Parametric and Logical Types for Model-Driven Engineering

Rick Murphy
U.S. Government
email: rick@rickmurphy.org

***Abstract*—This technical paper introduces a novel UML Profile to construct parametric and logical types for the advancing model-driven engineer. Parametric types admit parametric polymorphism under a relational interpretation of types. We introduce parametric disjunctive and conjunctive types that lift operations from the object to type levels, constructively. We call them logical types. Logical types represent a fragment of intuitionistic higher order logic under the Brouwer-Heyting-Kolgomorov (BHK) interpretation. We specify their construction informally in UML sequence diagrams and their semantics and operations on them formally in intuitionstic logic, category theory and commutative diagrams.**

We explain the advantages of parametric and logical types over the naive meta model approach and the approach to types and relations used in QVT, MOF and OCL. We provide semantics of parametric relational types as Kliesli categories; parametric logical data and type constructors as functors; and embedded product types as Hom-functors. We also provide the semantics of parametric and logical types as elementary and effective topoi. We prove the soundness and completeness of the relevant fragment of the BHK interpretation in natural deduction and extend the effective topoi to an interpretation of realizability. A survey of the publicly available literature reveals no evidence of a similar approach for the advancing model-driven engineer. The approach is novel in its definition of the profile and application of algebraic types in model-driven engineering.

I. Introduction

Model-Driven engineers have sought to clarify, or formalize, the interpretation of standards and associated "models" through ontology, semantics and logic. Logic is sometimes seen as external to "modeling" and ontology a competitor. Integrating first order logic appears unattainable for practical purposes and type specification remains subordinate to set theoretic, class-based inheritance semantics.

Despite our best efforts, some desired outcomes remain out of reach. Proposals for unification through "meta modeling" such as the Meta Object Facility (MOF) naively underestimate the complexity of a required solution. Query View Transformation (QVT) remains largely unimplemented.

While we do not claim to solve the larger challenges of unification and transformation, this paper returns to foundations of programming languages in describing a technical approach using parametric and logical types we believe required to achieve the desired outcomes.

A. Overview

The rest of this paper is organized as follows. Section II reviews standards literature on parametric types as UML templates with references to programming languages and specification. Section III describes the formulation of logical types in a UML profile with data and type constructors. It contrasts parametric and logical type with a naive meta model approach and explains the technical advantage of parametric and logical types. Section IV illustrates the construction of and operations on logical types and describes their relational, set- and category- theoretic semantics. It contrasts parametric and logical types with QVT 1.3 and its dependency on MOF 2.5 and OCL 2.4 and explains the advantages of parametric and logical types over QVT. Section V explains the proof calculus and model theoretic interpretation of parametric and logical types in a fragment of Heyting algebra with natural deduction. We prove logical types are sound and complete. Section VI extends the interpretation of the parametric and logical types to realizability semantics in the effective topos. Section VI concludes and proposes future directions for our research.

II. Parametric Types

It well known that classes and data types are interpreted as sets of objects and values. [UML 2.5 18.3, UML 2.5 10.2] Less known is the interpretation of classes and types as relations. Reynolds first proposed to generalize homomorphisms from functions to relations for free algebras. His abstraction theorem introduces relational and parametric types. [Reynolds, 1983]

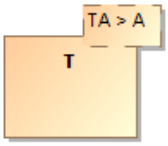
Consider the list L of ordered pairs L = [(0, True),(1, False)]. The relation inferred from the list is the parametric type Int Bool : the relation between the types of elements of ordered pairs in L. The counterintutive juxtaposition of Int and Bool may cause suspicion. Consider the more familiar declaration List Nat, the list of natural numbers where List is parametrized by the type of its elements : Nat.

More generally, the expression T A is the type constructor T parametrized by the type parameter A. The set of type parameters may be called the carrier set of T. Type parameters are polymorphic allowing the substitution of monomorphic types like Bool, Int or Nat for A. Any parametric type T A can be interpreted as the relation $\tau : T \iff A$. UML allows parametrized class and data type by declaration [UML 2.5 9.3.1].

Despite limited support for parametric types in graphical notation, some class and data type diagrams allow depiction of type parameters. See Figure 1. Observe the familiar box representing a class or data type labeled T is annotated with a box at its top right outlined in dash style. The annotated box

represents the binding $>$ of the parameter A to the template TA.

Figure 1. Parametric Type



Templates are model elements that are parametrized by other model elements. [UML 2.5 7.3.1] The term template originates in the C++ language which defines class and function templates. A template signature S on templatable elements TX defines a set of template parameters P bound to model elements $S_{TX} > \{TX : p \in P\}$.

We read from Figure 1, the template signature $S_{TA} > \{TA : A \in P\}$ binds parameter A to templateable element TA. The parameters in the signature are the "formal parameters" for which "actual parameters" will be substituted in a binding. Given a substitution, lets say of monomorphic type Int for parameter A, an operational semantics allows for instantiation of relational type $TInt$.

Templates derive from a more general formulation of parametric types. Mainstream programming languages like Java and C# call parametric types generics. [Bracha, 1998] Software architecture and algebraic specification allow parametrization of modules [Goguen, 1996] and algebraic specifications [Mossakowski, 2004]. Whereas today's mainstream programming languages implement parametric types, model-driven engineering can and must offer comparable syntax and semantics.

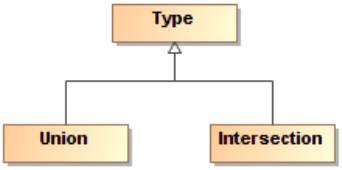
III. Logical Types

Current standards specify operations and operators in an object language. A class could define a boolean operation that implements bitwise boolean (&, |) or boolean logical (&&, ||) operators. [UML 2.5 9.6] OCL defines the semantics of the following boolean operators for basic types (and, or, xor, implies, not). [OCL 2.4, A 2.1.3]

"Meta models" typically characterize inheritance hierarchies of imperative programs in the abstract syntax of a "meta language." The meta language might specify a class or type. The object language inherits properties and operations and may represent the instance of the class or value of type, or a specialization at a level below the meta language.

Figure 2 illustrates a naive approach to type specification taken from an Object Management Group submission. Type semantics were imputed to be a class-like structure with properties. Constructors and operations were disallowed. Values comprised the extent imputed to be set. Union and Intersection contained no properties other than those contained in Type. Generalization implied intensional equality. No other model element specified the semantics of union or intersection.

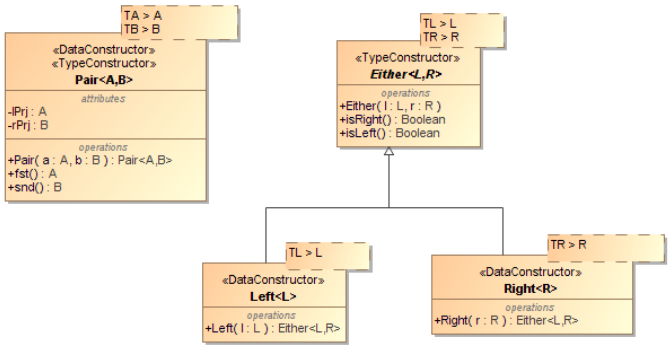
Figure 2. Naive Metamodel Approach



We conservatively extend the use of metamodels with parametric types that introduce a typed logic into our meta language. We call them logical types because they lift logic from the object language to the type system. See Figure 3.

