A Closer Look at Constraints as Processes

N. Raja [*] R.K. Shyamasundar [†]

School of Technology & Computer Science
Tata Institute of Fundamental Research
Mumbai 400 005, India

Keywords: Programming calculi; Formal semantics; Program correctness; Concurrency; Compositional encoding.

1 Introduction

Every major paradigm of programming is built around a few key design concepts and principles. In order to study and analyse these concepts, it is essential to embody these conceptual mechanisms in a minimal core calculus. Such a core calculus helps in abstracting away from the syntax of particular programming languages which belong to that paradigm, thereby making it possible to analyse the paradigm in a syntax independent manner [12]. A well known instance of such a scenario is provided by the inter relationship between the paradigm of functional programming and the λ-calculus [2].

A major enterprise in the conceptual analysis of various paradigms, is the analysis of the expressive power of seemingly disparate paradigms of programming, in order to discover underlying similarities between them, and to carry over successful analysis techniques across paradigms. Once again, the availability of minimal core calculi helps in achieving this step, by providing a framework for the development of a semantic foundation for one paradigm in terms of another.

The γ-calculus [15] and the π-calculus [8] may be considered as representing minimal calculi which capture the core features of two distinct paradigms. The γ-calculus [15] embodies higher-order concurrent constraint programming [16], an instance of which is the Oz programming language

[*]Email address: raja@tifr.res.in; WWW: http://www.tcs.tifr.res.in/~raja
[†]Email: shyam@tcs.tifr.res.in; URL: http://www.tcs.tifr.res.in/~shyam

[17]. On the other hand the π-calculus [8] embodies interactive concurrent programming [7], an instance of which is the PICT [13] programming language.

The problem of relating the paradigm of higher-order concurrent constraint programming to the paradigm of interactive concurrent programming is an important one. It can be solved by relating the γ-calculus to the π-calculus. A solution to this problem has been attempted before by the paper titled *Constraints as Processes* [18], which tried to present a semantic preserving embedding from the γ-calculus to the π-calculus, along with a proof of correctness of the embedding.

However, we demonstrate that the embedding proposed by the above paper is incorrect. We construct a concrete counterexample, which involves two γ-terms that are behaviorally equivalent, but their corresponding π-terms that arise from the embedding are not equivalent, thus violating the correctness criterion of the embedding. We trace the source of error in the mapping proposed by the above paper to the violation of one of the basic principles of concurrent programming, namely the non-preservation of atomic transactions.

The rest of this paper is organized as follows: Section 2 presents a counterexample which shows that the correctness criterion is violated by the embedding; Section 3 traces the root cause of this error to the incorrect handling of atomic transactions by the embedding; Section 4 reviews related work; and finally Section 5 concludes the paper.

2 Logic variables and their concurrent updates

The γ-calculus contains the notion of logic variables, and uses elimination as a form of constraint solving. The semantics of the γ-calculus is defined by a structural congruence and a reduction relation [15]. An example of a reduction in γ-calculus is:

Example 2.1 (Reduction in γ-calculus)

$$\exists x \ (x = a \ \wedge \ E) \quad \longrightarrow \quad E\{a/x\} \qquad \text{if } x \neq a \text{ and } a \text{ free for } x \text{ in } E$$

where $\exists x \ (x = a \ \wedge \ E)$ is a γ-term, which comprises the conjunction of a subterm E with the “fact” that x is a. The reduction results in $E\{a/x\}$, which is E where the logic variable x is “eliminated” and replaced by a.

The π-calculus contains the notion of agents interacting by sending and receiving data over named ports, where the data communicated are again ports. The semantics of the π-calculus is also defined by a structural congruence and a reduction relation [7]. An example of a reduction in π-calculus is:

Example 2.2 (Reduction in π-calculus)

$$b(x).P \mid \bar{b} < a > .Q \quad \longrightarrow \quad P\{a/x\} \mid Q$$

where $b(x).P$ is an agent which receives something for x along b and continues as the subagent P, while $\bar{b} < a > .Q$ is another agent which transmits a along b and continues as the subagent Q. The agents interact when they are combined using the parallel operator "|", and reduce to $P\{a/x\} \mid Q$, where the bound occurrences of x in P are replaced by a.

In [18], a compositional encoding is presented from the γ-calculus [15] to the π-calculus [7, 8], along with a correctness proof. We do not repeat the complete definition of the encoding here. For details of the encoding, we refer the reader to the paper [18]. However, we provide those parts of the encoding that are relevant for the purposes of this work.

