Visually Embodying Well-Typedness of Algebraic Data Structures through *Maramafication*

Chide Groenouwe John-Jules Meyer

Universiteit Utrecht
Information and Computing Sciences
Utrecht, the Netherlands
{c.n.groenouwe|j.j.c.meyer}@uu.nl

Abstract

This paper presents a *maramafication* of an essential part of FPLs: the construction of well-typed algebraic data structures based on type definitions with at most one type parameter. Maramafication means the design of visual 'twins' of existing programming constructs using spatial metaphors rooted in common sense or inborn spatial intuition, to achieve self-explanatoriness. This is, among others, useful to considerably reduce the gap between programmers and non-programmers in the creation of programs, for educational purposes or for invoking enthusiasm among non-programmers.

1. Introduction

It would be highly beneficial if non-programmers could co-program software applications.

The Marama-paradigm, as introduced in previous work, is a paradigm that is under development with the intention to considerably reduce the gap between programmers and non-programmers. The basis of the Marama-paradigm consists of designing visual 'twins' of modern functional programming constructs using spatial metaphors rooted in common sense or inborn spatial intuition, making the constructs almost entirely self-explanatory. This work has coined the term *internal semantics* for this purpose: the semantics of constructs is evident without an external definition. This may lower the threshold for non-programmer participation to a great extent. This work coins the term *maramafication* for this design process, I.e. if such a twin has been designed for a language construct L from for example Clean(Brus et al. 1987) or Haskell(Hudak et al. 1992), it has been 'maramafied'.

This paper focusses on a fragment of the challenge: it presents a new way to visually represent a few aspects of polymorphy and algebraic data structures as they occur in modern functional programming languages, and does so in line with the aforementioned paradigm. The design is a 'modular': it can be adopted straightforwardly into any visual functional programming language.

Algebraic data structures are designed in such a way that type consistency is entirely forced by the form of the M-constructs (maramafied constructs). I.e. a user of these constructs cannot create a type incorrect value, simply because the 'pieces will not fit'. In thus sense, the approach in this paper is truly visual: the semantics of the visual blocks is embodied by their visual structure and spatial manipulation options, and do not require a definition by textual or spoken means. I.e. a beginner using the M-constructs can find out how to program with them, without any prior textual or spoken explanation about how these constructs work.

Another way to phrase it, is that the semantics of the visualisation of polymorphy and datastructures proposed in this paper solely relies on shared human intuition for manipulation of 3D objects.

In this article, the term *spatial necessity* is coined for the aforementioned property of the visual designs, the property that given the laws of mechanics (as far as they are intuitively understood by the majority of humans) it is only possible to construct something that is correct. An example of such a widely shared intuition on which the spatial necessity design paradigm can rely, is that most people from an already very young age will predict that a ball that is held in the air, and then let lose, will move downward.

The design covers algebraic data structures based on algebraic data type definitions with at most type parameter, and can cope with polymorphic constructors without arguments, and can classify a given algebraic data structure polymorphically (through 'type statements'), and is a firm basis for future extensions with multiple type parameters, and 'full' polymorphy, among others.

Section 2 starts with providing some examples that elucidate all features of the design. The sections after it provides the formal definition of the design and a mathematical proof that its M-constructs indeed exhibit the same relevant behaviours as their textual counterparts.

2. Examples

2.1 Textual Language: Frapoly

This section presents the textual language that is used in this article to show the textual equivalences of the M-constructs presented in this paper. Because the M-constructs covered in this paper only deal with a fragment of a modern functional programming language (FPL) such as Clean or Haskell, this article also defines a textual language that is (isomorphic to) the relevant fragment of such an FPL. The language is called Frapoly. For intuition, the following first introduces the language by example. (A formal specification is to be found in section 3.1.)

Example 1 (Algebraic Data Type Definitions)**.** The following *Algebraic Data Type Definitions* define a number of algebraic data types:

```
1 ::WeekendDay = Sat  | Sun
2 ::Bool       = True | False
3 ::List a     = Cons a (List a) | Nil
```

Note that this paper does not cover a visual counterpart to algebraic data type definitions. However, it is important to include them in the textual language for explanatory and definitory purposes.

Example 2 (Algebraic Data Structures)**.** The following is a comma-separated list of Algebraic Data Structures:

```
1 True, False, Cons True Nil,
2 Cons True (Cons True False).
```

The maramafication presented in this paper does not yet deal with function definition and function application. Therefore it is not possible to express type information about functions with the maramafied constructs. However, it already deals with polymorphy in relation to ADSs, and therefore needs a way to express polymorphic type information about ADSs, albeit a less expressive one than function type definitions. In the textual language, Frapoly, these expressions take the following form.

Example 3 (Type Statements)**.** The following statement

```
1   False <: a
```

states that the ADS `False` is subsumed by (`<:`) polymorphic type `a` ("has type `a`"). Note that the `<:` does not exist in normal FPLs, such as Clean or Haskell. It is introduced to equip the textual language with the bare minimum to elucidate the working of the M-constructs of this paper. Another examples is the following:

```
1   Cons True Nil <: List a
```

A non-example is the following

```
1   False <: List a
```

This statement is incorrect, because there is no instantiation of `a`, such that `False` has type `List a`.

Self-evidently, the language only allows programs with type-correct type statements. In a textual programming language this is normally enforced by the type-checker *after* writing the program (or at least, the sentence), while in the visual language, as we will see, it will be enforced *immediately* by spatial necessicity.

2.2 Visual Language: Madawipol-α

This section presents the visual language, for which the term Madawipol-α is coined, by example. For clarity for the reader, in some of the following examples the correspondence between parts of the visual and the textual representation is shown. It is essential to note that these are not put there for definitory purposes: for a user of the visual language the semantics is contained in the construction possibilities of the building blocks. The correspondence is merely put here to provide the reader of this paper, who is probably well-versed in textual functional languages, a quick insight into the fact that these visualisations indeed exhibit the same relevant behaviours as their textual counter parts.

2.2.1 Visual Algebraic Data Structures

Lets start with examples of the simplest ADSs: atomic ADSs, i.e. ADSs without arguments.

Example 4 (Atomic ADSs)**.** Given the algebraic data type definitions:

```
1 ::Bool       = True | False
2 ::WeekendDay = Sat  | Sun
```

then the values of type `Bool` are represented visually as given in fig. 1. In particular, note the form of the joint (formed by the lower part of the form), this is important for the subsequent examples.

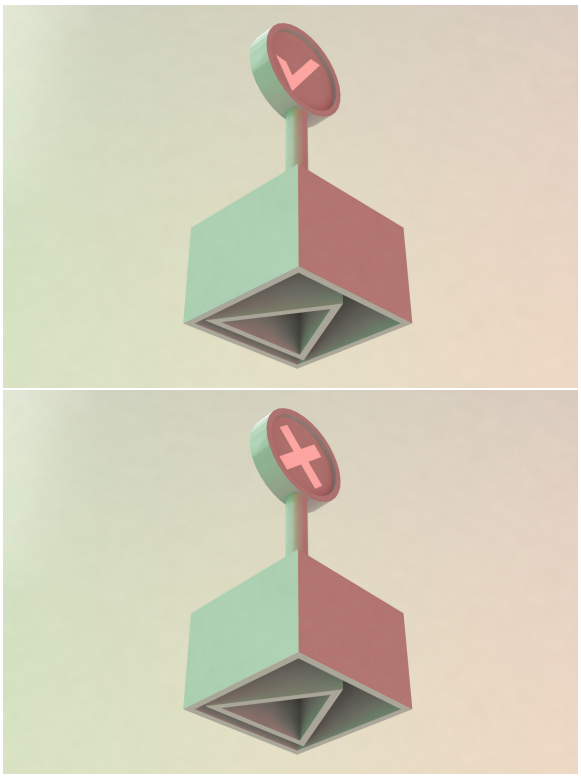

Figure 1. `True` and `False`.