Observe the parametrized type $Pair\langle A, B\rangle$. $Pair\langle A, B\rangle$ lifts conjunction into the type system. $Pair\langle A, B\rangle$ has two type parameters A and B representing the conjunctive operands. Type parameters are polymorphic and we construct a value p of $Pair\langle A, B\rangle$ from constructor $Pair(a : A, b : B)$ substituting values of monomorphic types for polymorphic type parameters. As expected, the value p includes references to both operands reduced to values of monomorphic types, or their references. Notice that $Pair\langle A, B\rangle$ has two properties $lPrj : A$, the left projection, and $rPrj : B$, the right projection, the types of which are the type parameters A and B respectively. References to values of monomorphic types for each respective operand is bound during construction. The operation fst returns the binding of type parameter A. snd returns the binding of type parameter B. $Pair\langle A, B\rangle$ may also be called product.

Figure 3. Applied Profile with Logical Types



Now that we have introduced conjunction, we likewise lift disjunction into the type system with the parametric type $Either\langle L, R\rangle$. A value e of $Either\langle L, R\rangle$ also has two type parameters L and R representing the disjunctive operands. Notice the names of the two type parameters L and R reflect the mnemonics left and right. Construction of $Left\langle L\rangle$ and $Right\langle R\rangle$ return their respective binding to type $Either\langle L, R\rangle$ such that the value e has a reference to only one of the operands, the other remains unevaluated. $Either\langle L, R\rangle$ is abstract and construction proceeds by binding a value of a

monomorphic type in the constructor of either $Left\langle L\rangle$ or $Right\langle R\rangle$. $Left\langle L\rangle$ and $Right\langle R\rangle$ inherit boolean operations that test for identity. $Either\langle L, R\rangle$ may also be called sum or coproduct.

We can reason constructively using the type system with logical types. Reasoning constructively means providing a witness to the proof of a proposition. [Wadler 2015] Consider a propositional logic with negation and implication. A constructive proof of DeMorgan's Law $\neg(P \wedge Q) \iff (\neg P) \vee (\neg Q)$ with parametric and logical types would have the type $Pair\langle A, B\rangle \rightarrow Not\ (Either\ (Not\ A), (Not\ B))$. See Figure 4. Given predicate $Not\ a = forall.\ a \rightarrow Bool$, $witness$ is evaluated and the proof given by pattern matching on the structure of the data constructors $Left\ id$ and $Right\ id$ and their application to the respective arguments in $Pair(x, y)$.

Figure 4. Constructive Witness

```
type Not a = a -> Bool

eval :: Pair Bool Bool -> Not (Either (Not Bool) (Not Bool))
eval (Pair(x,y)) (Left u) = u x
eval (Pair(x,y)) (Right u) = u y

-- true
witness = eval (Pair(True,True)) (Left id) &&
          eval (Pair(True,True)) (Right id)
```

Types in UML have either value specifications [UML 2.5, 8.6.22] derived from kinds of value classifiers [UML 2.5, 7.5.3.1] or are primitive in that they have no notation [UML 2.5, 21.3]. Value specifications are described as parameterable elements that may be exposed as a formal template parameter and provided as the actual parameter in the binding of a template. Primitive types are declared rather than constructed and most often constrain attribute values in a user defined class.

The notion of type and data constructors is absent from the UML 2.5 and other OMG standards. We extend the standards with a UML profile that allows type construction from type and data constructors. The profile is small requiring only two stereotypes : ≪DataConstructor≫ and ≪TypeConstructor≫ as extensions to the base class Class. [1]

Notice that we have applied our small profile to $Pair\langle A, B\rangle$ and $Either\langle L, R\rangle$. Left and Right are annotated with the stereotype ≪DataConstructor≫. Values are constructed only from data constructors in adherence to the "no junk" principle. Data constructors construct values from a value of a monomorphic type. For example, Left True constructs a value of type Either $\iff$ (Left $\iff$ Id_{Bool}, R). Alternatively Java syntax represents this type as $Either\langle Left\langle Bool\langle ?\rangle\rangle, \langle ?\rangle\rangle$.

Type constructors are relational. A type is indexed by the monomorphic type substituted for the polymorphic type parameter(s). Constant types like Bool or Int are interpreted as identity relations Id_{Bool} : Bool $\iff$ Bool and Id_{Int} : Int $\iff$ Int respectively. For types $\mathcal{A}$: A $\iff$ A′ and $\mathcal{B}$: B $\iff$ B′ the relation $\mathcal{A} \times \mathcal{B}$: (A × B) $\iff$ (A′ × B′) meaning pairs are related if their components are related. The relation $\mathcal{A} \rightarrow \mathcal{B}$: (A → B) $\iff$ $(A' \rightarrow B')$ is defined by $(f, f') \in \mathcal{A} \rightarrow \mathcal{B} \iff \forall\ (x, x') \in \mathcal{A}, (fx, f'x') \in \mathcal{B}$ meaning functions are related that takes related arguments into related values. Further, any type T can be interpreted as the relation τ : T $\iff$ T. [Wadler 1989]

Functional programmers will immediately recognize logical types. While the naive meta-model approach allows the assertion of an extent subject to intensional constraints under set-theoretic inheritance the advantage of logical types should be immediately obvious. (a) Logical types lift logic from the object language to the type system. We can reason with logic about types. (b) Logical types are constructive. The type and data constructors provide syntax and semantics of the logic. Constructive means more than having a constructor which the naive metamodel approach obviously lacks. The construction of a value of the type is considered a witness satisfying a proof of a proposition under the Curry Howard Correspondence [Wadler 2015]. Proofs describe algorithms from which we may formulate functional programs. The logic is intutitionistic rather than classical. [Pfenning 2000] We do not assert the law of excluded middle. (c) Logical types are relational because they are parametric. (d) Logical types allow parametric rather than ad-hoc polymorphism. Parametric polymorphism applies to a uniform range of types with common structure. [Strachey 1967]

IV. Construction of and Operations on Parametric and Logical Types

Parametric types are called generics in mainstream programming languages. This section provides the semantics of and illustrates the construction of and operations on parametric and logical types for model-driven engineering. It also explains their technical advantages over types and relations in QVT 1.3 and its dependencies on MOF 2.5, OCL 2.4.

An incomplete analysis reveals MOF 2.5 does not import UML Templates. OCL 2.4 defines TemplateParameterType without reference to UML Templates and admits only the informal semantics that TemplateParameterType "refer[s] to generic types" and is "used in the standard library to represent the parameterized collection operations." [OCL 8.2] OCL collections are not relational, they are set theoretic. [OCL 11.6] Collections are the only parametric types in the OCL standard library. [OCL 11] Further, OCL informally claims the collection type "is actually a template type with one parameter." [OCL 11.6] OCL semantics described in UML do not differentiate values or evaluations for collection types as template parameter types. OCL values and evaluation packages do not admit function or relation. [OCL 10, 11] QVT 1.3 claims a "user friendly Relations meta-model and language" specified in natural language and first order predicate logic and a "standard transformation for any relations model to a trace models and a Core model with equivalent semantics." [QVT 6.1] QVT also defines TemplateParameterType without reference to UML Templates. It does not allow user defined

[1] Profiles [UML 2.5 12.3] is the correct extension for type construction. Recall EMOF and CMOF do not import templates from UML. [MOF Core 2.5.1 EMOF 12.4, CMOF 14.4] It is well known that generics are implemented separately as ECore.

parametric types and its only parametric types are those in the standard library [QVT 8.2.2.26] Its relations language specifies homomorphisms and constraints rather than relations. [QVT 7.1, 7.2] The QVT standard library for relations is the OCL standard library. [QVT 7.12] Although the standard library for operational mappings defines operations that return parametric types, the types returned are set theoretic. Operations do not return function or relation. [QVT 8.3.4 - 8.3.18]

Though incomplete, our analysis reveals QVT, MOF and OCL substantially reject Reynolds' approach to abstraction and parametric types. [Reynolds 1983] They are not higher order. Their specification is informal, object inheritance enhanced with object-level operations. There is no formal object calculus despite its availability. [Abadi 1995] Transformation is ad-hoc rather than parametrically polymorphic. Its claim to relations reveals homomorphisms and constraints rather than relational semantics. And its informal collections only approach to parametric types restricts their application significantly less than even mainstream programming languages like Java and C#.

