High-Level Modelling for Typed Functional Programming

Yusuf Moosa Motara (✉)

Department of Computer Science, Rhodes University, Makhanda 6140, South Africa
y.motara@ru.ac.za

Abstract. There is currently no way to model the high-level structural design of a functional system. Given the strong links between functional programming and mathematics, it is hypothesised that the language of mathematics can provide insight into how a functional system might be modelled. The approach is successful and both philosophy and the language of mathematics are used to identify the necessary modelling concepts and briefly outline some modelling notation alongside a small case study.

Keywords: Modelling · Structural modelling · Functional programming · Type model

1 Introduction

The structural design of functional systems—i.e., those written in a functional programming language, and/or obeying the conventions of the functional paradigm—cannot be modelled: neither modelling notation nor tools exist for this purpose. This paper contributes to addressing that research gap.

At the outset, the breadth of the research gap should be understood: there is no standard notation to model either typed or untyped functional systems at a high level, behaviorally or structurally. This is not to claim that no notation whatsoever exists: on the contrary, many ad-hoc notations exist (see, for example, the pages of [22, 44], and many other such works), and different kinds of diagrams such as UML’s Process Diagram or BPMN have been repurposed to fit the needs of the moment. None of these notations claim to be able to model functional systems *in general*, because they cannot. There remains, therefore, no standard way to model the structure of a functional system.

This work is part of a broader effort to allow functional programs to be modelled in an intuitive way. It focuses specifically on the *structural* modelling of *typed* functional programs at a *high level* and proposes a general, standard notation that should be applicable to all functional languages.

Computation in an object-oriented language is based on the idea that progress is made through communication between objects, each of which collaborates to achieve the goal of the system. By contrast, computation in a typed

functional language is based on the idea that values are transformed through pure functions, potentially resulting in a final value that is the end goal of the system¹.

There are several important features of a typed functional system which are either difficult or impossible to model using structural UML diagrams. These features are present in most typed functional languages. UML's *enumerations* are insufficiently powerful to represent sum types since the cases of such types are often linked to other data; the Composite design pattern has to be used instead. There is no good notation for modelling the functional semantics at the heart of functional programming, including lambda functions, nested functions, higher-order functions, partially-applied functions, composed functions, and closures. Most importantly, every structural UML diagram implicitly or explicitly requires some mapping of the domain to the idea of class- or object-like containers which are not necessarily present in functional languages. This means that while UML can be used to represent a wide range of systems, including real-world systems that do not involve computers, it must model them primarily as *entities* with relationships rather than as *operations* that involve entities. The underlying problem may be that the philosophy underlying such diagrams is fundamentally object-based [41].

The relationship between functional programming and modelling is a fundamentally mathematical one. This is in contrast to the relationship between object-oriented programming and modelling, which may be described as thoroughly pragmatic. The dominant object-oriented modelling language, UML, was created through a process of consensus and refinement [bezivin_uml_1999] and links to mathematics [11] or an underlying philosophy [12] have involved a great deal of retroactive patching. By contrast, the roots of functional programming are found in the lambda calculus [3], a mathematical description of what it means to compute. The combination of typing (via the Hindley-Milner type system [25]) led to the development of typed functional programming based on rigorous type theory [2].

The contributions of this work are as follows:

- A strong philosophical grounding is sought, found, and mapped between the language of mathematics and typed functional programming;
- A set of concepts common to all typed functional programming and not tied to any specific language are identified;
- A high-level structural modelling notation is briefly described, along with a small case study to demonstrate applicability.

The remainder of this paper is structured as follows. Section 2 justifies the approach used to search for a modelling solution. Section 3 discusses modelling, and explicitly situates and scopes the work within a particular modelling context. Section 4 looks at related work. A philosophical basis is laid out in Sect. 5 and an exploration of mathematical language is found in Sect. 6. The philosophy and language are combined in Sect. 7 to identify a set of concepts for modelling.

¹ Event-based functional languages, such as Elm, omit the idea of a final value.

A notation for modelling these concepts is first theoretically justified and then proposed (Sect. 8). A short case study is presented in Sect. 9 and conclusions are drawn in Sect. 10.

2 Approach

The majority of this paper traverses the ground of philosophy, language, mathematical discourse, and design—with a case study at the end. It takes time to build bridges between these areas and explain them in sufficient detail for our purposes. Why should such a long and winding path be taken, rather than simply finding some notation that looks usable and causes practitioners to nod their heads?

The design of systems is ultimately a purposeful activity: every system has some reason for existing, even if that reason is a purely exploratory one. This design will typically follow the ways of thought that have been developed by the designer [33], and will often be restricted by those ways of thought as well [4]. It is therefore of primary importance to ensure that system stakeholders are encouraged to think in a way that facilitates a design which can easily be encoded as a computer system, and which discourages ways of thinking which are difficult to encode. Three categories of common design-related thinking are categorized here:

1. Thinking in a problem domain. A person may think of a problem in terms of their field of specialization, such as accounting or botany, or in terms of some other influence, such as fishing or family discussions. The ease of translation into a computer system is dependent on the degree to which system language(s) are able to encode the original thinking.
2. Thinking in a programming language. A developer may think of a problem in terms that a language makes available to them. Ease of translation into a computer system is assured, but the degree to which that system reflects the actual problem domain will be relative to how accurately the problem domain can be expressed in the language. A further issue is that a design of this type may be inaccessible to stakeholders without the requisite technical knowledge.
3. Thinking in a notation. A stakeholder—whether developer, business analyst, or other person—may think of a problem in terms that a notation supports. Ease of translation into a computer system is related to the ease with which notational semantics are translatable into computational semantics, and the degree to which the system reflects the actual problem domain is relative to how accurately the problem domain can be expressed in the notation [35].

Domain-specific languages may successfully be used to bridge (1) and (2) [18], but introduce problems of their own and should therefore not be used without due consideration [24, p. 320]. It is category (3) that is most accessible to the widest audience, and which this paper is concerned with. However, the accessibility of this category introduces problems of its own since stakeholders must

use the same underlying philosophy to be able to participate in any modelling process that uses the notation. That philosophy is, in the object-oriented world, provided by an underlying theory of objects [40]—although the details of what an “object” itself is have been contested [1]. Without a shared underlying philosophy, it is argued that stakeholders will have difficulty conceptualizing their ideas notationally and will become frustrated with a notation rather than using it effectively.

An appropriate philosophical backing for modelling is important [12], and deep concern for philosophy when proposing a high-level model is not simply theoretical: it is rooted in our field’s history [10, p. 73]:

Most of our current ‘modelling languages’ (MLs) date back to the 1990s and therefore claim to be ‘general purpose’—a prime example being the Unified Modeling Language (UML) from the Object Management Group (OMG) [9]. Although claiming to support a wide range of modelling abstraction levels (i.e. analysis and design) in the one package, its history of development clearly indicates that it is highly focussed towards (low-level or detailed) design and even implementation (e.g. Java and C++-style concepts are evident). Some Domain-Specific Modelling Languages (DSMLs) show the same bias, when presented as a ‘UML Profile’.

