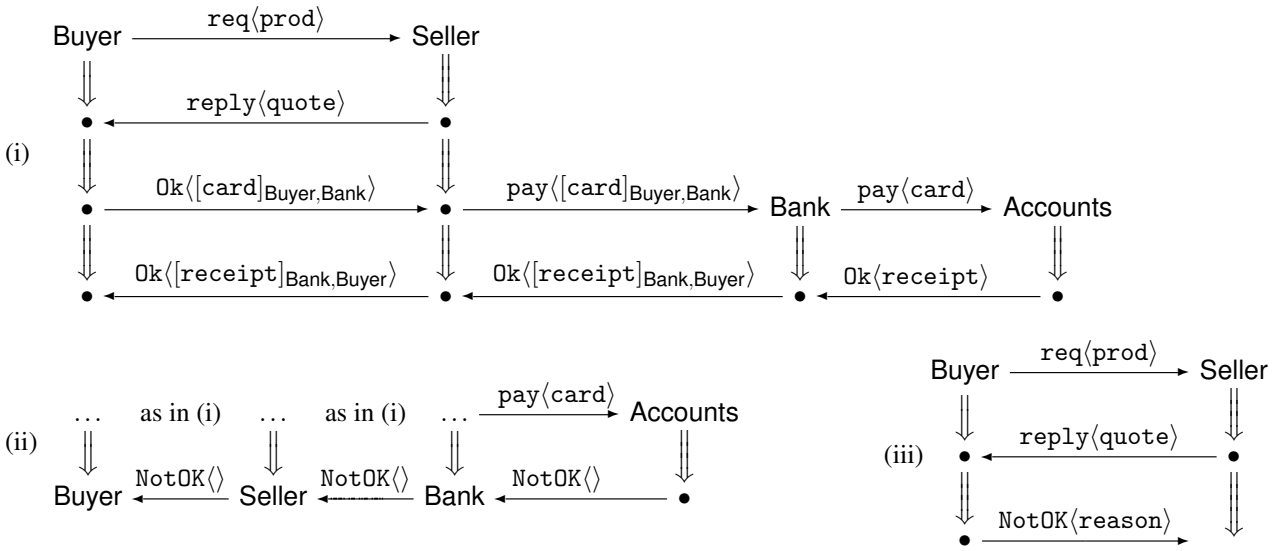

Figure 1: Abstract Bundle Semantics for the Buyer-Seller protocol

3 An execution model for Cryptoprotocols

A node is a *regular* node of a protocol Π if it lies on a strand of Π, not on an adversary strand.

Definition 1 (Deliver-once)**.** *Suppose that S is a set of messages, and $\mathscr{B}$ is a bundle. $\mathscr{B}$* delivers messages in S only once *if there exists an injective function $f : R \to T$, where*

R is the set of regular nodes n in $\mathscr{B}$ such that a member of S is received on n, and

T is the set of regular nodes n in $\mathscr{B}$ such that a member of S is transmitted on n.

When $\{S_i\}_{i\in I}$ is a family of sets indexed by $i \in I$, we say that $\mathscr{B}$ is deliver-once *for $\{S_i\}_{i\in I}$ when $\mathscr{B}$ delivers messages in each S_i only once.*

We typically apply this definition when I is a set of values that will be generated freshly, and S_i is a set of messages of particular forms containing one such value i ($K_{j,k}$ in the example below).

Cryptoprotocol Example. The Buyer-Seller cryptoprotocol implements the choreography example of Section 2. It provides parametric strands that define the behaviors of the principals as they send and receive encrypted messages to provide security services for the behaviors in the choreography. The central idea is that the first message between a pair of participants P_1, P_2 contains a nonce-like, freshly generated symmetric session key $K_{1,2}$ to be used for all the messages that the choreography requires. It is created by P_1 and delivered to P_2 encrypted with a long term public encryption key associated with P_2. The latter is an asymmetric encryption key. The generation of $K_{1,2}$ provides guarantees of freshness, and since it is associated with a unique session, prevents messages from one session from being accepted in any other session. We do not define the whole protocol here, but in Fig. 2 indicate the desired run in the main case, where no messages are rejected with a `NotOK` reply.

Abstraction and Correctness. A function α from messages to messages is an *abstraction map* if (1) $\alpha(a)$ contains no cryptographic operators, and no nonces or keys, and (2) the parameters appearing in $\alpha(a)$ always appear in a.

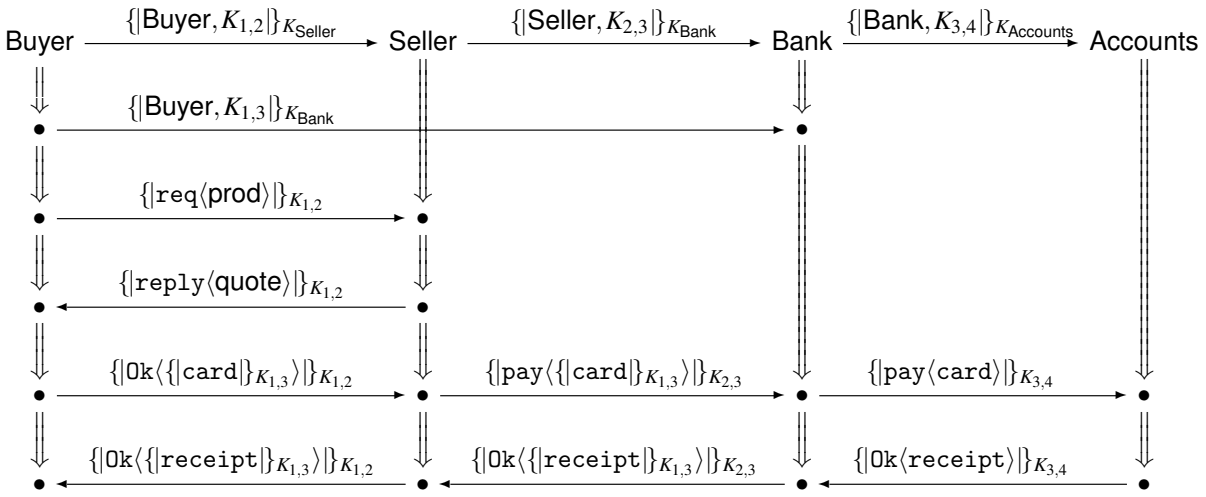

Figure 2: Successful run of the Cryptoprotocol

For instance, α could map $\{|\texttt{reply}\langle\texttt{quote}\rangle|\}_{K_{12}}$ to $\texttt{reply}\langle\texttt{quote}\rangle$, etc. We say that an abstract strand s is an *image* of a cryptographic strand s_c if the transmission and reception nodes of s form a subsequence of the result $\alpha(s_c)$ of mapping α through the sequence s_c.

An abstract bundle $\mathscr{B}$ is an *image* of a cryptographic bundle $\mathscr{B}_c$ if (1) there is a bijection ϕ between regular strands s_c in $\mathscr{B}_c$ and regular strands s in $\mathscr{B}$; (2) $\phi(s_c)$ is always an image of s_c; and (3) the transmission relation $\rightarrow_{\mathscr{B}}$ is formed by connecting nodes of $\mathscr{B}$ such that $m \rightarrow_{\mathscr{B}} n$ implies $m_c \preceq_{\mathscr{B}_c} n_c$, for some concrete nodes of which m, n are images.

Definition 2 (Faithfulness)**.** *Cryptoprotocol* Π *is* faithful to *choreography C if there is an abstraction function* α *such that:*

1. *If* $\mathscr{B} \in [\![C]\!]$ *is in the ABS of C, then* $\mathscr{B}$ *is an image of some bundle* $\mathscr{B}_c$ *of* Π*;*
2. *If* $\mathscr{B}_c$ *is a bundle of* Π*, then we may partition* $\mathscr{B}_c$ *into sub-bundles* $\{\mathscr{B}_i\}_i$ *such that:*
   (a) *if* $m \in \mathscr{B}_i$*,* $n \in \mathscr{B}_j$*, and* $m \preceq_{\mathscr{B}_c} n$*, then* $i = j$*;*
   (b) *If* $\mathscr{B}_0$ *is an image of any* $\mathscr{B}_i$*, then there is a* $\mathscr{B} \in [\![C]\!]$ *and a substitution* σ *such that* $\mathscr{B}_0$ *is an initial sub-bundle of* $\sigma(\mathscr{B})$*.*

If $\{S_i\}_{i\in I}$ *is a family of sets of messages, then* Π *is* faithful to C assuming the deliver-once property for $\{S_i\}_{i\in I}$ *if the above holds for bundles of* Π *that are deliver-once for* $\{S_i\}_{i\in I}$*.*

4 Concluding Remarks

We have introduced two execution models, one for a choreography—assuming no compromised participants—and one for cryptoprotocols with deliver-once assumptions. The abstract bundle semantics gives a set of bundles representing all the possible runs of the protocol described by a choreography.