QVT 1.3 depends on MOF 2.5 and OCL 2.4. Despite the early optimism surrounding the standardization of QVT, its authors report only two discouraging implementations of QVTr, its relations language, and no implementations of QVTc, its core language. Most revealing for an OMG standard, there are no commercial implementations, since 2002. In response, Willink proposes three new intermediate languages and a dependency on the Action Language Foundation for UML (ALF). In his "progressive architecture approach." [Willink 2016] We do not discuss the progressive architecture approach or ALF.

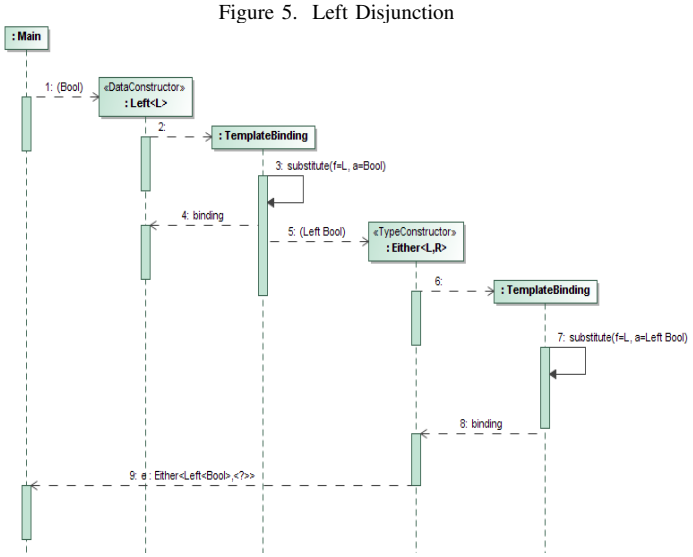
Our novel UML profile offers technical advantages over types and relations in QVT, MOF and OCL as explained in the examples below. Example A : Left Disjunction illustrates the binding of a monomorphic type to a polymorphic type in the Left disjunct using a set theoretic interpretation of the parameters to build intuition. Example B : Conjunction proceeds with relations on Pair in the set theoretic function space where the parameters PlusOne and Nat. Example C : Composition and Parametricity provides insight into the effective use of abstraction rather than type construction. We demonstrate composition of parametric and logical types with polymorphic type variables and theorems induced by parametricity. [Wadler 1989] Example D : Categorical Semantics provides an incomplete categorical semantics from earlier examples. We explain 1) relational type constructors in **Rel** the Kleisli category of **Set**; 2) functors on data and type constructors; 3) the embedding of products in **Set**, the category of sets with the Hom functor; and 4) an incomplete examination of elementary and effective topoi using relational type constructor Pair with function and natural numbers parameters.

A. Left Disjunction : (Left $\iff$ Bool) $\rightarrow$ (Either $\iff$ ((Left $\iff$ Bool) $\iff$ R))

Given parametrized data constructors $Left\langle L\rangle$ and $Right\langle R\rangle$ and type constructor $Either\langle L, R\rangle$, using set theoretic type parameters we interpret the left data constructor as Left $\iff$ Bool and the type constructor as Either $\iff$ ((Left $\iff$ Bool) $\iff$ R). Data construction proceeds by substituting an instance of the monomorphic type Bool for the polymorphic type parameter L in $Left$. See Figure 5. Notice $Left\langle L\rangle$ is stereotyped appropriately as $\ll DataConstructor\gg$. Type construction of $Either\langle L, R\rangle$ proceeds with the substitution of the actual parameter for the formal for type parameter L. Type parameter R remains unevaluated.

Notice the result e is typed $Either \iff ((Left \iff Bool) \iff R)$ representing the left disjunct. Java generics programmers will recognize the syntax in step 9 of the sequence diagram as $Either\langle Left\langle Bool\rangle, \langle ?\rangle\rangle$. The right disjunct proceeds analogously as $Either \iff (L \iff (Right \iff Bool))$.

Figure 5. Left Disjunction



Reynolds showed that the standard interpretation of the first order typed lambda calculus in the category of sets cannot be extended to an interpretation of the second order typed lambda calculus. [Reynolds, 1984] However, his notion of what constitutes the extension of a second order from first order interpretation was later narrowed to show the exclusion was due to the non-constructive nature of sets. Hyland showed it possible for elementary toposes to embed the second order lambda calculus in completely standard way. [Hyland 1982]

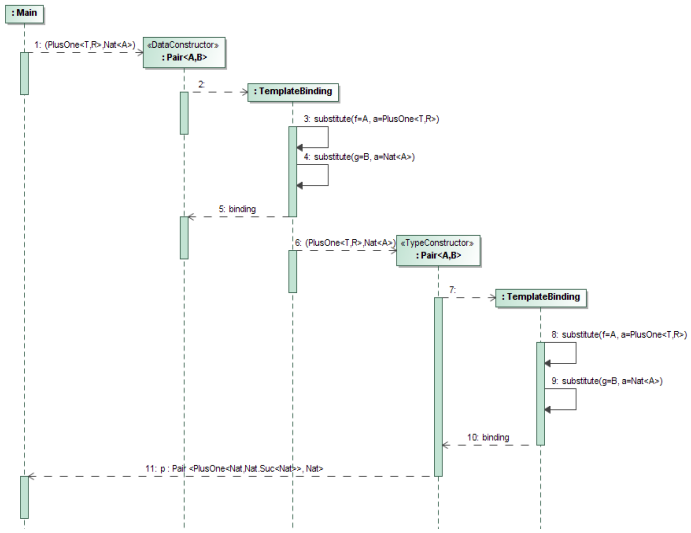
An elementary topos is an interpretation of higher order constructive logic in which polymorphic types are interpreted as sets, functions interpreted as exponential and products interpreted as indexed collections. Pitts showed the completeness and full embedding of the second order typed lambda calculus in a topos. [Pitts 1987] Note this result does not contradict Reynolds due to the absence of the Law of Excluded Middle in intuitionistic logic. We discuss the effective topos below reminding the reader that logical types are constructive and intutitionistic.

