Algebra of Parallel Programming in Agda – Text!

Thomas Bååth Sjöblom

September 4, 2012

1 Introduction

[TODO: Safety] [TODO: Easier (maybe) to prove correctness along the way] [TODO: Parallel] [TODO: Parsing]

2 Introduction stuff

2.1 Agda

Agda is a dependently typed programming language invented at Chalmers Norell [2007].

2.2 Category theory

Category theory is a theory for unifying mathematics by placing importance on things that are kind of like functions.

There are two reasons for introducing category theory. The first is that it will give us a clean and point-free way of reasoning about programs. The second is that it provides a semantics for the [TODO: develop above sentences, AoP works in **Fun**, but what is up with the initial algebra semantics? are lists thought of as sets $\subseteq$ **Fun**, the set of all lists with elements in A is obviously a set, but is it the best view, what. no wait, nothing important here D:]

The presentation here is based on (the early chapters of) Awodey [2010] and Mac Lane [1998]. The material can also be found in Bird and Moor [1997], but the authors have reversed the function arrows and writes $f : Y \leftarrow X$ for $f : X \rightarrow Y$, something that we feel actually reduces the readability a great deal.

Definition 2.1. A category $\mathcal{C}$ is a collection $\mathcal{O}_\mathcal{C}$ of objects and a collection $\mathcal{A}_\mathcal{C}$ of arrows $f : X \rightarrow Y$, where $X \in \mathcal{O}_\mathcal{C}$ is the domain of f and $Y \in \mathcal{O}_\mathcal{C}$ is the codomain of f. The collections $\mathcal{O}_\mathcal{C}$ and $\mathcal{A}_\mathcal{C}$ are required to satisfy the following:

- For every object $X \in \mathcal{O}$, there is an identity arrow $\mathrm{id}_X : X \rightarrow X$.
- For every pair $f : A \rightarrow B$ and $g : B \rightarrow C$ of arrows, there is an arrow $g \circ f : A \rightarrow C$

- For every three arrows $f : A \to B$, $g : B \to C$ and $h : C \to D$, we have $h \circ (g \circ f) = (h \circ g) \circ f$
- For every arrow $f : A \to B$, we have $f \circ id_A = id_B \circ f = f$.

These requirements just say that the arrows should behave as functions with compositions and identity functions.

Example 2.1. Indeed, the basic example we will work with is the category **Fun** where the objects are sets and the arrows are functions. This is a category because [TODO: fulfils the axioms].

Traditionally in category theory (see for example Awodey [2010] and Mac Lane [1998]), this category is called **Set**, but we will call it **Fun** for three main reasons:

1. It's the name used in Bird and Moor [1997].
2. In category theory, the arrows are usually the most importand part of a category.
3. Both it and the next category we will introduce have the same objects.

Example 2.2. A second example of a category which we will expand on is the category **Rel** where the objects are again sets, but this time, the arrows are relations. We write $R : Y \leftarrow X$, where X is the domain and Y the codomain of R, following the notation for relations used in Mu et al. [2009]. Here, composition of $R : Z \leftarrow Y$ and $S : Y \leftarrow X$ is defined as $R \circ S = \{(a, c) : \exists b : aRb\}$

Concretely, relations are defined as subsets of $Y \times X$ [TODO or other way around?], but this is a view that we try to avoid with by introducing category theory

More examples of categories include sets with some structure:

Example 2.3. In the category **Grp**, the objects are groups and the arrows are group homomorphisms, i.e., functions ϕ that respect the group operations: $\phi : (G, +) \to (H, \times)$ such that $\phi(x + y) = \phi(x) \times \phi(y)$ and $\phi(-x) = \phi(x)^{-1}$.

Example 2.4. Other examples along the same lines include for example the category **Rng** of rings, and the category **Mon** of monoids, in both of which the arrows are the [TODO word] homomorphisms. One can also go further and add other kinds of structure, to get for exampl **Top**, the category of topological groups, where the objects are topological groups (groups where the operations are continous) [TODO continue]

[TODO two more examples: poset and matrixes] And finally, we present two examples that are quite unlike the previous ones, which present the kind of things that are included in the category definition. Since they are very different from functions, they motivate further specialization of the category definition. We also note that they (like all the categories we have mentioned) have some relation to the later chapters in this thesis.

Example 2.5. The first strange example is a so called poset category. Let X be any partially ordered set, that is, a set with a partial order, that is a relation $\leq\ : Y \leftarrow X$ satisfying the following:

1. For every $x \in X$, $x \leq x$.
2. For every x, y, if $x \leq y$ and $y \leq x$, then $x = y$.
3. For every x, y, z, if $x \leq y$ and $y \leq z$, then $x \leq z$.