In future work, we intend to adapt these ideas to justify a method for generating cryptoprotocols that are faithful to a choreography.

References

[1] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *19th International Conference on Concurrency Theory (Concur'08)*, LNCS, pages 418–433. Springer, 2008.

[2] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centred Programming for Web Services. In *16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[3] Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan, and James J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. Available at `http://www.msr-inria.inria.fr/projects/sec/sessions/cryptographic-protocol-synthesis-and-verification-for-multiparty-sessions.pdf`, July 2008.

[4] Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan, and James J. Leifer. A secure compiler for session abstractions. *Journal of Computer Security*, 16(5):573–636, 2008.

[5] Daniel Dolev and Andrew Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.

[6] Joshua D. Guttman, Jonathan C. Herzog, John D. Ramsdell, and Brian T. Sniffen. Programming cryptographic protocols. In Rocco De Nicola and Davide Sangiorgi, editors, *Trust in Global Computing*, number 3705 in LNCS, pages 116–145. Springer, 2005.

[7] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP Proceedings*, LNCS. Springer, March 2009.

[8] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), December 1978.

[9] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(1), 1999.

Towards the Safe Programming of Wireless Sensor Networks

Francisco Martins
FCUL & LASIGE
Lisboa, Portugal
fmartins@di.fc.ul.pt

Luís Lopes
FCUP & CRACS/INESC-Porto
Porto, Portugal
lblopes@dcc.fc.up.pt

João Barros
FEUP & IT
Porto, Portugal
jbarros@fe.up.pt

Abstract

Sensor networks are rather challenging to deploy, program, and debug. Current programming languages for these platforms suffer from a significant semantic gap between their specifications and underlying implementations. This fact precludes the development of (type-)safe applications, which would greatly simplify the task of programming and debugging deployed networks. In this paper we define a core calculus for programming sensor networks and propose to use it as an assembly language for developing type-safe, high-level programming languages.

keywords: Sensor Networks, Programming Languages, Process-Calculi.

1 Introduction and Motivation

Wireless sensor networks are composed of huge numbers of small physical devices capable of sensing the environment and connected using ad-hoc networking protocols over radio links. These platforms have several unique characteristics when compared with other ad-hoc networks. First, sensor networks are often designed for specific applications or application domains making software reusability and portability an issue. Sensor devices have very limited processing power (CPU), available memory, and battery lifetime, and are often deployed at remote locations making physical access to the devices (*e.g.* for maintenance) difficult or impossible. For these reasons, programming these large scale distributed systems can be daunting. Programs must be *lightweight*, produce a *small memory footprint*, be *power conservative*, be *self-reconfigurable* (i.e. may be reprogrammed dynamically without physical intervention on the devices) and, be *(type-)safe*.

To date several programming languages and run-time systems have been proposed for wireless sensor networks [1] that address some of the above issues, but few tackle the *safety* issue. Regiment [6], a strongly typed functional *macroprogramming* language, is the closest to achieve this goal by providing a type-safe compiler. However, Regiment is then compiled into a low-level *token machine language* that is not type-safe. This intermediate language is itself compiled into a nesC implementation of the run-time based on the *distributed token machine* model, for which no safety properties are available. In fact, in general, an underlying model with well studied operational semantics for sensor networks seems to be lacking. The absence of such a model reveals itself as a considerable semantic gap between the semantics of the (sometimes high-level) programming languages and their respective implementations.

In this paper we propose Callas, a calculus for programming sensor networks, based on the formalism of process calculi [2, 5], that aims to establish a basic computational model for sensor networks. The goal is to diminish the above mentioned semantic gap by proceeding bottom-up, using Callas as a basic assembly language upon which high-level programming abstractions may be encoded as semantics preserving, derived constructs. Callas is an evolution from a previous proposal [3] by the authors, which unlike its original sibling provides: (a) decoupled semantics for in-sensor computation (associated with the *application layer*) and networking (associated with the *data-link* and *network* layers); b) support for a form of timed *events* and; c) an *event-driven* semantics.

2 Overview of Callas

The syntax of Callas is provided by the grammar in Figure 1. Let $\vec{\alpha}$ denote a possibly empty sequence $\alpha_1 \dots \alpha_n$ of elements of some syntactic category α. We let l range over a countable set of *labels* rep-

$S ::=$	*Sensors*	$C ::=$	*Code*
$\mathbf{0}$	empty network	$\{l_i = (\vec{x}_i)\, P_i\}_{i \in I}$	code module
$\mid S \mid S$	composition		
$\mid [R \triangleright C, T]^{I,O}_{p,t}$	sensor	$P ::=$	*Processes*
		v	value
$v ::=$	*Values*	$\mid l(\vec{v})$ **every** v **expire** v	call
b	built-in value	$\mid$ **system** $l(\vec{v})$	system call
$\mid x$	variable	$\mid$ **send** $l(\vec{v})$	communication
$\mid C$	code	$\mid$ **receive**	communication
		$\mid$ **install** v	install code
$m ::= \langle l(\vec{v}) \rangle$	messages	$\mid$ **let** $x = P$ **in** P	sequence
$I, O ::= m_1 :: \cdots :: m_n$	message queues	$R ::= P_1 :: \cdots :: P_n$	run-queue

Figure 1: The syntax of Callas.

resenting function names, and let x range over a countable set of *variables*. These sets are pairwise disjoint.

A network S is an abstraction for a network of real-world sensors connected via radio links. We write it as a flat, unstructured collection of sensors combined using the parallel composition operator. The empty network is represented by symbol $\mathbf{0}$. A sensor $[R \triangleright C, T]^{I,O}_{t,p}$ is an abstraction for a sensor device. It features a run-queue of processes (R), in which the front process is running. Its memory stores both the code for the applications (C) and a table of timers for function calls (T). These components constitute the application layer of the protocol stack for the sensor. The interface with the lower level networking and data-link layers is modeled using incoming (I) and outgoing (O) queues of messages. The sensors have a measurable position (p) and their own clocks (t), and are able to measure some physical property (e.g. temperature, humidity) by calling appropriate *system* functions. The code in C consists of a set of named functions. The syntax $l = (\vec{x})P$ represents a function, where l is the name, $(\vec{x})$ the parameters, and P the body. The I (O) queue buffers messages received from (sent to) the network. Messages are just packaged function calls $\langle l(\vec{v}) \rangle$. Finally, T is a set that keeps information on timers for function calls. For each timer, a tuple is maintained with the following information: the call to be triggered, the timer period, the time after which the timer expires and, the time of the next call.

A process P can be: a synchronous system call – **system** $l(\vec{v})$; a code installation – **install** C' (that adds the set of functions in C' to C); an asynchronous remote call – **send** $l(\vec{v})$ (that adds a message $\langle l(\vec{v}) \rangle$ to the O queue); a receptor – **receive** (that gets a similar message from the I queue); a timer – $l(\vec{v})$ **every** v **expire** v (that calls a function $l(\vec{v})$ periodically until it expires); and, finally a **let** construct that allows the processing of intermediate values in computations. The later is also useful to derive a basic sequential composition construct (in fact, **let** $x = P$ **in** $P' \equiv P\ ;\ P'$ with $x \notin \mathrm{fv}(P')$). We make frequent use of this construct to impose a more imperative style of programming. We also write $l(\vec{v})$ **every** 0 **expire** 0 as $l(\vec{v})$ for convenience.

Values v are the data exchanged between sensors and comprise basic values (b) that can intuitively be seen as the primitive data types supported by the sensor's hardware, and collections of functions (C).

In the sequel we present two small examples of programs written with Callas. Both examples have two components: the code to be run at a base-station (*sink*) and the code to be run at each of the sensors.