The correctness criterion of the encoding is formalized in the above paper by requiring the following property: If two γ-calculus terms are behaviorally equivalent, then the encodings of these terms into the π-calculus are also equivalent, and vice versa. In this section, we show that the encoding provided in the above paper does not satisfy the stated correctness criterion. In particular, we shall present a counterexample which involves two γ-terms G and G' (say) which are behaviorally equivalent, but their encodings into the π-calculus represented by P and P' (say) respectively are not equivalent, thus contradicting the correctness result.

We construct the following scenario involving two distinct logic variables trying to update to each other concurrently. Let x and z be two logic variables of γ-calculus. The terms represented by the equations $\exists x(x = z)$ and $\exists x(x = z \wedge z = x)$ are behaviorally equivalent in the γ-calculus. However the encoding leads to two π-terms which are not equivalent in the π-calculus.

The γ-calculus has three kinds of basic entities, *names*, *variables*, and *terms*. Names and variables are also called references. Our focus will be on the encoding of variables, and on update operations involving them. The encoding maps each logic variable of γ-calculus to a corresponding π-calculus process term, which is called the *handler* agent for that variable. For instance, the logic variable x of γ-calculus is mapped to the handler agent $V(\underline{x})$, which is a process term of π-calculus. $V(\underline{x})$ is defined as:

$$
\begin{array}{lll}
V(\underline{x}) & \overset{def}{=} & \quad x \triangleright \texttt{value}(c, \_).\overline{c} < \underline{x} > .V(\underline{x}) \\
 & & + \; x \triangleright \texttt{update}(\underline{u}). \\
 & & \qquad [x = u](V(\underline{x}), \overline{u \triangleright \texttt{value}} < \nu c, x > .c(\underline{u}). \\
 & & \qquad\qquad\qquad [x = u](V(\underline{x}), R(\underline{x}, \underline{u}))) \\
 & & + \; x \triangleright \texttt{eq}(\underline{u}, y, n). \\
 & & \qquad (\; V(\underline{x}) \\
 & & \qquad \mid \overline{u \triangleright \texttt{value}} < \nu c, x > .c(\underline{u}).[x = u](\overline{y}, \overline{x \triangleright \texttt{eq}} < \underline{u}, y, n >))
\end{array}
$$

where,

$$
\begin{array}{lll}
R(\underline{x}, \underline{u}) \equiv & & !u \triangleright \texttt{name}(\underline{a}).\overline{x \triangleright \texttt{name}} < \underline{a} > \\
 & \mid & !x \triangleright \texttt{value}(c, v).[u = v](\overline{c} < \underline{u} >, \overline{u \triangleright \texttt{value}} < c, v >) \\
 & \mid & !x \triangleright \texttt{update}(\underline{v}).\overline{u \triangleright \texttt{update}} < \underline{v} > \\
 & \mid & !x \triangleright \texttt{eq}(\underline{v}, y, n).\overline{u \triangleright \texttt{eq}} < \underline{v}, y, n >
\end{array}
$$

The handler agent interacts with the environment to perform operations involving the corresponding logic variable. In particular it also accomplishes update operations on the logic variable through the interaction "$x \triangleright$ $\texttt{update}(\underline{u})$" which tells x to update to the reference u. The γ-calculus equation "$x = z$" imposes the equality of x and z on all terms. The encoding of such a constraint is given by

$$[\![x = z]\!] \;=\; \overline{x \triangleright \texttt{update}} < \underline{z} >$$

A variable x which has been updated to a variable z is handled by $R(\underline{x}, \underline{z})$. In the encoding, equivalence classes of logic variables are represented in the form of trees, constructed using π-calculus processes. The encoding imposes certain conditions in order to ensure that the tree representations do not become circular. In particular when a logic variable (say x) is processing an '$x \triangleright \texttt{update}$' request, no other logic variable is allowed to update to x. This is ensured by disabling '$x \triangleright \texttt{value}$' requests during update sections, and vice versa, since another variable must read the value of the reference it is updating to. The encoding does indeed take measures to prevent circularities when x is told to update to itself either directly or indirectly. However, the encoding does not prevent incorrect behavior in cases where two distinct logic variables are trying to update to each other concurrently.

This can be demonstrated as follows. Consider another logic variable z which is distinct from the logic variable x. The π-calculus process $V(\underline{z})$ denotes the corresponding handler agent that encodes z. Let logic variables

x and z be such that they have so far not been subjected to any update operations. Now, consider the encodings of the two behaviorally equivalent γ-calculus terms, namely $\exists x(x = z)$ and $\exists x(x = z \wedge z = x)$.

The encoding of the γ-calculus term $\exists x(x = z)$ is given by

$$[\![\exists x(x = z)]\!] \; = \; (\nu x)(V(\underline{x}) \mid V(\underline{z}) \mid \overline{x \triangleright \texttt{update}} < \underline{z} >)$$

which, after several reductions, results in:

$$(\nu x)(R(\underline{x}, \underline{z}) \mid V(\underline{z}))$$

thereby achieving the required update.

