Compiling Effectful Terms to Transducers

Prototype Implementation of *Memoryful Geometry of Interaction*

Koko Muroya Toshiki Kataoka Ichiro Hasuo
Dept. Computer Science, University of Tokyo
{muroykk,toshikik,ichiro}@is.s.u-tokyo.ac.jp

Naohiko Hoshino
RIMS, Kyoto University
naophiko@kurims.kyoto-u.ac.jp

In this preliminary report for *LOLA 2014*, we present a prototype implementation of the *memoryful GoI* framework in [Hoshino, Muroya and Hasuo, CSL-LICS 2014] that translates lambda terms with algebraic effects to transducers. Those transducers can be thought of as "proof nets with memories" and are constructed in a compositional manner by means of coalgebraic component calculi. The transducers thus obtained can be simulated in our tool, too, helping us to scrutinize the step-by-step interactions that take place in higher-order effectful computation.

Geometry of Interaction (GoI) Girard's *Geometry of Interaction (GoI)* [6] is interaction based semantics of linear logic proofs and, via suitable translations, of functional programs in general. The mathematical cleanness of GoI has successfully identified various essential structures in computation; moreover its use as a compilation technique from programs to state machines—"GoI implementation," so to speak—has been worked out by Mackie, Pinto, Ghica and others [3, 4, 10, 11].

GoI is "Memoryless" In the common presentation of GoI by *token machines* [10], a λ-term induces a *proof net* on which a token runs through and computes the semantic value of the term, the latter being a cut-elimination invariant. A *state* of a token machine is the current position of the token; a *state transition* of a token machine is then a movement of the token, from one position to another. It is notable that the underlying proof net—the "graph" on which the token moves around—is static and remains unchanged.

This *memoryless* nature of GoI is a big advantage in view of simplicity: it allows us to analyze complicated higher-order computation in the elementary terms of nodes and edges in graphs (or: *links* and *edges* in proof nets). The same nature, however, poses certain limitations to the use of GoI, too. For example, it is long known in the community that additive connectives in linear logic—and similarly coproduct types like $\mathtt{bit} = 1 + 1$—call for special care in GoI interpretation that is far from trivial. One solution by so-called *additive slices* [9], which can be thought of as an additional "memory" layer on proof nets.

Another example of the limitations of "memoryless GoI" manifests itself in presence of *computational effects*. Consider the call-by-value evaluation of the term

$$\mathtt{P} \quad = \quad (\lambda \mathtt{x} : \mathtt{nat}.\, \mathtt{x} + \mathtt{x})\, (\mathtt{3} \sqcup \mathtt{5}) \quad : \mathtt{nat} \tag{1}$$

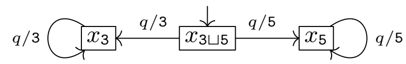
where the subterm $\mathtt{3} \sqcup \mathtt{5}$ returns $\mathtt{3}$ or $\mathtt{5}$ nondeterministically. Obviously the term is expected to yield $\mathtt{6}$ or $\mathtt{10}$ (but not $\mathtt{8}$). However the usual GoI interpretation can yield $\mathtt{8}$ too: in the interaction between the subterms $\lambda\mathtt{x}.\, \mathtt{x} + \mathtt{x}$ and $\mathtt{3} \sqcup \mathtt{5}$, the value of the latter is queried twice, to which the subterm $\mathtt{3}\sqcup\mathtt{5}$ can answer differently. Here what is needed is some memory mechanism that allows the subterm $\mathtt{3}\sqcup\mathtt{5}$ to *remember* the choice that it has made, and to *stick* to it.

"Memoryful" GoI Motivated by a similar (but more complicated) technical challenge we encountered in the semantics of a quantum λ-calculus [7], in [8] we introduced the *memoryful GoI* framework that systematically equips proof nets with memories.[1]

Let T be a monad on $\mathbf{Set}$ and Σ be a set of algebraic operations, as in [12]. Our framework yields a translation of a λ-term t (in which algebraic operations σ can occur) to a *(stream) transducer*—also called a *Mealy machine* or a *sequential machine*—that itself has a T-effect. The latter is concretely given by

$$\big(\ X,\ X \times A \overset{c}{\to} T(X \times B),\ x_0 \in X\ \big)\ \ ;$$

it is a state machine that transforms streams over A to those over B, in a way that depends on the internal state $x \in X$. An example is shown below that would adequately model the term $\mathtt{3} \sqcup \mathtt{5}$ in (1).