Streaming data. The program that runs on the sink starts by installing a code module that contains the function gather in the local memory (*C*). This function simply logs the arguments using a built-in system call. It then broadcasts a call to setup with the desired period (the units are in internal clock cycles of the sensor) and time interval for the sampler process. The call is placed in the output queue of the sink (*O*).

```
// sink
install { gather = (x,y) system log(x,y) };
send setup(period,interval)

// sensors
install { setup  = (x,y) sample() every x expire y
          sample = ()    let x = system time() in
                         let y = system data() in
                         send gather(x,y)                }
```

Note that the routing of messages is transparent at this level. It is controlled at the network and data-link layers and we simulate this by having an extra semantic layer for the network (c.f. Figure 3). In this example, the messages from the sink are delivered to every sensor that carries a setup function. The information originating in the sensors, in the form of gather messages, on the other hand, is successively relayed up to the sink (since sensors have no gather functions implemented).

Each sensor installs two functions: setup, and sample. When a sensor receives a call to setup from the network (through *I*), it sets up a timer to periodically call sample. When this function is executed the local time and the desired data are read with system calls and a gather message is sent to the network carrying those values.

The maximum value of a data attribute and the MAC address of the sensor that reads it. This example follows much the same principles of the above, except that it is a single shot request. Instead of computing the maximum value of the data attribute only at the sink, we optimize the program so that each sensor has two attributes max_data and max_mac that keep, respectively, the maximum value for the data that passed through the sensor, and the associated MAC address.

```
// sink
install { gather = (x,y) system log(x,y) };
send setup()

// sensors
install { setup  = ()    let x = system data() in
                         let y = system mac()  in
                         install { max_data = () x
                                   max_mac  = () y };
                         send gather (x,y)

          gather = (x,y) let val = max_data() in
                         if x > val then
                            install { max_data = () x
                                      max_mac = () y };
                            send gather(x,y);            }
```

Sensors read their data and MAC addresses and send gather messages to the network with that information. Each time such a message is relayed by a sensor on its way to the sink, the relaying sensor checks whether it is worth to send the data forward by comparing it with the local maximum. This strategy manages to substantially reduce the required bandwidth at the sensors closest to the sink. The sink implementation of gather stops the relaying and logs the data. Note that in this example, to simplify, more than one value may be recorded at the sink. Also, we use an **if**–**then** construct that is not provided

$$\frac{\text{noEvent}(T,t)}{[\mathscr{C}[\![\textbf{system } l(\vec{v})]\!] :: R \triangleright C, T]^{I,O}_{p,t} \to [\mathscr{C}[\![v]\!] :: R \triangleright C, T]^{I,O}_{p,t+1}} \quad \text{(R-SYSTEM)}$$

$$\frac{\text{noEvent}(T,t)}{[\mathscr{C}[\![\textbf{install } C']\!] :: R \triangleright C, T]^{I,O}_{p,t} \to [\mathscr{C}[\![\{\}]\!] :: R \triangleright C + C', T]^{I,O}_{p,t+1}} \quad \text{(R-INSTALL)}$$

$$\frac{\text{noEvent}(T,t)}{[\mathscr{C}[\![\textbf{send } l(\vec{v})]\!] :: R \triangleright C, T]^{I,O}_{p,t} \to [\mathscr{C}[\![\{\}]\!] :: R \triangleright C, T]^{I,O::\langle l(\vec{v})\rangle}_{p,t+1}} \quad \text{(R-SEND)}$$

$$\frac{\text{noEvent}(T,t)}{[\mathscr{C}[\![\textbf{receive}]\!] :: R \triangleright C, T]^{\langle l(\vec{v})\rangle::I,O}_{p,t} \to [R :: l(\vec{v}) \triangleright C, T]^{I,O}_{p,t+1}} \quad \text{(R-RECEIVE)}$$

$$\frac{\text{noEvent}(T,t)}{[\mathscr{C}[\![\textbf{let } x = v \textbf{ in } P]\!] :: R \triangleright C, T]^{I,O}_{p,t} \to [\mathscr{C}[\![P[v/x]]\!] :: R \triangleright C, T]^{I,O}_{p,t}} \quad \text{(R-LET)}$$

$$\frac{\text{noEvent}(T,t)}{[v :: P :: R \triangleright C, T]^{I,O}_{p,t} \to [P :: R \triangleright C, T]^{I,O}_{p,t+1}} \qquad \frac{\text{noEvent}(T,t)}{[R \triangleright C, T]^{I,O}_{p,t} \to [R \triangleright C, T]^{I,O}_{p',t}} \quad \text{(R-NEXT,R-MOVE)}$$

$$\frac{C(l) = (\vec{x})P \quad \text{noEvent}(T,t)}{[\mathscr{C}[\![l(\vec{v})]\!] :: R \triangleright C, T]^{I,O}_{p,t} \to [\mathscr{C}[\![P[\vec{v}/\vec{x}]]\!] :: R \triangleright C, T]^{I,O}_{p,t+1}} \quad \text{(R-CALL)}$$

$$\frac{l \notin \text{dom}(C) \quad \text{noEvent}(T,t)}{[\mathscr{C}[\![l(\vec{v})]\!] :: R \triangleright C, T]^{I,O}_{p,t} \to [\mathscr{C}[\![\{\}]\!] :: R \triangleright C, T]^{I,O}_{p,t+1}} \quad \text{(R-NO-FUNCTION)}$$

$$\frac{T' = T \uplus (l(\vec{v}), v_1, t+v_2, t+v_1) \quad \text{noEvent}(T,t)}{[\mathscr{C}[\![l(\vec{v}) \textbf{ every } v_1 \textbf{ expire } v_2]\!] :: R \triangleright C, T]^{I,O}_{p,t} \to [\mathscr{C}[\![l(\vec{v})]\!] :: R \triangleright C, T']^{I,O}_{p,t+1}} \quad \text{(R-TIMER)}$$

$$\frac{t \leq v_2 \quad T' = T \uplus (l(\vec{v}), v_1, v_2, t+v_1)}{[R \triangleright C, T \uplus (l(\vec{v}), v_1, v_2, t)]^{I,O}_{p,t} \to [l(\vec{v}) :: R \triangleright C, T']^{I,O}_{p,t}} \quad \text{(R-TRIGGER)}$$

$$\frac{t > v_2}{[R \triangleright C, T \uplus (l(\vec{v}), v_1, v_2, t)]^{I,O}_{p,t} \to [R \triangleright C, T]^{I,O}_{p,t}} \quad \text{(R-EXPIRE)}$$

$$\frac{S \to S'}{S \,|\, S'' \to S' \,|\, S''} \qquad \frac{S_1 \equiv S_2 \qquad S_2 \to S_3 \qquad S_3 \equiv S_4}{S_1 \to S_4} \quad \text{(R-NETWORK, R-CONGR)}$$

Figure 2: Reduction semantics for sensors.

in the base calculus but that can easily be added for convenience.

Unlike the previous example, here every sensor will relay gather messages only after some internal processing, by its own version of the function.

Semantics. The calculus has two variable binders: the **let** and the function constructs, inducing the usual definition for free and bound names. We present the reduction relation with the help of a structural congruence, as it is usual [4]. Here, $[R \triangleright C, T]^{I,O}_{p,t} \equiv [R \triangleright C, T]^{I,O}_{p,t}\{\mathbf{0}\}$ is the only non-standard rule and provides a sensor with a conceptual *membrane* that engulfs neighboring sensors as they become engaged in communication. This prevents the reception of duplicate copies of the message during transmission. The reduction relation is inductively defined by the rules in Figures 2 and 3. Since processes evaluate to values, we allow for reduction within the **let** construct and therefore present the reduction relation using the following reduction contexts: $\mathscr{C}[\![\cdot]\!] ::= [\,] \mid \textbf{let } x = \mathscr{C}[\![\cdot]\!] \textbf{ in } P$. The reduction in a sensor is driven by the process at the front of the run-queue.