(references in original). The core issue here is that languages *currently* popular should not have an outsize impact on notation that may outlive them; without the stabilizing influence of an explicit underlying philosophy, the link between notation and language becomes contaminated by specific language constructs. What looks “usable” or “familiar” today is often a function of the programmer’s experience and should not be discounted, but should not be considered as a sufficient reason for particular notation to exist.

The link between problem domain and notation is also an area of concern. If the notation makes it easy to model the problem domain in a way that is unsuitable for implementation, conflict ensues [35, p. 238]:

[I]t was not just that UML contained semantic ambiguities and inconsistencies, but rather that the increased prominence given to particular modelling notations had in turn placed a premium on carrying out certain kinds of analysis and design activity. Analysts were enthusiastically adopting new approaches to conceptualising their system, eventually becoming trapped in unproductive arguments over the objects populating the system and the proper representation of the control structure of the system. Designers were then refusing to implement the models produced by the analysts, since it was often impossible to map from use case models and sequence diagrams onto anything that a conventional software engineer would recognise.

Once again, such issues could be mitigated with reference to some shared underlying philosophy that guides analysis, design, and implementation.

Finding a suitable underlying philosophy and creating a conceptual bridge to link problem domain, language, and notation is seen as a way of avoiding some of the difficulties that attended the birth, development, and use of UML. The specific notation itself, as an artifact of this search, is relatively unimportant as long as *some* viable notation can be demonstrated. This is not stated to dismiss the importance of a concrete notation, but to elevate the importance of its foundation. By analogy, one can say that any specific software system is relatively unimportant when compared to the importance of the principles of software design which guided its creation and the creation of thousands of other software systems.

Furthermore, researchers with access to the underlying foundation of a notation are free to propose other notation that is based upon it, and that notation is likely to be coherent with—or superior to—what is proposed in this paper. Researchers are also free to critique, supplement, and revise the foundation, thus making for a firmer foundation for future notation. However, researchers *without* access to the foundation of a notation must simply guess at the philosophical underpinnings and linkages by working backwards from the notation, leading to unnecessary misunderstandings and unproductive competing proposals on implicit grounds of language-preference, problem-domain utility, and so on.

3 Modelling Context

Models obtain their power through their ability to “be used in place of what they model” [28]. This is most often a system which is part of a larger domain of interest which, itself, is part of the totality of reality [11]. A useful general modelling language should be able to model many domains of interest and, ideally, much of reality. This aligns well with the idea of a general programming language, while not detracting from the importance and utility of specialist modelling languages and programming languages.

Models may be either structural or behavioural. A structural model expresses the elements of a model that exist and the way in which elements of the model relate to each other. A behavioural model expresses the way in which a system accomplishes a task. This work is concerned with structural modelling only. With that said, the term “structural” may also encompass behavioral aspects of a functional system. This is because functional programming is declarative, and describing the relationships between declarations also goes some way towards expressing how a task is accomplished. Nevertheless, the distinction between behavioural and structural is a useful one since it excludes models which focus primarily on behaviour rather than structure.

A model may be used as either a description or a specification [28]. The distinction is in where the truth of the system lies: in the former case, it is with the system, and in the latter case, it is with the model. Many excellent specification languages already exist for the functional paradigm (e.g. [15, 29]), and this work attempts to be complementary to these. It proposes, therefore, a descriptive modelling language. This does not preclude a role similar to that

of a specification since a descriptive model which is developed earlier in the development process may be used as a prescriptive model—a loose specification, if so desired—and can be regarded as having a more descriptive role during and after development [28].

Following on from Stachowiak [37] as reported by Kühne [19], a model must have the qualities of mapping, reduction, and pragmatism: it must be based on some system (“mapping”), reflect only the relevant parts of that system (“reduction”), and be usable in place of the original system with respect to some purpose (“pragmatism”). Kühne simplifies both mapping and reduction under the term “projection”, and sets out the view that abstraction may be described through function composition [19, p. 371]:

$$\alpha = \tau \circ \alpha' \circ \pi$$

Model abstraction (α) consists of projection (π), some further abstraction (α') on elements (including relationships), and a translation τ to another representation, i.e., the modelling language. With projection π we associate any filtering of elements both reducing their number and individual information content.

Given some actual system S and some notation N , with an abstract model M , one can think of these as having the types $\alpha : S \rightarrow N$, $\pi : S \rightarrow M$, $\alpha' : M \rightarrow M$, and $\tau : M \rightarrow N$.

The “further abstraction (α')” aspect requires additional explanation: why is it required? Kühne explains the necessity by differentiating between two kinds of model:

- **Token models** are those where there is “a one-to-one correspondence between relationships and elements in the model $\mathcal{M}$ and a subset of these in system $\mathcal{S}$ ” [19, p. 373]. Token models can therefore be used as direct representations for the actual system, and this remains true even when token models refer to other token models, or when the “mapping” part of projection elides or combines some of the finer details of the actual system. Token models capture the system-specific aspects of a system. A blueprint, for example, is a token model.
- **Type models** are those which operate by trait-classification rather than by replication and/or combination of a system’s elements. A trait may be identified, from an object-oriented point-of-view, with a UML interface in a class diagram: it specifies which behaviours should be present *without* specifying which specific class or properties must be used to provide those behaviours. It therefore captures “*universal* aspects of a system’s elements by means of *classification*” [19, p. 374] (emphasis in original).

In a token model, the α' function is unnecessary since projection is sufficient; there is no further abstraction and one can regard α' as being the identity function. In a type model, however, the α' function serves to classify elements

and relationships prior to translation. The proposed modelling language in this work aims to show a high-level design-oriented view, and is therefore closer to a type model than a token model.

Lastly, a model must have a particular intention: a reason for existing or a purpose that it must be fit for. The intention of a model is “a mixture of requirements, behavior, properties, and constraints, either satisfied or maintained by the [model]” [28, p. 350]. The intention of the proposed notation is to allow a modeller to express the underlying design of a system in a way that is comprehensible to others.

In summary, a model created using the notation described in this work would be structural; applicable to many domains of interest; descriptive; type-model focused; and aimed at exposing a comprehensible system design.

4 Related Work

This section has been split into two subsections. The first of these considers the applicability of mainstream modelling languages which were developed without considering either functional programming or its mathematical basis. The second broadens the related work to consider mathematically-oriented notations and notations which have been specifically designed with functional languages in mind.

4.1 Mainstream Modelling Languages

UML’s behavioural diagrams are more useful for representing the semantic structure of a functional program. The Interaction and State diagrams, in particular, are easy to adapt for the structural modelling of simple functional systems. Interaction diagrams could be used to represent functions as blocks, with labeled data going between them; conversely, State diagrams could be used to represent data as blocks, with labeled functions joining them. These diagrams do not scale: more complicated systems involving sum types with four or more cases quickly become a nightmare of lines, diamonds, and boxes.