On the other hand the encoding of the other behaviorally equivalent γ-calculus term $\exists x(x = z \wedge z = x)$ is given by the following parallel composition:

$$[\![\exists x(x = z \wedge z = x)]\!] \; = \; (\nu x)(V(\underline{x}) \mid V(\underline{z}) \mid \overline{x \triangleright \texttt{update}} < \underline{z} > \; \mid \overline{z \triangleright \texttt{update}} < \underline{x} >)$$

The above parallel composition, after several reductions, results in the following parallel composition, which cannot evolve any further:

$$(\nu x)(\overline{z \triangleright \texttt{value}} < \nu c, x > .c(\underline{u}).[x = u](V(\underline{x}), R(\underline{x}, \underline{u}))$$

$$\mid \overline{x \triangleright \texttt{value}} < \nu c, z > .c(\underline{u}).[z = u](V(\underline{z}), R(\underline{z}, \underline{u})))$$

Thus, processing the request $\overline{x \triangleright \texttt{update}} < \underline{z} >$ in parallel with the request $\overline{z \triangleright \texttt{update}} < \underline{x} >$ fails to achieve the required update of the logic variables to each other. This can be clearly seen from the above parallel composition of terms which cannot evolve any further.

3 Discarded atomic transactions

The γ-calculus and the π-calculus are based on two very different paradigms of concurrent computation. The γ-calculus may be thought of as being built on the shared memory model of concurrent programming, while the π-calculus may be considered to be based on the message passing model. The γ-calculus utilizes a central shared memory, called a *blackboard* [15], and uses global substitution of variables to achieve state updates. The π-calculus utilizes message passing through named channels between concurrent processes.

The embedding of logic variables from the γ-calculus to the π-calculus needs careful consideration. With the underlying memory model of γ-calculus, operations which involve multiple logic variables appear very natural. One such operation is that of updating two distinct logic variables to each other. On the other hand, as we have seen in the last section, simulating such update operations of logic variables in a message passing model becomes error prone.

In the γ-calculus, the act of updating two distinct logic variables to each other is an atomic transaction [6], and it is essential that this atomicity be preserved by translations to the message passing paradigm. Such an updating process in the message passing paradigm, should be able to acquire its resources (namely the two logic variables) without leading to deadlock or starvation. One of the simplest ways of encoding operations involving more than one logic variable simultaneously, is to use the *two-phase locking* algorithm [6] familiar from distributed computing. Using a more complex encoding it is possible to program the two-phase locking strategy from the basic constructs of π-calculus. On the other hand, a more elegant alternative would be to extend the π-calculus with mechanisms which would simplify the translation of two-phase locking into it [3]. Of course, apart from two-phase locking, there are a number of other well known strategies to ensure atomicity of transactions [6].

4 Related Work

Related research may be classified under four main sub-headings. They are discussed in the following sub-sections.

4.1 From sequential logic programming to π-calculus

Sequential logic programming and sequential Prolog may be considered as special cases of concurrent constraint systems. Work reported in [5, 14] provides translations of logic variables, sequential unification, and other features of Prolog to π-calculus. However, these papers do not provide proofs of formal correctness, and also remain unencumbered by the complexities of concurrency in their source languages.

4.2 From concurrent constraints to enhanced π-calculus

There are papers [9, 15] which provide an embedding from different variants of concurrent constraint calculi to enhanced versions of the π-calculus.

However the embedding proposed by these papers [9, 15] require the support of extra enhancements to the primitives of π-calculus, such as variables, equations, and elimination rules, in order to define the mapping.

The update calculus [10] and the fusion calculus [11] are process calculi which posit entirely new primitives which are not present in the π-calculus. Such primitives are called *update* and *fusion* in the two calculi respectively. They provide the capability to perform actions which result directly in global side effects, or in side effects whose scope can be explicitly controlled with the help of a *scope* operator. Thus, it should be noted that in these calculi the basic mechanisms for updating a shared state are already present, and there is no simple way of translating such primitives to the π-calculus. The work reported in [19] embeds a calculus with concurrent constraints into the fusion calculus.

4.3 From concurrent constraints to basic π-calculus

The only work of this kind is that in [18], which tries to provide a translation from concurrent constraints to the minimal π-calculus without requiring any additional enhancements. However, as we have demonstrated, the embedding it proposes leads to deadlocks.

4.4 From other domains to extensions of π-calculus

The π-calculus has been extended in many other ways in response to motivations from various other application domains. Such extensions include the applied π-calculus [1] which has extensions like value passing, primitive functions, and equations among terms, in order to analyse security protocols. The extended π-calculus [3] considers extensions such as message loss, sites, timers, site failure, and persistence, to the asynchronous version of the π-calculus [4] so as to examine transactions in the presence of partial failures in distributed systems.