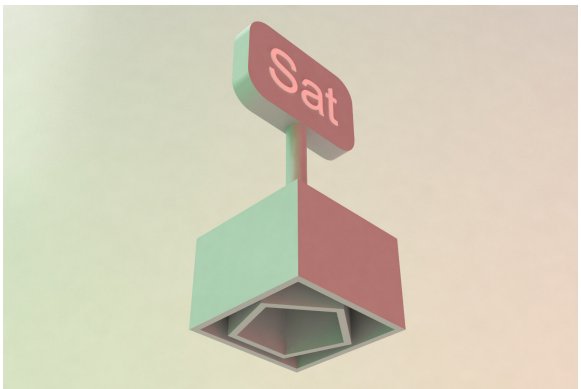

Figure 2. `Sat`.

Subsequently, lets now turn to a molecular ADS (an ADS that contains arguments), which makes use of the previous types.

Example 5 (Molecular ADS)**.** Given the algebraic data type definition:

```
1 List a :== Cons a (List a) | Nil
```

The following first focuses on values of type `List Bool`. The visual analogue to the constructor `Cons` is represented as given in fig. 3. Note the forms of its three joints.

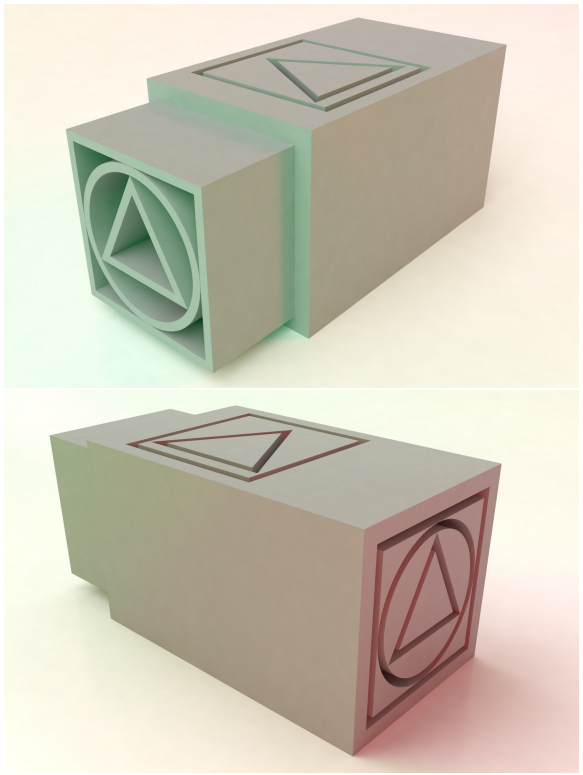

Figure 3. `Cons` of `List Bool` presented in two perspectives

The visual analogue to the constructor `Nil` is represented as given in fig. 4. Note the forms of its joint.

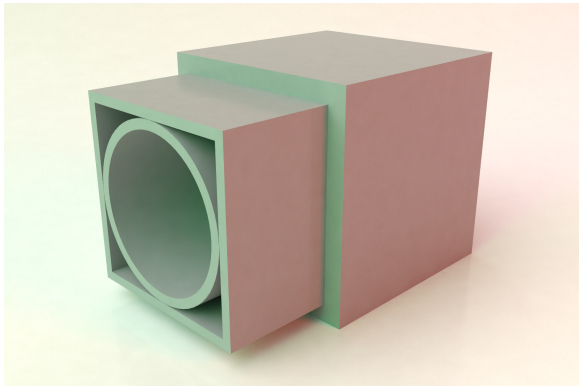

Figure 4. `Nil`

Now lets build values using these *M-constructor* blocks. M-constructor stands for 'maramafied constructor' – so within the context of this paper a constructor as it appear in Madawipol-α. A value `Cons True Nil` of the type `List Bool`, is represented visually as given in fig. 5.

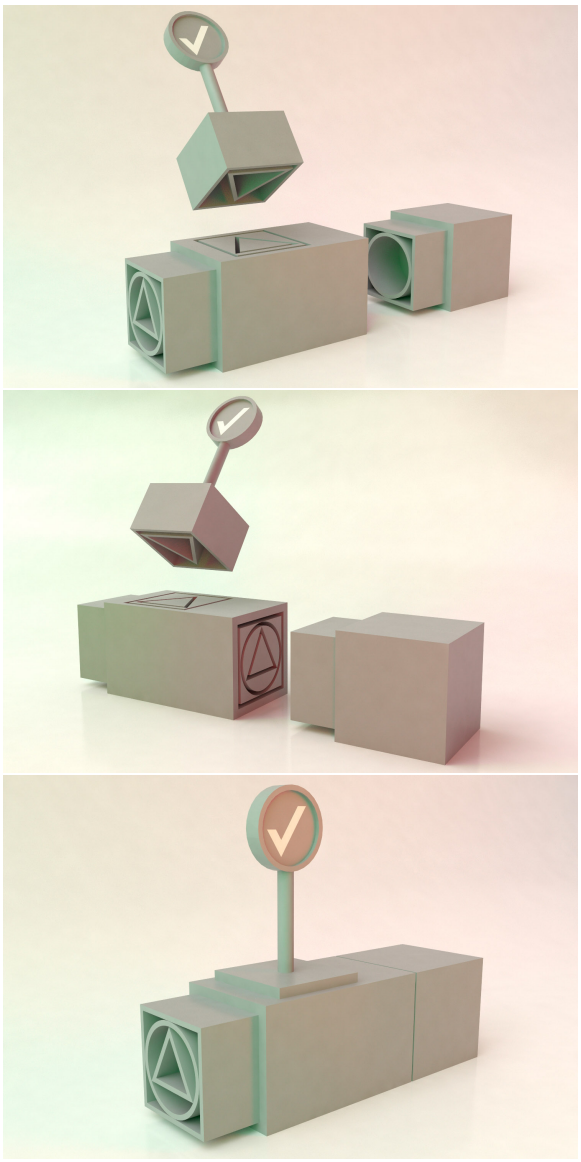

Figure 5. `Cons True Nil`

A value `Cons False (Cons True Nil)` is represented visually as given in fig. 6.

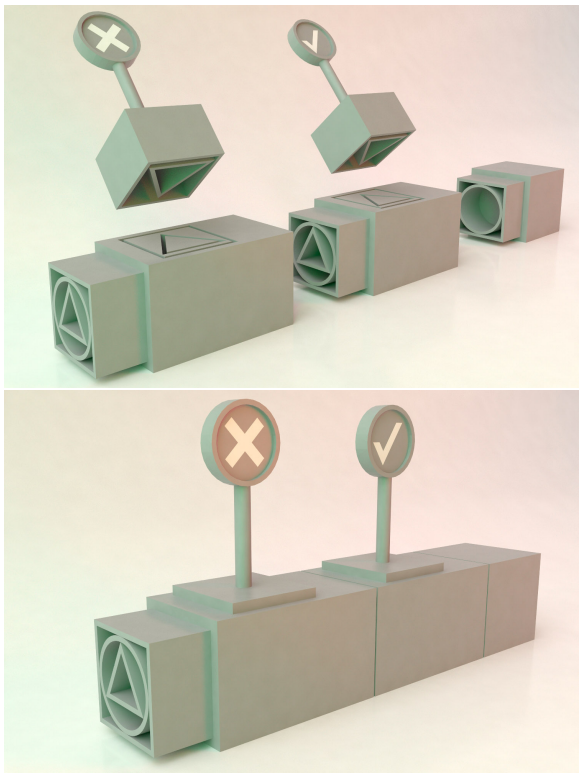

Figure 6. `Cons False (Cons True Nil)`, exploded and assembled view.

It may now be clear to the reader that, using the visual building blocks of `List Bool` as given in example 5, any value of `List Bool` can be created. On the other hand, it is not possible to create anything else than a valid ADS (or fragment thereof), as becomes clear in the following example.

Example 6 (Spatial necessity of wellformedness)**.** Consider someone trying to fit a visual ADS of type `WeekendDay` into a visual ADS block of type `List Bool`, as suggested in fig. 7.

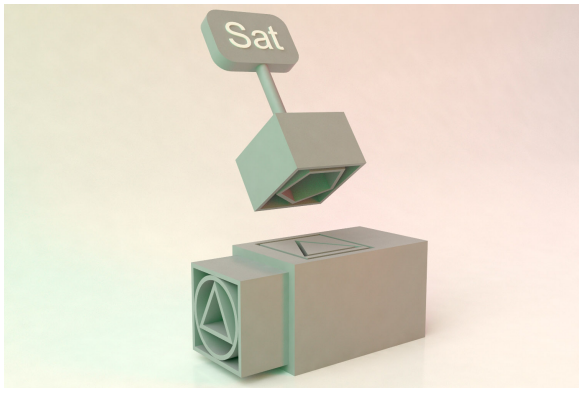

Figure 7. Spatial necessity: `Weekendday` does not fit into `List Bool`

It is clear that the building block of type `WeekendDay` simply will not fit.