There are other diagrams which are meant to represent processes, albeit in more limited contexts. Two of the most popular of these notations are Process Diagrams [30] from Business Process Modeling Notation (BPMN) and Data Flow Diagrams (DFDs). Both DFDs and Process Diagrams are behavioural models rather than structural models: they specify the way in which data flows between entities, rather than the relationships between entities themselves. As behavioural diagrams for functional programming, they have some merit; for example, the restrictions on how processes may be used at the sentence level map admirably to the way in which functions operate [21, p. 86]:

Processes cannot consume or create data. That means the process must have at least 1 input data flow (to avoid miracles), at least 1 output data flow (to avoid black holes) and should have sufficient inputs to create outputs (to avoid gray holes).

Nevertheless, both DFDs and Process Diagrams make function-type inputs or outputs difficult to represent naturally. The former is better at this than the latter since a DFD is to be viewed in conjunction with a Data Dictionary wherein the relevant function-type can be given a suitable name. Although this may be adequate, it is not necessarily a good fit since function inputs and outputs are very important and, ideally, should not be relegated to a separate document.

Fig. 1. Example of Tonic graphical notation (reproduced from [38, p. 36])

Tonic visualisations [38], inspired by BPMN and developed as the complement of GiN [13], are a specific adaptation that targets task-oriented programming (see Fig. 1) and are suitable for expressing aspects of functional programming.

4.2 Functional Programming and Modelling

Due to its mathematical roots, functional programmers have tended to use mathematics as the most intuitive and general modelling tool at their disposal. Category theory [23] provides a visual way to represent and reason about arrows and objects which are analogous to functions and types in a typed functional programming language. However, this visualisation is suitable only for considering functions and types, and there is no obvious way to extend its scope. Figure 2 demonstrates this by reproducing a representative figure from [23, p. 16]. It succinctly defines what a natural transformation $\tau : S \rightarrow T$ looks like by relating the categories S and T , but its ability to describe the domain itself has not been tested. The most promising works in this direction are likely [6] and [36] which try to make category theory accessible to others, enabling them to “think in categories” in the same way that budding programmers are encouraged to “think in objects”. While this is certainly a worthwhile goal, it nevertheless keeps functional programming opaque to those who have *not* been taught to think in such a way. Modelling is a complementary way to improve the accessibility of functional programming.

Motara [27] suggests a novel way to use string diagrams [34] to represent lower-level function manipulation. Their work focuses on behavioral modelling and the syntax and semantics are not applicable to structural modelling. Other

$$\begin{array}{ccc}
c & S_c & \xrightarrow{\tau_c} T_c \\
\downarrow f & \downarrow S_f & \downarrow T_f \\
c', & S_{c'} & \xrightarrow{\tau_{c'}} T_{c'}
\end{array}$$

Fig. 2. Modelling in category theory

works which use string diagrams are [14] and [34] where they are used to increase the accessibility of category-theoretical manipulations in a novel way. An alternative graphical approach is taken by Eklund [5] in a paper which is focused on understanding monadic composition. Once again, however, all these cannot readily be extended to the modelling that is desired in this work.

Type-driven development and domain-driven design, as exemplified by [8, 44] for functional languages, model within a programming language. They pragmatically take advantage of type systems to explicate a domain, creating small domain-specific languages that map naturally to the domain that is being modeled. While this style of modelling is successful, there is no common notation for it and it is sometimes only accessible to developers rather than a broader audience (see, for example, Chapter 4 of [8]).

5 Philosophical Underpinnings

A late-Wittgensteinian [43] language-centric approach will be taken as a philosophical basis. This approach was selected based on Wittgenstein’s language-centric and context-dependent view of problems and philosophy. This view, it could be argued, is a good fit for the functional paradigm since functional designs often strongly emphasise language use: see, for example, [9, 16, 32]. Furthermore, Wittgenstein’s later philosophy is well-regarded in philosophical circles [20] and has the benefit of having been examined and analysed for over half a century. Such a philosophical foundation is more likely to be stable than one which is created ad-hoc for a particular goal.

Wittgenstein’s philosophy can be broken up into “early” and “late” eras, with the early Wittgenstein approach finding its zenith in the *Tractatus Logico-Philosophicus* [42]. Later Wittgenstein turned away from significant aspects of the earlier work and can be most clearly seen in *Philosophical Investigations* [43], which forms the basis of this summary. That work is written as a series of numbered paragraphs and, for the purposes of this work, it is only necessary to consider paragraphs up to ≈ 140 .

Philosophical Investigations is written as a response to both the attempted formalization of language into strict logical propositions and to the broader consideration of what constitutes philosophy itself. Wittgenstein therefore writes about language itself, how it is constructed, and how it is meant. To assist a reader who wishes to consider the source material, the original numbered paragraphs from [43] which form the basis of each part of this summary are included in parentheses for the remainder of this section.

In Wittgenstein's estimation the fundamental starting-point is to consider language not as words strung together within a grammatical framework, but as moves within a *language-game*. A language-game is the "game" of communication that sets up a context within which language is used and within which one party or another may make "moves", and within which there are many varieties of expression (23). For example, the word "fire" has a very different meaning depending on whether one is in a crowded movie theatre or at a shooting range or in a Human Resources meeting. Similarly, a word may be a command or a question or something else entirely, with its meaning hinging on expression or tone or something else (21); and how we categorise words depends upon both the aim of the classification and upon our own subjectivity (17). Within a language-game, the meaning of a sentence is more important than the way in which the sentence is constructed and, indeed, two sentences or words which mean the same thing but are otherwise different should be considered to be the same (20, 24, 138). All language-games depend on implicit presuppositions (31) which may naturally be assumed given strong enough evidence (33) for them. A word by itself, depending on the situation in which it is used, may be a sentence on its own (49).

Language-games may include names. A name *signifies* a thing, but is not that thing, much as the name-tag which is attached to a thing is not the thing itself (15, 40). A name continues to exist because the meaning of a name continues to exist, even if the bearer of the name no longer exists (41, 55). This aspect of language allows us to talk about bearers which have been destroyed. A name has no meaning whatsoever outside of a particular language-game, and the mere naming of something is not—until the name is used—a move in a language-game (49). Since all names have a meaning which must necessarily exist for them to be used within their language-game, it makes no sense to talk about whether a name "exists" or not (50, 57, 58). The meaning behind a name, and how to use it, must necessarily be known before a name is defined (31). Similarly to names, all words in a language game are ways to represent other things (50).

Demonstratives (such as "that" and "here") are a special case of words which require a bearer; however, this fact alone does not make a demonstrative into a name (9, 45). Each language-game may contain words that have specific uses within that game (10, 11, 43), and very little is gained by considering them to be more similar than they are (14).

There is nothing that is natively composite, outside of a particular language-game (47). Even within a particular language-game, what is "composite" may be defined variously as the game progresses (48). What is important is not the "simplified" or "composite" forms of things, howsoever they may be defined, but the avoidance of misunderstandings (48) since there may be times when a "simplified" form (e.g. "brush and stick") is neither more fundamental nor more simple than the "composite" form (e.g. "broom") (60–63); but this depends, ultimately, on the language-game that is in use (64).

