On the (co-) limit of a functor, νF, a whirlwind tour into the extreme technology and exciting work at Functor.

Let’s remind ourselves about what a functor really is and why it is so important today to software engineering as we witnessed advanced so well aligned with the technology pioneered by the **Functor research spinoff** ([bit.ly/go-F](https://bit.ly/go-F)), that is, **Functor Group AB** with subsidiaries including **Functor AB** and Functor Consulting AB, with headquarters in Sweden.

WE TURN NEXT GENERATION SOFTWARE TECHNOLOGY INTO GROWTH
WHAT’S YOUR MISSION?
JOIN US AT FUNCTOR

We might just have a rather exciting opportunity for you at Functor. Send your application AND PUSH THE LIMIT OF FUNCTORS FURTHER FORWARD, go here:

➤ bit.ly/angel-job
➤ functor.se/jobs

THE NEXT PARADIGM

www.functorgroup.com
A **functor** $F: A \rightarrow B$ from a category $A$ to another category $B$ is a structure-preserving function from $A$ to $B$. The concept arises in a branch of mathematics known as category theory. Intuitively, if $A$ is seen as a network of arrows (morphisms) between objects and then $F$ maps that network onto the network of arrows (morphisms) of $B$, this gives a picture of $A$ in $B$.

The logician Alfred Tarski, back in 1955, presented pioneering work within set theory, which subsequently has been generalised within the field of category theory. Tarski is well-known for his (disputed) notion of truth, eg. $\llbracket$the snow is white$\rrbracket$ if the snow is white. The search for truth goes back in time well before, and certainly including, Aristotle, the latter stating eg.

```
“ to say of what is that it is not, or of what is not that it is, is false, while to say of what is that it is, or of what is not that it is not, is true.”
```
In more recent years, the Swedish logician Professor Per Martin-Löf as offered a philosophical theory of **meaning explanations**, **giving a philosophical epistemological logical intuitionistic founding of “truth”, what it means to “know”,** etc. This was done by Per Martin-Löf within the theory of **dependent types**, that came to be known as Martin-Löf type theory. Per Martin-Löf’s work essentially gave birth to **a new research field within computer science and progress has accelerated immensely over the last decade in this field – type theory**, We should remind the reader that constructive set theory and logic goes far back in time. Functor does not quite come out of thin air, the journey began long ago within mathematics and logic but it was Martin-Löf who brought it all into, first, theoretical computer science. Type theory meant a leap from pure mathematics into programming (**BHK**), although it took several decades to make it all practical and five years for Functor
to make the final leap into industrial technology, which has proven very valuable in the industry by now. In summary, Functor’s technology, while as unique, ground-breaking, powerful, while also, somewhat mind-boggling even to us, having enormous scalability, stands strongly on giants. Functor stands on more than a complete research field (type theory) that before Functor has seen only one application in the industry (natural language processing), and never, amazingly, before been disseminated, developed, aligned with practices, etc, and made into powerful programming tools ready to be used by the industry out of the box. Admittedly, this was a major challenge in the development of our technology and indeed products. They are designed to work seamlessly, appeal to a competent software engineer immediately, and offer augmentation and automation, while demanding no change, fitted to most toolings that are actually being used by the industry, and, certainly, addressing the most pressing problems, which the industry are more than aware of, as Functor gives remedy for pains, risks and high costs.
Martin-Löf type theory, then, has had global impact not only within mathematics but also in philosophy, our understanding of natural languages, linguistics, statistics. This Swedish research subsequently and gradually was absorbed by computer science, and Per Martin-Löf named **CONSTRUCTIVE PROGRAMMING**, by simply paraphrasing the title (as we did at STEW 2012) of one of his later papers.

---

WITH IOT EXPECTED TO INCLUDE 26 BILLION CONNECTED “THINGS” BY 2020, THE NUMBER OF POTENTIAL ATTACK POINTS WILL SKYROCKET.
The **CONSTRUCTIVE PROGRAMMING PARADIGM** is therefore something that originates from Per Martin-Löf, although eg. Functor Scalor™ has an all-new operational semantics turning Martin-Löf type theory into a high-performance metaprogramming platform. From this core semantics, scrutinized by Per himself, we’ve added a component-model as well as object-based programming, in the spirit of Guy Steele’s “Growing a Language” approach (there core language, which we obviously named Score™, is extremely fast and extremely minimal, executing on LLVM admitting speed of C for embedded domain-specific languages interoperating with all kinds of programming languages via LLVM, but let’s not leap ahead too far in this intentionally whirly tour where we emit a bit of the excitement around Functor Group with subsidiaries, and its research footing that took such long time to make impact, with enormous investments behind it implying obviously future vast global impact and literally a game changing technology for the whole industry to adopt over time).
Swedish Professor (now Emeritus) Per Martin-Löf’s work has moved, finally, into software engineering, found infinitely timely alignment with recent advances in that field, and with new industrial applications of significant value and with scalability that is nothing less than totally mind-blowing, all delivered by Functor, Sweden, a research spinoff from University of Cambridge, Oxford University and, indeed, Swedish universities.

