Towards Representation-Independent Logic and Data

CONSTANTINE THEOCHARIS, University of St Andrews, UK

Abstract TODO

CCS Concepts: • **Theory of computation** → **Categorical semantics**; **Abstraction**; *Operational semantics*; • **Software and its engineering** → *Translator writing systems and compiler generators.*

Additional Key Words and Phrases: Representation, Algebraic Data Types, Categorical Semantics, Compilation, Low-Level Data Structures

ACM Reference Format:
Constantine Theocharis. 2024. Towards Representation-Independent Logic and Data. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Let $\mathbb{S}$ and $\mathbb{L}$ be two 2-categories. Eventually we will require additional structure to be present in these, but for now we want to consider them as context categories for two different type theories. We will consider 2-morphisms to be rewriting rules, and 1-morphisms to be terms, in the style of [CITE].

We want to view $\mathbb{S}$ as the "high-level", nice category that we want to write our programs in. On the other hand, $\mathbb{L}$ is the "low-level", ugly category that we actually execute our programs in. It is ugly from the point of view of the programmer, but nice from the point of view of a machine, because it closely follows the way the machine actually executes the program.

Since we have a high-level and a low-level category, we want to have a way to translate between them. We will do this by defining two functors:

- A functor $C : \mathbb{S} \longrightarrow \mathbb{L}$, called the *compilation* functor. It takes a program in § and compiles it to a program in $\mathbb{L}$.
- A functor $Q : \mathbb{L} \longrightarrow \mathbb{S}$, called the *quoting* functor. it allows us to opaquely operate on program fragments from $\mathbb{L}$ within $\mathbb{S}$.

Importantly, both of these functors are *lax*, meaning that they only respect functoriality up to 2-morphisms. In other words, compiling a high-level program $C(f \circ g)$ is not necessarily the same as compiling $C(f)$ and then $C(g)$ separately, but there is a 2-morphism $C(f) \circ C(g) \Rightarrow C(f \circ g)$ implying that one evaluates to the other.

The quoting functor Q is left-adjoint to the compilation functor C, in the sense that compiling a quoted program yields the same program. Commonly this adjunction is strict.

2 FEATURES OF $\mathbb{S}$

Since $\mathbb{S}$ is the site of our high-level programs, we want it to have a lot of the things that we expect from a programming language. In particular, we want it to be cartesian closed, and we want it

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

to admit fixpoints of endofunctors. Furthermore, we expect that there exists a terminal object $1 \in \mathrm{Ob}(\mathbb{S})$.

3 FEATURES OF $\mathbb{L}$

We require much less convenience in the structure of $\mathbb{L}$, instead relying on the functor C to translate human-friendly programs into machine-friendly ones.

Commonly $\mathbb{L}$ will not be closed, and it might be a Kleisli category with respect to some monad (probably a monad holding the state of the stack/heap/environment/etc).

We will commonly want to restrict $\mathbb{L}$ to only be monoidal in some less-than-cartesian way, for example to model a linear logic for the tracking and preservation of resources.

4 COMPILATION

This setup of having two sites of "programming" is not a new concept. In fact, it could be argued that this is exactly how the process of standard compilation works. We have a high-level language, which is compiled into a low-level language, which is then executed on a machine.

However, the process of compilation is usually not considered to be part of the high-level program itself. Rather, it is the job of the compiler to figure out exactly what is the optimal representation of a high-level program in the low-level language. This is especially the case when the discrepancy between the levels becomes great, for example in functional languages such as Haskell or ML. In these languages, the programmer has very little control over the eventual representation of their program in the low-level language, unless they use specialised primitives which are meant to mirror the low-level capabilities of the target system.

This is what we are aiming to change with this formalism. We envision a process of compilation in which the programmer has a lot more control over the representation of their program in the low-level language, while still keeping a clear separation between that and the core logic of the high-level program itself. In a way, we are trying to create a principled and ubiquitous version of "directive annotations" in high-level languages which aim to address a particular low-level concern about a program (for example, whether to keep the field of a data type behind a pointer.)

5 TRANSLATION OF LOGIC AND DATA

In the present categorical presentation of this process, we want each program to construct its own customised compilation functor C. This functor will be constructed by the programmer, just like the rest of the high-level logic of the program.