The reader is encouraged to try other combinations that are not type-correct. These are simply spatially impossible to construct.

It is moreover possible to create 'nested' structures, in the visual language, as given in the following example.

Example 7 (Nested structures)**.** Consider the type `List (List Bool)`. All visual building blocks needed to build values of this type are those already provided in fig. 1, fig. 4 and fig. 3, in addition to the one given in fig. 8.

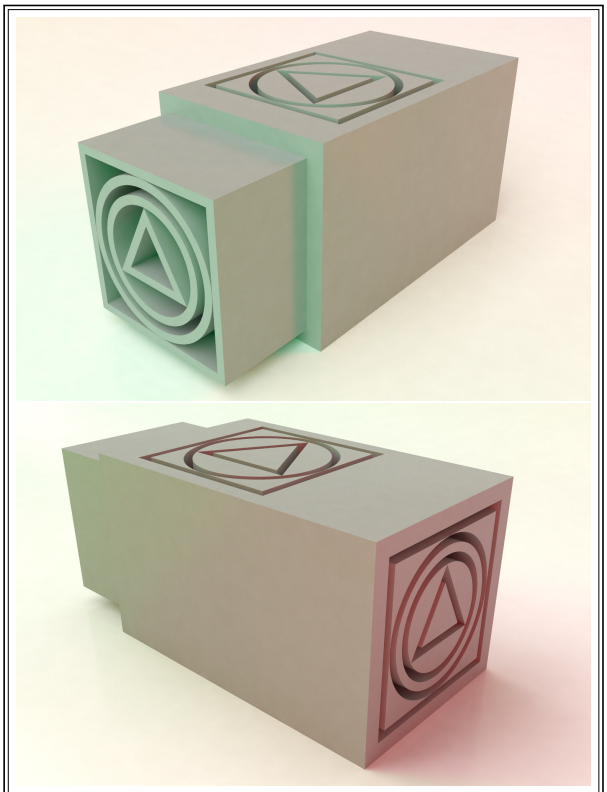

Figure 8. `Cons` of `List (List Bool)`

An example of a value, `Cons (Cons True (Cons False Nil)) (Cons (Cons True Nil) Nil)` (in a sugared form: `[[True, False], [True]]`) is provided in fig. 9.

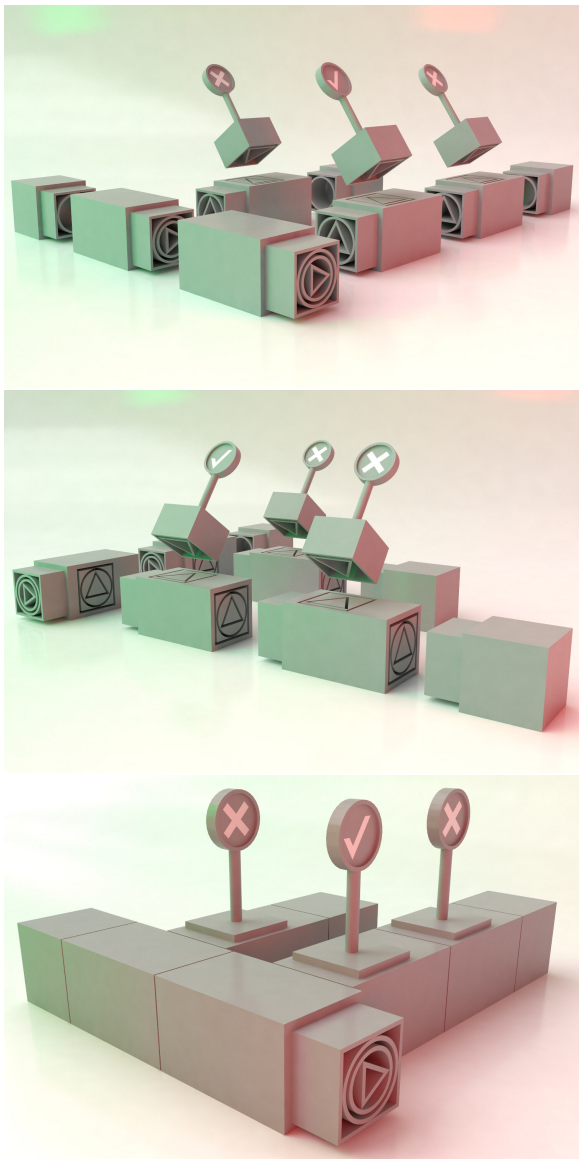

Figure 9. Example of a value of type `List (List Bool)`

The reader is encouraged to try visualise more complex values of this type, and also to try to construct non-well-formed examples (which should be impossible).

2.2.2 Visual Polymorphy

Visualising polymorphy comes with at least one big challenge if it has to be realised by 'spatial necessity'. How can one make ADSs of different types fit into the *same* female joint? The previous examples, already tacitly contained design decisions that make it possible to realise polymorphy in an elegant way. The following exemplifies the realisation of polymorphism.

Example 8 (Visual Polymorphy)**.** Consider the following type statement.

```
1 True <: a
```

(Hence: "There is a substitution s for type variable `a`, such that `True` is of type s.") The visual counterpart to this statement, that additionally also expresses its type correctness, is as given in fig. 10.

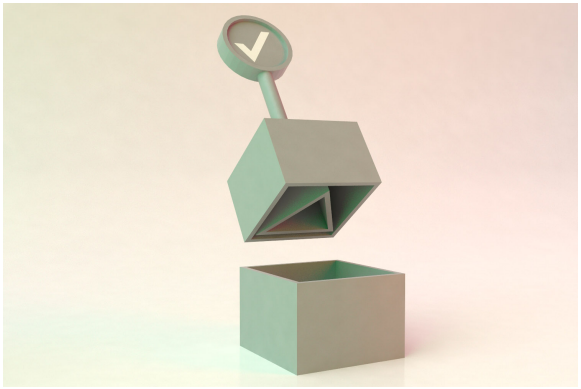

Figure 10. `True <: a`

It is clear the male joint of the visual `True` fits into the polymorphic female joint. Now also consider the statements:

```
1 Sun <: a
2 Cons True Nil <: a
```

These also 'fit' into the aforementioned polymorphic female joint, as is made clear in fig. 11.

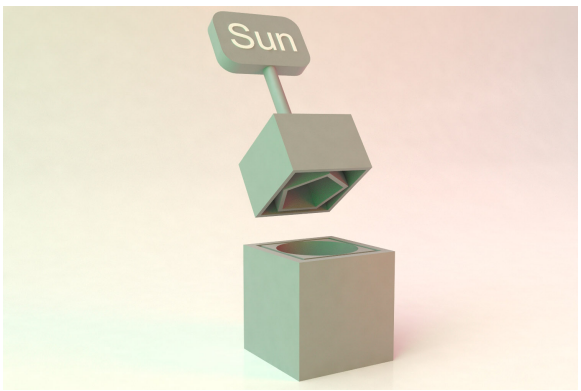

Figure 11. `Sun <: a`

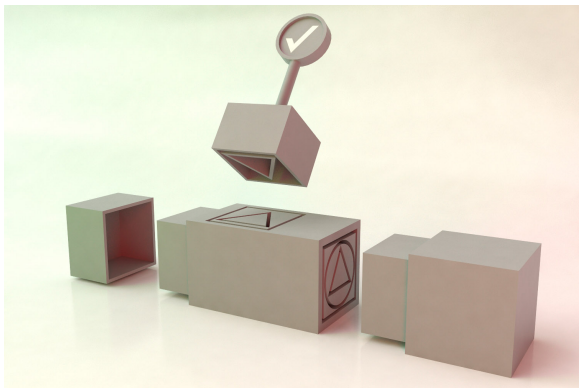

Figure 12. `Cons True Nil <: a`

Lets now consider a correct, and an incorrect type-statement:

```
Cons True Nil <: List a
Sun           <: List a
```

The visual equivalent indeed does not allow the ADS `Sun` to be fitted into the polymorphic female joint `List a`, as can be seen in fig. 13.