(2)

Here the machine can initially respond to a query q with $\mathtt{3}$ or $\mathtt{5}$; however, after that the machine sticks to the same choice by remembering the choice by means of its internal state. We use such a transducer as a *memoryful node* (or a "link") of a proof net, or their composite (i.e. a *memoryful proof net*).

What is notable about our framework is that the term-to-transducer translation is based on denotational semantics—given by a category of suitable partial equivalence relations (PERs)—and hence is *correct by construction* and *compositional*. The construction of the denotational model relies on the categorical axiomatization of GoI by Abramsky, Haghverdi and Scott [1, 2]: it allows one to derive a Cartesian closed category from a traced monoidal category $\mathcal{C}$ with suitable additional structures. What we do in [8] is take as $\mathcal{C}$ the category of *resumptions*, i.e. transducers modulo a suitable behavioral equivalence. This in fact is already done in [2]: our technical novelty is systematic use of *component calculi*—those calculi for composing transducers, formulated in coalgebraic terms—in composing resumptions.

Our Tool TtT This is a preliminary report on the implementation of memoryful GoI. Our tool is called *TtT*—short for "Terms to Transducers"—and is implemented in Haskell. It consists of two parts: *TtT Compiler* and *TtT Simulator*.

TtT Compiler implements the translation sketched in the above. We express a transducer (with the effect T) as a Haskell program of the type `Td m x a b`:

```
type  Td m x a b  =  (x, a) -> m (x, b)
```

where m is the Haskell monad that corresponds to T, x is the type for a state space, a is the input type and b the output type.

The transducer obtained from an (effectful) λ-term is then executed by TtT Simulator, in a meticulous way where every movement is recorded. Recall that a transducer here is much like a proof

[1] The word "memory" here is almost synonymous with "internal states"; we stick to the former so as to distinguish from *states* as a computational effect.

net equipped with memories: TtT Simulator records every movement of a token on it, together with the change of memories (from $x \in X$ to x') caused by the visit of the token. Examples of simulation results are shown shortly.

TtT Compiler is parametric in the choice of a monad T and algebraic operations, exploiting Haskell's support of computational effects as monads. TtT Simulator, however, is necessarily monad-specific; currently it supports the powerset monad (nondeterminism) and the distribution monad (probability).

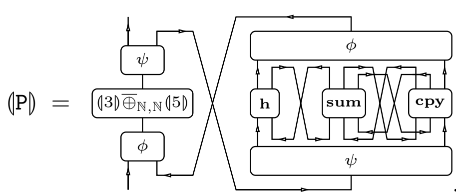
TtT as a Prototype We emphasize that what our tool TtT currently does is *of no practical use whatsoever*: after all it translates an effectful λ-term—that may well be simply expressed as a Haskell program—to another Haskell program that is way more complicated and runs more slowly. Nevertheless we believe this is a worthwhile venture, for the following three reasons.

The first reason is theoretical: our memoryful GoI framework in [8] seems to be a useful theoretical tool that gives us insights into higher-order computation with effects. Automating the translation—that is painfully complicated when done by hand—will hence meet some theoreticians' needs.

The second reason is speculative but practical: we wish to follow the path of [3, 4, 10, 11] and use memoryful GoI as a compilation technique to hardware. In our case this will specifically mean to take hardware that natively supports the effect T, and compiling λ-terms with T-algebraic effects to it. Doing so for emerging computing paradigms like probabilistic and quantum programming will have big practical impacts—not only because programs will execute faster (see e.g. [5]), but also because the compilation (based on denotational semantics) is correct by construction. The current toy tool of TtT will then form a basis of such practical compilers.

The third reason is: it's simply a lot of fun to see higher-order effectful computation in action—or *GoI at work*. We hope the reader will be convinced by the following examples.

