Submitted to MFPS 2024

Staging Inductive Types to Optimised Data Structures

Constantine Theocharis Christopher Brown

University of St Andrews
St Andrews, Fife, UK

1 Introduction

The techinque of program staging aims to separate the high-level structure of a program in a way that is convenient for abstraction and manipulation, from the low-level eventual representation of the program that is efficient for machine execution. This is done by separating a language into two parts: the *meta* fragment and the *object* fragment. The meta fragment is the site in which the program is synthesised, and the object fragment is the output of the synthesis process. This is made possible by the ability to manipulate object-level fragments inside the meta language.

The approach we follow mirrors the setup of [8]. We consider a *two-level type theory* $\mathbb{G}$, with the following

2 Preliminaries

3 Technique

Consider a lambda calculus with endofunctors and least fixpoints $\mathbb{H}_\lambda$, defined by the grammar

$$\begin{aligned}
x, y, X & && \text{(variables)}\\
l & && \text{(labels)}\\
F & ::= \Lambda X.\, l\,(\, l : A^*\,)^* && \text{(type endofunctors)}\\
A, B & ::= X \mid A \to B \mid \mu F \mid F(A) && \text{(types)}\\
t, u, v & ::= \lambda x.t \mid x \mid t\,u \mid F/l \mid \textsf{case}\ t\ \textsf{of}\ (\, u \mapsto v\,)^* \\
& \quad \mid \textsf{let}\ x = t\ \textsf{in}\ u \mid \textsf{wrap} \mid \textsf{unwrap} \mid \textsf{fold} \mid \textsf{unfold}\,. && \text{(terms)}
\end{aligned}$$

We will usually abbreviate F/l as l when the functor is clear from the context.

In this language, we can define the natural numbers as a least fixpoint

$$\mathsf{Nat} := \mu(\Lambda X.\, \mathsf{Nat}\,(\mathsf{z} : (\,), \mathsf{s} : X))$$

and define the usual operations, such as addition

$$\begin{aligned}
\mathsf{add} := \textsf{fold}(\lambda\, x.\, \lambda\, y.\, \textsf{case}\ x\ \textsf{of}\ & \mathsf{z} \mapsto y \\
& \mathsf{s}\, x' \mapsto \mathsf{s}\,(\mathsf{add}\, x'\, y))
\end{aligned}$$

or multiplication

$$\mathsf{mult} := \lambda\, x.\, \lambda\, y.\, \textbf{case}\ x\ \textbf{of}\ \mathsf{z} \mapsto \mathsf{z} \\ \mathsf{s}\, x' \mapsto \mathsf{add}\, y\, (\mathsf{mult}\, x'\, y)\,.$$

4 Categorical model

In this section we explore a category-theoretic formulation of the process of compilation.

Definition 4.1 A *compilation functor* is a lax functor $\mathrm{C} : \mathbb{S} \longrightarrow \mathbb{L}$ between weak 2-categories $\mathbb{S}$ and $\mathbb{L}$.

Each of these categories is interpreted a language, where $\mathbb{S}$ is the source high-level language and $\mathbb{L}$ is the target low-level language. In the style of [CITE], the 2-category structure of these languages represents

- contexts in the language as objects,
- terms in the language as 1-cells, and
- term rewriting rules as 2-cells.

C being lax is because compositions of morphisms might only be preserved modulo reduction rules.

4.1 Quoting

One desirable feature of compilation for our concerns is the ability of the high-level language to represent "quoted" terms of the low-level language.

Definition 4.2 A *quoting* functor associated with a compilation functor C is an injective-on-objects, fully faithful functor Q, right adjoint to C,

$$\mathbb{S} \underset{\mathrm{Q}}{\overset{\mathrm{C}}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathbb{L}$$

such that the adjunction is strict. From this, it follows that the counit of the adjunction is a natural isomorphism (so that compilation cancels out quoting). It also follows that $\mathbb{L}$ is a full subcategory of $\mathbb{S}$.

Definition 4.3 A *free quoting structure* on a compilation functor $\mathrm{C} : \mathbb{S} \to \mathbb{L}$ consists of

- an extension $\mathbb{S}^\star$ of $\mathbb{S}$, constructed by adding a new object QA for each object $A \in \mathbb{L}$, and a new morphism $Qf : QA \to QB$ for each morphism $f : A \to B$ in $\mathbb{L}$, such that $Q\,\mathrm{id}_A = \mathrm{id}_A$, and $Q(f \circ g) = Qf \circ Qg$, and
- an extension $\mathrm{C}^\star : \mathbb{S}^\star \to \mathbb{L}$ of the compilation functor C which maps each QA to A and each Qf to f.

Lemma 4.4 *The functor* Q *defined by the free quoting structure by* $\mathrm{Q}(A) = QA$ *and* $\mathrm{Q}(f) = Qf$ *is a quoting functor.*

Proof. *TODO* □

4.2 Intermediate representations

Most practical compilation processes involve more than two languages; often a source language is transformed into a sequence of increasingly low-level intermediate languages, before finally being translated into the target language.

Definition 4.5 An *n-stage* compilation functor is a sequence of compilation functors

$$\mathbb{S} \xrightarrow{\mathrm{I}_1} \mathbb{I}_1 \xrightarrow{\mathrm{I}_2} \cdots \xrightarrow{\mathrm{I}_{n-1}} \mathbb{I}_{n-1} \xrightarrow{\mathrm{C}} \mathbb{L}$$

starting at $\mathbb{S}$, passing through a sequence of intermediate languages $\mathbb{I}_i$, and ending at $\mathbb{L}$.

4.3 Normalisation

Definition 4.6 A *normalisable* 2-category is a 2-category $\mathbb{C}$ with an endofunctor $\mathrm{Nm} : \mathbb{C} \longrightarrow \mathbb{C}$ such that

- If $\mathrm{Nm}(f) = g$ then exists a 2-cell $h : f \Rightarrow g$.
- $\mathrm{Nm} \circ \mathrm{Nm} = \mathrm{Id}$.

Definition 4.7 A normalisation functor $\mathrm{Nm} : \mathbb{S} \longrightarrow \mathbb{S}$ is *compilation-compatible* if there exists a lax natural isomorphism $\mathrm{C} \cong \mathrm{C} \circ \mathrm{Nm}$ in $[\mathbb{S}, \mathbb{L}]$.

4.4 Picking a compilation functor

To aid in the process of picking a compilation functor, we will define a category of representations of $\mathbb{S}$ in $\mathbb{L}$, denoted $\mathrm{Repr}^{\mathbb{S}}_{\mathbb{L}}$.

The categories $\mathbb{S}$ and $\mathbb{L}$ come with an identity-on-objects normalisation endofunctor, which we will denote Norm:

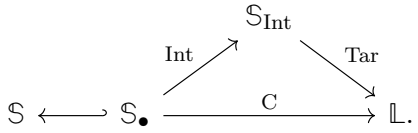
$$\mathrm{Norm}_{\mathbb{S}} : \mathbb{S} \longrightarrow \mathbb{S}$$
$$\mathrm{Norm}_{\mathbb{L}} : \mathbb{L} \longrightarrow \mathbb{L}$$

We require that $\mathbb{S}_\bullet$ is closed under the normalisation functor; reducing a representable program should yield another representable program. Finally, we require the following constraints for all 2-categories involved:

- If $\mathrm{Norm}(f) = g$ then exists a 2-cell $h : f \Rightarrow g$.
- $\mathrm{Norm} \circ \mathrm{Norm} = \mathrm{Id}$.

4.5 Intermediate representations

We will focus on the cases when the compilation functor C factors into two pieces through a subcategory $\mathbb{S}_{\mathrm{Int}}$, which we will call the *intermediate subcategory* of $\mathbb{S}$:



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 in which we execute our programs. 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 $\mathrm{C} : \mathbb{S}^\bullet \longrightarrow \mathbb{L}$, called the *compilation* functor. It takes a program in $\mathbb{S}$ and compiles it to a program in $\mathbb{L}$.
- A functor $\mathrm{Q} : \mathbb{L} \longrightarrow \mathbb{S}$, 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 $\mathrm{C}(f \circ g)$ is not necessarily the same as compiling $\mathrm{C}(f)$ and then $\mathrm{C}(g)$ separately, but there is a 2-morphism $\mathrm{C}(f) \circ \mathrm{C}(g) \Rightarrow \mathrm{C}(f \circ g)$ 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.

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

4.7 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.

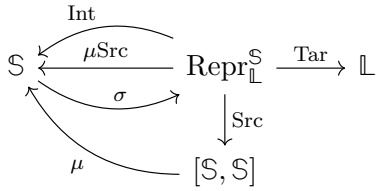
Commonly $\mathbb{L}$ will not be closed, but for this paper, we will assume it is. It might sometimes be a Kleisli category with respect to some monad (probably a monad holding the state of the stack/heap/environment/etc).

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.

4.8 The category $\mathrm{Repr}^{\mathbb{S}}_{\mathbb{L}}$

The data about how to translate high-level programs into low-level programs will be stored in a category $\mathrm{Repr}^{\mathbb{S}}_{\mathbb{L}}$.

Consider the following diagram:



(1)

The category $\mathrm{Repr}^{\mathbb{S}}_{\mathbb{L}}$ is the category of representations of $\mathbb{S}$ in $\mathbb{L}$. There are a few key functors here:

- $\mathrm{Src} : \mathrm{Repr}^{\mathbb{S}}_{\mathbb{L}} \longrightarrow [\mathbb{S}, \mathbb{S}]$ is the source functor. It takes a representation of $\mathbb{S}$ in $\mathbb{L}$ and returns the original high-level inductive type as an endofunctor in $[\mathbb{S}, \mathbb{S}]$. Can be considered one of the bundle projections.
- $\mathrm{Tar} : \mathrm{Repr}^{\mathbb{S}}_{\mathbb{L}} \longrightarrow \mathbb{L}$ is the target functor. It takes a representation of $\mathbb{S}$ in $\mathbb{L}$ and returns the low-level representation of the given high-level inductive type. Can be considered the other bundle projection.
- $\mu : [\mathbb{S}, \mathbb{S}] \longrightarrow \mathbb{S}$ is the functor that takes an endofunctor in $[\mathbb{S}, \mathbb{S}]$ and returns the least fixpoint of that endofunctor in $\mathbb{S}$.
- $\sigma : \mathbb{S} \longrightarrow \mathrm{Repr}^{\mathbb{S}}_{\mathbb{L}}$ is the functor that takes an object in $\mathbb{S}$ and returns the representation of that object in $\mathbb{L}$. It is a section of $\mu\mathrm{Src}$.

Each object $R \in \mathrm{Repr}^{\mathbb{S}}_{\mathbb{L}}$ contains the following data:

- An object $\mu\mathrm{Src}_R \in \mathbb{S}$ which is the source type.
- An endofunctor $\mathrm{Src}_R : \mathbb{S} \longrightarrow \mathbb{S}$ whose fixpoint is $\mu\mathrm{Src}_R$.
- An object $\mathrm{Tar}_R \in \mathbb{L}$ which is the low-level representation of $\mu\mathrm{Src}_R$.
- An object $\mathrm{Int}_R \in \mathbb{S}$ which represents an intermediate descriptive object that captures information about $\mu\mathrm{Src}_R$ during a translation process. In simple cases the intermediate object will be something like $\mathrm{Int}_R = \mu\mathrm{Src}_R + Q\mathrm{Tar}_R$.
- A morphism
$$\mathrm{IntAlg}_R \in \mathbb{S}(\mathrm{Src}_R(\mathrm{Int}_R), \mathrm{Int}_R)$$
which calculates the structure of the intermediate descriptive object by a fold over the syntactical term structure of the high-level program. For example, this can condense sequences of constructors of the high-level inductive type into a single constructor of the intermediate descriptive object, depending on what the algebra dictates.

- A morphism
$$\mathrm{IntCoAlg}_R \in \prod_{L \in \mathrm{Repr}_{\mathbb{L}}^{\mathbb{S}}} \mathbb{S}(\mathrm{Int}_L^{\mathrm{Src}_R(\mathrm{Int}_R)}, \mathrm{Int}_L^{\mathrm{Int}_R})$$
which calculates the structure of a function on the intermediate object, given a function on the source object. It is almost a coalgebra, if the bound of the product was over all of $\mathbb{S}$ rather than just the image of Int. Syntactically, it is used to transform pattern matches on the high-level inductive type into pattern matches on the intermediate descriptive object.
- A morphism
$$\mathrm{Comp}_R \in \mathbb{S}(\mathrm{Int}_R, Q\mathrm{Tar}_R)$$
that "compiles" the intermediate descriptive object into the low-level representation (quoted).
- A morphism
$$\mathrm{Decomp}_R \in \mathbb{S}(Q\mathrm{Tar}_R, \mathrm{Int}_R)$$
that "decompiles" the low-level representation into the intermediate descriptive object.

Each morphism $f \in \mathrm{Repr}_{\mathbb{L}}^{\mathbb{S}}(R, R')$ contains the following data:

- TODO: Basically a morphism of each of the above data

From all this data, we should be able to construct the following functions (morphisms in **Set**):

- A function
$$\mathrm{int}_{\sigma,\Gamma,T} : \mathbb{S}(\Gamma, T) \to \mathbb{S}(Q\mathrm{Tar}_{\sigma\Gamma}, \mathrm{Int}_{\sigma T})$$
which applies all the IntAlg and IntCoAlg morphisms by folding and unfolding over the syntactical term structure of the high-level program.
- A function
$$\mathrm{comp}_{\sigma,\Gamma,T} : \mathbb{S}(Q\mathrm{Tar}_{\sigma\Gamma}, \mathrm{Int}_{\sigma T}) \to \mathbb{S}(Q\mathrm{Tar}_{\sigma\Gamma}, Q\mathrm{Tar}_{\sigma T})$$
which applies the Comp morphism to the intermediate descriptive object.
- A function
$$\mathrm{un}Q_{\sigma,\Gamma,T} : \mathbb{S}(Q\mathrm{Tar}_{\sigma\Gamma}, Q\mathrm{Tar}_{\sigma T}) \to \mathbb{L}(\mathrm{Tar}_{\sigma\Gamma}, \mathrm{Tar}_{\sigma T})$$
which essentially unquotes the result, yielding a term in the low-level language.

Finally, for a given section σ of the bundle of representations, we should be able to define the functor C as follows:

$$\begin{aligned} \mathrm{C}_\sigma &: \mathbb{S} \longrightarrow \mathbb{L} \\ \mathrm{C}_\sigma(T) &:= \mathrm{Tar}_{\sigma T} \\ \mathrm{C}_\sigma(f) &:= \mathrm{un}Q_{\sigma,\Gamma,T} \circ \mathrm{comp}_{\sigma,\Gamma,T} \circ \mathrm{int}_{\sigma,\Gamma,T}(f) \end{aligned}$$

NOTICE: We have not at all defined what the internal language of $\mathbb{S}$ or $\mathbb{L}$ looks like. An advantage of this formalism is that the compilation process in terms of algebras and coalgebras can be defined independently of the actual syntax and semantics of the languages (modulo some requirements such as cartesian closedness for $\mathbb{S}$).

4.9 Properties of the compilation functor

The compilation functor should be coherent with respect to operational reduction in both languages.

We can draw the following lax-commutative square:

$$\begin{array}{ccc} \mathbb{S} & \xrightarrow{\mathrm{eval}} & \mathbb{S} \\ \downarrow{\scriptstyle \mathrm{C}} & \swarrow & \downarrow{\scriptstyle \mathrm{C}} \\ \mathbb{L} & \xrightarrow{\mathrm{run}} & \mathbb{L} \end{array} \tag{2}$$

5 Properties

6 Examples

6.1 Manual boxing and sequences from lambda calculus with simple constructions

We will define and work in the internal languages of $\mathbb{S}$ and $\mathbb{L}$. By internal language we mean that the objects of each category are the types, and the morphisms are the terms within a given context.

For example, a morphism in $\mathbb{S}$

$$f \in \mathbb{S}(\Gamma, A)$$

will correspond to a term inside a context in the internal language of $\mathbb{S}$,

$$\Gamma \vdash f : A$$

.

6.2 Definition of $\mathbb{S}$

$$\begin{array}{rll} x, y, X & & \text{(variables)} \\ L & & \text{(type labels)} \\ F & ::= \Lambda X.\, L\,(\,x\,A^*\,)^* & \text{(type endofunctors)} \\ A, B & ::= X \mid A \to B \mid \mu F \mid F(A) \mid \mathbf{Q}C & \text{(types)} \\ t, u, v & ::= \lambda x.t \mid x \mid t\,u \mid F/x \mid \textsf{case}\ t\ \textsf{of}\ (\,u \mapsto v\,)^* \mid \mathbf{q}(c) & \\ & \quad \mid \textsf{let}\ x = t\ \textsf{in}\ u \mid \textsf{wrap} \mid \textsf{unwrap} \mid \textsf{fold} \mid \textsf{unfold} & \text{(terms)} \end{array}$$

Why this language?

- We want to be able to express inductive data types $\mu x.A$, for lists, numbers, trees, etc.
- We also want to be able to handle quoted terms of the lower language: $\mathbf{Q}C/\mathbf{q}(t)$, so that we can define translation functions in the language.
- We want to be able to explicitly label certain types and their inhabitants $(l, A)/(l, t)$, even though they might be functionally identical to some others. This is so that we can consider them as different objects in the category, and thus have the compilation functor produce different results for each one of them.
- We want to be able to express the usual constructs of a functional language: functions, pairs, sums, recursion, etc.

Still to do: typing rules, operational semantics.

Still to do: need an extra calculus of representations so that we can define the extra data from $\mathrm{Repr}^{\mathbb{S}}_{\mathbb{L}}$ i.e. algebras and coalgebras.

Typing rules (assuming weakening, contraction and, exchange) are below. Contexts are visually shown as lists of pairs of variables and type, but when considering $\mathbb{S}$ each morphism is identified modulo alpha-equivalence.

$$\mathrm{List}_T = \mu(\Lambda X.\mathrm{List}_T\,(\mathrm{nil}, \mathrm{cons}\,T\,X))$$

$$\textsc{Fn}\ \frac{\Gamma \vdash A\,\mathrm{type} \quad \Gamma \vdash B\,\mathrm{type}}{\Gamma \vdash A \to B\,\mathrm{type}} \qquad \textsc{TyWeak}\ \frac{\Gamma \vdash A\,\mathrm{type}}{\Gamma \vdash A\,\mathrm{type}[X]}$$

$$\textsc{TyId}\ \frac{}{\Gamma \vdash X\,\mathrm{type}[X]} \qquad \textsc{TySub}\ \frac{\Gamma \vdash A\,\mathrm{type}[X] \quad \Gamma \vdash B\,\mathrm{type}}{\Gamma \vdash A[X \mapsto B]\,\mathrm{type}}$$

$$\textsc{Endo}\ \frac{\forall i \in I, j \in J_i.\ \Gamma \vdash A_{ij}\,\mathrm{type}[X]}{\Gamma \vdash (\Lambda X.\, L\,(x_i\,(A_{ij})_{j \in J_i})_{i \in I})(A)\,\mathrm{type}} \qquad \textsc{Mu}\ \frac{\Gamma \vdash F(A)\,\mathrm{type}}{\Gamma \vdash \mu F\,\mathrm{type}}$$

$$\textsc{Var}\ \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \qquad \textsc{Lam}\ \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : A \to B} \qquad \textsc{App}\ \frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f\,a : B}$$

$$\textsc{Ctor}\ \frac{F = \Lambda X.\, L\,(x_i\,(A_{ij})_{j \in J_i})_{i \in I} \quad \Gamma \vdash F(A)\,\mathrm{type} \quad \exists i \in I, j \in J_i.\ \Gamma \vdash a_{ij} : A_{ij}[X \mapsto A]}{\Gamma \vdash F/x_i\,(a_{ij})_{j \in J_i} : F(A)}$$

$$\textsc{Case}\ \frac{F = \Lambda X.\, L\,(x_i\,(A_{ij})_{j \in J_i})_{i \in I} \quad \Gamma \vdash t : F(T) \quad \forall i \in I.\ \Gamma, (y_j : A_{ij}[X \mapsto T])_{j \in J} \vdash u_i : U}{\Gamma \vdash \mathbf{case}\ t\ \mathbf{of}\ (\,x_i(y_j)_{j \in J_i} \mapsto u_i\,)_{i \in I} : U}$$

$$\textsc{Let}\ \frac{\Gamma \vdash a : A \quad \Gamma, x : A \vdash b : B}{\Gamma \vdash \mathbf{let}\ x = a\ \mathbf{in}\ b : B} \qquad \textsc{Fold}\ \frac{\Gamma \vdash F(A)\,\mathrm{type}}{\Gamma \vdash \mathbf{fold} : (F(A) \to A) \to \mu F \to A}$$

$$\textsc{Unfold}\ \frac{\Gamma \vdash F(A)\,\mathrm{type}}{\Gamma \vdash \mathbf{unfold} : (A \to F(A)) \to A \to \mu F}$$

Fig. 1. Typing rules for $\mathbb{S}$

6.3 Definition of $\mathbb{L}$

$$\begin{aligned}
x, y & && \text{(variables)}\\
n & && \text{(finite natural numbers, word size)}\\
s &::= \varnothing \mid [s, t] && \text{(sequences)}\\
S &::= \varnothing \mid [S, C] && \text{(type sequences)}\\
C, D &::= C \multimap D \mid \Sigma x_n.C \mid \mathbf{W} \mid S[n] \mid C^n \mid \Box C \mid \mathbf{I} && \text{(types)}\\
t, u &::= \lambda x.t \mid x \mid t\,u \mid \langle n, t\rangle \mid \mathbf{letpair}(t, u, x.y.v) \mid n \mid \mathbf{box}(t) && \\
&\quad \mid \mathbf{letbox}(t, u, x.v) \mid s \mid \mathbf{letseq}(t, u, x.v) \mid \mathbf{num}(t) \mid \star && \text{(terms)}
\end{aligned}$$

Why this language?

- Once again we want a language based on the lambda calculus—for now we will not go too low-level.
- We want this language to be able to represent boxed types, because we want to have lower-level control over the memory representation of inductive data types (usually in the form $(l, \mu x.A)$).
- We allow a restricted form of dependent types in the form of $\Sigma x_n.C$ types parameterised over some word size $\mathbf{W}$. This is because sequence types C^n are indexed by W, and if we have some sequence that is of some runtime size, we want to be able to represent the fact that some stored size is the size of that sequence.
- We want to be able to represent sequences C^n for the reason above. We can also have sequence types which can be used to model disjoint unions (same as unions in C). The only difference here is we can represent tagged unions by using the Σ type: $\Sigma x_2.[A, B][x]$ is kind of like $A + B$ in the high-level language.
- Due to the presence of boxed types, we want the language to be linear. In other words we do not allow weakening or contraction of the context (unclear if we want to allow contraction or not—we do not have

explicit destructors so maybe we really want an affine system).

- We don't care about labels here since that is only a concern during compilation.
- We have different let expressions for the different linear types, to be able to extract their inner data all at once or not at all.
- The category $\mathbb{L}$ is still a monoidal closed category (specifically a symmetric monoidal closed category), so we still allow lambdas to capture variables (i.e. closures). Compiling these to boxed closures is a different matter..

Still to do: typing rules, operational semantics.

6.4 Lists to arrays and natural numbers to big unsigned integers

$$\text{List}(A) := \mu X.(A \times X + 1) \qquad \text{(lists)}$$
$$\text{Array}(A) := \Sigma n.\square A^n \qquad \text{(arrays)}$$

$$\text{Nat} := \mu X.(1 + X) \qquad \text{(numbers)}$$
$$\text{BigUInt} := \Sigma n.\square \mathbf{W}^n \qquad \text{(big unsigned integers)}$$

Fully write out the representations of these types in $\mathbb{S}$ and $\mathbb{L}$.
Find less trivial examples.

6.5 $\mathbb{S}$ and $\mathbb{L}$ in one using dependent types

We should be able to embed the data of $\text{Repr}_{\mathbb{L}}^{\mathbb{S}}$ inside $\mathbb{S}$ if the latter has sufficient quantification structure. In other words, its internal language is some kind of dependent type theory.

7 Related work

TODO: BibLatex
Bit Stealing Made Legal: https://dl.acm.org/doi/pdf/10.1145/3607858
Type fusion: https://www.cs.ox.ac.uk/ralf.hinze/publications/AMAST10.pdf
Unrolling lists: https://dl.acm.org/doi/10.1145/182409.182453
An automatic object inlining optimization and its evaluation: https://dl.acm.org/doi/10.1145/349299.349344
Selection of representations for data structures: https://dl.acm.org/doi/pdf/10.1145/872736.806944
Linear/non-Linear Types For Embedded Domain-Specific Languages: https://core.ac.uk/download/pdf/214213829.pdf
Staged compilation with two-level type theory: http://arxiv.org/abs/2209.09729

8 Conclusion

8.1 Ideas about future work

- Algebra-coalgebra pairs as a way to interpret inductive data types and their recursive control-flow in low-level categories. (this work)
- Coherence conditions on an algebra-coalgebra pair to ensure that a chosen representation is faithful. (this work ?)
- Custom shortcuts for derived transformations based on the algebra-coalgebra pairs, for fine-tuning the representation of compound operations.

- Algebra-coalgebra pair generators for equivalence classes of isomorphic data types, to automatically generate representations of commonly seen structures.
- Solving for the representation that optimises some metric of a chosen set of operations (e.g. space complexity, time complexity, constant factors etc.) through various static and dynamic techniques
- Restriction of the context category of the source language in terms of its monoidal structure, to prevent certain low-level operations from being expressible at all.
- Relaxing well-foundedness, to model more complicated control-flow structures such as coroutines, continuations, and so on.

9 [TEMP] Notes

10 Document plan

We probably need to have the following sections:

- Introduction
- Preliminaries – category theory, conventions, basic structure of the problem
- Categorical model of (compilation?) Maybe it should be called something else
- An example with the simply-typed lambda calculus and a language with explicit boxing
- Embedding the representations into the high-level language – dependent types
- Related work
- Conclusion + future
- Would be nice to have an artifact: Agda formalisation?

References

[1] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor Mcbride, and Peter Morris. Indexed containers. *J. Funct. Programming*, 25:e5, January 2015.

[2] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-Level type theory and applications. May 2017.

[3] S Boulier. Extending type theory with syntactic models. November 2018.

[4] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 182–194, New York, NY, USA, January 2017. Association for Computing Machinery.

[5] Edwin C Brady and Kevin Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. *SIGPLAN Not.*, 45(9):297–308, September 2010.

[6] Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. April 2019.

[7] Brandon Hewer and Graham Hutton. Quotient haskell: Lightweight quotient types for all. *Proc. ACM Program. Lang.*, 8(POPL):785–815, January 2024.

[8] András Kovács. Staged compilation with two-level type theory. September 2022.

[9] M Sato, Takafumi Sakurai, and Yukiyoshi Kameyama. A simply typed context calculus with first-class environments. *J. Funct. Log. Prog.*, pages 359–374, March 2001.

[10] Marcos Viera and Alberto Pardo. A multi-stage language with intensional analysis. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. unknown, October 2006.

[11] Jeremy Yallop. Staged generic programming. *Proc. ACM Program. Lang.*, 1(ICFP):1–29, August 2017.

[12] Robert Atkey Sam Lindley Yallop. Unembedding Domain-Specific languages. https://bentnib.org/unembedding.pdf. Accessed: 2024-2-22.