One might remark that this sounds like a lot of work for the programmer. Imagine having to not only write a program, but also write a customised compiler for that very program, and doing that each time you want to write a program. However, we will see that this is not the case. In fact, we will see that the compilation functor can be constructed automatically based on a set of core "data type representations" that are provided by the programmer. These data type representations will be the only thing that the programmer will have to write, and they will be reusable across many different programs.

Furthermore, there will be a trade-off at play. The more custom parts of the compilation functor the programmer writes, the more control they have over the eventual representation, but the more coupling there is between the high-level program and the low-level representation. Crucially however, this coupling is never present in the high-level program itself, but rather sits alongside it.

6 ALGEBRA-COALGEBRA PAIRS AS THE PRIMITIVE OF REPRESENTATION

Consider that we have an endofunctor $F : \mathbb{S} \longrightarrow \mathbb{S}$ which admits a least fixpoint, which we will denote μF. This is a certain kind of free structure in $\mathbb{S}$: an inductive data type. We will mostly focus on least fixpoints, however it might be worth considering the dual case of greatest fixpoints as well.

We want to be able to represent this in $\mathbb{L}$. We will do this by defining an algebra and a coalgebra in $\mathbb{S}$ over a chosen object $I_F \in \mathrm{Ob}(\mathbb{S})$,

$$F(I_F) \underset{u}{\overset{w}{\rightleftarrows}} I_F \ . \tag{1}$$

This pair of morphisms allows us to "peel away" a layer of F from I_F, as well as "wrap" a layer of F around I_F. Conceptually, I_F represents an intermediary descriptive object which is used to gather and encode data about the structure of an inhabitant of μF.

Furthermore, I_F must satisfy a second property: there must exist another pair of morphisms

$$I_F \underset{i}{\overset{r}{\rightleftarrows}} Q(R_F) \ . \tag{2}$$

with the quoted object $Q(R_F)$, making it so that $R_F \in \mathrm{Ob}(\mathbb{L})$ is the chosen low-level representation of μF.

In general we do not expect that r and i, or w and u are strict inverses. In fact, them being strict inverses would imply a very trivial correspondence that probably does not contain significant information. However, we do expect that there exist two pairs of 2-morphisms

$$v : u \circ w \Rightarrow \mathrm{id}_{F(I_F)} \text{ and } v' : w \circ u \Rightarrow \mathrm{id}_{I_F} \tag{3}$$

and

$$\mu : i \circ r \Rightarrow \mathrm{id}_{I_F} \text{ and } \mu' : r \circ i \Rightarrow \mathrm{id}_{Q(R_F)} \ . \tag{4}$$

TODO: do we really need both directions here? These 2-morphisms are the key to the correspondence between the high-level and low-level representations. They ensure that no matter how different the composition of the encoding and decoding morphisms is from "doing nothing", the result always evaluates to the same term.

7 AN EXAMPLE: LISTS AS HEAP ARRAYS

We will define and work in the internal languages of $\mathbb{S}$ and $\mathbb{L}$.

For $\mathbb{S}$:

$$\begin{aligned}
&x, y && \text{(variables)}\\
&A, B ::= A \to B \mid A \times B \mid A + B \mid \mu x.A \mid 1 && \text{(types)}\\
&t, u, v ::= \lambda x.t \mid x \mid t\,u \mid (t, u) \mid \mathsf{p}_1(t) \mid \mathsf{p}_2(t) \mid \mathsf{inl}(t) \mid \mathsf{inr}(t) \mid \mathsf{case}(t, x.u, y.v) \mid \star && \text{(terms)}
\end{aligned}$$

For $\mathbb{L}$:

$$\begin{aligned}
&x && \text{(variables)}\\
&n && \text{(finite natural numbers, word size)}\\
&s ::= \varnothing \mid [s, t] && \text{(sequences)}\\
&A, B ::= A \multimap B \mid \Sigma x.A \mid \mathsf{W} \mid A^n \mid \Box A \mid \mathsf{I} && \text{(types)}\\
&t, u ::= \lambda x.t \mid x \mid t\,u \mid \langle n, t\rangle \mid n \mid s \mid \mathsf{box}(t) \mid \mathsf{letbox}(t, x.t) \mid \mathsf{letseq}(t, x.t) \mid \mathsf{num}(t) \mid \star && \text{(terms)}
\end{aligned}$$

8 SLICE CATEGORIES AND DEPENDENT TYPES

ACKNOWLEDGMENTS

Acknowledgments

Received 28 February 2024