Within sensors reduction proceeds without obstacle while the internal clock t is not such that a timed call must be triggered. This is controlled by the predicate noEvent that checks the time of the next activation for every timed call against the current time. When a value of t is reached that implies the triggering of a call, the rules R-TRIGGER and R-EXPIRE come into action. Rule R-TRIGGER preempts

$$\frac{\text{inRange}(p,p') \qquad (I'',O'') = \text{networkRoute}(m,I',O')}{[R \triangleright C,T]^{I,m::O}_{p,t}\{S\} \mid [R' \triangleright C',T']^{I',O'}_{p',t'} \rightarrow [R \triangleright C,T]^{I,m::O}_{p,t}\{S \mid [R' \triangleright C',T']^{I'',O''}_{p',t'}\}} \quad \text{(R-BROADCAST)}$$

$$[R \triangleright C,T]^{I,m::O}_{p,t}\{S\} \rightarrow [R \triangleright C,T]^{I,O}_{p,t} \mid S \quad \text{(R-RELEASE)}$$

Figure 3: Reduction semantics for sensor networks.

the running process at the front of the run-queue and places a timed function call $l(\vec{v})$ to be executed. The execution of the call is delegated to rule R-CALL. Note that only calls to functions installed in C are allowed. Other calls are discarded by rule R-NO-FUNCTION. If the timer has expired, rule R-EXPIRE removes the corresponding tuple from T.

Network level reduction proceeds concurrently with in-sensor processing. It handles the distribution of messages placed by the sensors in their output queues O. Each time a new sensor is added to the membrane of a broadcasting sensor, a function networkRoute decides where the message in the O queue of the broadcasting sensor should be copied to in the new sensor. The function can be thought off as implementing the routing protocol for the sensor network. The message broadcast ends with the destruction of the membrane, the captive sensors becoming again free to engage in communication.

3 The Type System

The syntax for types τ include the types for built-in values β, the types for functions $\vec{\tau} \rightarrow \tau$, where $\vec{\tau}$ is the type of the parameters of the function and τ is its return type, and the types for code module $\{l_i\colon \vec{\tau}_i \rightarrow \tau_i\}_{i \in I}$ that is a record type gathering type information for each function of the code module.

Type judgments for values are of the form $\Gamma_S;\Gamma_C \vdash v\colon \tau$, where Γ_S and Γ_C are code module types representing the types for the built-in functions of the sensor and for functions installed in the sensor memory. The rules are standard so we omit them due to space constraints.

A partial list of the typing rules for processes and sensor networks is presented in Figure 4. The judgments for processes are the same as for values. Rule T-SYSTEM ensures that no user-defined function is executed as a system call and that a system call always belongs to a predefined set typed in Γ_s ($\Gamma_S \vdash l\colon \vec{\tau} \rightarrow \tau$). On the other hand, user-defined functions must be installed in the sensor memory (C) (Rule T-INSTAL). Notice that when installing a function l_i its type must be preserved ($\Gamma_C \vdash l_i\colon \vec{\tau}_i \rightarrow \tau_i$). Broadcasting a call (Rule T-SEND) is only possible for user-defined functions. Since the call is asynchronous the return type of the **send** process is the empty object (cf. the return call of a system call, which is synchronous). Firing an event (Rule T-TIMER) amounts to calling a user-defined function locally. The premises of Rules T-SEND and T-TIMER are the same, since the system does not distinguish local from remote functions. Such a refinement may be interesting and can be easy added.

Typing judgments for sensor networks are of the form $\Gamma_S;\Gamma_C \vdash S$. We only present the rule for typing a sensor (Rule T-SENSOR): the run-queue must be well typed, meaning each process in the run-queue is well typed; the set of installed functions must be well typed, as well as the built-in values t and p, the input and output queues I and O, and the event table T.

Typing the run-queue (rule omitted), the input and output queues, and the event table is equivalent to typing each element of the structure (Rules T-COMM-QUEUE and T-EVENT-QUEUE). Notice that each element of the input (output) queue is typable if it can be fired. The same holds for timed calls ($l(\vec{v})$).

From a preliminary analysis of the reduction relation and of the type system we expect, based on previous results, that subject reduction and type safety can be proved.

$$\frac{\Gamma_S \vdash l\colon \vec{\tau} \to \tau \qquad \Gamma_S;\Gamma_C \vdash \vec{v}\colon \vec{\tau}}{\Gamma_S;\Gamma_C \vdash \textbf{system}\ l(\vec{v})\colon \tau} \qquad \frac{\Gamma_S;\Gamma_C \vdash v\colon \{l_i\colon \vec{\tau}_i \to \tau_i\}_{i\in I} \qquad \forall i \in I.\Gamma_C \vdash l_i\colon \vec{\tau}_i \to \tau_i}{\Gamma_S;\Gamma_C \vdash \textbf{install}\ v\colon \{\}}$$

(T-SYSTEM, T-INSTAL)

$$\frac{\Gamma_C \vdash l\colon \vec{\tau} \to \{\} \qquad \Gamma_S;\Gamma_C \vdash \vec{v}\colon \vec{\tau}}{\Gamma_S;\Gamma_C \vdash \textbf{send}\ l(\vec{v})\colon \{\}} \qquad \frac{\Gamma_C \vdash l\colon \vec{\tau} \to \{\} \qquad \Gamma_S;\Gamma_C \vdash \vec{v}\colon \vec{\tau} \qquad \Gamma_S;\Gamma_C \vdash vv\colon \vec{\beta}}{\Gamma \vdash l(\vec{v})\ \textbf{every}\ v\ \textbf{expire}\ v\colon \{\}}$$

(T-SEND, T-TIMER)

$$\frac{\begin{array}{c}\Gamma_S;\Gamma_C \vdash R \qquad \Gamma_S;\Gamma_C \vdash C\colon \{l_i\colon \vec{\tau}_i \to \tau_i\}_{i\in I} \\ \forall i \in I.\Gamma_C \vdash l_i\colon \vec{\tau}_i \to \tau_i \qquad \Gamma_S;\Gamma_C \vdash tp\colon \vec{\beta} \\ \Gamma_S;\Gamma_C \vdash I \qquad \Gamma_S;\Gamma_C \vdash O \qquad \Gamma_S;\Gamma_C \vdash T\end{array}}{\Gamma_S;\Gamma_C \vdash [R \triangleright C, T]^{I,O}_{t,p}} \qquad \frac{\Gamma_S;\Gamma_C \vdash l(\vec{v})\colon \_ \qquad \Gamma_S;\Gamma_C \vdash I}{\Gamma_S;\Gamma_C \vdash \langle l(\vec{v})\rangle :: I}$$

(T-SENSOR, T-COMM-QUEUE)

$$\frac{\Gamma_S;\Gamma_C \vdash l(\vec{v})\colon \_ \qquad \Gamma_S;\Gamma_C \vdash T \qquad \Gamma_S;\Gamma_C \vdash v_1 v_2 t\colon \vec{\beta}}{\Gamma_S;\Gamma_C \vdash T \uplus (l(\vec{v}), v_1, v_2, t)}$$

(T-EVENT-QUEUE)

Figure 4: Typing rules for processes, sensor networks, and queues (partial list).

4 Discussion

Programming languages based on type safe specifications are fundamental for applications where development and debugging can be complex. Sensor networks are one such case. Difficulties in physically accessing deployed sensors, resource limitations of the devices and, dynamic ad-hoc routing protocols, all conspire to make the programming and debugging of these infra-structures a difficult task.

Type safe Callas will allow the seamless dynamic reprogramming of sensor networks, ensuring programmers that no run-time errors will be introduced. Dynamic reprogramming is supported directly in Callas, using its ability to ship code as arguments in messages and to install the code in remote sensors. This will make the programmer's task less daunting, when compared with most current languages and systems. Furthermore, the next step in this work will be to provide a virtual machine that can be proved sound relative to the Callas calculus and to use it as the specification for a basic run-time system. This approach solves portability problems between sensor platforms while also providing a semantically robust run-time.

Acknowledgements. The authors are partially supported by project CALLAS of the Fundação para a Ciência e Tecnologia (contract PTDC/EIA/71462/2006).

References

[1] B. Garbinato, H. Miranda, and L. Rodrigues, editors. *Middleware for Network Eccentric and Mobile Applications*, chapter 2. Springer-Verlag, 2009.

[2] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *European Conference on Object-Oriented Programming (ECOOP'91)*, number 512 in LNCS, pages 133–147. Springer-Verlag, 1991.

[3] L. Lopes, F. Martins, M. S. Silva, and João Barros. A Process Calculus Approach to Sensor Network Programming. In *International Conference on Sensor Technologies and Applications (SENSORCOMM '07)*, pages 451–456. IEEE Computer Society, 2007.

[4] R. Milner. A Calculus of Communicating Systems. Number 92 in LNCS. Springer-Verlag, 1980.

[5] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[6] R. Newton, Arvind, and M. Welsh. Building up to Macroprogramming: An Intermediate Language for Sensor Networks. In *Information Processing in Sensor Networks (IPSN'05)*, pages 37–44, 2005.

