# Intensional and univalent aspects of rule-based programming

Ralph Matthes

CNRS, Institut de Recherche en Informatique de Toulouse (IRIT),
University of Toulouse

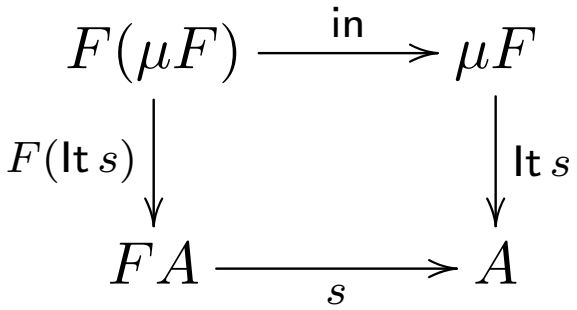
EUTypes 2018 Working Meeting, Radboud University,
Nijmegen, The Netherlands
January 24, 2018


# Abstract

I will discuss various views of what “intensional” datatypes could be, and approaches to obtain them. In particular, this includes univalent foundations.
Some other benefits of univalent foundations for programming will be mentioned, including for SQL query rewrites (triggered by the work of Chu et al on HoTTSQL).

# Outline

# Categorical inspiration from initial algebras




- $F$ is a functor ($F$ used both for object and morphism mapping)
- $\mu F$ is the carrier of an initial algebra in
- $s$ is an $F$-algebra
- $\mathsf{It}\, s$ is the "mediating morphism"

This is very far from being "intensional"!

# Why not intensional?

- is set in a category, thus in terms of a model rather than an algorithm
- how does a commuting diagram commute (w. r. t. which notion of equality)?
- an initial algebra requires uniqueness of $\mathsf{It}\, s$ (w. r. t. which notion of equality?)

# monotone inductive types

Started with a textual note sent as email by Thorsten Altenkirch in the early nineties. Question (simplified): can one have termination guarantees when programming iteratively not with syntactically identified classes of "functors"—in reality type transformers—$F$ but assuming that $F$ is "monotone", as witnessed by a suitable term of type

$$\mathsf{mon}F :\equiv \forall A \forall B.(A \to B) \to (FA \to FB)$$

?

Yes, this is can be done, and the witness for $F$ need not even be

- fixed (once and for all for a given $F$)
- a closed term (it may be a variable)

# one possible formulation

For each type transformation $F$ assume a type $\mu F$. Now I simplify to a Curry-style formulation and to partially applied function symbols in the following typing rules for the term formers. I omit typing contexts.

$$\frac{t : F(\mu F)}{\mathsf{in}\, t : \mu F} \qquad \frac{m : \mathsf{mon} F \quad s : F\, A \to A}{\mathsf{It}\, m\, s : \mu F \to A}$$

Rewrite rule:

$$\mathsf{It}\, m\, s(\mathsf{in}\, t) \longrightarrow s\big(m(\mathsf{It}\, m\, s)t\big)$$

Comparison with initial algebra diagram:

- $F$ arbitrary type transformer
- a computation rule
- no uniqueness, hence no good laws
- a witness $m$ recorded for each iterative definition

# strong normalization

In the framework of second-order polymorphic lambda-calculus (system F), the new rules do not break strong normalization, i. e., it still holds that all sequences of computation steps eventually end. This can even be shown by a syntactic encoding into system F that properly simulates reduction steps.

The type transformation $F$ can be $\lambda X.X \to 1$ or $\lambda X.X \to X$ for which closed terms of type mon$F$ exist for trivial reasons. Or $\lambda X.X \to 0$ where such a term does not exist. I do not claim that the iterator is useful in those cases.

# another possible formulation

$$\frac{m : \forall A.(\mu F \to A) \to (F(\mu F) \to FA) \quad t : F(\mu F)}{\mathsf{in}\, m\, t : \mu F} \qquad \frac{s : FA \to A}{\mathsf{It}\, s : \mu F \to A}$$

Rewrite rule:

$$\mathsf{It}\, s(\mathsf{in}\, m\, t) \longrightarrow s\big(m(\mathsf{It}\, s)t\big)$$

Recall the first formulation:

$$\frac{t : F(\mu F)}{\mathsf{in}\, t : \mu F} \qquad \frac{m : \mathsf{mon} F \quad s : FA \to A}{\mathsf{It}\, m\, s : \mu F \to A}$$

Rewrite rule:

$$\mathsf{It}\, m\, s(\mathsf{in}\, t) \longrightarrow s\big(m(\mathsf{It}\, m\, s)t\big)$$

# strong normalization of alternative formulation

$m$ is not even a proper monotonicity witness, but also this system can be embedded into system F with simulation of reduction.