Transducers, Derived and Executed Consider the term `P` in (1) with nondeterministic choice $\sqcup$ in it. Its translation to a (three-state, nondeterministic) transducer $(\!|\mathtt{P}|\!) \colon 3 \times \mathbb{N} \to \mathcal{P}(3 \times \mathbb{N})$, after some manual simplifications, is depicted below (manually).



(3)

The figure is a *string diagram* in the traced monoidal category of resumptions; the box $(\!|3|\!)\overline{\oplus}_{\mathbb{N},\mathbb{N}}(\!|5|\!)$ in (3) is the (equivalence class of) the transducer (2), after suitably encoding messages like q or `3` as natural numbers. The diagram (3) can be identified with a proof net via the "Int construction." Specifically: notice a horizontal axis of symmetry; folding the diagram on the axis then gives a proof net, where the four ϕ and ψ nodes get identified and yield two nodes that are understood as $⅋$ or $\otimes$ links in the proof net.

The tool TtT generates the transducer $(\!|\mathtt{P}|\!)$ inductively by the derivation of the type of `P`. The outcome is a Haskell program of type `Td [] x Int Int` where: `Td` is as defined before; `[]` is the list monad; and `x` is a type that stands for a three-element set. Presenting it graphically as in (3) seems to take a lot of efforts and is left as future work.

Instead of pictorial presentation, we "execute" the transducer with TtT Simulator and observe its dynamic behaviors. The simulation result of $(\!|\mathtt{P}|\!)$ for `P` in (1) consists of 741 lines and it yields `6` and `10` as possible outcomes, successfully excluding `8`. Similarly, for the term $(\lambda f.\, f\, \mathtt{0} + f\, \mathtt{1})\, (\lambda x.\, \mathtt{3} \sqcup \mathtt{5})$ we have a simulation result of 4526 lines that yields $\mathtt{6}, \mathtt{8}, \mathtt{8}, \mathtt{10}$ as possible outcomes (duplication of `8` is due to different final states of the transducer).

Simulation Results We present actual simulation results for simpler terms: $\mathtt{3} \sqcup \mathtt{5}$, written as `3|_|5` in Fig. 1; and $(\lambda x.\, x)\, \mathtt{1}$ in Fig. 2.

```
1  .    ---- dd<42,137> --->
2  |    Query:[ [3|_|5] @ Nothing ]
3  +-.    ---- dd<42,137> --->
4  | |    Query:[ [3]|_|5 @ * ]
5  | |      h; k_3; h
6  | |    [ [3]|_|5 @ * ]:Answer
7  | |    ---- dd<42,3> --->
8  | |  [ [3|_|5] @ Just (Left (*)) ]:Answer
9  | |  ---- dd<42,3> --->
10 | Result: 3 / State: Just (Left (*))
11 `-.    ---- dd<42,137> --->
12   |    Query:[ 3|_|[5] @ * ]
13   |      h; k_5; h
14   |    [ 3|_|[5] @ * ]:Answer
15   |    ---- dd<42,5> --->
16   |  [ [3|_|5] @ Just (Right (*)) ]:Answer
17   |  ---- dd<42,5> --->
18   Result: 5 / State: Just (Right (*))
```

Figure 1. Simulation Result for $\mathtt{3} \sqcup \mathtt{5}$

The convention is as follows. A token carries a natural number n around—or more precisely, an edge e.g. in (3) consists of $|\mathbb{N}|$-many "pipes" and a token goes through the n-th—and this is denoted by `---- n --->`. Therefore expressions like `dd<42,137>` in Fig. 1 stand for a certain natural number. We however chose to supplementarily use the *dynamic algebra* notation (`g` for *left* and `d` for *right*, in French, see e.g. [9, 11]), for readability and efficiency.

In Fig. 1, between movements `---- n --->` of the token are expressions like `Query:[ [3|_|5] @ Nothing ]` and `h; k_3; h`. The ones like the former stand for the token's *entrance to* and *exit from* a transducer. Specifically, p`:[`t`@`x`]`—with a subterm t' of t designated by `[`t'`]`—means: the token entered to (a copy of) the transducer $(\!|t'|\!)$, whose current state is x, at its *port* p. Typical ports include: `Query` (a top-level query) and `Answer x` (an answer to the previous query on `x`; see Fig. 2, lines 24 and 44). Similarly, `[`t`@`x`]:`p means the token departed from the port p.