Programming Idioms for Transactional Events

Matthew Kehrt Laura Effinger-Dean Michael Schmitz Dan Grossman

University of Washington

{mkehrt,effinger,schmmd,djg}@cs.washington.edu

Abstract

Transactional events (TE) are an extension of Concurrent ML (CML), a programming model for synchronous message-passing. Prior work has focused on TE's formal semantics and its implementation. This paper considers programming idioms, particularly those that vary unexpectedly from the corresponding CML idioms. First, we solve a subtle problem with client-server protocols in TE. Second, we argue that CML's `wrap` and `guard` primitives do not translate well to TE, and we suggest useful workarounds. Finally, we discuss how to rewrite CML protocols that use abort actions.

1 Introduction

Transactional events (TE) provide powerful message-passing facilities for concurrent programming [1]. TE extends Concurrent ML (CML) [3] with a sequencing primitive `thenEvt`, which allows an arbitrary number of sends or receives per event. `thenEvt` is powerful enough that programmers can implement patterns such as n-way rendezvous (synchronizing multiple threads at once) or guarded receive (successfully synchronizing only if a received value satisfies a predicate).

In previous work [1, 2], we and other researchers explored the semantics and implementation of transactional events. However, no existing work discusses practical programming idioms for TE. We have found in the course of our research that some common CML idioms are surprisingly difficult to reproduce in TE. This paper presents our solutions to these problematic idioms, which include client-server protocols and programs using CML's `wrap`, `guard`, and `wrapAbort`.

TE is implemented for Haskell and Caml. In our examples, we use Caml, which is call-by-value.

2 Background

TE [1, 2] and CML [3] are closely related paradigms for concurrent programming that have been implemented for several languages. This section briefly reviews TE and explains how it differs from CML.

In CML, threads send values on channels. A channel of type `'a chan` carries values of type `'a`. Communication is *synchronous*: a send blocks until matched with a receive in another thread.

An *event* is a description of a communication to be performed. The functions `sendEvt` and `recvEvt` produce events that describe sends and receives, respectively. *Synchronizing* on an event with the function `sync` performs the event; `sync` has type `'a event -> 'a`. Events may be composed of other events. For example, the function `chooseEvt` constructs an event that, when synchronized on, performs exactly one of two events. The types of these and other functions appear in Figure 1.

TE extends CML with a new function `thenEvt`, which allows the sequencing of events. Synchronizing on `thenEvt ev f` does the following: (1) synchronize on `ev` to produce a result `v`; (2) apply `f` to `v` to produce a new event `ev2`; (3) synchronize on `ev2` to produce a final result. Critically, these three steps are all-or-nothing: if the second event cannot successfully synchronize, then the first event does not (appear to) happen. Therefore, we say that events built using `thenEvt` are *transactional*. A single event may communicate with multiple threads, so the success of one synchronization may entail the success of an arbitrary number of synchronizations in other threads.

Two more CML/TE functions are useful in combination with `chooseEvt` and `thenEvt`. `alwaysEvt` is the event that always succeeds: `sync (alwaysEvt x)` is equivalent to `x`. `neverEvt` is the event that never succeeds: `sync neverEvt` blocks forever.

The following example tries to do either a single send, or a receive followed by a send:

```
type 'a chan
type 'a event
val newChan : unit -> 'a chan
val sync : 'a event -> 'a
val sendEvt : 'a chan -> 'a -> unit event
val recvEvt : 'a chan -> 'a event
val chooseEvt : 'a event -> 'a event
                -> 'a event
val thenEvt : 'a event -> ('a -> 'b event)
              -> 'b event
val alwaysEvt : 'a -> 'a event
val neverEvt : 'a event
val guard : (unit -> 'a event)
            -> 'a event
val wrap : 'a event -> ('a -> 'b)
           -> 'b event
val wrapAbort : 'a event
                -> (unit -> unit)
                -> 'a event
```

Figure 1: Types for several key CML/TE functions.

```
sync (chooseEvt (sendEvt c1 5)
                (thenEvt (recvEvt c2) (fun x -> sendEvt c3 x)))
```

3 Server loops

One common concurrent programming idiom is a server thread that repeatedly handles requests from multiple clients. In CML, a server is often implemented as an infinite loop that calls `sync` at every iteration. In this section, we discuss why TE requires more sophisticated servers and how to implement them.

Consider the following function, which spawns a new thread to act as a server. The server sends increasing integers on a channel so that clients get a unique integer every time they receive on the channel.

```
let simpleIncrementServer () =
  let c = newChan () in
  let rec f y = sync (sendEvt c y); f (y + 1) in
  Thread.create f 0; c
```

A simple client receives on the server's channel.

```
let simpleIncrementClient c = sync (recvEvt c)
```

Both client and server are perfectly valid as both TE and CML. However, in TE `thenEvt` can be used to write other clients that interact with this server in unexpected ways. The following code receives two integers and adds them together in a single event:

```
let complexIncrementClient c =
  sync (thenEvt (recvEvt c) (fun x ->
        thenEvt (recvEvt c) (fun y ->
        alwaysEvt (x + y))))
```

If this client and the server were to synchronize on their events, neither would succeed. The client could receive one integer from the server. The server event would block waiting for the client event to complete, as they would be participating in the same transaction. Meanwhile, the client would block waiting for another integer. Both sides would be stuck, so this particular client cannot successfully synchronize with the simple increment server.

A similar problem arises when *two* client threads each receive an integer from the server and then communicate with each other in the same event. A transaction consisting only of these two events and

the server event would not succeed. The server would need to send to both threads, but the server's event does a single send.

To solve this problem, we want a server that can perform multiple sends in a single synchronization. In the TE code below, the server synchronizes on a single event that can perform an arbitrary number of sends. After each send, the event chooses between either returning the sent value or recursively calling the server function.

```
let betterIncrementServer () =
  let c = newChan () in
  let rec evtLoop x =
    thenEvt (sendEvt c x)
            (fun _ -> chooseEvt (alwaysEvt (x + 1)) (evtLoop (x + 1))) in
  let rec serverLoop x = serverLoop (sync (evtLoop x)) in
  Thread.create serverLoop 0; c
```

However, this problem can occur with many different server and client combinations. A better solution would be to create a generic combinator for an event that is repeated one or more times.

For this purpose, we define a function, `serverLoop`, suitable for creating servers. It takes two arguments and loops forever. The first argument is a pair, `(ev, b)`, of an event and any value. The second argument is a function, `f`. After synchronizing on `ev` to produce a value `a`, `serverLoop` calls `f` on the pair `(a,b)` to produce a new `(ev,b)` pair, with which it recurs. `b` acts as a loop-carried state for `serverLoop`. Programmers can use `serverLoop` much like they use `fold` to process lists.

Internally, `serverLoop` uses a second function, `evtLoop`, that creates an event that synchronizes on `ev` and then nondeterministically chooses between returning or recurring with the result of calling `f` to produce a new event and loop-carried state. In other words, `evtLoop` does exactly what `serverLoop` does but, crucially, in a single transaction. Overall, `serverLoop` sequentially synchronizes on the events computed by `f`, but transactions may end (starting the next transaction) at any point in the sequence. Thus, clients can communicate with the server any number of times in one synchronization.

```
(* evtLoop :
     ('a event * 'b) -> (('a * 'b) -> ('a event * 'b)) -> ('a * 'b) event *)
let rec evtLoop (ev, b) f =
  thenEvt ev (fun a -> chooseEvt (alwaysEvt (a, b)) (evtLoop (f (a, b)) f))

(* serverLoop : ('a event * 'b) -> (('a * 'b) -> ('a event * 'b)) -> 'c *)
let rec serverLoop (ev, b) f = serverLoop (f (sync (evtLoop (ev, b) f))) f
```

Using `serverLoop` to construct an increment server is straightforward. We spawn a new thread to run `serverLoop` called with (1) an initial event paired with the initial loop-carried counter and (2) a function to construct the next event and counter by incrementing the counter and creating the next send event.

```
let incrementServer () =
  let c = newChan () in
  Thread.create (serverLoop ((sendEvt c 0), 0))
                            (fun (_, x) -> (sendEvt c (x+1), x+1)); c
```

4 `wrap` and `guard`

`wrap` and `guard` (see Figure 1) add post- and pre-processing, respectively, to CML events. `sync (wrap ev f)` synchronizes on `ev`, then applies `f` to the result. `sync (guard g)` synchronizes on the result of `g ()`. In this section, we discuss how to adapt programs that use these functions to TE.