At the core of Martin-Löf type theory lies the category theoretic notion of a functor. Indeed, a **functor** from $A$ to $B$, is a structure-preserving map much like a homomorphism between, let’s say, an algebraic structure such as a **group** in classical (non-constructive well-founded) Zermelo-Fraenkel (ZF) set theory, though the concept reoccurs in all sorts of foundations of mathematics and has its incarnations in category theory including internalisations within one single category.
More precisely, any category $A$ has an identity functor $\text{id}_A : A \to A$, which leaves the objects and arrows of $A$ unchanged. The action of $\text{id}_A$ is, as it were, to do nothing either on objects or arrows (morphisms). Given functors $F : A \to B$ and $G : B \to C$ there is a composite functor $GF : A \to C$ sometimes written $G \circ F$. Tarski is famous, among many other things, for his fixpoint theorem, saying, in general terms, that a certain structure of ordered sets (a complete lattice, to be precise) has a least fixpoint of an order-preserving function, this is formerly expressed somewhat differently, and a specialised variation is known as the Knaster–Tarski theorem.
Over the last decades, we have witnessed great advances within theoretical computer science and work in mathematics and logic has increasingly been leveraged and transformed the subject itself. What to Tarski was an order-preserving function, is of course a homomorphism, and in modern denotational (or mathematical operational, MSOS, etc) semantics of programming languages it often means exactly a functor.

**Functors are of course ubiquitous** in category theory at large. They are further, **fundamental to the semantics of types and datatypes**, and indeed in the semantics of **Martin-Löf type theory** and its various incarnations, including in the mathematical semantics of **Functor Scalor™ meta-programming platform** where “everything is your code”, plugging into existing code bases and tooling and bringing static analysis and the power of DDD.
DDD is an abbreviation for domain-driven design, and is a method that has over the last years become extremely popular in the industry. The mantra is “the domain model is the code”, and it is the logical extension of agile practices, whilst capitalising on new pattern languages and domain-specific languages building on MDA (model-driven architecture), MDD (model-driven development) and MDE (model-driven engineering, model-driven engineering (MDE), model-based engineering (MBE), … Indeed, DDD did not arise from thin air. Software engineer engineering gurus, some indeed are with us at Functor, can today connect the dots between these and other trends in software engineering practices. There is long-standing efforts in software engineering in both the industry and in academia (often overlapping in this field) and these and other acronyms actually has been hallmarks on the journey which we now see converged into domain engineering.