B. Conjunction : $(Pair \iff ((A \times B) \iff (A' \times B'))) \rightarrow (Pair \iff (A \times B))$

Recall the carrier set of $Pair\langle A, B\rangle$ is the product $A \times B$ and that pairs are related if their components are related. The relation $\mathcal{A} \times \mathcal{B} : (A \times B) \iff (A' \times B')$ is defined by $((x, y), (x', y')) \in A \times B \iff (x, x') \in A$ and $(y, y') \in B$. We examine the special case where given $(f \times g) \iff (x, y) \rightarrow (fx, gy)$ again under a set theoretic interpretation of the parameters. [Wadler 1989] See Figure 6.

Construction proceeds with the substitution of both monomorphic actual parameters for both polymorphic formal parameters in the data constructor resulting in the binding $Pair \iff ((PlusOne \iff (T \iff R)) \times (Nat \iff A))$. Type construction proceeds similarly resulting in the binding of type $Pair \iff ((PlusOne \iff ((Nat \iff A) \iff (Suc \iff (Nat \iff A)))) \times (Nat \iff A))$. Note the result type p in familiar Java generics syntax is $Pair\langle PlusOne\langle Nat\langle ?\rangle, Nat.Suc\langle Nat\langle ?\rangle\rangle\rangle, Nat\langle ?\rangle\rangle$.

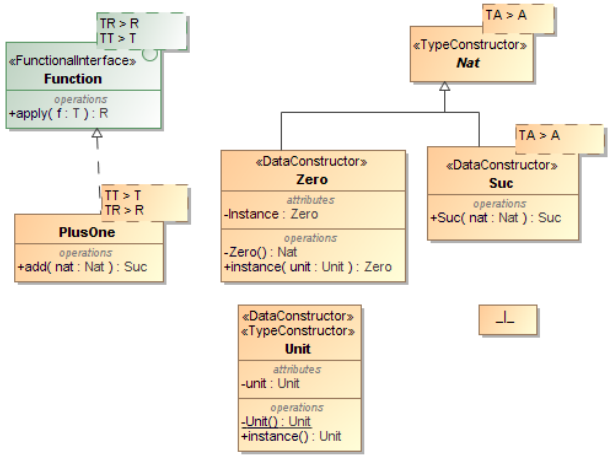
Figure 6. Conjunction



Function application is the evaluation of $PlusOne\langle T, R\rangle\ Nat\langle A\rangle$. Figure 7 defines the parametrized function $PlusOne\langle T, R\rangle$ as the implementation of the appropriately stereotyped $\ll FunctionalInterface\gg$ beginning with Java 8. $PlusOne\langle T, R\rangle$ has one operation $add(nat : Nat\ A) : Suc\ A$. It inherits operation $apply(f : T) : R$ where T and R are type parameters. $PlusOne\langle T, R\rangle$ is data constructor $Suc\ A$ in disguise. A comparable UML profile for relevant Java Generics would complement the type construction profile proposed here.

Figure 7 also defines two types useful in the application of our profile : $Unit$ and $\perp$. $Unit$ the type with one value $Unit$ and $\perp$ the abstract class with no values. $Unit$ is both a data and type constructor. Data constructor $Zero$ is constructed from $Unit$. $Zero$ and $Unit$ are appropriately singletons with static with private constructors. Parametrized data constructor $Suc\ A$ takes as argument a value of $Nat\ A$ and returns its successor in $Nat\ A$. Notice the abstract type constructor $Nat\ A$ is constructed only from either the disjunction of $Zero$ or $Suc\ A$. Finally $\perp$ is the abstract type with no values. It is neither a type or data constructor.

Figure 7. Function Application



Recall the evaluation of $Pair \iff (PlusOne \iff (T \iff R)) \times (Nat \iff A)$ is set theoretic function application resulting in the product $\langle R = Suc\ (Nat\ A), T = Nat\ A\rangle$ where $T < R$.

C. Composition and Parametricity

While construction of monomorphic types from polymorphic constructors satisfies our intuition of building something, operations on polymorphic functions can be equally useful. Consider composition defined as $\circ : (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$. Polymorphic functions on parametric and logical types compose such that $f \circ g = f(g(x))$. Given functions prodSum :: Pair a (Either c b) → Either (a, c) (a, b) and sumProd :: Either (a, c) (a, b) → Pair a (Either c b) ⇒ { (prodSum ∘ sumProd) :: Either (a, c) (a, b) → Either (a, c) (a, b) , (sumProd ∘ prodSum) :: Pair a (Either c b) → Pair a (Either c b) }. See Figure 8.

Figure 8. Function Composition

```
prodSum :: Pair a (Either c b) -> Either (a, c) (a, b)
prodSum (Pair (x,e)) =
    case e of
      Left y -> Left (x, y)
      Right z -> Right (x, z)

sumProd :: Either (a, c) (a, b) -> Pair a (Either c b)
sumProd e =
    case e of
      Left (x, y) -> Pair (x, Left y)
      Right (x, z) -> Pair (x, Right z)

comp :: Either (a, c) (a, b) -> Either (a, c) (a, b)
comp = prodSum . sumProd

comp' :: Pair a (Either c b) -> Pair a (Either c b)
comp' = sumProd . prodSum
```

Walder observed the composition of polymorphic functions with monomorphic functions induced theorems for free. [Wadler 1989] Given a polymorphic function fn :: Left x →

Left x, the monomorphic functions ord :: Char → Int and chr :: Int → Char and the functor mapEither :: (a → c) → (b → d) → Either a b → Either c d we can for example state the following theorem : (mapEither ord chr ∘ fn) = (fn ∘ mapEither ord chr). Recall from Figure 6 the second monomorphic parameter chr is not used by data constructor Left.

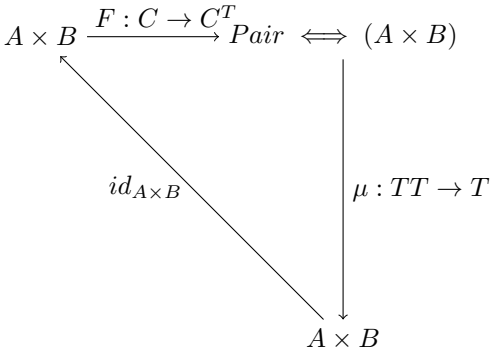
D. Categorical Semantics

Goguen first proposed the explicit junction between computer science and category theory. [Goguen, 1989] The Categorical Manifesto [Goguen, 1989] promulgates seven dogmas for using category theory to the faithful. We examine the categories of particular interest to an incomplete and preliminary exposition of parametric and logical types to model driven engineering in adherence to our belief in the manifesto.

1) Relational Type Constructors: . The category **Rel** is the category whose objects are sets and whose morphisms are binary relations between sets. A morphism $R : A \to B$ between sets A and B is the relation $R \subseteq A \times B$. Relations $R : A \to B$ and $S : B \to C$ may be composed such that $(a, c) \in S \circ R \iff \{b \in B; (a, b) \in R \text{ and } (b, c) \in S\}$.

Given a monad $T = (T, \mu, \eta)$ in **Cat** where $T : C \to C, \mu : TT \to T$ and $\eta : Id_C \to T$, the Kleisli category C_T of the monad T is the subcategory of the Eilenberg-Moore category C^T on the free T-$algebras$. If $U : C^T \to C$ is the forgetful functor and $F : C \to C^T$ is the free algebra functor, then the Kleisli category is the full subcategory in the image of F. **Rel** is the Kleisli category for **Set**. So for parametric type constructor Pair $\iff A \times B$, we interpret Pair as the Kleisli category on the carrier set $A \times B$. See Figure 9. Note that $F : C \to C^T$ is the type constructor for $Pair\langle A, B\rangle$ which wraps $A \times B$ in $Pair$. It is called the section of μ. $\mu : TT \to T$ unwraps $A \times B$ and is called the retraction of F where $\mu \circ F = id_{A \times B}$.