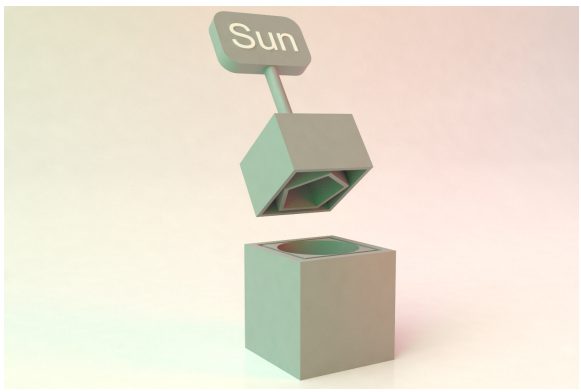

Figure 13. `Sun <: List a`

Nested polymorphic types are also no problem. Observe the following respectively incorrect and correct type-statement:

```
Cons True Nil <: List (List Bool)
Cons
   (Cons True (Cons False Nil))
   (Cons
      (Cons True Nil)
      Nil ) <: List (List Bool)
```

[TODO provide picture.]

3. Definition of Madawipol-α

This article suffices with a sketch of the definition and proof of correctness of the design of Madawipol-α. The goal of this article is to evoke the reader's understanding of the (correctness of the) design. An overly technical proof is beyond the scope of this article and would even be unnecessarily obfuscating. Moreover, a treatment of type statements is omitted – the structure of the proofs and definitions for these are essentially the same as for algebraic data structures.

The syntax and semantics of Madawipol-α are defined by means of a translation $trans$ from Madawipol-α to Frapoly and its inverse $transT^{-1}$. This is sufficient for defining the syntax: the latter is simply equal to the range of $trans$. However, it is not sufficient for defining Madawipol-α's semantics. A translation does not yet *prove* that the target language exhibits the same relevant behaviours as the original language. The proof-sketch is realised by first defining a mapping between the relevant behaviours of both languages, and then showing that the given translation preserves these behaviours. Another way to phrase it is that an expression in Frapoly should have the same relevant behaviours as its translation into Madawipol-α. Assuming that the chosen textual language has a well-defined semantics, the target language then has then been proven to 'inherit' this well-defined semantics. In more mathematical terms, the translation defines an isomorphism between two sets of language expressions with regard to the structure expressed by these relevant behaviours.

3.1 Definition of Frapoly

Frapoly is a fragment of an existing FPL with strong static typing such as Haskell or Clean. It only allows defining algebraic data type definitions and expressing algebraic data structures, and a fragment of type declarations, coined 'type statements', and does not contain function definitions. For the algebraic data structures and algebraic data type definitions, it follows the syntax of Clean(). A Frapoly 'program' consists of algebraic data type definitions, and free-standing algebraic data structures, both following the syntax as specified in the Clean language report(van Eekelen et al. 2011). The expressions use no other constructs than provided in the examples so far: The algebraic data structures are free of variables, and simply consist of a tree of constructor-applications, in which each constructor is expressed by its name. The leafs consist of constructors that do not take arguments. The algebraic data type definitions are limited to type constructors with at most one type parameter. Each right hand side alternative consists of a constructor name and a sequence of types. No quantifiers or other constructs are used.

Definition 1 (Frapoly(sub) expressions)**.** Given a set of algebraic data type definitions $ADTDset$ in Frapoly. Then $ADTDset_{constrs}$ is the set of constructors occurring in $ADTDset$ and $ADTDset_{tyConstrs}$ the set of type constructors occurring in $ADTDset$. $ADSset(ADTDset)$ is the set of all possible ADSs that are well-typed with respect to $ADTDset$.

3.2 Translation $trans$

Madawipol-α's expressivity is limited to two basic constructs: algebraic data structures and type statements. One of these is, therefore, always provided as an argument to the function $trans$. However, Madawipol-α does not contain ways to express algebraic data type definitions, but assumes these pre-exist. I.e. Madawipol-α is in fact a set of languages, each set of algebraic data type definitions inducing another instance of Madawipol-α. $trans$'s arguments therefore, include, next to an algebraic data structure or type statement to be translated a set of algebraic data type definitions (both in Frapoly) onto a construct in Madawipol-α. [1] Moreover, ad-

[1] Note that in future work, also algebraic data type definitions will be maramafied, and this allows the algebraic data structures to be defined in terms of these maramafied algebraic data type definitions instead. Then, Madawipol *can* in principle be expressed as one single language, although for technical reasons it may still be advantageous to formulate the translation as it is given in this paper.

ditional 'atomic' translation information[2] is needed to fully specify the translation. For this, this paper coins the term translation configuration. For example, a joint-form for each type should be provided in the translation configuration. It is not trivial what further information is minimally needed to be able to define a complete and valid translation. We believe we found an elegant minimal set, which, however, may need future extension.

The following first specifies the translation configuration.

3.2.1 Translation configuration

First some preliminary definitions are needed. In these definitions, each joint form is formalised by defining it as a subset of $\mathbb{R}^2$. The subset corresponds to the collection of points that one sees when viewing the bottom with a line of sight that is perpendicular to the bottom. I.e. if you would use the male version of the joint form as a kind of stamp on a piece of paper, these are the points that would appear on that paper. Moreover, the origin of $\mathbb{R}^2$, is, by definition, aligned with the center of the complete joint. The formalisation abstracts from depth and height information of the actual joints. After all, if one assumes that male and female joints have matching heights and depths, the only aspect that determines whether they fit is the information provided in the given formalisation.

Definition 2 (Basic Madawipol-α Typing Notions)**.**

1. The function $SqReg$ defines a square region within $\mathbb{R}^2$ with edge length l:

$$SqReg(l) = \{(x,y) \in \mathbb{R}^2 | -\tfrac{1}{2}l \leq x \leq \tfrac{1}{2}l \text{ and } -\tfrac{1}{2}l \leq y \leq \tfrac{1}{2}l\}. \quad (1)$$

   It is an auxiliary function, used to define some of the joint forms.
2. The alignment square $alignSq$ is the outermost square band occurring in all joints. The inner edge of this square band has, by definition, a fixed length $innerLengthAlignS$ and an outer edge length $outerLengthAlignS$. It is defined as follows:

$$alignSq = SqReg(outerLengthAlignS) - SqReg(innerLengthAlignS). \quad (2)$$

3. The permitted zone $pZone$ is the zone that a type-constructor form (see below) may maximally occupy. It is a square-area that lies centered within the alignment square. The square area of the permitted zone has an edge length of $lengthPZone$. $pZone$ is defined as:

$$pZone = SqReg(lengthPZone). \quad (3)$$

4. A type-constructor form $tyConstrForm$ is intended to correspond one-to-one with a type-constructor occurring in a given set of algebraic data type definitions in Frapoly. It should comply with the following conditions. (1) It is contained in the permitted zone so $tyConstrForm \subset pZone$. (2) It forms a connected space.[3] (3) It is closed (contains its boundaries). (4) It must be a surface, so, $\forall p \in S$: there is a disc $D \subset S$ such that $p \in D$. A noa line segment without thickness. (5) It may contain a finite (but not infinite) number of holes. Examples with respectively 0, 1 and 2 holes are a cross with thick lines, an annulus, and an 8 with thick lines.
5. (Auxiliary definition) Given $S \subset \mathbb{R}^2$ which is bounded[4]. Then the total fill of S, $Fill(S)$, is defined as follows. S is bounded, so there exists a disc D such that $S \subset D$. Now, $Fill(S)$ consists of all $x \in D$ for which holds that $x \notin S$ and all paths from x to points outside D cross S. An example, if S has an '8'-shape, its total fill consists of the two holes in the '8'.
6. A polymorphic space mapping $polySpaceMp$ is a function that maps each type-constructor form $tyConstrForm$ to its *polymorphic subspace*. The latter is the form that is The latter must be a connected surface without holes that lies within the total fill of $tyConstrForm$, i.e. $polySpaceMp(tyConstrForm) \subset Fill(tyConstrForm)$.
7. A maximal space mapping $maxSpaceMp$ is a function that maps each type-constructor form $tyConstrForm$ to the total space it occupies including its polymorphic subspace. Thus, formally it is defined as:

$$maxSpaceMp(tyConstrForm) = tyConstrForm \cup polySpaceMp(tyConstrForm). \quad (4)$$

8. The vertical joint size $verticalJntSize$ is the size of joints perpendicular to their 2D joint form. So, if one orients the joint form horizontally, it is the vertical size of the joint. For female joints, this is their depth, for male joints their height. All joints have the same vertical size.