5 Conclusion

The task of relating core calculi which correspond to different programming paradigms is an important one. However, as shown in this paper, one needs to be aware of the various subtleties that need to be taken care of in order to avoid pitfalls and inaccuracies that are likely to arise in proposed embeddings. The purpose of this paper is not to merely act as a bug report, but instead to effectively demostrate that the problem of relating higher-order concurrent constraint programming to the paradigm of interactive concurrent programming still remains open. Though we have hinted at using standard algorithms from distributed programming to solve the problem correctly, nevertheless the task of constructing a fully abstract encoding between the two paradigms will prove to be highly intricate and non trivial to achieve.

A useful intermediate step which might help in the construction of elegant encodings between various calculi, would be the provision of simple extensions for the core calculi. Analogous to similar situations in logic, such extensions could take the form of *conservative extensions* of the minimal calculi, in order to retain the minimality feature of the core calculi. While in logic the notion of conservative extension has the clear meaning of preserving satisfaction of original sentences, in the world of calculi for computation it should mean conservativity in dynamics and in behavioural semantics. The update calculus and the fusion calculus cannot be considered to be conservative extensions of the π-calculus because there is no simple way of translating the primitives of these calculi to the π-calculus. Formulated in this manner, conservative extensions would equip the underlying core calculi with higher-level constructs which provide a convenient handle to reduce the complexity of various embeddings, and would also enrich the core calculi by making them more elegant frameworks for the analysis of real-world programming language characteristics and features.

Acknowledgements

Our thanks to the referees for their helpful comments. Our thanks to Ms. Margaret D'Souza for typing and proofreading this paper.

References

[1] M. Abadi, C. Fournet, Mobile Values, New Names, and Secure Communication, Proceedings of POPL'01, ACM Press (2001) 104–115.

[2] H.P. Barendregt, Functional Programming and Lambda Calculus, Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (1990) 321–363.

[3] M. Berger, K. Honda, The Two-Phase Commit Protocol in an Extended Pi-Calculus, Proc. Express'00, Electronic Notes in Theoretical Computer Science, Vol. 39, No. 1 (2000).

[4] K. Honda, M. Tokoro, An Object Calculus for Asynchronous Communication, ECOOP'91, LNCS **512**, Springer (1991) 133–147.

[5] B.Z. Li, A π-Calculus Specification of Prolog, ESOP'94, LNCS **788**, Springer (1994) 379–393.

[6] N.A. Lynch, M. Merritt, W.E. Weihl, A. Fekete, Atomic Transactions, Morgan Kaufmann (1993).

[7] R. Milner, Communicating and Mobile Systems: The Pi Calculus, Cambridge University Press (1999).

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

[9] J. Niehren, M. Muller, Constraints for Free in Concurrent Computation, Asian Computing Science Conference, ACSC'95, LNCS **1023**, Springer (1995) 171–186,.

[10] J. Parrow, B. Victor, The Update Calculus, Proceedings of AMAST'97, LNCS **1349**, Springer (1997) 409–423.

[11] J. Parrow, B. Victor, The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes, Proceedings of LICS'98, IEEE Press (1998) 176–185.

[12] B.C. Pierce, Foundational Calculi for Programming Languages, Handbook of Computer Science and Engineering, CRC Press (1997) 2190–2207.

[13] B.C. Pierce, D.N. Turner, Pict: A Programming Language Based on the Pi-Calculus, Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press (2000) 455–494.

[14] B.J. Ross, A π-Calculus Semantics of Logical Variables and Unification, Proc. of First North American Process Algebra Workshop, Workshops in Computing, Springer-Verlag (1993) 216–230.

[15] G. Smolka, A Foundation for Higher-Order Concurrent Constraint Programming, Proceedings of Constraints in Computational Logics, Lecture Notes in Computer Science, Volume 845, Springer (1994) 50–72.

[16] G. Smolka, The Oz Programming Model, Computer Science Today, Lecture Notes in Computer Science, vol 1000, Springer (1995) 324–343.

[17] G. Smolka, The Definition of Kernel Oz, Constraints: Basics and Trends, Lecture Notes in Computer Science, Volume 910, Springer (1995) 251–292.

[18] B. Victor, J. Parrow, Constraints as Processes, Proceedings of CONCUR'96, Lecture Notes in Computer Science, Volume 1119, Springer (1996) 389–405.

[19] B. Victor, J. Parrow, Concurrent Constraints in the Fusion Calculus, Proceedings of ICALP'98, LNCS **1443** (1998) 455–469.