Figure 9. Section and Retraction



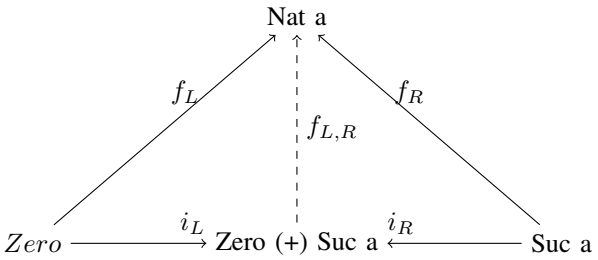
$$A \times B \xrightarrow[F : C \to C^T]{} Pair \iff (A \times B) \xrightarrow[\mu : TT \to T]{} A \times B$$

Section Retraction

2) Functors on Data and Type Constructors: Given categories D and T the functor F maps the objects of D to the objects of T and the arrows of D to the arrows of T. $F : D \to T$. Type and data constructors provide a familiar example of functor from computer science where D is the category of data constructors and T type constructors. [Pierce 1988] Consider the diagram D with data constructors $Zero$ and $Suc\ a$ as objects in the category T of type constructor $Nat\ a$. See Figure 10.

In the functor $F : D \to T$ the arrow f_L maps the object $Zero$ to $Nat\ a$ and f_R maps object $Suc\ a$ to $Nat\ a$. The sum $Zero\ (+)\ Suc\ a$ is the disjoint union of the objects $Zero$ and $Suc\ a$ induced by the injections i_L and i_R. The arrow $f_{L,R}$ maps the arrows i_L and i_R to the arrows f_L and f_R respectively. Nat a is the coproduct of $Zero$ and $Suc\ a$ if there exist injections i_L and i_R satisfying a universal property $f_{L,R}$ such that $f_L = f_{L,R} \circ i_L$ and $f_R = f_{L,R} \circ i_R$.

Figure 10. Coproduct



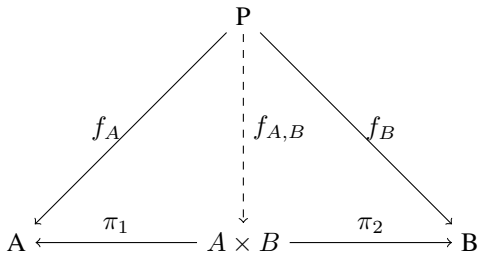
More generally a bifunctor B is simply a functor whose domain is the product of two categories. $B : L \times R \to T$ Recall the left and right disjuncts from Example A. The combined Left and Right disjuncts comprise a bifunctor from the respective data constructors to type constructor Either. See Figure 10. Notice the function mapEither in Haskell syntax takes objects in Either a b to objects in Either c d though the assignment operation and the arrows of Either a b to the arrows of Either c d through function application. The result type of functor mapEither :: $(a \to c) \to (b \to d) \to Either\ a\ b \to Either\ c\ d$ is called a sum type or coproduct.

Figure 11. Bifunctor Either

```
-- bifunctor either
mapEither :: (a -> c) -> (b -> d) -> Either a b -> Either c d
mapEither f _ (Left x) = Left (f x)
mapEither _ g (Right x) = Right (g x)
```

3) Category of Products in $Pair\langle A, B\rangle$: Recall that $Pair\langle A, B\rangle$ is the type constructor on the product of two sets A and B. The Cartesian product of sets $A \times B$ is the set of all ordered pairs (a, b) whose first coordinate is a and second is b. Also recall $Pair\langle A, B\rangle$ is equipped with operations $A\ fst()$ and $B\ snd()$ from which we recover the first and second coordinates of pairs respectively. Given the category P with objects A and B, the product $A \times B$ is an object in P along with the arrows $\pi_A : A \times B \to A$ and $\pi_B : A \times B \to B$ satisfying the universal mapping property that for every object P and pair of morphisms F_A and F_B there is a unique morphism $f_{A,B} : P \to A \times B$ such that the following diagram commutes. π_A and π_B are the projections of $A \times B$ from which we recover the respective objects comprising the product.

Figure 12. Product



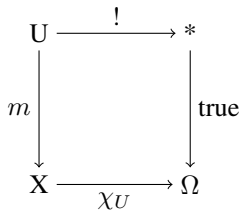
Given any category **C** with any objects A and B we call the set of arrows $Hom(A, B) = \{f \in \mathbf{C} \mid f : A \to B\}$ its Hom-set. $Hom(A, B)$ is an object S in **Set**, the category of sets with arrows $h : S \to S'$. Any arrow $g : B \to B'$ in **C** induces a function $Hom(A, g) : Hom(A, B) \to Hom(A, B')$ in **Set** such that $Hom(A, g) = g \circ f$ determines the covariant representable functor $Hom(A, -) : \mathbf{C} \to$ **Set**. [Awodey 2005] Further any object P and pair of arrows $p_1 :\to A$ and $p_2 \to B$ determine an element (p_1, p_2) of the set $Hom(P, A) \times Hom(P, B)$ in **Set**. And for any object P in a category **C** with products the covariant representable functor $Hom_C(P, -) : \mathbf{C} \to$ **Set** preserves products if given functor $\mathbf{F} : \mathbf{C} \to \mathbf{D}$ and $A \times B$ in **C** and $F(A \times B)$ in **D**, then $F(A \times B) \to F\ A \times F\ B$. [Awodey 2005]

The category of sets **Set** helps build intuition in our categorical semantics. The empty set $\emptyset$ serves as the initial object. Every singleton $\{*\}$ is a terminal object. The product in **Set** is the Cartesian product of sets. The coproduct is the disjoint union. And **Set** is complete and co-complete meaning all small limits and co-limits exist. We have constructed the functor category $\mathbf{Set}^{\mathbf{C}}$ from product category **C**. If **C** is a small category with covariant functors from **C** to **Set** and natural transformations as morphisms, then the functor category $\mathbf{Set}^{\mathbf{C}}$ is a topos.

4) Elementary and Effective Topoi: Categorical interpretation of topoi originated in topological spaces. The abstract properties of a topos **Top** are those of **Set** with an internal logic. That internal logic is intutitionistic and higher order thereby inducing both a Heyting Category and providing the categorical semantics of the logic imputed to parametric logical types. We de-emphasize the geometric interpretation of **Top** in favor of the logical one.

An elementary topos **Top** is a category which has finite limits, is Cartesian closed and has a subobject classifier. A subobject classifier in a category **C** with finite limits is a monomorphism $true : * \to \Omega$ from the terminal object such that for every monic $m : U \hookrightarrow X$ in **C** there is a unique arrow $\chi_U : X \to \Omega$ such that the following pullback diagram commutes. See Figure 13.