Definition 3 (Basic Madawipol-α Notions and Definitions)**.**

- The notion *M-constructor* stands for a maramafied constructor: a constructor as it appears in Madawipol-α. Its precise definition follows later.
- The notion *proto-M-constructor* stands for an M-constructor that is not yet in its final form. Such constructors are not part of Madawipol-α, but are needed in the *process* of defining Madawipol-α.

For clarity, the text that follows now, makes use of one running example. Part of of this running example is the following set of algebraic data type definitions:
$ADTDset =$

```
::WeekendDay = Sat  | Sun
::Bool       = True | False
::List a     = Cons a (List a) | Nil
```

Given a set of algebraic data type definitions $ADTDset$, then a translation configuration $tConf$ for this set consists of the following parts:

- $ADTDset$: to contribute to consiseness in notation, the set of algebraic data type definitions that is associated with $tConf$ is also included. This allows the ommission an explicit mentioning of $ADTDset$ when dealing with translation configurations.
- *type-constructor mapping*: A mapping $tyConstrFormMp$ from each type-constructor occurring in $ADTDset$ to a type-constructor form (see definition 2). Hence, its type is:

$$tyConstrFormMp : ADTDset_{tyConstrs} \rightarrow 2^{\mathbb{R}^2}$$

- *constructor block mapping*: A mapping $coBlckMp$, which associates a given constructor $constr$ that occurs in $ADTDset$ with a solid 3D object that is the M-constructor in its rough form: it does not yet contain the joints. Its type is as follows.

$$coBlckMp : ADTDset_{constrs} \rightarrow 2^{\mathbb{R}^3}$$

  Moreover, the locations where its joints are to be created (see $aLocMp$ and $rLocMp$) are (1) flat surfaces with the same orientation as the corresponding joint. (2) sufficiently deep, hence deeper than $verticalJntSize$. It should map each constructor to

[2] Atomic in the sense that it is information that is provided with some of the languages atoms of Frapoly, such as individual type constructor symbols

[3] Topological term for consisting out of 'one piece'.

[4] topological term for 'finitely sized'

a unique proto-M-constructor, which even remains unique after the joints have been added. (That means that the distinction between two different proto-M-constructors is *never* at the location of the surfaces where the joints are created. The distinction will then be 'overwritten'.)

- *constructor argument-location mapping*: A mapping $aLocMp$, which, given a constructor $constr$ from $ADTDset$, specifies the location and orientation of the joints that correspond to the arguments in the textual form. It has the following type:

$$aLocMp \quad : \quad (constrT, \mathbb{N}) \quad \rightarrow \quad jntLocT, \quad (5)$$

where $jntLocT$ is a joint location, which is a pair that consists of a coordinate in $\mathbb{R}^3$ which indicates the location of the center of the joint, and the orientation of the joint using the axis-angle representation():

$$jntLocT = (jntCenterT, jntOriT)$$
$$jntCenterT = \mathbb{R}^3$$
$$jntOriT = (unitVectorT, jntOriT),$$

where $unitVectorT$ is the set of unit vectors, determining the rotation axis, and $jntOriT$ the set of angles $[0°, 360°]$.

- *constructor's result-type location mapping*: A mapping $rLocMp$, which, given a constructor $constr$ from $ADTDset$, specifies the location of the result type that corresponds to the result type of $constr$. It has the following type:

$$rLocMp : constrT \rightarrow jntLocT$$

$aLocMp$ and $rLocMp$ satisfy the condition that they map a given constructor to non-overlapping joint locations.

- *scalingMap*: A mapping $scalingMp$, which maps each type constructor to a scale factor. This factor is, loosely speaking, used to scale down the maramafied arguments to a maramafied type constructor, so that they fit within its polymorphic subspace. Its type is as follows:

$$scalingMp : tyConstrT \rightarrow \langle 0, 1\rangle$$

Scaling of a subset of $\mathbb{R}^2$ with a scaling factor is a simple linear transformation with respect to the origin:

Definition 4 (scaling)**.** Given a scaling factor sf and $S \subset \mathbb{R}^2$. Then the scaling operator $\times$ is defined as:

$$sf \times S = \{(x, y) | \exists (a, b) \in S : x = sf \cdot a \text{ and } y = sf \cdot b\}. \quad (6)$$

Definition 5 (translation configuration)**.** Given a set of algebraic data type definitions $ADTDset$. A translation configuration $tConf$ for $ADTDset$ (also written as $tConf_{\,ADTDset}$) is a tuple

$$(ADTDset, tyConstrFormMp, coBlckMp, aLocMp, rLocMp, scalingMp, verticalJntSize). \quad (7)$$

which meets the following conditions:

- For any two type constructors tc_1, tc_2 occurring in $ADTDset$, $maxSpaceMp(tc_1)$ and $maxSpaceMp(tc_2)$ should not or only partially overlap. I.e.:

$$maxSpaceMp(tc_1) \not\subseteq maxSpaceMp(tc_2) \quad (8)$$
$$maxSpaceMp(tc_2) \not\subseteq maxSpaceMp(tc_1). \quad (9)$$

- For any type constructor tc holds:

$$scalingMp(tc) \times pZone \subset polySpaceMp(tc). \quad (10)$$

I.e., after applying the scaling factor associated with a type constructor to the permitted zone, it lies within its polymorphic subspace.

Notation 1 (Ommission of translation configuration)**.** The translation configuration is frequently needed in the definitions to come. Therefore, if it is clear from the context which translation configuration is intended, it is omitted from expressions.

3.2.2 Translation *trans*

Before the definition of the translation, first some auxiliary definitions are required. A brief overview of these is as follows:

- The *type translator* $tyTrans$: a function that maps types from Frapoly to joint forms.
- $creaFemJnt$: a function that creates a female joint on a proto-M-constructor.
- $creaMaleJnt$: a function that creates a male joint on a proto-M-constructor.
- $mConstructorSet$: the set of all possible M-constructors.
- $MADSset$: the set of all M-ADSs.
- $FinMADSset$: the set of all finished M-ADSs.
- $tyTrans$: a *type annotator*: a function that annotates algebraic data structures from Frapoly with typing information.
- $\overset{n}{\leftarrow}$: an operator that fits maramafied algebraic data structures to M-constructors.

The type translator is defined as follows.

Definition 6 (type translator)**.** Given a translation configuration $tConf_{\,ADTDset}$, and a (non-function) type in accordance with $ADTDset$: $ty = tc_1\ tc_2\ \ldots\ tc_n\ [tparam]$. $tparam$ is present if and only if tc_n takes a type parameter. Then

$$tyTrans(ty, tConf_{\,ADTDset}) = (typeConstrForms, polyForm), \quad (11)$$

where

$$\begin{aligned} typeConstrForms = {} & tyConstrFormMp(tc_1) \cup \\ & scalingMp(tc_1) \times (tyConstrFormMp(tc_2) \cup \\ & scalingMp(tc_2) \times (tyConstrFormMp(tc_3) \cup \\ & \vdots \\ & scalingMp(tc_{n-2}) \times (tyConstrFormMp(tc_{n-1}) \cup \\ & scalingMp(tc_{n-1}) \times (tyConstrFormMp(tc_n)[\cup \\ & polySpaceMp(tc_n)]) \ldots), \end{aligned} \quad (12)$$

and

$$\begin{aligned} polyForm = {} & scalingMp(tc_1) \times scalingMp(tc_2) \times \ldots \\ & scalingMp(tc_{n-1}) \times polySpaceMp(tc_n). \end{aligned} \quad (13)$$

The 2D joint forms of the female and male joints are now defined as follows.

Definition 7 (female joint form and male joint form)**.** Given $T, P \subset \mathbb{R}^2$ (where T is intended to specify type constructor forms, and P the polymorphic subspace). Then

$$femFormMp(T, P) = T \cup P, \quad (14)$$

while

$$maleFormMp(T, P) = T. \quad (15)$$

$creaMaleJnt$ is defined as follows.