DDD minus boilerplate is Functor LDD™, on top of Functor Scalor™.
**Functor Scalor™ offers the first technological foundation for this new kind of domain engineering.** While DDD involves pattern languages and a lot of boilerplate code, its value proposition is clear enough. Functor Scalor™ is a metalanguage designed to help in domain engineering while interoperating with programming languages via LLVM bindings. Concepts like bounded context, continuous integration context map and notions such as entity, value object, aggregate, domain event, service, repository, factory, etc, burdens the DDD approach. Some years ago we noted certain advances in software engineering that seemed to align surprisingly and yes even amazingly well with the totally game-changing technology we have been developing and sold in products to the industry for some time now. In particular, Bjaner’s contributions was key. His work on domain engineering turned out to fit hand-in-glove with our technology and Functor Scalor™, and we made Functor Scalor™ into a tool where the DDD boilerplate and
indeed the formal methods part of Bjarner’s approach, is internalised – automated. We aid DDD by simplifying it and Functor Scalor™ admits even stronger value proposition if a customer wish to go further and harvest our technology, which customers can do in instalments, which helps litigate the perception of risk since there is a continuum of value that can be drawn from Functor Scalor™ beyond the strong selling point of dramatically helping with DDD and giving it high-precision static analysis, taking care of all that boilerplate code so that the domain models can be refined, tested, and statically analysed with high performance and dramatically improved productivity, not to speak of code maintainability and several other metrics beyond risk litigation, cost, cutting lead times, improving performance as more computations are done statically rather than at run-time. Functor Scalor™ subsumes 95% of the DDD activities and offers automation.
In effect, we are allowing developers to continue with DDD, but dramatically increase efficiency while harvesting all the values we bring with new static analysis, test automation, and so on. In saying we have the first foundation for DDD, it may suffice to note that the very first appearance of Martin-Löf type theory in theoretical computer science was as a metalanguage (Harper, Honsell, Plotkin, 1993) in which not only domain models, or DSLs for that matter, were modelled, but, hold your breath, entire programming languages. Recall that the only other area where type theory has been disseminated into the industry, that is beyond software development with Functor, is natural language processing (Ranta, et al, GF, etc). We understand at Functor it can be hard to grasp all this and our customers are focused on their problems and the simplest most valuable solution, not the long story of science etc to give them this. Investors, too, go wild if they hear about a paradigm.
Experts within software engineering can certainly trace the progress over the last three decades and the convergence into agile and then domain engineering. DDD is now turned into Functor LDD™, which means that things gets a lot simpler as the tool, here Functor Scalor™, is fitted to the task and strongly founded on all this research turned into technology for domain engineering. It’s quite hard to grasp all this, but so-called industrial software (engineering) gurus, indeed some found at Functor, can certainly connect the dots in hindsight with the acronyms mentioned above, and others having been practised, refined, iteration by iteration, into what is not just about any software development project, with none of the boilerplate code and more effective testing and automation based on a new kind of dependent types-based program analysis.

In the field of programming and programming language research, as it has advanced, have come to **identify functors with types** in a certain
manner. The signature of a type, (co)algebraic datatype or otherwise, is, semantically speaking, a particular fixpoint of a functor. Such fixpoints are denoted $\mu F$ or $\nu F$ for the colimit respective limit of the functor, as Smyth and Plotkin expounded first in the context of recursive domain equations such as $D \equiv D \rightarrow D \perp$, the semantic domain of lambda calculus. Such equations are indeed root of the entire discipline of domain theory, intersecting mathematics and computer science today, pioneered by Dana Scott.
Functors are key to our understanding of programming today, and advances from the work of the Swedish mathematician Per Martin-Löf has now made its way also into software engineering, with Functor, Sweden, at the very forefront, with a metalanguage platform Functor Scalor™ and a new approach to testing and static analysis in the Functor Prevent™ automation tool.

Functor’s tools have associated methods that are equally aligned to advances in the industry and the current and emerging practices as they are to the great advances in the discipline of computer science with regard to programming languages, program analysis, and so on.
There is obviously much more to say, and this exposition is intentionally informal in nature, leaving much untold and with little specificity.

At Functor AB and Functor Group AB, all these research advances, the mathematics, type theory, logic, program analysis, theorem solving, and so on, are inside the tools and further wrapped in state-of-the-art industrial methods.

This makes our tools, despite a complete research field going right into our core technology, and what we have touched upon above, etc, familiar to any competent software engineer in the industry, augmenting current practices rather than changing them, at the very frontier of technological progress.
JOIN OUR TEAM AT FUNCTOR
PUSH THE LIMIT OF FUNCTORS

We do not discard that you see now a frankly mind-blowing and exciting opportunity to push the limits of functors further still with us at Functor. To apply to be part of our teams, you want to go here:

▶️ [functor.se/jobs](http://functor.se/jobs)

FUNCTOR SWEDEN
www.functorgroup.com
WE TURN NEXT GENERATION SOFTWARE TECHNOLOGY INTO GROWTH

FUNCTOR SWEDEN

www.functorgroup.com
WE TURN NEXT GENERATION SOFTWARE TECHNOLOGY INTO GROWTH

FUNCTOR SWEDEN

www.functorgroup.com
WE TURN NEXT GENERATION SOFTWARE TECHNOLOGY INTO GROWTH

FUNCTOR SWEDEN

www.functorgroup.com
WE TURN SOFTWARE TECHNOLOGY INTO GROWTH

FUNCTOR SWEDEN

www.functorgroup.com
FUNCTOR SWEDEN

www.functorgroup.com
WE TURN NEXT GENERATION SOFTWARE TECHNOLOGY INTO GROWTH

FUNCTOR

SWEDEN

www.functorgroup.com
FUNCTOR
SWEDEN
www.functorgroup.com