Figure 13. Pullback in Elementary Topos



The object Ω is called the object of truth values, it contains the generic subobject true and $\chi_U : X \to \Omega$ is called the characteristic map of the subobject. Every monic $m : U \hookrightarrow X$ arises as a pullback of the generic subobject along χ_U. The pullback of a monic is a monic and because there is only one arrow ! to $*$ then χ_U is a monic. Subobjects generalize the notion of subsets. In the category of sets the two element set **2** = {f,t} is the truth object Ω. The subset classifier $t : * \to \Omega$ identifies element t as the generic subset. And given the subset $S \subseteq X$ the characteristic function $\chi_U : X \to \Omega$ is defined as $\chi_S(x) = t$ if $x \in S$ and $\chi_S(x) = f$ if $x \notin S$. ! is the function from any subset to $*$ in **Set**.

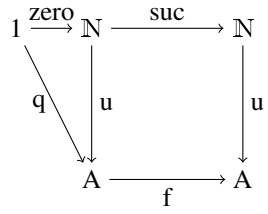
We derived the functor category $\mathbf{Set}^{\mathbf{C}}$ from the product of sets in type constructor $Pair \iff A \times B$. Any elementary topos such as **Set** is cartesian closed. Any category **C** is cartesian closed if it has a terminal object $*$ and for all objects A and B both a product $A \times B$ and an exponential B^A. Any singleton $\{*\}$ in $\mathbf{Set}^{\mathbf{C}}$ is the terminal object. Recall the covariant representable functor $Hom_C(P, -) : \mathbf{C} \to$ **Set** preserves products $A \times B$ in set as $F(A \times B) \to F\ A \times F\ B$. The exponential B^A is the set of all functions $B^A : Hom(A, g) = Hom(A, B) \to Hom(A, B')$ in $\mathbf{Set}^{\mathbf{C}}$.

Finally, recall that in addition to our UML profile we define $Unit$ as a singleton class with one value $Unit$ and $\perp$ the abstract class with no values. See Figure 7. $Unit$ is the terminal object and $\perp$ the initial object allowing finite limits and colimits in elementary topoi. $Unit$ may also be named $*$ or 1 as above.

Recall from Example B, the product from which the functor category $\mathbf{Set}^{\mathbf{C}}$ is derived is $(PlusOne \iff (T \iff R) \times (Nat \iff A)$. The effective topos **Eff** is an example of an elementary topos with a natural numbers object. **Eff** is an environment for higher order recursion where the functor $Hom(1, -)$ preserves the internal logic of **Eff** in $\mathbf{Set}^{\mathrm{Eff}}$.

A natural numbers object in a topos **Top** with a terminal object 1 is an object $\mathbb{N}$ equipped with an arrow $zero : 1 \to \mathbb{N}$ and an arrow $suc : \mathbb{N} \to \mathbb{N}$ such that for every other diagram $1 \xrightarrow{q} A \xrightarrow{f} A$ there is a unique morphism $u : \mathbb{N} \to A$ such that the following diagram commutes. See Figure 14.

Figure 14. Effective Topos Diagram



The pair (q, f) is called the recursion data for u. Notice $q = u \circ zero$ and $\forall n \in \mathbb{N}, u(suc\ n) = f(u\ n)$.

Consider function space evaluation in Example B. We interpret the construction in the effective topos **Eff** with a natural numbers object $Nat\ A$ and an object $PlusOne : Nat\ A \rightarrow Suc\ (Nat\ A)$, and the arrow $u : Nat\ A \rightarrow PlusOne$. **Eff** also has terminal object $Unit$ and initial object $\perp$. Following the recursion data in Figure 14, we construct $zero : Unit \rightarrow Zero$ and $suc : ((Nat \iff A) \rightarrow (Suc \iff (Nat \iff A))$. Substituting $PlusOne$ for f we have $q = u \circ Zero$ and $u\ (suc\ n) = PlusOne\ (suc\ n)$ the required proof that the diagram commutes, noting $u = PlusOne$. Also recall that function space evaluation is the ordered product $((PlusOne \times (Nat\ A)) \times Nat\ A)$. $Nat\ A$ and $PlusOne$ are inductive types interpreted as an initial algebra of an endofunctor in **Eff** with finite limits and co-limits.

While we admit to an incomplete analysis and do not claim to solve all the challenges of QVT, MOF and OCL, parametric and logical types offer the following technical advantages: 1) They represent in part Reynold's abstraction theorem following his finding "The way out of this impasse is to generalize homomorphisms from functions to relations" in a simple UML profile. [Reynolds 1983] 2) They admit a higher order intuitionistic logic on relations in which their construction and proof can be formally specified. 3) They allow parametric polymorphism with advantages in composition and parametricity. 4) Their semantics can be formulated in categories without abuse, formalizing definitions and theories from computer science, carrying out proofs, discovering and exploiting relations with other fields, dealing with abstraction and representation independence. [Goguen 1989]

Equipped with conjunctive and disjunctive parametrized logical types through our type construction profile we gain both a formal type and categorical interpretation in our meta modeling practice. Work is not complete on the logic. We prove that the logic is sound and complete.

V. Proof Calculus for Logical Types

Although logical types offer certain advantages over the naive meta model approach, we need to assess their use in a proof system. We want our proof system to be sound and complete. Given introduction and elimination rules our system is sound when from the introduction rules we prove the elimination rules do not invalidate the introduction rules by deriving new consequences. The system is complete when we can reconstruct arbitrary proofs created from the introduction rules using the elimination rules without loss of information.

Logical types comprise a fragment of Heyting algebra (HA). HA is a formalization of the BHK interpretation of intuitionistic logic. HA is a bounded lattice with join $A \vee B$ and meet $A \wedge B$ operations. It admits $\top$, aka true, $\perp$ aka absurd and implication $A \Longrightarrow B$. True is constructed, absurd is not. We will not use the universal $\forall$ or existential $\exists$ quantifiers. HA does not admit the law of excluded middle, therefore the contradiction $A \wedge \neg A$ is not a rule of intuitionistic logic. We follow Pfenning's approach to intuitionistic natural deduction. [Pfenning 2004]

A. Introduction and Elimination Rules

A verificationist approach begins with definitions and proves the definitions from introduction rules. We proceed by induction over the structure of the data and type constructors using natural deduction. Natural deduction is a proof calculus with a line denoting an inference and introduction and elimination rules to its right. With introduction rules we read the inference from the bottom to the top. With elimination rules we read from the top to bottom. Labeled propositions which encode an interpretation are said to be judgments.