An inexact meaning is still eminently usable, and the drawing of boundaries around it does not necessarily make it more useful, except in the more specialized

case where a word has a niche meaning that is amenable to such boundaries being drawn (69, 81, 139). It is often context or examples—the manner in which it is used—which make the meaning most clear (29, 71). Meanings should therefore be separated only to the extent that, within the language-game, they are needed to avoid misunderstanding (87, 88, 98, 99). A word may have many meanings, each of which independently support the word, and no fixed meaning (77, 79, 87); and the same word, used in a different way, may result in a different meaning (140).

All philosophical problems are, in fact, problems of language (109). These arise because philosophers insist on trying to understand concepts and words—such as “truth”, “world”, and “self”—in isolation and divorced from any language-game, which is precisely where they are most meaningless (105–108). Instead, philosophers should restrict themselves to describing (and never explaining) things within the context of their language-game (109, 125); this is the only way in which problems may be solved. Any such solution is one way in which a problem may be solved, but not necessarily the only way (131, 132), and is not generalisable to largely-unrelated cases (133).

6 The Language of Mathematics

The Language of Mathematics [7] demonstrates a way in which arbitrary symbolic and textual mathematics, as written in standard works and textbooks, can be parsed, understood, and represented with full semantics using Discourse Representation Theory [17]. Critically, the work uses linguistic theory to understand mathematics as a language and then encode it within a modified Discourse Representation Structure (DRS). Such a DRS is capable of translating and encoding the lambda calculus [3], the basis of all functional programming, as well as type-theoretic logic [2]. *The Language of Mathematics* identifies as many features of natural-language mathematics as possible, and strives to find an encoded form of those features without loss of semantics. This work takes precisely the opposite approach and asks: if we consider a functional representation to be the encoded form of a system, is it possible to obtain a more natural language form without loss of semantics?

6.1 Natural Language Structure

For the convenience of the reader, certain terms will be written in **boldface**. These terms are those which will be particularly important in later discussion.

One apparent difference between mathematical language and a program is that mathematics is typically written either in the form of an argument, with various statements bolstering some conclusions or results, or in the form of an exploration where background knowledge is described. These two forms map neatly to the ideas of a functional program and a functional library respectively: the former makes at least one argument about inputs and transformations and outputs, and the latter describes tools that may be used in the course of such an argument.

Mathematical language is written in one of two **modes**: formal and informal. Formal statements are possible to evaluate objectively. Informal statements, such as “It is *interesting* that the Fibonacci sequence appears in many natural contexts”, give opinions but are not subject to computational evaluation.

Mathematical language may be textual—expressed in English²—and/or symbolic, with symbols being used “to abbreviate material that would be too cumbersome to state with text alone” [7, p. 17]. Symbols are often **embedded** within textual material and their **abbreviative** use makes it much easier to convey complex ideas in a small amount of space; indeed, the argument is made that “modern mathematics would quickly become unreadable” [7, p. 18] without such use of symbols. Symbols which represent terms can be embedded in contexts that accept a noun, and symbolic formulae can be embedded in contexts that accept a clause or sentence. Symbolic terms often carry **presuppositions**; for example, “ $\sqrt{x}$ ” presupposes that x has a square root, i.e. that x is nonnegative” [7, p. 31], assuming that a real-valued solution is desired.

An important feature of mathematical language is **adaptivity**: the way in which the textual and/or symbolic lexicons are updated with more nuanced meanings as additional mathematical definitions are encountered. For example, $\frac{3}{4}$ may initially be understood as “three parts out of a four-part whole”, but may later be understood to also mean “three divided among four entities” when the appropriate mathematical definitions are encountered. This also brings into focus the critical importance of **definitions** in mathematical language.

Mathematical prose is commonly organized into blocks. The most important blocks used in mathematical language appear to be:

- **Lemma**, denoting a minor result that is useful on the path to a greater goal;
- **Definition**, which updates textual and/or symbolic lexicons;
- **Theorem**, denoting a major and important result;
- **Proposition**, denoting a result that is more important than a Lemma, but less important than a Theorem;
- **Corollary**, denoting a consequence that naturally follows from the truth of a Lemma, Theorem, or Proposition.

Blocks are often numbered so that they can be referred to unambiguously from other parts of an argument. “Proof” blocks, denoting the means by which a “lemma”, “theorem”, or “proposition” are shown to be true, exist only as part of these other blocks. “Proof” blocks are only used to separate the result from reasoning and are not numbered. Such a block is therefore more usefully regarded as a part of one of the other named blocks than as having an independent existence. Most blocks represent the *behavioral* component of mathematical text and make an argument that links the entities from “Definition” blocks together.

Mathematical blocks often involve the use of **variables**. These are used as a form of anaphor³ and are often scoped to the block itself. It is asserted that

² Or other natural language.

³ The use of words to refer to other entities without naming them, e.g. “The lady ate food that *she* enjoyed”. Pronouns are a form of anaphor.

“[t]hey cannot be eliminated precisely because anaphor is not powerful enough to replace them” [7, p. 31] and this observation is likely to be true in the case of a functional system as well.

A careful reading of [7] reveals several intra-block formal-mode mathematical rhetorical constructs beyond (and including) those obviously classed as rhetorical (see [7, p. 77–82]). The identified rhetorical constructs are:

- **Variable definition.** Variables are often defined intensionally (i.e. by predicate). *Examples:* “Let $x \in \mathbb{N}$ ”; “Let K be a ring”.
- **Naming.** This names a particularly important result, often a Theorem, so that it can be referred to by name. *Example:* “Theorem 2.4 ‘Sigmund’s Paradox’”.
- **Presupposition.** This is used to attach a restriction to the use of a construct. *Example:* “ $\sqrt{n}$ is defined for all $n \geq 0$ ”.
- **Consequence.** This qualifies the condition(s) under which a definition is true. *Example:* “If $I \times A = A$ and $A \times I = A$, then I is the identity matrix”.
- **Cross-reference.** This is used to refer unambiguously to a result demonstrated elsewhere. *Example:* “By Sigmund’s Paradox (Theorem 2.4), ...”.
- **Conclusion.** This is the final result of a Lemma, Proposition, or Theorem.
- **Product type.** This creates a named grouping. *Example:* “A polite sentence P consists of a subject, a predicate, and a politeness modifier”.
- **Sum type.** This creates a discrete, named set of elements. *Example:* “We say that 2, 3, 5, 7, and 11 are members of the set of initial primes Q ”.

Note that common sentences such as “Given a set of sets S , the powerset $P(S)$ is the set of all subsets of S ” may contain more than one of the identified constructs.

7 The Bridge over the River Wittgenstein

A bridge between *The Language of Mathematics* (LoM) and typed functional programming (TFP) will be created in this section, using *Philosophical Investigations* (PI) to go between the two. The intermediary philosophical link is crucial for being able to take ideas from one side to the other in a principled and theoretically justifiable way. LoM presents a coherent and clear account of mathematical language; PI provides a coherent and clear account of natural language. To the best of the author’s knowledge, there is no comparable account for TFP, and attempting to establish one *a priori* risks creating a biased design based on the author’s subjective experiences with functional languages.