The latter class of expressions like `h; k_3; h` on Fig. 1, line 5 and `phi` on Fig. 2, line 3 are referred to as *bookkeeping*. For example, `h; k_3; h` means the token traversed an $\mathbf{h}$ node, a $\mathbf{k}_3$ node and then an $\mathbf{h}$ node. The definition of these nodes (as pure functions like $\mathbf{h} \colon \mathbb{N} + \mathbb{N} \to \mathbb{N} + \mathbb{N}$) is found in [8]. As we noted before, these bookkeeping nodes correspond to logical connectives in a proof net; therefore they change the number carried by the token. From line 1 to 4 in Fig. 2, `g` is added by `phi` i.e. $⅋$; this is much like in the dynamic algebra presentation [9].

Let us discuss effects. Branching occurred in Fig. 1, line 3, as depicted by the lines on the left. As we discussed about the term (1), different branches must result in different states $x \in X$, or "memories"; we see this is indeed the case in Fig. 1, lines 8 and 16, hence eventually lines 10 and 18. The states `Just (Left (*))` and `Just (Right (*))` encode x_3 and x_5 in (2), respectively.

The indentation designates the depth of the subterm in focus. In Fig. 2, we see a query on `x` raised on line 24. The token is returned to shallower levels and is eventually passed on to the subterm `1` of $(\lambda x.\, x)\, \mathtt{1}$ (denoted by `(\x.x) [1]`) on line 33; and then the value `1` of `x` is obtained and carried by the token (in the form of `1` in `<42,1>`) to the originator on line 44.

We note that a query to a natural number value must be a token carrying a number `dd<`n`,`m`>`, where n and m are arbitrary (hence 42 and 137 in Fig. 1–2 are just arbitrary numbers). An answer to this query, that the value is l, is given by a token carrying `dd<`n`,`l`>`.

```
1 ---- dd<42,137> --->
2 Query:[ [(\x.x) 1] @ ({_: *}, *) ]
3   phi
4   ---- gdd<42,137> --->
5   Query:[ [\x.x] 1 @ {_: *} ]
6     h
7   [ [\x.x] 1 @ {_: *} ]:Answer
8   ---- dgdd<42,137> --->
9   psi; psi; phi
10  ---- gdd<42,137> --->
11  Query:[ (\x.x) [1] @ * ]
12    h
13  [ (\x.x) [1] @ * ]:Answer
14  ---- dgdd<42,137> --->
15  psi; e; phi; phi
16  ---- dd<0,gdd<42,137>> --->
17  Query:[ [\x.x] 1 @ {_: *} ]
18    h; v
19    0 {
20      psi
21      ---- dd<42,137> --->
22      Query:[ (\x.[x]) 1 @ * ]
23        h
24      [ (\x.[x]) 1 @ * ]:Query x
25      ---- <42,137> --->
26      phi
27    } 0
28    u; h
29  [ [\x.x] 1 @ {_: *} ]:Answer
30  ---- dd<0,d<42,137>> --->
31  psi; psi; e'; phi
32  ---- dd<42,137> --->
33  Query:[ (\x.x) [1] @ * ]
34    h; k_1; h
35  [ (\x.x) [1] @ * ]:Answer
36  ---- dd<42,1> --->
37  psi; e; phi; phi
38  ---- dd<0,d<42,1>> --->
39  Query:[ [\x.x] 1 @ {_: *} ]
40    h; v
41    0 {
42      psi
43      ---- <42,1> --->
44      Answer x:[ (\x.[x]) 1 @ * ]
45        h
46      [ (\x.[x]) 1 @ * ]:Answer
47      ---- dd<42,1> --->
48      phi
49    } 0
50    u; h
51  [ [\x.x] 1 @ {_: *} ]:Answer
52  ---- dd<0,gdd<42,1>> --->
53  psi; psi; e'; phi
54  ---- dgdd<42,1> --->
55  Query:[ (\x.x) [1] @ * ]
56    h
57  [ (\x.x) [1] @ * ]:Answer
58  ---- gdd<42,1> --->
59  psi; phi; phi
60  ---- dgdd<42,1> --->
61  Query:[ [\x.x] 1 @ {_: *} ]
62    h
63  [ [\x.x] 1 @ {_: *} ]:Answer
64  ---- gdd<42,1> --->
65  psi
66 [ [(\x.x) 1] @ ({_: *}, *) ]:Answer
67 ---- dd<42,1> --->
68 Result: 1 / State: ({_: *}, *)
```

Figure 2. Simulation Result for $(\lambda x.\, x)\, 1$

Finally let us speak about making $|\mathbb{N}|$-many copies of a transducer—which interprets the $!$ modality that is implicit in the Girard translation $A \to B = {!A} \multimap B$. The bookkeeping function $v\colon \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ in Fig. 2, line 18, splits $|\mathbb{N}|$-many pipes into $|\mathbb{N}| \cdot |\mathbb{N}|$-many pipes; and lines 19 and 27 mean the token went to the 0-th bunch of pipes, i.e., to the 0-th copy of the transducer `[ (\x.[x]) 1 @ * ]`. The state `{_: *}` that occur e.g. on line 17 stands for the function $\mathbb{N} \to 1, n \mapsto *$, meaning that the state of every copy of the transducer is the unique one $*$.

Acknowledgments

Thank are due to Ugo Dal Lago, Claudia Faggian, Dan Ghica and Akira Yoshimizu for useful discussions, and to the anonymous referees for useful comments. K.M. and I.H. are supported by Grants-in-Aid for Young Scientists (A) No. 24680001, JSPS; T.K. is supported by Grant-in-Aid for JSPS Fellows; and N.H. is supported by Grants-in-Aid for Young Scientists (B) No. 26730004, JSPS.

References

[1] S. Abramsky. Retracing some paths in process algebra. In *CONCUR*, pages 1–17, 1996.

[2] S. Abramsky, E. Haghverdi, and P. Scott. Geometry of interaction and linear combinatory algebras. *Math. Struct. in Comp. Sci.*, 12:625–665, 2002.

[3] O. Fredriksson and D. R. Ghica. Seamless distributed computing from the geometry of interaction. In C. Palamidessi and M. D. Ryan, editors, *Trustworthy Global Computing*, volume 8191 of *Lecture Notes in Computer Science*, pages 34–48. Springer Berlin Heidelberg, 2013.

[4] D. R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 363–375. ACM, 2007. ISBN 1-59593-575-4.

[5] D. R. Ghica, A. I. Smith, and S. Singh. Geometry of synthesis IV: compiling affine recursion into static hardware. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *ICFP*, pages 221–233. ACM, 2011. ISBN 978-1-4503-0865-6.

[6] J.-Y. Girard. Geometry of interaction 1: Interpretation of system F. In S. V. R. Ferro, C. Bonotto and A. Zanardo, editors, *Logic Colloquium '88 Proceedings of the Colloquium held in Padova*, volume 127 of *Studies in Logic and the Foundations of Mathematics*, pages 221–260. Elsevier, 1989.

[7] I. Hasuo and N. Hoshino. Semantics of higher-order quantum computation via geometry of interaction. In *LICS*, pages 237–246, 2011.

[8] N. Hoshino, K. Muroya, and I. Hasuo. Memoryful geometry of interaction: From coalgebraic components to algebraic effects. In *CSL-LICS*, 2014. To appear.

[9] O. Laurent. A token machine for full geometry of interaction. In *TLCA*, pages 283–297, 2001.

[10] I. Mackie. The geometry of interaction machine. In R. K. Cytron and P. Lee, editors, *POPL*, pages 198–208. ACM Press, 1995. ISBN 0-89791-692-1.

[11] J. S. Pinto. *Implantation Parallèle avec la Logique Linéaire (Applications des Réseaux d'Interaction et de la Géométrie de l'Interaction)*. PhD thesis, École Polytechnique, 2001. Main text in English.

[12] G. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.