Pair Introduction : We read the judgment on the bottom to infer the judgment(s) on top using the rule to prove the inference. Here the first judgment is the data constructor Pair (A,B) true. The introduction rule (Pair) I reads that if judgment Pair (A,B) is true, then judgment A must be true and if judgment Pair (A,B) is true then judgment B must be true. We verify the inference holds in a model theoretic interpretation.

$$\frac{A\,\text{true} \qquad B\,\text{true}}{Pair(A,B)\,\text{true}}\ (\text{Pair})\ I$$

Pair Elimination : We read the judgments on top to infer the judgments on the bottom using each elimination rule to prove the inference. The first judgment is again the data constructor Pair (A,B). By elimination rule (Pair) E_1, we infer the judgment A true. Likewise, by elimination rule (Pair) E_2, we infer the judgment B true.

$$\frac{Pair(A,B)\,\text{true}}{A\,\text{true}}\ (\text{Pair})\ E_1 \qquad \frac{Pair(A,B)\,\text{true}}{B\,\text{true}}\ (\text{Pair})\ E_2$$

Either Elimination : A pragmatist approach proceeds by inference from how a judgment is used. Observe the type constructor Either (L, R) true on top. For disjunction we need only to prove one data constructor is true while preserving a model theoretic interpretation. To its right we define two hypothetical judgments. One, a proof of C given data constructor Left true. Second, a proof of C given Right true. We read the judgment Either (L, R) true to mean there are no interpretations where either Left is false or Right is false. Elimination rule (Either) $E^{x,y}$ reads that if we have a proof of C from Left true than we can conclude C. Or if we have a proof of C from Right true we can conclude C. Variables x and y are exclusive to the hypothetical judgment in which they are introduced.

$$\dfrac{\text{Either } (L,R)\,\text{true} \qquad \begin{array}{c}\overset{x}{\overline{\quad}}\\ Left\ \text{true}\\ \vdots\\ C\,\text{true}\end{array} \qquad \begin{array}{c}\overset{y}{\overline{\quad}}\\ Right\,\text{true}\\ \vdots\\ C\,\text{true}\end{array}}{C}\ \text{Either}\ E^{x,y}$$

Either Introduction : Notice it would be unsound to infer judgment Left true from judgment Either (L,R) true in every model theoretic interpretation of classical logic. From judgment Either (L,R) true on the bottom we read (Either) I_1 to mean that there is at least one model theoretic interpretations of Left true in intuitionistic logic, while recalling we do not admit the law of excluded middle. (Either) I_2 likewise means that from judgment Either (L,R) true there is at least one model theoretic interpretation in intuitionistic logic for Right true.

$$\dfrac{\text{Left}\quad \text{true}}{Either(L,R)\,\text{true}}\ (\text{Either})\ I_1 \qquad \dfrac{\text{Right}\quad \text{true}}{Either(L,R)\,\text{true}}\ (\text{Either})\ I_2$$

B. Sound and Completeness Justification

We prove soundness by reduction and completeness by expansion. Reduction checks that no new consequences are discovered when applying the elimination rules to the proofs derived from the introduction rules : we do not gain information. Complete checks that we do not lose information when reconstructing an arbitrary proof from the elimination rules. Sound and complete checks are called justifications.

Pair Reduction : Reading the judgment from type constructor Pair (A,B) true assume arbitrary proofs $\mathcal{D}$ of A and $\mathcal{E}$ of B from the pair introduction rule (Pair) I. By elimination rule (Pair) E_1 we reduce Pair (A,B) true to A true, the subject of proof $\mathcal{D}$. Likewise by elimination rule (Pair) E_2 we reduce Pair (A,B) true to B true, the subject of proof $\mathcal{E}$. Our justification is sound. We have used all the elimination rules to the introduction rule and we have gained no information. Soundness is local to the instant proof.

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E}\\ A\,\text{true} & B\,\text{true}\end{array}}{Pair(A,B)\,\text{true}}\ (Pair)I}{A\,\text{true}}\ (Pair)E_1 \quad \Longrightarrow_R \quad \begin{array}{c}\mathcal{D}\\ A\end{array}$$

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E}\\ A\,\text{true} & B\,\text{true}\end{array}}{Pair(A,B)\,\text{true}}\ (Pair)I}{B\,\text{true}}\ (Pair)E_2 \quad \Longrightarrow_R \quad \begin{array}{c}\mathcal{E}\\ B\end{array}$$

Pair Expansion : From an arbitrary proof $\mathcal{D}$ of type constructor Pair (A,B) true we expand elimination rules (Pair) E_1 and (Pair) E_2 to prove A true and prove B true from introduction rule (Pair) I. We read from the top left that from elimination rule (Pair) E_1 we can conclude A true from data constructor Pair (A,B) true. Expansion allows "copies" to check completeness of expansion, therefore we use a copy of arbitrary proof $\mathcal{D}$ concluding B true from elimination rule E_2. We then verify that no information list lost by concluding both A true and B true from the introduction rule Pair (A,B) true and our local expansion is justified.

$$\begin{array}{c}\mathcal{D}\\ Pair(A,B)\end{array} \quad \Longrightarrow_E \qquad //$$

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{D}\\ Pair(A,B)\,\text{true} & Pair(A,B)\,\text{true}\end{array}}{\begin{array}{cc}A\,\text{true} & B\,\text{true}\end{array}}\ \begin{array}{cc}(Pair)E_1 & (Pair)E_2\end{array}}{Pair(A,B)\,\text{true}}\ (Pair)I$$

Either Reduction : Recall introduction rule (Either) I_1 justifies a conclusion that under a model theoretic interpretation of intuitionistic logic there is at least one case where data constructor Left is true from type constructor Either (L,R). We name this proof $\mathcal{D}$. While preserving a model theoretic interpretation we require only proof of C from either data constructor Left or data constructor Right. By elimination rule (Either) $E^{x,y}$ we reduce one of the two hypothetical proofs to C. We choose the Left judgment, arbitrarily named $\mathcal{E}$. Similarly in our second case we provide proof of C from judgment Right arbitrarily named $\mathcal{F}$. Our local reduction has gained no information and local reduction is sound. Note that due to the expanded size of the proofs we elide true from judgments in this and all future proofs without loss of information.

$$\dfrac{\begin{array}{ccc}\mathcal{D} & \overset{x}{\overline{\quad}} & \overset{y}{\overline{\quad}}\\ Left & Left & Right\\ & \mathcal{E} & \mathcal{F}\end{array}}{\dfrac{Either(L,R) \qquad C \qquad C}{C}\ (Either)E^{x,y}}\ (Either)I_1 \quad \Longrightarrow_R \quad \dfrac{\begin{array}{c}\mathcal{D}\\ \overset{x}{\overline{\quad}}\\ Left\\ \mathcal{E}\\ C\end{array}}{}$$

$$\dfrac{\begin{array}{ccc}\mathcal{D} & \overset{x}{\overline{\quad}} & \overset{y}{\overline{\quad}}\\ Right & Left & Right\\ & \mathcal{E} & \mathcal{F}\end{array}}{\dfrac{Either(L,R) \qquad C \qquad C}{C}\ (Either)E^{x,y}}\ (Either)I_2 \quad \Longrightarrow_R \quad \dfrac{\begin{array}{c}\mathcal{D}\\ \overset{y}{\overline{\quad}}\\ Right\\ \mathcal{F}\\ C\end{array}}{}$$

Either Expansion : From an arbitrary proof $\mathcal{D}$ of type constructor Either (L,R) we reconstruct the proof created from the introduction rules (Either) I_1 and Either I_2 using the elimination rule (Either) $E^{x,y}$ without loss of information. We say that local expansion justifies completeness. Recall that from (Either) I_1 we conclude there is at least one model theoretic interpretation of data constructor Left. Recall that from (Either) I_2 we conclude there is at least one model theoretic interpretation of data constructor Left in intuitionistic logic. We conclude (Either) $E^{x,y}$ from either Left or Right

without losing information. Our local expansion is justified. Recall true is elided from judgments without loss of meaning.

$$\begin{array}{c} \mathcal{D} \\ Either(L,R) \end{array} \Longrightarrow_E \qquad //$$

$$\dfrac{\begin{array}{ccc} & \begin{array}{c}\overline{\quad x \quad} \\ Left\end{array} & \begin{array}{c}\overline{\quad y \quad} \\ Right\end{array} \\ \mathcal{D} & \overline{\quad (Either)I_1 \quad} & \overline{\quad (Either)I_2 \quad} \\ Either(L,R) & Either(L,R) & Either(L,R) \end{array}}{Either(L,R)}\ (Either)E^{x,y}$$

We are done. We have justified claims that logical types are sound and complete using their type and data constructors using Gentzen's natural deduction as our proof calculus. No information is gained or lost. Using both verificationist and pragmatist approaches to justification our system is "in harmony."