Whenever possible, the link should not be made *directly* between TFP and LoM because this risks conflating the former with the latter. While the two may be similar, they are not the same, and pretending that they are serves no purpose. PI serves as a guard against this tendency and forces the modelling to be done on the level of a human-focused language-game. Conversely, attempting to establish a typed functional programming language-game with only the philosophy of language-games makes misclassification errors more likely and unnecessarily

discards the touchstone of mathematics. Anaphora, for example, may be sought and “found” in typed functional programming when variables [7, p. 31] are likely to be a more appropriate abstraction.

Mathematics cannot be fully understood out of its context or in an isolated way; the same is true of words in a language-game, and functions and types and their relationships in typed functional programming. PI describes many varieties of expression, and mathematics restricts itself to either the formal or informal. The closest analogue to an informal mode in TFP may be programming comments.

Wittgenstein’s discussion of ambiguity can be broken down into (at least) the following distinct points:

1. A word may have many meanings, each of which support the word independently, or no fixed meaning.
2. A word may have an inexact meaning, as long as it can be distinguished from other words.
3. A word’s meaning may change as the game progresses.

All of these are traits hold true in mathematics: for example, “prime” has many meanings, “interesting” has no fixed meaning, “abstract” may have an inexact meaning, and examples of adaptivity have already been given. In typed functional programming, the fold operation (and other parametrically polymorphic operations) can have many meanings, shadowing makes it possible for a name to have localized and global meanings, and the meaning of words such as “authorised” or “valid” may change as more moves are made. Yet, just as in the case of a language-game, essential meaning is preserved despite—and sometimes because of—ambiguity.

Mathematical blocks do not appear to have any explicitly described counterpart in the philosophy of language-games. However, *Philosophical Investigations* consists of numbered paragraphs which are set up such that they may reference each other, thus implicitly taking on a structure of blocks and cross-references. It can therefore be said that a paragraph is analogous to a block, and the structure of the text forms a presupposition [43, par. 31] that is important for the semantics of the text. On the mathematical side, the importance of blocks in structuring mathematical language is overwhelmingly clear: there are few mathematical texts that do *not* follow this convention, and cross-referencing between blocks is critical. Similarly, the importance of numbered paragraphs in Wittgenstein’s own implicit language game is critical for cross-referencing purposes.

On the typed functional programming side, a plausible analogue is ostensibly the abstraction of packages/modules/namespaces which most languages have. This analogue is not without its problems, however: such containers may be used to package a wide variety of functionality from GUI components to algorithms to service interfaces. It seems unreasonable to insist that each of these forms is either the same as all the others, or to create distinctions—with no principled basis—between “kinds” of containers. There therefore appears to be no direct analogue for mathematical blocks.

Mathematical rhetoric is used to form sentences through which “moves” are made within the language-game of mathematical language. Rhetoric links blocks, which delineate an overall structure, and argumentation together, building on already-demonstrated results to develop a richer mathematical narrative. Types and functions perform a similar role in typed functional programming; see, for example, [44] where functions are used to transform types (and hence meaning) from more basic forms to more sophisticated ones. Types naturally encode rhetorical “sum type” and “product type” constructs, and functions naturally encode the rhetorical constructs of “presupposition” (as logic) and “conclusion” (as return values). However, simple functions and types do not allow one to express *general* narratives such as “Any valid calculation must remain within particular bounds”. Parametrically polymorphic types, combined with functions that obey certain “laws” by convention, give rise to applicative functors, monads, and other such constructs. These constructs can be used to express richer narratives. Ironically, these constructs can also be formidable barriers to understanding. The fundamental issue, covered well in [31], is subtle but pervasive throughout typed functional programming: abstract knowledge of parametrically polymorphic functions and the transformations that they *potentially* enable is not sufficient to combine them sensibly or construct a cohesive narrative from them. The typed functional programming domain has many ways to describe functions and transformations (lambda, higher-order, functor, applicative, arrow, ...) but no way to describe how to link these into a cohesive narrative. The “moves” made by sophisticated typed functional narratives are difficult to discern because a *design*-relevant rhetoric to describe those moves is almost entirely missing.

Mathematical language uses variables instead of “demonstratives” and similar anaphora, but the typed functional paradigm lacks a similar exclusive way to identify other entities. Instead, features such as arguments, closed-over values, types, and namespacing are used to refer to particular kinds of entities in different contexts. A similar situation occurs when considering the idea of language-game names, which can be neatly mapped to textual/symbolic definitions in mathematical language. Typed functional programming defines multiple named entities such as named functions, types, and modules/packages, all of which may be used as names in different contexts.

7.1 Proposed Basis

The following principles were applied to arrive at a suitable TFP modelling language:

1. If similar language exists in LoM, PI, and TFP, then it is clearly important in all three and should be represented in modelling language for TFP.
2. If similar language exists in LoM and PI, but not in TFP, then it is likely to be part of a modelling vocabulary that must be developed for TFP and should be represented in a TFP modelling language.
3. If certain language exists only in LoM, then it is likely to have a use only in LoM and should not be included in a TFP modelling language.

4. If multiple expressions of a language construct exist in LoM, and fewer analogous expressions exist in PI and/or TFP, then it is possible that the expanded set of language constructs is only necessary in LoM because of the specific requirements of mathematics. On the basis that meanings should only be separated to the extent that this is necessary (see [43, par. 88, 89, 98, 99]), a reduced set of language constructs—ideally, only those which are necessitated by PI—should be represented in a TFP modelling context.
5. If multiple expressions of a language construct exist in TFP, and fewer analogous expressions exist in PI and/or LoM, then it is possible that the expanded set of language constructs is only necessary in TFP because of the specific requirements of TFP. This does not necessarily mean that such constructs are necessary in a *modelling* or design context. On the basis that meanings should only be separated to the extent that this is necessary, a reduced set of language constructs—ideally, only those which are necessitated by PI—should be represented in a TFP modelling context.

Principles (4) and (5) are the most controversial since the case could be made for an expanded TFP modelling language rather than a reduced one. Such a case has not been made because it is thought to be better to create a smaller initial language that can be expanded rather than a larger language that may later have constructs removed from it.

Recall that Kühne’s model of abstraction [19] involves projection (π , consisting of both mapping and reduction), then further abstraction (α'), and lastly translation (τ) to the modelling language. This section jointly considers both π and α' , with the goal of outlining a standardised design language and vocabulary which can later be translated into a modelling language. That translation must take into account additional design factors around notation—the “Physics of Notation” [26, 39]—and will be presented in Sect. 8.

A majority of concepts and ideas can pass seamlessly, with a one-to-one correspondence, over the philosophical bridge that links LoM and TFP. A π function thus encompasses modes, symbols, definitions, and variables.

The following points sketch the outlines of a principled α' :

1. High-level design rhetoric is almost entirely missing from TFP, but exists as sentences in both LoM and PI. Such rhetoric must be created, but much of it is used in LoM in a *behavioural* context and is not necessary for a *structural* model. Two rhetorical constructs, “sum” and “product” distinction, exist already in TFP.
2. The concept of structured blocks exists in LoM, but has an implicit existence as paragraphs in PI. The primary purpose of blocks in both is to separate and allow for easy cross-referencing. Relevant block structures must be created, and must be amenable to cross-referencing.
3. LoM variables and definitions both have multiple representations in TFP. They will be coalesced into the simpler representations from LoM.