Then we get a category if we take as objects elements $x \in X$ and include an arrow $x \rightarrow y$ if and only if $x \leq y$. By 1, there is an identity arrow for every object. By 3, if there are arrows $x \rightarrow y$ and $y \rightarrow z$, there is an arrow $x \rightarrow z$. Finally, we note that [?]. Hence This satisfies the axioms for a category. Requirement 2 guarantees that [?]

Example 2.6. In our final example, we let the objects be the natural numbers $\mathbf{N}$, and let the arrows be matrixes with coefficients in some (associative) ring. Hence this, too, forms a category.

[TODO subcategory, important for defining datatypes in **Rel**, example of **Ab**]

There are a number of operations that can be performed to create new categories:

Definition 2.2. A *functor* $F : \mathcal{C} \rightarrow \mathcal{D}$ is a pair of mappings: $F_{\mathcal{O}} : \mathcal{O}_{\mathcal{C}} \rightarrow \mathcal{O}_{\mathcal{D}}$ and $F_{\mathcal{A}} : \mathcal{A}_{\mathcal{C}} \rightarrow \mathcal{A}_{\mathcal{D}}$ satisfying

- If $f : X \rightarrow Y$, then $F_{\mathcal{A}} f : F_{\mathcal{O}} X \rightarrow F_{\mathcal{O}}$.
- If $f : X \rightarrow Y$ and $g : Y \rightarrow Z$, then $F_{\mathcal{A}}(g \circ f) = (F_{\mathcal{A}} g) \circ (F_{\mathcal{A}} f)$.
- The identity arrow $\mathrm{id}_X : X \rightarrow X$ is mapped to the identity arrow in $F_{\mathcal{O}} X$: i.e., $F_{\mathcal{A}}\,\mathrm{id} = \mathrm{id}_{F_{\mathcal{O}} X}$

In what follows, we will refer to both mappings $F_{\mathcal{O}}$ and $F_{\mathcal{A}}$ with just F.

Some

Definition 2.3. A funtor $F : \mathcal{C}C \rightarrow \mathcal{C}C$ is called an *endofunctor*.

[TODO: Examples of functors: product, sum, expand to polynomial functors => regular functors? what are they and are they included in GADT-paper] [TODO: Natural transformation]

[TODO: Algebra]

Definition 2.4. An algebra is

2.3 Algebra of Programming

One point of the introduction of category theory was to allow us to generalize som common list functions to arbitrary datatypes.

An inductive datatype is a

[TODO: What happens to algebra of programming when using dependent types? There are the papers: Gambino and Hyland [2004] and Hamana and Fiore [2011]

2.4 Parsing

Parsing is the process of turning a sequence of tokens (for example a string contianing a computer program [TODO]) into a data structure suitable for [something]. It is an important step in compilation. [TODO : need to learn more about basic parsing stuff]

However, in this thesis, we will not focus much on the parsing stuff. This chapter is here mainly as a motivation for what is to come (and so that I will learn some real world stuff!) [TODO: Example]

2.4.1 Grammar

A grammar is a collection of rules, called productions, that The productions are made up of terminals and nonterminals. Additionally, there is a start symbol that specifies [TODO WHAT]

Definition 2.5. A *grammar* is a tuple (N, T, S, P) where N is the set of nonterminals, T is the set of terminals, S is the start symbol, and P is the set of productions. A *production* is a [TODO thing] sequence of terminals and nonterminals followed by and arrow and another sequence of terminals and nonterminals. [TODO write in symbols] $t_1 \cdots t_n \to s_1 \cdots s_m$, t_i, $s_i \in N \cup T$

We will only consider context free grammars. These are grammars A language generated by a context free grammar is called a context free language. It is well known that every context free grammar can be turned into a

2.4.2 Special kind of grammar – Chomsky normal form

It is well known [cite?] that any context free grammar can be turned into one in Chomsky normal form, that is a grammar where [TODO WHAT IS THIS?]. For grammars in Chomsky normal form, it is possible to

2.5 Transitive Closure

The result of the parse is the set of all [TODO expand]

2.6 CYK Algorithm

[TODO what is it? valiant is related to it, or they wouldn't be on same wikipedia page!]

2.7 Matrices

We define a type of abstract matrices,

Definition 2.6. A matrix $A = (a_{ij})$ is functions $Fin \to Fin \to R$

or, in Agda

```
Matrix :    Set
Matrix m n = (i : Fin m)  (j : Fin n)  R
```

We include also our definitions of addition and multiplication of matrices as examples of how things look.

```
  -- Matrix addition
_M +_ : {m n} → Matrix m n → Matrix m n → Matrix m n
_M +_ A B =  i j  (A i j) R + (B i j)
  -- Matrix multiplication
_M ∗_ : {m n p}  Matrix m n  Matrix n p  Matrix m p
A M ∗ B =  i j  ( k  A i k)  ( k  B k j)
```

2.8 Triangular matrices

One of the main objects we will work with in this thesis is triangular matrices, and we will only ever consider upper triangular matrices with only zeros on the diagonal. For brevity, and because of the similarity with the Agda code we will present soon, we therefore make the following definition.