Definition 8 ($creaMaleJnt$)**.** Given a joint form $J \subset \mathbb{R}^2$, a joint location $jntLoc$, a vertical joint size $verticalJntSize$ and a solid $C \subset \mathbb{R}^3$, which has a flat surface at the location and orientation specified in $jntLocT$. Then

$$creaMaleJnt(J, jntLoc, C, verticalJntSize)$$

maps to C onto which a male joint is attached by first translating and rotating J according to $jntLoc$, and then extruding it perpendicularly away from the surface of C over a distance of $verticalJntSize$.

Definition 9 ($creaFemJnt$)**.** $creaFemJnt$ is defined analogously to $creaMaleJnt$, however, instead of extruding, it is used to carve away material from C by extruding the translated and rotated J perpendicularly *into* the surface of C, and extracting the resulting form from C.

A type annotation of a constructor is defined as follows.

Definition 10 (constructor type annotation)**.** Given a constructor $constr$ and a set of algebraic data type definitions $ADTDset$. Also, suppose that the type definition for $constr$ in $ADTDset$ is as follows:

$$::typeConstr\ param\ \texttt{=}$$
$$constr\ t_1[param] \ldots t_n[param], \quad (16)$$

(Note that type definitions of alternative constructors are left out of the algebraic data type definition written above, if there are any.) Here, $t_i[param]$ is a type in which the type parameter $param$ occurs. Moreover, suppose tha $someType$ is a type in accordance with $ADTDset$. Then

$$constr\texttt{:[}t_1[someType]\ \ t_n[someType]\ \texttt{->}$$
$$typeConstr\ someType\texttt{]} \quad (17)$$

is a constructor that is annotated in accordance with $ADTDset$. Note that the above follows the functional style of defining types of constructors.

An auxiliary notion: a type is *closed* if it does not contain any type parameters. I.e. if it is 'fully instantiated' and cannot be instantiated any further. This notion is important, because Madawipol-α only contains M-constructors which arguments have a closed type.

$transAnConstr$ is defined as follows.

Definition 11 ($transAnConstr$)**.** Given a translation configuration $tConf_{\,ADTDset}$. Moreover, let $constrAn$ be a constructor annotated in accordance with $ADTDset$, for which holds that the annotation only contains arguments with a closed type (this includes the case that it has no arguments), hence,

$$constrAn = constr\texttt{:[}type_{a_1} \ldots type_{a_n} \rightarrow type_{res}\texttt{]}, \quad (18)$$

where $type_{a_i}$ is closed for all i (including the case $n = 0$, so if there are no arguments). Then:

$$transAnConstr(constrAn, tConf_{\,ADTDset}) =$$
$$creaFemJnt(femFormMp(tyTrans(type_{a_1})), aLocMp(constr, 1),$$
$$creaFemJnt(femFormMp(tyTrans(type_{a_2})), aLocMp(constr, 2),$$
$$\vdots$$
$$creaFemJnt(femFormMp(tyTrans(type_{a_n})), aLocMp(constr, n),$$
$$creaMaleJnt(maleFormMp(tyTrans(type_{res})), rLocMp(constr),$$
$$coBlckMp(constr)) \ldots). \quad (19)$$

$mConstructorSet$ is now simply defined as the image of the function $transAnConstr$:

Definition 12 ($mConstructorSet$)**.** Given a translation configuration $tConf_{\,ADTDset}$.

$$mConstructorSet(tConf_{\,ADTDset}) =$$
$$\{mc | \exists ac.transAnConstr(ac, tConf_{\,ADTDset}) = mc\}. \quad (20)$$

Note that the argument to $mConstructorSet$ will not always be mentioned.

Some further basic Madawipol-α notions can now be introduced.

Definition 13 (M-ADS)**.** Given is a translation configuration $tConf_{\,ADTDset}$.

- An *M-ADS* (maramafied ADS) is an object $mAds$ for which holds: $mAds \in mConstructorSet$ or $mAds$ can be created by joining the joints of several M-constructors from $mConstructorSet$. In the latter case, $mAds$ must form a single undivided structure, hence, all M-constructors must be (in)directly connected to each other. Moreover, the structure must be free of cycles: there is exactly one path (sequence of connected constructors) between any two M-constructors in $mAds$.

- An *M-ADS* (maramafied ADS) is an object $mAds$ for which holds: $mAds \in mConstructorSet$ or $mAds$ can be created by fitting together the joints of several M-constructors from $mConstructorSet$. In the latter case, $mAds$ must form a single undivided structure, hence, all M-constructors must be (in)directly connected to each other. Moreover, the structure must be free of cycles: there is exactly one path (sequence of connected constructors) between any two M-constructors in $mAds$.
- A joint that is not connected to another joint, is called on *open joint*. The opposite is a *closed joint*.
- An M-ADS is a *finished M-ADS* if it does not contain open female joints. It is an *unfinished M-ADS* if it is not a finished M-ADS.
- $MADSset(tConf)$ is the set of all M-ADSs, including the unfinished ones.
- $FinMADSset(tConf)$ is the set of all finished M-ADSs.

Theorem 1. *All M-ADSs have exactly one open male joint.*

Proof. This can be easily shown by induction on the structure of M-ADSs.

If an M-ADS consists of one M-constructor, the property holds. This follows directly from the definition of $transAnConstr$: it creates exactly one male joint. If the property holds for $mAds_1$ and $mAds_2$, then it also holds for $mAds_1$ joined with $mAds_2$ if they can be joined. After all, the single male joint of $mAds_1$ must be joined with a female joint of $mAds_2$, which leaves open the single joint of $mAds_2$, or vice versa. The remaining single male joint cannot be connected without creating a cycle, which is prohibited according to definition 13. □

Definition 14 (outermost M-constructor)**.** Given an M-ADS $mAds$. The outermost M-constructor of $mAds$ is the M-constructor which male joint is not connected to a female joint. (There is exactly one M-constructor with this property according to theorem 1.)

Next, the definition of $trans$ requires the constructors in an ADS of Frapoly to be annotated with their exact type. This paper

assumes the type inferencing algorithm for Frapoly as a given, because Frapoly is a fragment of an existing modern functional programming language, such as Haskell or Clean. This type inferencer, in its turn, can be used to define a type annotator for ADSs. It is a function that annotates an ADS from Frapoly with typing information. To be precise: it annotates each constructor that occurs in a given ADS with its inferred type. The definition of a type annotator is straightforward, therefore, this paper will suffice with providing examples.

Example 9 ($tyAn$)**.**

$$tyAn(\texttt{Cons True Nil}) = \texttt{Cons:[Bool (List Bool) -> List Bool] True:[Bool] Nil:[List a]} \quad (21)$$

Note that $tyAn$ does not annotate molecular (sub)expressions as a whole. It merely annotates the *individual* constructors occurring in it. Therefore, the following is a non-example:

Example 10 (Non-example $tyAn$)**.** `(Cons:[Bool (List Bool) -> List Bool]  True:[Bool] Nil:[List Bool]):[List Bool]`

The (informal) definition of the infix operator $\overset{n}{\leftarrow}$ is as follows.

Definition 15 ($\overset{n}{\leftarrow}$)**.** Given (finished or unfinished) M-ADS $mAds$, which may contain open female joints in its outermost M-constructor, and finished M-ADS $mAdsC$ (so one without any open female joints). Then:

$$mAds \quad \overset{n}{\leftarrow} \quad mAdsC \quad (22)$$

is the result of (spatially) fitting $mAdsC$ into the n^{th} argument-position of the outermost M-constructor of $mAds$. If the male joint of $mAds$ does not fit into the n^{th} argument position, or if that position does not exist or is already taken, then the operator maps to the constant unjoinable, which stands for "unjoinable".

For simplicity, the assumption is that the negation of the conditions in the last sentence is an adequate formalisation of *real* joinability, i.e. that the physical M-ADSs really fit together in the mentioned way. This assumption, however, does not always hold: the M-ADS s may physically get 'into each other's way' depending on the shape of the M-constructors and the structure that one is trying to build. With minor adjustments, such as the introduction of 'flexible' M-constructors, the assumption can be made to always hold (future work).