A case has already been made for including definitions in the modelling basis. Natural analogues for other blocks were not found in typed functional

programming, but some commonalities clearly exist between natural and mathematical language. The remaining blocks were therefore classified as follows:

- “Lemma”, “proposition”, and “theorem” blocks differ in the importance accorded to them and are also relatively hierarchical. While some natural language texts do contain such divisions, many others do not, and PI has little to say about them. It is plausible that they could be coalesced into a single construct.
- “Corollary” blocks exist in LoM, but have no explicit existence in PI. They should therefore be ignored.

8 Notation

The notation is guided by best-practice principles from the literature, which will be detailed first. The actual notation follows as a separate subsection.

8.1 Design Process

This work will use the Physics of Notation Systematized (PoN-S) design process [39], which aims to create workable artifacts that follow the principles of good notation suggested by Moody [26], and which is very briefly summarized below. Such principles aim to improve “the speed, ease, and accuracy with which a representation can be processed by the human mind” [26, p. 757].

The PoN-S process begins by looking at cognitive fit: whether the notation will fit the task and the audience.

Given a task and audience, the second step of PoN-S is to determine the symbols to be used in the notation. This involves three principles: semiotic clarity, semantic transparency, and perceptual discriminability. A symbol has semiotic clarity when it maps to one (or zero) concepts, and when each concept is mapped to a maximum of one symbol. Each symbol, by the principle of semantic transparency, should suggest its semantics; and each must be perceptually discriminable (i.e. visually distinguishable) from other symbols.

The symbols should ideally be enhanced to improve the speed, ease, and accuracy of their processing. This involves improving their visual expressiveness through the use of different visual characteristics (position, shape, size, colour, hue, orientation, and texture), limiting the number of symbols (graphic economy), and using text to improve the clarity and expressiveness of symbols (dual coding).

Lastly, PoN-S calls for identification of legitimate ways in which symbols may be combined. This specifically requires forethought about the complexity management of a notation: what looks reasonable for a few symbols may turn into a chaotic mess when hundreds or thousands of symbols are involved. A validation step then ends the PoN-S process.

8.2 Proposed Notation

Section 7.1 expands on both projection (π) and further abstraction (α'), leaving only translation (τ) to a notation to be considered. This section deals with that translation. The audience for the notation is *inter alia* students, developers, and business analysts; in other words, a broad and general audience which has some technical background and may be interested in software, software features, and software design, but may not necessarily be *au fait* with the details of software development. The task is to allow a typed functional system’s structure to be expressed, modified, and understood by this audience.

For ease of reference in other works, this preliminary high-level notation can be called HL0 (pronounced “hello”).

As discussed in Sect. 3, a structural model expresses the elements of a model that exist and the way in which elements of the model relate to each other. A model, in addition, must have the quality of pragmatism. This leads naturally to the question of which relationships one should express in order for the model to be useful. While there are many competing answers to this, a reasonable start might be to consider some common questions that people have about language in general, and attempt to model those relationships:

1. “What words exist, and what do they mean?” is answered by a dictionary.
2. “Which words are similar?” is answered by a thesaurus.
3. “What is the ancestry of this word?” is answered by a book of etymology.

Fig. 3. Notation for definition (left) and thesaurus (right)

The most significant notation is for definitions (see Fig. 3). The leftmost part of a definition is a *rub-’al-hizb*⁴ shape which contains symbolic notation. The *rub-’al-hizb* is chosen not only for its distinctive visual appearance, but also because it is found at Unicode codepoint 06DE. This makes it easy to integrate into text when one wishes to use parametric polymorphism.

On the right of a definition, expandable space exists for textual definition. If an alphabetic name in the symbolic definition may be substitutable with something else, then it will appear in boldface in the textual definition. Textual definitions often contain the “►” symbol which indicates different cases which are patterned on the definition, and which lead (via “→”) to some mapped

⁴ Pronounced “roob-El-Hizb”, with emphasis on the capitalized letters. IPA: [ru:b ʕl hizb].

entity⁵. The $\blacktriangleright$ is also used to indicate different cases in a sum type; if a product type exists, $\bullet$ would be used to describe the components of the group. A “_” symbol indicates a fall-through case.

Thesaurus notation (see Fig. 3) begins with a boldface title that describes the basis of the similarity. Each similar item is then listed in turn, with a short colon-prefixed description that expresses why it should be a member of the group.

Definitions should only make reference to definitions already defined above them, assuming a reading order of top-to-bottom and then left-to-right, and thesaurus elements should follow after all definitions. This causes symbols and their meanings to be (partially) ordered such that more basic definitions always precede more advanced ones: a *dictionary*. When one definition is a structural subset of another, effectively aliasing a particular part of the larger definition, then the subsidiary definition should be joined to its source by a diamond-terminated line: $\blacklozenge\text{---}\diamond$. A suggested name for this is the “alias” relationship. When one definition is built upon another, the subsidiary definition should be joined to its basis by a circle-terminated line: $\bullet\text{---}\circ$. A suggested name for this is the “relies” relationship⁶. In both cases, the black side indicates the origin. Together, these lines show *etymology* relationships. When a definition has relationships to many other definitions, it is given a thick black border instead of lines to avoid clutter. This indicates that it is fundamental to the problem domain.

In terms of notational clarity, perceptual discriminability is ensured through distinctive shapes, position, colour, and graphic economy. Semiotic clarity is good, given that this is a type model and thus classifies by traits at a high level; the use of $\odot$ and the shape of a symbolic definition enclosure relate $\odot$ intuitively to “some definition”.

Some symbols will have to be reserved by convention. The following subset of symbols covers all of the cases encountered thus far during the research:

- “(” and “)” for grouping;
- “ $\odot$ ” for indicating “some type or value”;
- subscripts for the cases of sum types;
- “ $\rightarrow$ ” for mapping cases;
- “ $\blacktriangleright$ ” and “ $\bullet$ ” for discrete cases and grouping respectively.

9 Case Study

Figure 4 refers. The case study chosen was FParsec⁷, an open-source library written in F#. This is a type model, not a token model, so only the structural

⁵ Although numeric subscripts were considered to distinguish between different parametrically polymorphic elements whenever necessary, this was found to be a more readable alternative.

⁶ A case could be made for “extension/specialization”, but this was thought to be too specific; in addition, those terms have particular and established OO meanings.

⁷ <https://www.quanttec.com/fparsec/>.

Fig. 4. Case study: FParsec

parts that were considered most important are shown. One consequence of this is that some lesser-documented parts of FParsec, such as its ability to parse expression trees with precedence, do not appear. The model is therefore not a 1-to-1 correspondence with a set of functions and types, but is an abstraction of the problem domain that can be mapped more easily to functional constructs.

The possibility of multiple implementations, all of which follow the same high-level design, is a strength of type modelling as compared to token modelling.