One can get inspiration from Herman Geuvers' TYPES 1992 paper and formulate systems with the schemes of iteration and of primitive recursion, but now again with monotonicity witnesses instead of a positivity restriction.

# functor laws?

We do not care about them for the non-functional verification of termination through types, but we will have a hard time programming an unfolding function for $\mu F$ without them.

Positive type transformers have "good" monotonicity witnesses if a bit of $\eta$-rules is present in the system. From this side, there is no need to further restrict to strictly positive types. There are reasonable uses of positive types that are not strictly positive. One I advocated: $\mu(\lambda X.A + ((X \to 0) \to 0))$, better written as $\mu X.A + \neg\neg X$. This is not just $\neg\neg A$ for which double-negation elimination would be for free, but the free construction of a supertype of $A$ with double-negation elimination.

# inductive families

The same questions can be asked for type transformations that arise as simultaneous fixed points.
My favourite example (not my invention):

$$TA = A + TA \times TA + T(1 + A)$$

is solved by the type transformation Lam that represents untyped lambda-terms with names of free variables in the argument type $A$.

The notion "nested datatypes" is common for those inductive families indexed over all possible argument types.
One can easily formulate an iteration principle for those that is again inspired from the initial algebra diagram.

# extra problems with monotonicity of inductive families

Is Lam itself monotone? A monotonicity witness of type monLam ought to do (parallel) variable renaming.
Can it be defined from the iteration principle for nested datatypes? Not quite in general. Generalized iteration is needed, and Ralf Hinze and coauthors advocate now the name “adjoint folds” for these schemes, the reason being that behind the used Kan extensions, there is the more general situation of adjoints.

# Is there a shortcut through an impredicative definition?

Nested datatypes can be translated into higher-order polymorphic lambda-calculus, and their reductions can be simulated there. The encoding of the "carrier" $\mu F$ automatically produces positive type transformers.
However, the generic monotonicity witness for $\mu F$ thus obtained breaks the abstraction: it is in terms of the encoding. Extensional models (parametric equality theory based on logical relations) can show that the programs have the right denotational semantics. This does not describe a computation in terms of the specification of the datatype. This does therefore not qualify as an intensional description.

# struggle to represent nested datatypes in Coq

I wanted to have the specification of an arbitrary nested datatype in a module type in Coq and allow myself the use of impredicativity of Set for an implementation of this module type so that the general iteration scheme is not just provable but holds judgmentally (proof by reflexivity, hence exploiting only convertibility).

The solution (system LNMIt) is unpleasant for several reasons:

- need for proof irrelevance for sort Prop
- monotonicity witness is not an instance of the iterator
- the constructor of the nested datatype has plenty of logical arguments, including a reference to the monotonicity witness, hence there is a mild form of induction-recursion
- extensionality assumptions have to be propagated

# remark on types for programming

Here, I take a specific view. While types are already preserved under reduction (subject reduction), the types also ensure termination. It is not the form of the datatypes that guarantees it but rather the existence of monotonicity witnesses of the right type.

The adjoint folds scheme rather ensures existence and uniqueness of the recursively defined functions that follow the scheme. Again, the scheme is governed by types and not by, e. g., an analysis of decrease of arguments in recursive calls.

# machine-checked verification of recursion schemes in the categorical model

With Benedikt Ahrens (to appear in TYPES 2015 post-proceedings), I used the UniMath library to give a faithful representation of the categorical development of heterogeneous substitution systems. They are an abstract description of what substitution should be on wellfounded and non-wellfounded syntax alike, with many forms of binding captured by the formalism.

This is applied category theory, with a bunch of categories involved, in particular different categories of endofunctors, i. e., constructed categories and not abstract ones. Function extensionality is very important for all reasoning about natural transformations.

# lacking extensionality

The theoretical approach (by Tarmo Uustalu and myself) is based on signatures with strength. Laws for the strength are part of the definition of the signature. (It is crucial for certified programming that laws for the ingredients are not just treated in comment but are fully integrated.)

Experience: the laws did not type-check since the monoidal laws on endofunctors hold pointwise, hence they hold provably, but not definitionally. We had to lay out the monoidal structure, even if all the required morphisms were pointwise the identity. We then identified our strength definition as an instance of a definition by Marcelo Fiore (much later than the original paper with Uustalu).