If one or both arguments of $\overset{n}{\leftarrow}$ are unjoinable, then the result is also unjoinable.

Moreover, $\overset{n}{\leftarrow}$ is left-associative.

This, finally, allows us to define $trans$ on ADSs of Frapoly:

Definition 16 ($trans$)**.** The translation of ADS ads is defined, recursively, as follows:

$$trans(ads, tConf_{\,ADTDset}) = transAn(tyAn(ads, ADTDset), tConf_{\,ADTDset}), \quad (23)$$

where $transAn$ is defined as follows. If $adsAn$ is atomic, then

$$transAn(adsAn, tConf_{\,ADTDset}) = transAnConstr(adsAn, tConf_{\,ADTDset}). \quad (24)$$

Note that $adsAn$ can be offered directly to $transAnConstr$, because an atomic ADS is equal to a constructor without arguments. If ads is molecular, then $adsAn$ can be written as

$$adsAn = constrAn \;\; argAn_1 \;\; \ldots \;\; argAn_n.$$

In that case:

$$\begin{aligned} transAn(adsAn, tConf_{\,ADTDset}) = \\ transAnConstr(constrAn, tConf_{\,ADTDset}) \overset{1}{\leftarrow} \\ transAn(argAn_1, tConf_{\,ADTDset}) \overset{2}{\leftarrow} \\ transAn(argAn_2, tConf_{\,ADTDset}) \overset{3}{\leftarrow} \\ \vdots \\ transAn(argAn_{n-1}, tConf_{\,ADTDset}) \overset{n}{\leftarrow} \\ transAn(argAn_n, tConf_{\,ADTDset}). \end{aligned} \quad (25)$$

An alternative notation for $trans$ is:

$$trans_{tc}(ads) = trans(ads, tc)$$

This allows expressing the type of $trans$ for a given $tConf_{\,ADTDset}$:

$$trans_{tConf} : ADSset(ADTDset) \rightarrow FinMADSset(tConf)$$

4. Semantical equivalence of Madawipol-α and Frapoly

As stated earlier, Madawipol-α's semantical equivalence to Frapoly is proven by showing that (1) relevant behaviours of Madawipol-α and Frapoly that are intended to correspond, can indeed be *proven* to correspond under the intended correspondence between expressions as defined by $trans$, and (2) that both languages are equally expressive. In other words, it is shown that $trans$ and the intended correspondences between relevant behaviours form an isomorphism. The following theorems expresses the required correspondences informally.

Informal Theorem 1 (correspondence of fitting joints together and constructor-application)**.** *Fitting a complete $M-ADS$ into an M-constructor corresponds with constructor-application in Frapoly.*

Informal Theorem 2 (correspondence of (joinability+ finishedness) and well-typedness)**.** *(joinability+finishedness) in Madawipol-α corresponds with well-typedness in Frapoly.*

Informal Theorem 3 (expressivity equivalence)**.** *Each finished M-ADS corresponds uniquely to a well-typed ADS of Frapoly.*

The most essential and interesting aspects of the design of Madawipol-α ly in Informal Theorem 2. Therefore, the proof in the rest of this paper mostly focus on this theorem. A proof of Informal Theorem 3 is left to the reader.

4.1 Correspondence of (joinability+finishedness) and well-typedness

First a few lemmas are needed that shows that the translation of *types* behaves well.

Definition 17. The *joint-form complement* $jntFormComplem$ of a (2D) joint form J is defined as

$$jntFormComplem(J) = pZone \backslash J.$$

(This is the relative complement of permitted zone $pZone$ with respect to J, so all points of $pZone$ that are not in J.)

Lemma 1. *A female joint and a male joint fit into each other, iff the joint-form complement of the 2D joint form of the female joint does not overlap the 2D joint form of the male joint.*

Intuitively this is clear, therefore the proof is omitted.

Lemma 2 (compatible joints correspond with unifiable types)**.** *A male and a female joint in Madawipol-α fit together iff the equation of the corresponding types in Frapoly is unifiable. (This statement*

is purely about the compatibility of the extruded joints forms as they are in themselves. If they are part of an M-constructor there may be other impediments than these forms.)

Formally expressed: Given a translation configuration $tConf$, and given two (non-function) types M and F. Then:

$$M = F \textit{ is unifiable } \Leftrightarrow$$
$$\textit{the male joint based on } maleFormMp(tyTrans(M, tConf))$$
$$\textit{fits into the female joint based on}$$
$$femFormMp(typeTrans(F, tConf)). \quad (26)$$

Proof. According to the definition of Frapoly, types M and F must have the following form:

$$M = tcM_1\ tcM_2\ \ldots\ tcM_n\ [tparam]$$

$$F = tcF_1\ tcF_2\ \ldots\ tcF_m\ [tparam]$$

The type parameter $tparam$ is present iff the type constructor occurring before it takes a type parameter. The following uses the abbreviations: $mM = maleFormMp(tyTrans(M, tConf))$, $mF = femFormMp(tyTrans(F, tConf))$
and $\overline{mF} = jntFormComplem(mF)$

($\Rightarrow$) Suppose M and F are unifiable, then with induction on n and m:

1. $n = m = 1$. In this case, it also holds that $M = F$ (modulo the *name* of the type parameter): in this case it is trivial that mM and $\overline{mF}$ do not overlap, and therefore, the corresponding male and female joints fit into each other (lemma 1).
2. Assume the statement holds for $n = c_n$ and $m = c_m$. Now, suppose that $n = c_n + 1$ and $m = c_m$. So, M's type has the form:

   $$M = tcM_1\ tcM_2\ \ldots\ tcM_{c_n}\, tcM_{c_n+1}\ [tparam].$$

   Now suppose the type M_{-1} is defined as follows:

   $$M_{-1} = tcM_1\ tcM_2\ \ldots\ tcM_{c_n}\ tparam.$$

   The $tparam$ in M_{-1} must exist, because tcM_{c_n} takes a type parameter as follows from the form of M. As given, M is unifiable with F. Then it is not difficult to see that M_{-1} is also unifiable also with F (left to the reader). Moreover, according to the induction hypothesis (in combination with lemma 2) $tyTrans(M_{-1})$ (abbreviated with mM_{-1}) does not overlap with $\overline{mF}$. If one analyses the definition of $tyTrans$, and the property that the permitted zone always lies within the polymorphic subspace, one sees that the difference in joint form between mM_{-1} and mM is contained within the polymorphic subspace of the type-constructor form associated with tcM_{c_n} to which all scaling factors of the previous type-constructor forms have been applied from the first to the last. This scaled down polymorphic subspace moreover, does not overlap with any type-constructor form of mM_{-1} – it lies enclosed within the inner most one, as a consequence of eq. (10) and item 6 of definition 2. Outside of this scaled down polymorphic subspace, the joint forms are identical. Now, one can distinguish the following cases:
   - $c_m = c_n$. In this case the polymorphic subspace of the female joint is equal to that of the male. Because the difference between mM_{-1} and mM is contained within the polymorphic subspace of M_{-1}, mM and $\overline{mF}$ do not overlap.
   - $c_m < c_n$. Now, the polymorphic subspace of the female joint is a superset of that of the male joint, so there is even more 'space'. Then by the same reasoning, mM and $\overline{mF}$ do not overlap.
   - $c_m > c_n$. Because M is unifiable with F, $tcM_{c_n+1} = tcFc_n + 1$. mF's form was already identical to that of mM up to the type-constructor form associated with tcM_{c_n} (because of the induction hypothesis). With the identity just given, this now also extends up to the type-constructor form tcf associated with tcM_{c_n+1}. Further up, Mm does not contain points that are in the polymorphic subspace of tcf, and therefore does not overlap any of the remaining forms of mF. After all, these remaining forms ly within the corresponding polymorphic subspace of mF, as follows from the definition of $tyTrans$. Hence, mM and $\overline{mF}$ do not overlap.
3. The proof for the case $n = c_n$ and $m = c_m + 1$ is analogous to the previous case.