- We begin at the top-left with the definition of a parser, which is followed by symbolic aliases for each of the cases. These ancillary definitions help us greatly later on. The $\odot$ definition is foundational for almost all of the other definitions on the page, and is therefore given a thick black border.
- The definition of $\odot^{f.\odot}$ is the first in which we see the $\odot$, which means “some type or value”. The text of definition also indicates that the definition is implemented as an operator in FParsec. The name “f” is repeated in boldface in the definition as a way of showing that it is treated as a substitutable name. All non-alphabetic characters, such as the “.”, are considered to be fixed parts of the symbolic definition.
- $\odot!\odot$ is the first appearance of $\blacktriangleright$ and $_$. The $\blacktriangleright$ indicates the start of each new case. In text, one can read the definition as:
 - “If the $\odot$ is a parser which has successfully recognized text (i.e. $\oplus$), then the result is the type that is specified after the ‘!’ mark.”
 - “Anything else results in a failed parser (i.e. $\otimes$).”
 Notice how the $\oplus$ alias of $\odot$ is used to make the mapping simpler to read.
- Skipping ahead, the $\odot[\odot]\odot$ parser is the first one in which we see the text “Equivalent to:”. Such textual definitions are quite common in functional programming, which is to be expected since functional programming naturally lends itself to composition of functions. The implementation code does not necessarily involve composition, though the “relies” relationships do indicate that core functionality is (plausibly) delegated to $\odot\#$ and $\odot+\odot$. “Equivalent to:” should therefore be regarded as referring to semantics.
- $\odot^{\text{opt}}$ shows a case where the alphabetic text is not freely substitutable. The textual definition makes it clear that the only two values that are possible are “0” and “1”.

In most cases, some attempt is made to express the human-relevant meaning of the operation apart from its formal semantics. For example, the textual definition of $\odot^{\text{opt}}$ begins with “Turn a failure into a qualified success”.

The first six definitions thus show most of the features of the “definition” element of the notation. Due to the way in which definitions are ordered, a reader will never have to do more than scan *up* to find the meaning of a definition. At the bottom-right, one can see two etymology sections which should be easy to distinguish visually. Each of these begins with a title describing the group, and a list of definitions follows. It is envisaged that the etymology elements will be most useful for stakeholders who want to understand subtle differences and those who are interested in the different ways to achieve a particular task.

10 Conclusion

This work identified a significant research gap and set out to discover whether the underlying mathematical background of functional programming could be

used to inform the high-level structural modelling of a functional system. The answer is an affirmative: the mathematical and functional sides, both grounded in a strong philosophical foundation, lead to a relatively simple diagrammatic notation that should be easy to build upon. Perhaps more importantly, the separation of conceptual semantics from the actual modelling notation, and the grounding within a stable philosophy, should make it possible to build further diagrams or a better notation using the same underlying concepts. Three relationships, inspired by etymology, thesauri and dictionaries, have been proposed.

As stated in the introduction, this work is part of a broader effort to allow functional programs to be modelled in an intuitive way. The modelling itself has been built up as carefully as possible, avoiding the pitfalls of starting with a model that is focused on a particular problem domain or choosing a particular language to be paradigmatic of the functional paradigm. If these precautions were not taken, the history of our field shows that it would be very easy to end up with either a domain-specific modelling language (DSML) or language peculiarities creeping into a modelling notation.

Programming languages are clear to programmers. A Haskell programmer reading the following Haskell code might have a good idea of what it does, even without any further program context:

```
deleteTask p ts = [ t | t <- ts, not (p t) ]
```

However, is it clear to non-programming stakeholders? It can be *made* clear, certainly, but non-programmers have to do the work. Modelling works from the other way around. It can be a shared notation for collaboration by making the otherwise-opaque (but programmer-friendly) parts more accessible. If the proposed notation or a successor achieves this outcome, then it is successful.

In a sense, structural modelling of a functional system is the easier kind of modelling since analogues from mathematical discourse are readily available. It is behavioural modelling and the extraction of a design-relevant rhetoric for functional programming that may be much more difficult. Many questions remain, and much future work remains to be done. Some of the most interesting immediate questions are:

- How “natural” is the notation for non-functional programmers, or for those who are learning functional programming? What changes should be made to evolve the notation?
 - How does one handle symbols and namespacing, so that the same symbols can be used in another context?
 - What is considered to be an overwhelming accumulation of symbols? What do rich, sparse, and poor symbolic vocabularies look like? What design guidelines should exist?
 - What is modelled in a type model depends on what *needs* to be modelled. Which needs are specific to functional programming?
- What might a notation based explicitly on categories, but rooted in the same philosophy, look like? Can previous work in this area [6, 14, 36] be used to imagine such a notation or improve the proposed notation?

- More of the underlying philosophy and linkages has been summarized than is strictly necessary for a purely structural notation. This is intentional and opens the door to the creation of behavioural models, task- or process-oriented models, and so forth. What might these look like?
- Real dictionaries include parts of speech (e.g. “v.”, “adj.”, “n.”, “informal”, etc.) and other annotations which tell the reader about the grammatical and contextual use of the word. Which equivalent notations might the field of functional programmers agree upon? Is it *useful*—and if so, why?—to annotate entries with “mon.”, “arr.”, “lazy”, “async”, or “app.”?
- It is plausible that the proposed relationships are not the only or most suitable types of relationships to represent. Are there complementary or more suitable relationships that deserve recognition?
- Is a similar notation possible for untyped functional programming? What changes would need to be made?
- How amenable are pure functional programs to model-driven engineering?

Acknowledgments. The author would like to thank the anonymous reviewers of both MODELSWARD’21 and TFP’21 for their constructive and helpful comments on an earlier version of this paper. Insightful and constructive pre-presentation reviewer comments also made this work much better than it would otherwise have been, as did questions and discussions at Lambda Days 2021. This work is based on the research supported partly by the National Research Foundation of South Africa (Grant Number: 116794). This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA and CORIANT. The author acknowledges that opinions, findings and conclusions or recommendations expressed here are those of the author and that none of the above mentioned sponsors accept liability whatsoever in this regard.

References

1. Armstrong, D.J.: The quarks of object-oriented development. Commun. ACM (2006). <https://doi.org/10.1145/1113034.1113040>
2. Cardelli, L.: Type Systems. The Computer Science and Engineering Handbook (2004). <https://doi.org/10.1145/234313.234418>
3. Church, A.: The Calculi of Lambda Conversion (AM-6). Princeton University Press, Princeton (1941). <https://doi.org/10.1515/9781400881932>
4. Denning, P.J.: The profession of IT: beyond computational thinking. Commun. ACM **52**(6), 28–30 (2009). <https://doi.org/10.1145/1516046.1516054>
5. Eklund, P., Galán, M.A., Medina, J., Ojeda-Aciego, M., Valverde, A.: A graphical approach to monad compositions. Electr. Notes Theoret. Comput. Sci. (2001). [https://doi.org/10.1016/S1571-0661\(05\)80041-6](https://doi.org/10.1016/S1571-0661(05)80041-6)
6. Fong, B., Spivak, D.I.: An Invitation to Applied Category Theory. Cambridge University Press (2019). <https://doi.org/10.1017/9781108668804>
7. Ganesalingam, M.: The Language of Mathematics. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-642-37012-0>
8. Ghosh, D.: Functional and Reactive Domain Modeling. Manning Publications Co., Shelter Island (2016)