The following code uses `wrap` to perform two receives in either order:

```
sync (chooseEvt (wrap (recvEvt c1) (fun x -> (x, sync (recvEvt c2))))
                (wrap (recvEvt c2) (fun x -> (sync (recvEvt c1), x))))
```

Suppose we were to rewrite this code using `thenEvt`:

```
sync (chooseEvt
  (thenEvt (recvEvt c1)
           (fun x -> thenEvt (recvEvt c2) (fun y -> alwaysEvt (x,y))))
  (thenEvt (recvEvt c2)
           (fun y -> thenEvt (recvEvt c1) (fun x -> alwaysEvt (x,y)))))
```

These two versions actually behave differently: the CML version performs the receives in separate synchronizations, while the TE version executes both in the *same* synchronization. Therefore the above TE code could not communicate successfully with code that performed two synchronizations, such as `sync (sendEvt c1 4); sync (sendEvt c2 5)`.

We can mimic `wrap`'s behavior in TE by thunking the second receive and executing the thunk after the first `sync` completes:

```
(sync (chooseEvt
  (thenEvt (recvEvt c1)
           (fun x -> alwaysEvt (fun () -> (x, sync (recvEvt c2)))))
  (thenEvt (recvEvt c2)
           (fun x -> alwaysEvt (fun () -> (sync (recvEvt c1), x))))) ()
```

With the use of two helper functions, the TE code approaches the elegance of the original CML code:

```
(* thunkWrap : 'a event -> ('a -> 'b event) -> (unit -> 'b event) *)
let thunkWrap ev f = thenEvt ev (fun x -> alwaysEvt (fun () -> f x))

(* syncThunked : (unit -> 'a) event -> 'a *)
let syncThunked ev = (sync ev) ()

let _ = syncThunked
  (chooseEvt (thunkWrap (recvEvt c1) (fun x -> (x, sync (recvEvt c2))))
             (thunkWrap (recvEvt c2) (fun x -> (sync (recvEvt c1), x))))
```

We have sacrificed some composability: `thunkWrap` returns a `(unit -> 'b) event` instead of a `'b event`, so it is more difficult than `wrap` to combine with other events. The semantics of `wrap` (processing an event's result after synchronization completes) and `thenEvt` (combining two synchronizations into one) seem to be incompatible, but we believe that thunked wrappers are a good compromise. Wrapping an already wrapped event does not require a second level of thunk, as this helper function demonstrates:

```
(* rewrap : (unit -> 'a) event -> ('a -> 'b) -> (unit -> 'b) event *)
let rewrap ev g =
  thenEvt ev (fun f -> let x = f () in alwaysEvt (fun () -> g x))
```

CML's `guard` is useful for encapsulating actions that need to happen prior to synchronization. For example, the following code adds a timeout to an event by spawning a thread to signal when to give up:

```
(* timeoutEvt : 'a event -> float -> 'a event *)
let timeoutEvt ev time = guard (fun () ->
  let timeoutChan = newChan () in
  Thread.create
    (fun () -> Thread.delay time; sync (sendEvt timeoutChan ())) ();
  chooseEvt ev (wrap (recvEvt timeoutChan) (fun () -> raise TimedOutExn)))
```

As with `wrap`, it is difficult to add `guard` to TE's interface because the guard function needs to execute outside of the synchronization. However, we can code up `timeoutEvt` in TE without `guard`:

```
let timeoutEvt ev time = chooseEvt ev
  (thenEvt (alwaysEvt ()) (fun () -> Thread.delay time; raise TimedOutExn))
```

Moreover, this function is more readable than the CML code. In the next section, we will see another example for which the TE implementation is more natural than the original CML program.

5 `wrapAbort` and abort actions

The `wrapAbort` function (see Figure 1) lets CML programs specify an action to take if an event is part of a `chooseEvt` that is synchronized on and another choice is taken. For example, `sync (chooseEvt (wrapAbort ev1 f) ev2)` will block until either `ev1` or `ev2` succeeds, and in the latter case it will execute `f ()`. `wrapAbort` is useful for client-server protocols that have more than one communication. The CML book [3] uses `wrapAbort` to implement mutual-exclusion locks with this interface[1]:

```
type lockServer
val acquireLockEvt : lockServer -> int -> unit event
val releaseLockEvt : lockServer -> int -> unit event
val mkLockServer : unit -> lockServer
```

Clients of this interface acquire or release locks (represented by `int`s) by synchronizing on events created with `acquireLockEvt` and `releaseLockEvt`. Synchronizing on `acquireLockEvt s i` blocks until the acquire succeeds; clients may abort acquires, perhaps with the `timeoutEvt` function from Section 4:

```
sync (timeoutEvt (acquireLockEvt server 1) 5.0)
```

Implementing `acquireLockEvt` with a single server thread in CML requires two communications: a request from the client with the lock ID, and a confirmation from the server when the lock is available. Abort actions are essential for implementing `acquireLockEvt`, as the first communication may affect the server's internal state by adding the request to a queue. If the client aborts after the first communication but before the second, it must tell the server to remove the request from the queue. Otherwise, the server would hang when trying to confirm the lock acquire. The form of `acquireLockEvt` is essentially:

```
let acquireLockEvt s id =
  guard (fun () -> (* send acquire request *);
                   wrapAbort (* receive acquire confirmation *)
                             (fun () -> (* do cancellation *)))
```

[1] It actually uses the equally expressive `withNack`; we discuss `wrapAbort` because we find it more intuitive.

The abort action effectively makes the two communications *transactional*: if the second communication aborts, the effects of the first communication are canceled. In TE, we can implement a lock server without an abort action. Our solution uses `thenEvt` and `neverEvt` and is similar to the guarded-receive pattern [1]. If the server receives a request for an unavailable lock (the server maintains a list of held locks), it returns `neverEvt` inside `thenEvt`. `neverEvt` never succeeds, so the program behaves as if the request did not yet occur, exploiting the transactional semantics of `thenEvt`. A full implementation is below[2]; we expect other CML protocols that use abort actions will adapt similarly to TE.

```
type request = Acquire of int | Release of int
type lockServer = request chan
let acquireLockEvt reqCh id = sendEvt reqCh (Acquire id)
let releaseLockEvt reqCh id = sendEvt reqCh (Release id)
let mkLockServer () =
  let reqCh = newChan () in
  let serverEvt heldLocks =
    thenEvt (recvEvt reqCh) (function
      | Acquire id -> if List.exists (fun id2 -> id = id2) heldLocks
                      then neverEvt
                      else alwaysEvt (id::heldLocks)
      | Release id -> alwaysEvt (List.filter (fun id2 -> id <> id2) heldLocks))
  in Thread.create (serverLoop (serverEvt [], ()))
       (fun (heldLocks, ()) -> (serverEvt heldLocks, ()));
     reqCh
```

6 Conclusion

Every programming model needs three things: a semantics, an implementation, and useful idioms. Reppy's dissertation on CML [3] presents all three to demonstrate that CML is a useful programming model. Prior TE research [1, 2] has concentrated on semantics and implementation. We have presented several important TE programming idioms, the subtleties of which surprised us during our research. First, writing client-server protocols in TE requires careful consideration of how `sync` interacts with `thenEvt`; our `serverLoop` function is a general solution to this problem. Second, `wrap` and `guard` are difficult to integrate with TE, and we have suggested alternatives that preserve most of the original semantics. Finally, we have discussed how protocols with abort actions may be rewritten with `thenEvt` and `neverEvt`.

References

[1] Kevin Donnelly and Matthew Fluet. Transactional events. In *11th ACM International Conference on Functional Programming*, 2006.

[2] Laura Effinger-Dean, Matthew Kehrt, and Dan Grossman. Transactional events for ML. In *13th ACM International Conference on Functional Programming*, 2008.

[3] John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[2] As an orthogonal issue, we use `serverLoop` from Section 3 to support multiple lock operations in one synchronization.