($\Leftarrow$) If M and F are not unifiable, then there is an $i \leq min(m, n)$ for which $tcM_i \neq tcF_i$. Take the smallest i for which this holds. Then, by definition of $tyTrans$, the corresponding type-constructor forms of the two have been scaled down exactly the same amount. Moreover, in their original size, both forms did not or only partially overlap (see definition 5). After scaling the same amount, this property will continue to hold. This means that mM and $\overline{mF}$ *do* overlap, and the corresponding joints do not fit. □

The correspondence of (joinability+ finishedness) and well-typedness is formally expressed as follows.

Theorem 2 (correspondence of (joinability+finishedness) and well-typedness)**.** *Given a translation configuration $tConf_{\,ADTDset}$. Then*

1. *(well-typedness in Frapoly implies (joinability+finishedness) in Madawipol-α)*

   $$ads \in ADSset(ADTDset) \Rightarrow$$
   $$trans(ads) \in FinMADSset(tConf). \quad (27)$$

2. *((joinability+finishedness) in Madawipol-α implies well-typedness in Frapoly)*

   $$mAds \in FinMADSset(tConf)$$
   $$\Rightarrow$$
   $$\exists ads \in ADSset(ADTDset) \textit{ such that}$$
   $$trans_{ADTDset}(ads) = mAds. \quad (28)$$

   Equivalently, the image of $trans_{tConf}$ is equal to all M-ADSs that can be constructed from $mConstructorSet$, so to $FinMADSset(tConf)$.

Proof.

1. (well-typedness in Frapoly implies (joinability+ finishedness) in Madawipol-α) Given is $ads \in ADSset(tConf)$. The proof is by induction on the structure of ads. If ads is atomic, it trivially holds, for $trans_{tConf}(ads) = transAnConstr(typeannotator(ads))$, the latter of which is by definition an element of $FinMADSset$. If ads is molecular, it *could* go wrong if the joints would not fit, which would lead to $trans$ mapping to unjoinable. The following shows, however, that this cannot happen. ads is well-typed, so it can be annoted:

   $$adsAn = tyAn(ads),$$

   and $adsAn$ has the following form:

   $$adsAn = constrAn\ argAn_1\ \ldots\ argAn_n.$$

   Because this expression is well-typed, the type annotation of $argAn_i$ is unifiable with the i^{th} argument of $constrAn$. From

the definition of $trans$ one can immediately see that $trans$ 'attempts' to join the translation of $argAn_i$ to the i^{th} argument position of the translation of $constrAn$. Using lemma 2 these joins must fit.

2. (joinability in Madawipol-α implies well-typedness in Frapoly) This can be shown by, among other things, again using lemma 2, and is left to the reader.

□

4.2 Correspondence of fitting joints together and constructor-application

Theorem 3 is expressed formally as follows.

Theorem 3 (correspondence of fitting joints together and constructor-application)**.** *When a constructor applied to arguments forms a well-typed ADS ads, then the translation of ads is equal to the translation of the arguments fitted into the corresponding joints of the translation of the constructor, where the types of the arguments and the constructor are as they occur in the annotated version of ads.*

In formal terms:
Given a well-typed ADS

$$ads = constrarg_1 \dots arg_n,$$

and

$$tyAn(ads) = constrAn\ \ argAn_1\ \ \dots\ \ argAn_n.$$

Then the following identity holds:

$$\begin{aligned} trans(ads) = \\ transAnConstr(constrAn, tConf_{\,ADTDset}) \overset{1}{\leftarrow} \\ transAn(argAn_1, tConf_{\,ADTDset}) \overset{2}{\leftarrow} \\ transAn(argAn_2, tConf_{\,ADTDset}) \overset{3}{\leftarrow} \\ \vdots \\ transAn(argAn_{n-1}, tConf_{\,ADTDset}) \overset{n}{\leftarrow} \\ transAn(argAn_n, tConf_{\,ADTDset}). \end{aligned} \quad (29)$$

Proof. It is no coincidence that this correspondence looks exactly like the definition of $trans$ – the latter has been defined according to this structure. The equivalence can easily be proven by induction on the structure of ADSs. □

5. Empirical study into comprehensibility by non-programmers

An empirical study among more than 200 pupils have been carried out by a team of talented secondary schools students to investigate the understandability of the designs for non-programmers. The results are promising, but have to be repeated by experienced researchers under controlled circumstances. As a side note it may be mentioned that the design of Marama evoked quite some enthusiasm: the team, consisting of 3 ladies and one gentleman was free to pick any assignment from different disciplines, but chose this one over another. This is quite unusual for the topic of programming language design, which is known not to be popular (at least not in the Netherlands).

6. Related Work

Task Oriented Programming (TOP) is a programming paradigm to efficiently develop internet applications that support human collaboration.(Achten et al. 2015) It is an extension of the functional programming paradigm. One of the essential features of the paradigm is that it allows users to assist in the understanding and construction of these applications, for example through Tonic (Stutterheim et al. 2015). The design of TOP so far, however, is not self-explanatory to non-programmers in many respects. The particular design in this paper, and in general – the Marama paradigm, therefore, could be of great value to the further development of TOP.

Other functional programming languages that apply visual constructs are often not targeted at non-programmers at all, and do not employ the principle of 'internal semantics' in their designs. Examples are *The Gem Cutter* (Evans et al. 2007), *Visual Haskell* (Reekie 1994) and the more recent Viskell().

Targeted at non-programmers, however, are *Tangible Values* (TVs) of Elliott (Elliott 2007). This work can complement this work. Tangible values unify program creation and execution, and are visual and interactive manifestations of pure values, including functions. However, the design is also not self-explanatory. Moreover, it does not give the user access to actual function definitions. The focus of this paper forms an important step towards the latter.

7. Conclusion

This paper has presented a *maramafication* of an important part of FPLs: the construction of well-typed algebraic data structures based on type definitions with at most one type parameter. Maramafication means the design of visual 'twins' of existing programming constructs using spatial metaphors rooted in common sense or inborn spatial intuition, to achieve self-explanatoriness. This is, among others, useful to considerably reduce the gap between programmers and non-programmers in the creation of programs, for educational purposes or for invoking enthusiasm among non-programmers.

The paper presented the most important parts of a proof of the congruence between the mentioned maramification and the original expressions.

8. Future Work

Further future work may include: extension to multiple type parameters, allowing constructors with polymorphic arguments, further empirical study into understandability and usefulness of the constructs by non-programmers and implementation of the presented design in an editor.

Acknowledgments

Jesse Nortier for his excellent work on the 3D rendering of the designs.

References

Peter Achten, Pieter Koopman, and Rinus Plasmeijer. *An Introduction to Task Oriented Programming*, pages 187–245. Springer International Publishing, Cham, 2015. ISBN 978-3-319-15940-9. doi: 10.1007/978-3-319-15940-9_5. URL `http://dx.doi.org/10.1007/978-3-319-15940-9_5`.

T. H. Brus, Marko C. J. D. van Eekelen, M. O. van Leer, and Marinus J. Plasmeijer. Clean: A language for functional graph writing. In Gilles Kahn, editor, *FPCA*, volume 274 of *Lecture Notes in Computer Science*, pages 364–384. Springer, 1987. ISBN 3-540-18317-5.

Conal Elliott. Tangible functional programming. In *International Conference on Functional Programming*, 2007. URL `http://conal.net/papers/Eros/`.

Luke Evans, Bo Ilic, and Edward Lam. The gem cutter a graphical tool for creating functions in the strongly-typed lazy functional language cal. Technical report, Business Objects, 2007. URL `http://resources.businessobjects.com/labs/cal/gemcutter-techpaper.pdf`.

P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language

(Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992. URL `http://www.haskell.org/definition/haskell-report-1.2.ps.gz`.

H. John Reekie. Visual haskell: A first attempt. Technical report, 1994. URL `http://ptolemy.eecs.berkeley.edu/~johnr/papers/postscript/visual-haskell.ps.gz`.

Jurriën Stutterheim, Rinus Plasmeijer, and Peter Achten. *Tonic: An Infrastructure to Graphically Represent the Definition and Behaviour of Tasks*, pages 122–141. Springer International Publishing, Cham, 2015. ISBN 978-3-319-14675-1. doi: 10.1007/978-3-319-14675-1_8. URL `http://dx.doi.org/10.1007/978-3-319-14675-1_8`.

Marko C. J. D. van Eekelen, Marinus J. Plasmeijer, and John van Groningen. 2011. http://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf.