Compiling Effectful Terms to Transducers

Prototype Implementation of Memoryful Geometry of Interaction

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

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

In this preliminary report for LOLA 2014, we present a prototype implementation of the memoryful GoI framework in [Hoshino, Muoya 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 coalescable 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 ![\bigoplus](https://latex.codecogs.com/svg.image?\bigoplus) in proof nets. The limitations are that additive connectives in linear logic and similarly coproduct types like ![\bigoplus](https://latex.codecogs.com/svg.image?\bigoplus) in proof nets can occur nondeterministically. 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

\[ P = (\lambda x : \text{nat}. x + x) (3 \bigoplus 5) : \text{nat} \]

where the subterm 3 !\(\bigoplus\) 5 returns 3 or 5 nondeterministically. Obviously the term is expected to yield 6 or 10 (but not 8). However the usual GoI interpretation can yield 8 too: in the interaction between the subterms ![\lambda x . x + x](https://latex.codecogs.com/svg.image?\lambda x . x + x) and 3 !\(\bigoplus\) 5, the value of the latter is queried twice, to which the subterm 3 !\(\bigoplus\) 5 can answer differently. Here what is needed is some memory mechanism that allows the subterm 3 !\(\bigoplus\) 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 ![\mathbb{T}](https://latex.codecogs.com/svg.image?\mathbb{T}) be a monad on ![\mathbb{Set}](https://latex.codecogs.com/svg.image?\mathbb{Set}) and ![\Sigma](https://latex.codecogs.com/svg.image?\Sigma) be a set of algebraic operations, as in [12]. Our framework yields a translation of a λ-term ![t](https://latex.codecogs.com/svg.image?t) (in which algebraic operations ![\sigma](https://latex.codecogs.com/svg.image?\sigma) can occur) to a (stream) *transducer*—also called a Mealy machine or a sequential machine—that itself has a ![T](https://latex.codecogs.com/svg.image?T)-effect. The latter is concretely given by

\[ (X, X \times A \xrightarrow{t} T(X \times B), x_0 \in X) \]

it is a state machine that transforms streams over ![A](https://latex.codecogs.com/svg.image?A) to those over ![B](https://latex.codecogs.com/svg.image?B), in a way that depends on the internal state ![x](https://latex.codecogs.com/svg.image?x) ∈ ![X](https://latex.codecogs.com/svg.image?X). An example is shown below that would adequately model the term ![3 !\bigoplus 5](https://latex.codecogs.com/svg.image?3 \bigoplus 5) in (1).

\[ q/5 \xrightarrow{\text{push } \left(\text{cons } x\right)} q/3 \xrightarrow{\text{pop } \left(\text{cons } x\right)} q/5 \]

Here the machine can initially respond to a query ![q](https://latex.codecogs.com/svg.image?q) with ![3](https://latex.codecogs.com/svg.image?3) or ![5](https://latex.codecogs.com/svg.image?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 us to derive a Cartesian closed category from a traced monoidal category ![C](https://latex.codecogs.com/svg.image?C) with suitable additional structures. What we do in [8] is take as ![C](https://latex.codecogs.com/svg.image?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 calculus—those calculi for composing transducers, formulated in coalescable 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](https://latex.codecogs.com/svg.image?T)) as a Haskell program of the type ![T \vec{m} n x a b](https://latex.codecogs.com/svg.image?T \vec{m} n x a b):

\[
\text{type } T \vec{m} n x a b = (x, a) \rightarrow m (x, b)
\]

where ![n](https://latex.codecogs.com/svg.image{n}) is the Haskell monad that corresponds to ![T](https://latex.codecogs.com/svg.image{T}), ![x](https://latex.codecogs.com/svg.image{x}) is the type for a state space, ![a](https://latex.codecogs.com/svg.image{a}) is the input type and ![b](https://latex.codecogs.com/svg.image{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.
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 \( \lambda \)-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 \( \lambda \)-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 \( \lambda \text{-calculus} \) 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 \( \langle P \rangle : 3 \times N \rightarrow \mathcal{P}(3 \times N) \), after some manual simplifications, is depicted below (manually).

\[
\langle P \rangle = \begin{cases} 1 & \text{if } \phi \text{ or } \psi \text{ is a terminal node,} \\ 0 & \text{otherwise.} \end{cases}
\]

The figure is a string diagram in the traced monoidal category of resumptions; the box \([3]_{\mathcal{T}(h,k_3,h)}\) 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 “\( \text{Int} \) construction.” Specifically: notice a horizontal axis of symmetry; folding the diagram on the axis then gives a proof net, via the “\( \text{Int} \) construction.” Specifically: notice a horizontal axis of symmetry; folding the diagram on the axis then gives a proof net, via the “\( \text{Int} \) construction.” Specifically: notice a horizontal axis of symmetry; folding the diagram on the axis then gives a proof net, via the “\( \text{Int} \) construction.” Specifically: notice a horizontal axis of symmetry; folding the diagram on the axis then gives a proof net, via the “\( \text{Int} \) construction.” Specifically: notice a horizontal axis of symmetry; folding the diagram on the axis then gives a proof net, via the “\( \text{Int} \) construction.”

The tool TTT generates the transducer \( \langle P \rangle \) inductively by the derivation of the type of \( P \). The outcome is 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 \( \lambda \)-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 \( \lambda \text{-calculus} \) 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 \( \langle P \rangle : 3 \times N \rightarrow \mathcal{P}(3 \times N) \), after some manual simplifications, is depicted below (manually).

\[
\langle P \rangle = \begin{cases} 1 & \text{if } \phi \text{ or } \psi \text{ is a terminal node,} \\ 0 & \text{otherwise.} \end{cases}
\]

The figure is a string diagram in the traced monoidal category of resumptions; the box \([3]_{\mathcal{T}(h,k_3,h)}\) 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 “\( \text{Int} \) construction.” Specifically: notice a horizontal axis of symmetry; folding the diagram on the axis then gives a proof net, via the “\( \text{Int} \) construction.” Specifically: notice a horizontal axis of symmetry; folding the diagram on the axis then gives a proof net, via the “\( \text{Int} \) construction.” Specifically: notice a horizontal axis of symmetry; folding the diagram on the axis then gives a proof net, via the “\( \text{Int} \) construction.” Specifically: notice a horizontal axis of symmetry; folding the diagram on the axis then gives a proof net, via the “\( \text{Int} \) construction.”

The tool TTT generates the transducer \( \langle P \rangle \) inductively by the derivation of the type of \( P \). The outcome is 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 \( \lambda \)-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 \( \lambda \text{-calculus} \) at work. We hope the reader will be convinced by the following examples.
Figure 2. Simulation Result for $(\lambda x.x)$ 1

Finally let us speak about making $[N]$-many copies of a transducer—which interprets the $!$ modality that is implicit in the Girard translation $A \rightarrow B = !A \rightarrow B$. The bookkeeping function $v : N \rightarrow N \times N$ in Fig. 2, line 18, splits $[N]$-many pipes into $[N] \cdot [N]$-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 $[(\lambda x.\ [x]) \ 1 \ 0 \ *]$. The state $\{_, *\}$ that occur e.g. on line 17 stands for the function $N \rightarrow 1, n \mapsto *$, 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