Definition 2.7. A matrix $A = (a_{ij})$ is *triangular* if all its entries on or below the diagonal are 0. That is, A is triagular if $i \geq j$ implies that $a_{ij} = 0$.

We generalize this to the following.

Definition 2.8. A matrix $A = (a_{ij})$ is *triangular of degree d* if all its entries fewer than d steps above the diagonal are 0. That is, A is triangular if $d > j - i$ implies that $a_{ij} = 0$.

Remark. Note that an $n \times n$ matrix that is triangular of degree $d \geq n$ equals the zero matrix. [TODO: Insert agda proof]

We prove the following lemma both formally and informally

Lemma 2.9. *If A is an $m \times n$ matrix that is triangular of degree d_1 and B is and $n \times p$ matrix that is triangular of degree d_2, then the product AB is an $m \times p$ matrix that is triangular of degree $d_1 + d_2$.*

Proof. Let $(c_{ij}) = AB$, and let k and l be such that $d_1 + d_2 > k - l$. We have that

$$c_{kl} = \sum_{i=1}^{n} a_{ki} b_{il} \tag{1}$$

We want to show that $c_{kl} = 0$.

□

Heh, triangularity is a homomorphism from matrices with multiplication to $\mathbf{Z}$ with addition

2.9 Valiants Algorithm

Valiant's algorithm was initially introduced to show that parsing could be done as quickly as matrix multiplication. It turns out to be an algorithm where most of the work happens in parallel, and this might be useful for developing parallel parsers.

The algorithm takes as input an upper triangular $n \times n$ matrix A (with 0:s on the diagonal):

$$A = \begin{pmatrix} 0 & a_{1\,2} & \dots & a_{1\,n} & \\ \vdots & 0 & a_{2\,3} & \dots & a_{2\,n} \\ \vdots & & & & \end{pmatrix} \tag{2}$$

and splits the matrix into four parts $A_{1\,1}$, $A_{1\,2}$, $A_{2\,1}$ and $A_{2\,2}$ along the diagonal:

$$A = \begin{pmatrix} A_{1\,1} & A_{1\,2} \\ A_{2\,1} & A_{2\,2} \end{pmatrix} = \begin{pmatrix} A_{1\,1} & A_{1\,2} \\ 0 & A_{2\,2} \end{pmatrix} \tag{3}$$

so that $A_{1\,1}$ and $A_{2\,2}$ are both upper triangular

[TODO requirements on the operations, is it enough to have the ring stuff, does + need to be idempotent? (like union) because otherwise, it seems like the algorithm will fail when applied to a non-associative ring (there will be $k \cdot (x_1 \cdots x_k)$ instead of $(x_1 \cdots x_k)$). So I don't think the algorithm actually works even on non-associative rings, because we can't have inverses for addition (since $x + x = x$ we get that $x = x + (x - x) = (x + x) - x = x - x = 0$ for all x, bad.]

3 Work

Our goal is to prove the correctness of Valiant's algorithm, and to do this by presenting a derivation of it from a sensible specification. The first step to do this is to actually formulate the algorithm in Agda, in a way that avoids explicit recursion (using ideas from Bird and Moor [1997]. The next (and final) step is then to find a derivation from the specification to the algorithm. [TODO : related to real programming? not very maybe since we know the final algorithm]

3.1 Formulation

The algorithm is made up of two parts, what we refer to as the recursion step and the overlap step. We note that the overlap step is also recursive, and that it is there that the most work is needed to find an appropriate formulation.

We introduce two main recursive datatypes, *Tri* and *SplitMat*. The datatype *Tri* is esentially a binary tree with a *SplitMat* in each splitting, and empty leaves. It is used to represent upper triangular matrixes with zero diagonal and elements from **R**. The type *SplitMat* on the other hand is essentialy a tree with four subtrees at each splitting (and some extra stuff to make the matrix dimensions fit) and information in the leaves. It is to represent . [TODO why the different tree formats?]. In Agda, they are defined by

```
data SplitVec : Splitting  Set where
  one : (x : R)  SplitVec one
  two : {s s}  SplitVec s  SplitVec s  SplitVec (deeper s s)
data SplitMat : Splitting  Splitting  Set where
  Sing : (x : R)  SplitMat one one
  RVec : {s s}  SplitVec (deeper s s)  SplitMat one (deeper s s)
  CVec : {s s}  SplitVec (deeper s s)  SplitMat (deeper s s) one
  quad : {r r c c}  SplitMat r c  SplitMat r c
    SplitMat r c  SplitMat r c
    SplitMat (deeper r r) (deeper c c)
data Tri : Splitting  Set where
  one : Tri one
  two : {s s}  Tri s  SplitMat s s
    Tri s
    Tri (deeper s s)
```