Softly safely spoken: Role playing for Session Types

Elena Giachino*
Dipartimento di Informatica
Università degli Studi di Torino
Torino, Italy
PPS - Université Paris Diderot
Paris, France
giachino@di.unito.it

Matthew Sackman, Sophia Drossopoulou, Susan Eisenbach,
Department of Computing
Imperial College
London, England
ms@doc.ic.ac.uk, sd@doc.ic.ac.uk, sue@doc.ic.ac.uk

Abstract

Session types have made much progress at permitting programs be statically verified concordant with a specified protocol. However, it is difficult to build abstractions of, or encapsulate Session types, thus limiting their flexibility. Global session types add further constraints to communication, by permitting the order of exchanges amongst many participants to be specified. The cost is that the number of participants is statically fixed.

We introduce Roles in which, similarly to global session types, the number of roles and the conversations involving roles are statically known, but participants can dynamically join and leave roles and the number of participants within a role is not statically known. Statically defined roles which conform to a specified conversation can be dynamically instantiated, participants can be members of multiple roles simultaneously and can participate in multiple conversations concurrently.

1 Introduction

Communication is the single most important faculty of computing beyond the definition of computation itself. Of course, communication is built into computation itself, be it defined by the Turing Machine or the λ-calculus, by the ability to pass names, values or state between instructions. Increasingly, as programs are targeting distributed and highly parallel environments, safe communication between such programs is of the utmost importance and subject to a great deal of study.

Session types [8, 14] have provided a means to define protocols, to parametrise communication channels by such protocols, and to statically enforce that use of those communication channels matches the protocol specified. Thus upon any given dyadic communication channel, the two participants will not both be blocked, each waiting for the other to speak; and a value sent as one type will be received as the same type. Session types cater for branching and looping control-flow structures, and there have been several implementations of session types in a number of different languages [11, 13, 6, 7, 12, 10].

A language that supports session types can allow many channels to be defined and used at the same time, and enforces that communication actions upon those channels obey the requirements of their session types. However, the session types only specify the behaviour of a single channel at a time, and so it is difficult to reason about or understand the communication within a larger system, with many channels in use often reusing the same session type. Global session types [4, 9] solve this problem by allowing the protocol to be specified not in terms of channels, but in terms of participants, the types of values to be sent between participants and the order in which individual communication actions are to occur. Thus it is very easy to see the communication of the whole system. The global session type is then projected to a series of individual session types for the channels that each participant needs to use.

The disadvantage to this is that the global session type now specifies not only the communication, but also the number of participants involved. Without global session types, new participants can be created dynamically, their names can be communicated amongst existing participants, and new channels can be dynamically created to include the new-comers into the existing conversation. We feel that this is a valuable property that caters for a vast number of important use-cases.

*Partially supported by the Fondation Sciences Mathématiques de Paris.

One further limitation of both session types and global session types is that it is not easy to encode broadcast semantics. One-to-many and many-to-one (and many-to-many) are all possible (though only through specifying multiple consecutive send and receive actions), but only when the number of participants on each side is known. In the case of global session types, this must be known statically, but even without global session types, polling a large number of channels is not a well supported feature, and it is hard, if not impossible, to permit new channels to be created and received upon during an on-going multi-receive, or to release channels to participants that have died. Very few implementations of session types even cater for receiving on a set of channels ([11] is one of the few that does).

We introduce a language with support for Roles. Roles can contain an arbitrary number of participants, and participants are free to join and leave Roles at any time, subject to type-safety constraints. Communication is then specified by a `conversation`, which defines names for roles, names for channels between roles, and gives the dyadic session types for those channels. Because channels are between roles, and not between individual participants, communication is by definition broadcast, with many-to-many semantics. This design restores some of the dynamic flexibility of session types and allows us to use Roles naturally in a number of useful examples, but also retains the high-level declaratory communication plan of global session types.

2 The Auction

The design of our language has been motivated by a number of use-cases, one of which is an *ebay*-style auction. There are three distinct sorts of participants (or *roles*) in such an auction: the auctioneer, the bidders and the audience. As is the case with a real *ebay* auction, bidders can dynamically join and leave the auction after the start and before the end. At any point during the auction, a member of the audience may become a bidder by placing a bid, and bidders who are no longer interested in making any further bids may become members of the audience. Our language does not place any constraints on the number of participants within a role; this is an important feature for being able to model this auction without having to statically prescribe the number of bidders.

The auction takes place in a room in which everyone bar the auctioneer is a member of the audience. The auctioneer names the item, and states the reserve price. After this, bidders may call out their bids (implicitly becoming bidders rather than plain members of the audience). The bidders can't see who else is bidding, nor hear each other's bids, but they can hear the auctioneer announce what the current leading bid is and by whom, whenever a new high bid is received. Like *ebay*, the auction is timed so bids can received up until the time at which the auction ends.

$$
\begin{array}{l}
\texttt{conversation}\ \mathit{Auction}\ \{ \\
\qquad \texttt{role}\ \mathit{Auctioneer} \\
\qquad \texttt{role}\ \mathit{Audience} \\
\qquad \texttt{role}\ \mathit{Bidders} \\
\qquad \texttt{channel}\ \mathit{k1}\ \mathit{k1'}\ (\mathit{Auctioneer}, \mathit{Audience}) : !\mathit{Item}.!\mathit{Price}.\mu T.!(\mathit{Price}, \mathit{Bool}, \mathit{Pid}).T \\
\qquad \texttt{channel}\ \mathit{k2}\ \mathit{k2'}\ (\mathit{Auctioneer}, \mathit{Bidders}) : \mu\, T.?\mathit{Price}.T \\
\qquad \}
\end{array}
$$

First we define the *conversation*. The conversation exists to name a specification of roles and channels. When a conversation is instantiated, all the roles within it are created (initially without participants), as are the channels between the roles. A conversation is itself a type, and similar to a class in an object-oriented sense.

new Auction(a);

```
join (a.Audience);
a.k1′ ? (i : Item) [ ];
μ X.a.k1′ ? (current : Price, done : Bool,
              leader : Pid) [ ];
  μ Y.if done
       then { }
       else {if interested (i, current)
               then {join (a.Bidders);
                     a.k2′ ! (myNextBid (i, current),
                              self ());
                     leave (a.Bidders);
                    }
               else { }
             a.k1′ ? (current : Price, done : Bool,
                      leader : Pid) [ ];
             if done then { }
               else if (leader ≡ self ())
                       then X;
                       else Y;
            }
```

```
join (a.Auctioneer);
currentPrice = reservePrice;
leader = null;
a.k1 ! (Item);
a.k1 ! (currentPrice, False, leader);
μ X.if isFinished ()
       then {a.k1 ! (currentPrice, True,
                     leader); }
       else {a.k2 ? (bid : Price, bidder : Pid)
               [gotBid (bid, bidder)];
               gotBid (bid, bidder);
             X; }
function gotBid (bid, bidder){
  if bid > currentPrice
    then {a.k1 ! (bid, isFinished (),
                  bidder);
          currentPrice = bid;
          leader = bidder; }
    else { }
}
```

Figure 1: Example programs for the bidders (left) and auctioneer (right)

The conversation declares its name (*Auction*), the names of the three roles within each *Auction* conversation (i.e. *Auctioneer*, *Audience* and *Bidders*) and creates two channels. One channel is between the Auctioneer and the Audience and the other channel is between the Auctioneer and the Bidders. The channel between the *Auctioneer* and *Audience* is called *k1* when it is being used by the *Auctioneer* and is called $k1'$ when it is being used by the *Audience*. That is, channel endpoints have distinct names. Also specified with each channel is its session type. The session type is given for the left endpoint (e.g. *k1* and *k2*), and the dual is calculated in the normal way for the other channel end point (e.g. $k1'$ and $k2'$). The session types for channels are as standard, with send, receive and recursion (defined using μ as standard, and thus must be contractive), but we omit branching intentionally as it can be simulated through `join`() and `leave`(), and because it is unclear what the desired behaviour should be when multiple participants within the same role select different branches within the same offer action.

All audience members share the same behaviour: they can each choose whether or not to bid and how much to bid, but the communication actions must all conform to the session types declared with the channels. Members of the audience will receive the item name, and will then repeatedly receive a tuple, containing the new leading bid, a boolean indicating whether the auction has finished yet, and the name of the leading bidder. From the bidders, the auctioneer will repeatedly receive new bids.

Example programs are shown for the bidders and auctioneer in figure 1 in a π-calculus style syntax. The bidder starts by joining the audience. This allows them to hear the auctioneer announcing the item and the reserve price. Whilst the auction is still in progress, each bidder decides for themselves whether they are interested in making a bid on the current item (*i*) given its current price (*current*). If so, they will join the *Bidders* role and send their bid. They then leave the *Bidders* role and wait to see what happens. Eventually, the auctioneer will announce the new leading bid. If the new leading bid is the bid that the bidder just placed then the bidder returns to waiting for the next bid to be made (or exits if the time limit

is up).