9. Gibbons, J.: Functional programming for domain-specific languages. In: Zsóok, V., Horváth, Z., Csató, L. (eds.) CFP 2013. LNCS, vol. 8606, pp. 1–28. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15940-9_1
10. Henderson-Sellers, B., Gonzalez-Perez, C., Eriksson, O., Agerfalk, P.J., Walkerden, G.: Software modelling languages: a wish list. In: 2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering, pp. 72–77. IEEE (2015). <https://doi.org/10.1109/MiSE.2015.20>
11. Henderson-Sellers, B.: On the Mathematics of Modelling, Metamodelling, Ontologies and Modelling Languages. Springer, Heidelberg (2012). <https://doi.org/10.1007/978-3-642-29825-7>
12. Henderson-Sellers, B.: Why philosophize; why not just model? In: Johannesson, P., Lee, M.L., Liddle, S.W., Opdahl, A.L., López, Ó.P. (eds.) ER 2015. LNCS, vol. 9381, pp. 3–17. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25264-3_1
13. Henrix, J., Plasmeijer, R., Achten, P.: GiN: a graphical language and tool for defining iTask workflows. In: Peña, R., Page, R. (eds.) TFP 2011. LNCS, vol. 7193, pp. 163–178. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32037-8_11
14. Hinze, R., Marsden, D.: Equational reasoning with lollipops, forks, cups, caps, snakes, and speedometers. J. Log. Algebraic Methods Program. (2016). <https://doi.org/10.1016/j.jlamp.2015.12.004>
15. Huet, G.: Axiomatisations, proofs, and formal specifications of algorithms: commented case studies in the Coq proof assistant. In: Schwichtenberg, H. (ed.) Logic of Computation. NATO ASI Series, pp. 157–199. Springer, Heidelberg (1997). https://doi.org/10.1007/978-3-642-59048-1_5
16. Jeurings, J., Magalhães, J.P., Heeren, B.: Generic programming for domain reasoners. In: Trends in Functional Programming, vol. 10, pp. 1–16 (2014)
17. Kamp, H., Van Genabith, J., Reyle, U.: Discourse representation theory. In: Gabbay, D., Guenther, F. (eds.) Handbook of Philosophical Logic, pp. 125–394. Springer, Dordrecht (2011). https://doi.org/10.1007/978-94-007-0485-5_3
18. Kosar, T., Mernik, M., Carver, J.C.: Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. Empir. Software Eng. **17**(3), 276–304 (2012). <https://doi.org/10.1007/s10664-011-9172-x>
19. Kühne, T.: Matters of (meta-)modeling. Softw. Syst. Model. **5**(4), 369–385 (2006). <https://doi.org/10.1007/s10270-006-0017-9>
20. Lackey, D.P.: What are the modern classics? The Baruch poll of great philosophy in the twentieth century. Philos. Forum **30**(4), 329–346 (1999). <https://doi.org/10.1111/0031-806X.00022>
21. Li, Q., Chen, Y.-L.: Data flow diagram. In: Modeling and Analysis of Enterprise and Information Systems: From Requirements to Realization, pp. 85–97. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-89556-5_4
22. Lonsdorf, B.: Professor Frisby’s Mostly Adequate Guide to Functional Programming (2020)
23. Mac Lane, S.: Categories for the Working Mathematician. Springer, New York (1978). <https://doi.org/10.1007/978-1-4757-4721-8>
24. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4), 316–344 (2005). <https://doi.org/10.1145/1118890.1118892>
25. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978). [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4)

26. Moody, D.: The “physics” of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Softw. Eng.* (2009). <https://doi.org/10.1109/TSE.2009.67>
27. Motara, Y.M.: String diagrams for modelling functional programming. In: 2020 2nd International Multidisciplinary Information Technology and Engineering Conference (IMITEC), pp. 1–7. IEEE, Kimberley (2020). <https://doi.org/10.1109/IMITEC50163.2020.9334072>
28. Muller, P.-A., Fondement, F., Baudry, B., Combemale, B.: Modeling modeling modeling. *Softw. Syst. Model.* **11**(3), 347–359 (2012). <https://doi.org/10.1007/s10270-010-0172-x>
29. O’Regan, G. (ed.): *Mathematical Approaches to Software Quality*, pp. 1–32. Springer, London (2006). https://doi.org/10.1007/1-84628-435-9_1
30. Object Management Group: *Business Process Model and Notation (BPMN)*, Publication Title: Object Management Group specification (2013)
31. Petricek, T.: What we talk about when we talk about monads. *The Art, Science, and Engineering of Programming* (2018). <https://doi.org/10.22152/programming-journal.org/2018/2/12>
32. van der Ploeg, A.: Monadic functional reactive programming. In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell - Haskell 2013* (2013). <https://doi.org/10.1145/2503778.2503783>
33. Schulte, C., Magenheimer, J., Niere, J., Schäfer, W.: Thinking in objects and their collaboration: introducing object-oriented technology. *Comput. Sci. Educ.* **13**(4), 269–288 (2003). <https://doi.org/10.1076/csed.13.4.269.17492>
34. Selinger, P.: A survey of graphical languages for monoidal categories. In: Coecke, B. (ed.) *New Structures for Physics. Lecture Notes in Physics*, vol. 813. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-12821-9_4
35. Simons, A.J.H., Graham, I.: 30 things that go wrong in object modelling with UML 1.3. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) *Behavioral Specifications of Businesses and Systems*, pp. 237–257. Springer, Boston (1999). https://doi.org/10.1007/978-1-4615-5229-1_17
36. Spivak, D.I.: *Category Theory for the Sciences*. The MIT Press, Cambridge (2014)
37. Stachowiak, H.: *Allgemeine Modelltheorie*. Springer, Vienna (1973). <https://doi.org/10.1007/978-3-7091-8327-4>
38. Stutterheim, J.: *A Cocktail of Tools: Domain-Specific Languages for Task-Oriented Software Development*. Radboud University (2017)
39. da Silva Teixeira, M.D.G.: *An Ontology-Based Process for Domain-Specific Visual Language Design*. Universidade Federal do Espírito Santo (2017)
40. Wand, Y., Weber, R.: On the ontological expressiveness of information systems analysis and design grammars. *Inf. Syst. J.* **3**(4), 217–237 (1993)
41. Weber, R.: Evaluating and Developing Theories in the Information Systems Discipline. *J. Assoc. Inf. Syst.* **13**(1), 1–30 (2012)
42. Wittgenstein, L.: *Tractatus Logico-Philosophicus*. Kegan Paul, London (1922)
43. Wittgenstein, L., Anscombe, G.E.M., Hacker, P.M.S., Schulte, J.: *Philosophical investigations*. Chichester, Wiley-Blackwell (2009)
44. Wlaschin, S.: *Domain Modeling Made Functional*. The Pragmatic Bookshelf (2018)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.