If we allow arbitrary recursion, we can express the algorithm as:

```
  -- Recursion step:
valiant : {s} → Tri s → Tri s
valiant one = one
valiant (two A C B) = two A′ (valiantOverlap A′ C B′) B′
  where A′ = (valiant A)
    B′ = (valiant B)
  -- Overlap step:
valiantOverlap : {s s} → Tri s → SplitMat s s → Tri s → SplitMat s s
valiantOverlap {one} {one} A′ C B′ = C
valiantOverlap {one} {deeper s s} A′ (RVec v) B′ = RVec (vectmul v B′)
valiantOverlap {deeper s s} {one} A′ (CVec v) B′ = CVec (tvecmul A′ v)
valiantOverlap {deeper s s} {deeper s s} (two A A A) (quad C C C C) (two B B B) = quad X X X
  where X = valiantOverlap A C B
    X = valiantOverlap A (splitAdd C (splitMul A X)) B
    X = valiantOverlap A (splitAdd C (splitMul X B)) B
    X = valiantOverlap A (splitAdd C (splitAdd (splitMul A X) (splitMul X B))) B
```

Immediately, it is fairly obvious that the recursion step is simply a catamorphism. That is, we define a function *foldTri*:

```
foldTri : {b} {s : Splitting} {B : Splitting → Set b} (one′ : B one) (two′ : {s s} → B s → SplitMat s
foldTri one′ two′ one = one′
foldTri one′ two′ (two T R T) = two′ (foldTri one′ two′ T) R (foldTri one′ two′ T)
```

Which allows us to express the recursion step as

```
  -- Helper function for overlap
valiantOverlap' : {s s} → Tri s → SplitMat s s → Tri s → Tri (deeper s s)
valiantOverlap' T R T = two T (valiantOverlap T R T) T
valiantFold : {s}  Tri s  Tri s
valiantFold = foldTri one valiantOverlap'
```

The overlap step on the other hand requires quite a bit of work. First we note that the result of computing the transitive closure X_1 of C_1 is required when computing the closures X_2 and X_3 of C_2 and C_3 respectively. And that X_2 and X_3 are required for computing the transitive closure X_4 of C_4.

3.1.1 Nested Recursion

Even if we ignore the problem of sharing – ideally, we do not want to recompute the value of X_1 in the calculation of both X_2 and X_3, we still have the problem that we need to include a nested recursive call somehow in the algorithm (we want to compute $f(g(f(x)))$ for some particular f and g). For the computation of X_4 we actually need another level of nested recursion.

This seems similar to course-of-value recursion [thanks Patrik], which has been studied by **?**. However, in that case, the components involved are all sub structures of the input structure. Here, the recursion proceeds on elements that are structurally smaller but not sub structures (even though they are shaped like them). Another thing to note is that course-of-value recursion is equivalent to primitive recursion [WIKIPEDIA (only natural numbers)].

Since the recursive calls all happen on smaller structures, we can be sure that the function terminates. Maybe this is something that needs dependent types to work properly?

3.2 Derivation

3.3 Method 1

4 Other stuff

"Bird and Moor" or "Bird and de Moor"

References

Steve Awodey. *Category theory.* Oxford University Press, Oxford, 2nd ed. edition, 2010. ISBN 978-0-19-958736-0 (hbk.).

Richard Bird and Oege de Moor. *Algebra of programming.* Prentice Hall, London, 1997. ISBN 0-13-507245-X.

Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In *Types for proofs and programs*, volume 3085 of *Lecture Notes in Comput. Sci.*, pages 210–225. Springer, Berlin, 2004. doi: 10.1007/978-3-540-24849-1_14. URL `http://dx.doi.org/10.1007/978-3-540-24849-1_14`.

Makoto Hamana and Marcelo Fiore. A foundation for gadts and inductive families: dependent polynomial functor approach. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, WGP '11, pages 59–70, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0861-8. doi: 10.1145/2036918.2036927. URL `http://doi.acm.org/10.1145/2036918.2036927`.

Saunders Mac Lane. *Categories for the working mathematician.* Springer, New York, 2. ed. edition, 1998. ISBN 0-387-98403-8 (hardcover : alk. paper) ;.

Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. Algebra of programming in agda: Dependent types for relational program derivation. *Journal of Functional Programming*, 19(05):545–579, 2009. doi: 10.1017/S0956796809007345.

Ulf Norell. *Towards a practical programming language based on dependent type theory.* Chalmers University of Technology, Göteborg, 2007. ISBN 978-91-7291-996-9. Diss. Göteborg : Chalmers tekniska högskola, 2007.