Lesson learned: UniMath is still asking for a quite intensional thinking for the formulation of assertions. Only the proofs are eased.

# heterogeneous substitution systems concretely in Set

The previous results were obtained for an abstract base category that has right Kan extensions. Vladimir Voevodsky wanted to see a development suitable for the sets in the sense of univalent foundations. In joint work with Ahrens and Anders Mörtberg (accepted for publication in JAR), we adapted the well-known approach with $\omega$-cocontinuous functors to UniMath. We had to use a different general recursion scheme from Bird and Paterson. It does not come from initiality but from the construction of the nested datatype as $\omega$-colimit.

The creation of a library of $\omega$-cocontinuous functors was the harder task. We also extended this to slice categories to capture simple type information, see the TYPES 2017 talk.

Criticism: definitional equality I had previously with ordinary Coq seems lost.

# hope from the Bachelor thesis 2017 by Felix Rech

Felix Rech studies $M$-types—the coinductive reading of the functors underlying $W$-types. He takes the corecursor that had been constructed previously in univalent foundations and transforms it by help of propositional truncation into a corecursor that has the usual corecursion equation definitionally (he also gives a construction of $W$-types that are carved out of the $M$-types and also fulfill the recursive equations definitionally).

Propositional truncation (called “squash” in the talk by Nicolas Tabareau on Monday) turns types into propositions of univalent foundations. The elimination rule only allows to create other propositions from it. But much more freely than with monads: the proposition need not be a truncation. It can have computational content that can be extracted when programming the destructor.

# Intermezzo on data from propositions

UniMath has in `UniMath/Combinatorics/FiniteSets.v` the function `fincard` that extracts the cardinality from every Bishop-finite type (see talk by Niels van der Weide of Tuesday), which is determined by the mere existence of a standard finite set of $n$ elements that is equivalent. The real proof work for this is found in `squash_pairs_to_set` in `UniMath/Foundations/Sets.v`.

Not so surprising: Martín Escardó recalled in his introduction to univalent foundations in Birmingham in Dec. that from the mere existence of a zero of an endofunction over $\mathbb{N}$, one can extract a zero since the least such zero is captured by a proposition.

# Summary—univalent foundations at the service of recursion schemes

- category theory needs function extensionality which is a consequence of univalence
- function extensionality does not enter the type-checker, hence more subtle statements are needed—may give rise to see the general mathematical structure
- recursion schemes typically uniquely determine a function; packaging this function with the property gives a proposition (making it usable in presence of propositionally truncated types) and still contains access to the function as data

# Are univalent sets good as cardinalities?

By univalence, types that are in bijective correspondence are equal (involves the "graduate student theorem"). Sloppily, this means that they share all properties. But, of course, they only share properties that can be expressed for any type. We saw before that finiteness is such a property, and even the cardinality can be extracted.

Wilmer Ricciotti made on Tuesday quite some comments on the PLDI 2017 paper on HoTTSQL by Shumo Chu, Konstantin Weitz, Alvin Cheung and Dan Suciu. I'll make some extra comments since I promised them in the abstract.

SQL query rewriting is certainly an activity in the field of programming, and we see a rather unusual role of types here as multiplicities, but on the level of types. No quotienting is needed—equality is already guaranteed.

# Claims by the HoTTSQL authors

Univalent foundations allow a practical implementation of semi-ring SQL semantics. This means in particular that the types in the universe fulfill the laws of semi-rings. The latter is true by univalence.

Predicates should not have a bag but a set semantics. The full version of the paper shows the semantics of equality between expressions as equality type between the individual semantics. If those are not 0-truncated, the result will not be a proposition. However, why would one go beyond Set as cardinalities?

Propositional truncation mediates between bag and set view.

Infinite cardinalities are needed since, otherwise, one would have to carry around information about finite support.

# some thoughts

- How much does it help to have equality instead of some setoid equivalence between these "cardinalities"? I mostly see that individual rules are verified.
- No convertibility helps with these "cardinalities". However, the rules that are verified are schematic and will hardly profit from advanced automatized arithmetic.
- The HoTT book introduces the type of cardinalities as $0$-truncation of Set. This does not seem needed, but it might be cleaner.
- Is this in the topic of programming with types? At least, the tool developers implement a formal semantics that computes types (the well-studied type-level programming).