VI. Realizability in the Effective Topos

This section introduces realizability in the effective topos to the model-driven engineer. Ontology and fragments of first order and description logics often challenge the advancing model-driven engineer to further refine their approach to model specification and interpretation. Introducing realizability in the effective topos exposes engineers to the computability theory underlying constructive or intuitionistic logic. Our main thrust is to contrast non-standard truth values in a Heyting algebra with a classical interpretation of first order logic.

We claim the construction of a type from a value is a witness satisfying the proof of a proposition. Kleene proposed the constructive proof of a proposition in the Brouwer-Heyting-Kolmogorov (BHK) interpretation of inutionistic logic could be formulated in terms of numerical codes that are effectively computable. [Kleene 1945] Given a Godel numbering in n $\in \mathbb{N}$ and recursive function $f\ n$, then $f\ n$ is the result of applying f to n. The result $f\ n$ is a realizer. And we say that $f\ n$ realizes formula A in a Heyting algebra or is a realizer of the BHK interpretation. We make no claims of efficiency in Godel numbering.

Effective topoi are elementary topoi with a natural numbers object and an internal logic with finite limits and co-limits. The natural numbers object is recursive. Recall the internal logic of the effective topos is also a Heyting algebra. Also recall model theory based in classical logic admits an interpretation I of a model M when the set of truth values $V = \{True, False\}$ satisfy the model. [Tarski 1944]

Realizability in the effective topos implies for each sentence ψ in a Heyting algebra HA that $f\ n$ realizes ψ in HA. We define this subset $P(\mathbb{N})$ of $\mathbb{N}$ as the set of non-standard truth values of HA. More generally a $P(\mathbb{N})$ valued predicate on a set S is a function $f : S \to P(\mathbb{N})$ that associates with each predicate in S a non-standard truth value that is a subset of $\mathbb{N}$. [Vermeeren 2009] Given the non-standard truth values $f\ n$ realizes the sentences ψ in HA and the interpretation I of model M in HA is constructed.

Recall the subobject classifier $true : * \to \Omega$ from the terminal object to the object of truth values. Under a realizability interpretation the subobject classifier becomes $(f\ n)\ \psi : * \to \Omega$. Likewise the characteristic function $\chi_U : X \to \Omega$ becomes $\chi_U = (f\ n)\ \psi$ if $n \in P(\mathbb{N})$ and $\chi_U = \perp$ otherwise.

VII. Conclusion

Parametric and logical types offer technical advantages over both the naive approach to meta models and QVT and its dependencies on MOF and OCL. Parametric types allow parametric polymorphism following Reynolds' abstraction theorem. Parametric types may be formally specified in type and category theory. They are composeable and induce theorems that may be proven. We have shown how UML Templates allow the specification of parametric types in UML.

Logical types lift logic from the object language to the type system, constructively. Our novel UML profile of data and type constructors and a few utility classes allowed an interpretation of intuitionistic logic. We specified informal semantics in sequence diagrams and formal semantics in type and category theory and demonstrated their realizability in the effective topos. We proved their soundness and completeness in a fragment of a Heyting algebra using natural deduction.

There's no shortage of future work following from parametric and logical types in model-driven engineering. Standardizing the profile may be of interest both to researchers in the algebraic data types community who seek industrial relevance for their research and model-driven engineers seeking an alternative to first order valuations of classical logic, description logic or ontology. Further technical work abounds. Subtopoi of the effective topos appear relevant as is homotopy. Further developing realizability in the effective topos will allow model-driven engineers to better contrast classical model theory with realizability. And further developing the BHK interpretation and its extension to dependent types as well as a proof calculus in the sequent calculus also appears relevant.

Acknowledgment

The author would like to thank George Thomas whose friendship and support made this paper possible long before it began.

References

[1] J. Reynolds, *Types, Abstraction and Parametric Polymorphism*. Pittsburg, PA. Carnegie Mellon. 1983.

[2] Object Management Group, *Unified Modeling Language, 2.5*. Needham , MA. Object Management Group. 2015.

[3] Object Management Group, *Object Constraint Language, 2.4*. Needham , MA. Object Management Group. 2014.

[4] Object Management Group, *Meta Object Facility 2.0 Query View Transformation Language 1.3*. Needham , MA. Object Management Group. 2016.

[5] G. Bracha et al, *Extending the Java Programming Language with Type Parameters*. Menlo Park, CA. Sun Microsystems. 1998.

[6] J. Goguen, *Parameterized Programming and Software Architectures*. Proceedings of Fourth IEEE International Conference on Software Reuse. IEEE. 1996.

[7] T. Mossakowski et al, *Common Algebraic Specification Language*. Common Framework Initiative. ESPRIT COFI Wroking Group. 2004.

[8] M. Abadi, *An Imperative Object Calculus*. Proceedings of the Second ACM SIGPLAN Workshop on State in Programming Languages. Technical Report UIUCDCS-R-95-1900, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.
[9] P. Wadler, *Theorems for Free*. 4'th International Conference on Functional Programming and Computer Architecture, London. 1989.
[10] C. Strachey, *Fundamental Concepts in Programming Languages Lecture Notes*. Copenhagen, Denmark. International Summer School in Computer Programming. 1967.
[11] J. Goguen, *Categorical Manifesto*. Technical Monograph PRG-72. London, UK. Oxford Computing Lab, Programming Research Group. 1989.
[12] J. Reynolds, *Polymorphism is Not Set Theoretic*. Valbonne, France. Research Report RR-0296. INRIA. 1984.
[13] J.M.E. Hyland, *The Effective Topos*. Amsterdam, North Holland. L.E.J. Brouwer Centenary Symposium. 1982.
[14] A. Pitts, *Polymorphism is Set Theoretic, Constructively*. Edinburgh, UK. Proceedings of the Summer Conference on Category Theory and Computer Science, Springer Lecture Notes in Computer Science. 1987.
[15] B. Pierce, *A Taste of Category Theory for Computer Scientists*. Pittsburg. Carnegie Mellon. 1988.
[16] S. Awodey, *Category Theory*. London, UK. Oxford Logic Guides. 2005.
[17] E. Willinik, *Local Optimizations in Eclipse QVTc and QVTr using the Micro-Mapping Model of Computation*. EXE@Models. 2016.
[18] F. Pfenning, *Intuitionistic Natural Deduction*. Pittsburg. Carnegie Mellon. Chapter 2. Automated Theorem Proving. 2004.
[19] S. Kleene, *On the Interpretation of Intuitionistic Number Theory*. Journal of Symbolic Logic. 1945.
[20] A. Tarski, *Introduction to Logic: and to the Methodology of the Deductive Sciences*. Dover books on Mathematics. 1936.
[21] S. Vermeeren, *Realizability Toposes*. Darwin College, Belgium. 1936.
[22] S. Lee, *Basic Subtoposes of the Effective Toposes*. Self. 2012.