All participants within a role must obey the session types of the role's channels. If the session type of a channel indicates the next action is to send a value on that channel then *all* the participants within that role must send upon that channel. Consequently, when receiving, the receive action may receive only a single value (if there was only one participant in the role containing the dual endpoint of the channel), or multiple values. We did not wish to specify sets or lists of values within our language, so instead, our receive construct has two continuations, the first of which (indicated in square brackets) is invoked on all but the last value of the receive action, whilst the second continuation is invoked on the very last value. This allows the continuations to be invoked as soon as possible and prevents having to wait for all the participants within the sending role to send their messages.

For the auctioneer, the same continuation is specified in both cases (i.e. many bidders may be within the *Bidders* role at the same time and so are placing bids as part of the same send action): the bid that is received is checked to see if it is higher than the current price, and if so, the bid is announced. Eventually, the auctioneer will notice that the auction has expired, will reannounce the winning bid, and will exit.

3 The language

The auction example in the previous section makes use of all the syntax of our language. The metavariable `C` ranges over conversation names, `r` ranges over role names, `k` ranges over channel names and `y` ranges over conversation variables. We write $\overline{\texttt{Rdec}}$ as a shorthand for $\texttt{Rdec}_1 \ldots \texttt{Rdec}_n$ (and similarly for `Kdec`). We abbreviate $\texttt{k}_1 : \texttt{T}_1, \ldots, \texttt{k}_n : \texttt{T}_n$ by writing $\overline{\texttt{k}} : \overline{\texttt{T}}$.

Cdec	::=	conversation C $\{\overline{\texttt{Rdec}}, \overline{\texttt{Kdec}}\}$	Conversation declaration
Rdec	::=	role r	Role declaration
Kdec	::=	channel k k′ (r,r′):T	Channel declaration
P	::=	new C(y); P	New conversation
	|	join(y.r); P	Join
	|	leave(y.r); P	Leave
	|	y.k!(e); P	Send
	|	y.k?(x)[P]P	Receive
	|	P‖P	Parallel composition
	|	μX.P	Recursion
	|	X	Process variable
	|	**0**	Inaction
e	::=	x | y | v | ...	Expression

The declaration conversation C $\{\overline{\texttt{Rdec}}, \overline{\texttt{Kdec}}\}$ defines a conversation of name C and contains the declarations of the roles that take part in the conversation and the declarations of the channels used by the roles within the conversation to communicate. A role declaration of the form role r defines a role of name r. A channel declaration of the form channel k k′ (r,r′):T specifies that the channel is shared by the roles r (with local name k) and r′ (with local name k'). T will be the session type of k at r and the dual of T will be the type of k′ at r′. The session type T must be contractive and is defined by the grammar T ::=!(t).T |?(t).T | end | μY.T | Y.T where Y ranges over session type recursion variables. The dual of a session type is obtained by syntactically substituting ! with ? in the standard way.

Joining a role enables a process to use the channels that are connected to the role as specified by the conversation. We wish to ensure that all participants within a role receive the same set of messages within each receive action. This means that on the sending side, it must not be possible for a participant to join a role and then send a message as part of a send action for which other participants within that role

have already sent. If this were permitted then participants receiving these messages would potentially receive different messages depending on when they see what they each believe is the last message within the receive action. As such, when joining a role, the participant may only send messages as part of send actions which have yet to be started by any participant within the role.

Similarly, to avoid having to preserve message queues indefinitely, when joining a role, the participant may only participate in receive actions that are either yet to start, or have started but have not yet been fully received by all participants within the role. The operational semantics for join uses the inferred types of the channels within the role as used by the process, ensures these are reductions of the declared types of the roles as defined by the conversation, and only reduces when the runtime types of the channels matches the inferred used of the channels.

The process $\texttt{new}\,\texttt{C}(\texttt{y});\texttt{P}$ creates a new instance of the conversation C and continues as the process P in which the local conversation variable y is substituted by the new instance. The process $\texttt{join}(\texttt{y.r});\texttt{P}$ joins the role r within the conversation y. The process $\texttt{leave}(\texttt{y.r});\texttt{P}$ leaves the role r within the conversation y, loses the right to access the channels of y and then continues as P. Sending and receiving are the only two actions that are specified in the session type of a channel.

Sending is asynchronous, and the receive will start as soon as any of the values to be sent as part of the sending action are received. The process $\texttt{y.k!}(\texttt{e});\texttt{P}$ sends the result of the evaluation of e on the channel k of the conversation y and continues as P. The process $\texttt{y.k?}(\texttt{x})[\texttt{P}']\texttt{P}$ receives a value from all the processes in the role of the dual of the channel k, performing $\texttt{P}'$ on all but the last value, and continuing with P on the final value to be received. The parallel composition, recursion and inaction are standard. An expression e could be a value, a variable or other expression coming from the host language.

4 Related work and Conclusions

Session types were first presented by Honda, Vasconcelos and Kubo [8] and have received a great deal of subsequent study and refinement [12, 2, 6, 14, 13, 7, 4, 5, 10].

In these subsequent papers, constructs to support sessions were added to different several languages, and despite session types being able to guarantee many desirable properties (in some cases including progress, see [6, 5] among others), those languages were not able to represent more complex models of interaction at a sufficiently high level to permit an intuitive overview of the communication patterns. To overcome this limitation, multiparty sessions [9, 1, 3] have been proposed.

The existing work on multiparty sessions was the inspiration for us to model interactions between roles instead of processes, with the idea to describe those scenarios in which many participants follow the same communication behaviour (e.g. the bidders in the auction). We believe that our conversations (in the spirit of [3, 9]) make it easy to reason about the overall communication among roles.

In order to maximise potential concurrency, we invested much effort to ensure that processes within a role do not have to operate within lockstep but can each manipulate channels freely at their own rate, provided they do not violate the constraints placed upon their actions by the session types of the channels. This increases the flexibility of the semantics and allows greater levels of parallelism to be exploited. The trade off is that joining and leaving a role dynamically becomes more tricky to define safely.

The flexibility afforded by participants being able to join and leave roles has allowed us to simplify the standard session type semantics. We no longer explicitly cater for delegation because it can be simulated by a participant leaving a role and then the target participant joining the same role, thus gaining access to the role's channels. Similarly, branching can be encoded through changing roles, although we are continuing to explore including branching directly within our language. We are currently in the process of developing a type system to enfore the safety requirements of our language and hope to present this in future work.

Acknowledgements We thank Professor Mariangiola Dezani for making this work possible.

References

[1] Eduardo Bonelli and Adriana Compagnoni. Multipoint session types for a distributed calculus. In *TGC'07*, volume 4912 of *LNCS*, pages 240–256. Springer, 2007.

[2] Eduardo Bonelli, Adriana Compagnoni, and Elsa Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *J. Funct. Progr.*, 15(2):219–248, 2005.

[3] Luis Caires and Hugo Torres Vieira. Conversation Types. In *ESOP'09*, 2009. To appear.

[4] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[5] Mariangiola Dezani-Ciancaglini, Ugo de' Liguoro, and Nobuko Yoshida. On progress for structured communications. In *TGC'07*, volume 4912 of *LNCS*, pages 257–275. Springer, 2008.

[6] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.

[7] Pablo Garralda, Adriana Compagnoni, and Mariangiola Dezani-Ciancaglini. Bass: Boxed ambients with safe sessions. In *PPDP'06*, pages 61–72. ACM Press, 2006.

[8] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*. Springer, 1998.

[9] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284, 2008.

[10] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP 2008*, pages 516–541. Springer, 2008.

[11] Matthew Sackman and Susan Eisenbach. Session Types in Haskell: Updating Message Passing for the 21st Century. Technical report, June 2008.

[12] Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the behavior of objects and components using session types. In *FOCLASA'02*, volume 68(3) of *ENTCS*, pages 439–456. Elsevier, 2002.

[13] Vasco T. Vasconcelos, Simon Gay, and António Ravara. Typechecking a multithreaded functional language with session types. *Theorical Computer Science*, 368(1-2):64–87, 2006.

[14] Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type disciplines for structured communication-based programming revisited. In *SecRet'06*, volume 171 of *ENTCS*, pages 73–93. Elsevier, 2007.