Homotopy Types and Formal Program Specification

Pedro Cunha[1] and Mário Florido[2]

[1] Vortex CoLab, Rua de Serpa Pinto, 44, 4400-012 Vila Nova de Gaia, Portugal
pedro-miguel.cunha@vortex-colab.com
[2] LIACC & Dep. de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto, Rua do Campo Alegre, s/n, 4169-007 Porto, Portugal
amflorid@fc.up.pt

Homotopy Theory is a branch of topology that treats (continuous) paths in topological spaces.

Martin-Löf's Dependent Type Theory [7] consists of three main classes of types: the Simple Types, the Dependent Types and the Identity Types. The first is well-known: the Simple Types [3] form the basis of many programming languages, such as Haskell or C. In Type Theory, these are usually taken to be: the Function Types, $A \to B$, whose elements are simple functions; the Pair Types, $A \times B$, whose elements are simple pairs (both of these are like the corresponding types in Haskell); and the Sum Types, $A + B$, whose elements are sets of elements annotated with left or right (which has a clear correspondence with the Either type in Haskell). The second is less well-known: the Dependent Types [4] can be seen as generalizations of the Simple Types. Of these, there are two relevant ones: the Dependent Function Type (or Dependent Product Type), usually noted with a $\prod$ and the Dependent Pair Type, usually noted with a $\sum$. Lastly are the Identity Types, [5] which are the way Martin-Löf tries to capture equality in his type system. It can only be formed between elements of the same type: if $a, b : A$, we can form the type $a =_A b$. This means, in particular, that elements of different types *cannot be compared using equality*.

In the 80s, Grothendieck [3] proposed that ∞-groupoids should constitute particularly adequate models for homotopy types. Shortly after, in 1990, Voevodsky and Kapranov published a proof that Grothendieck's intuition was right [9]. In 1996, Hoffman and Streicher published [4], in which they presented a groupoid interpretation of Martin-Löf's type theory. Voevodsky noticed this and connected the dots: it was possible to interpret types as homotopy types. And so, in an unpublished "short note" of Voevodsky in 2006, [10], *Homotopy Type Theory* (HoTT) was born.

Homotopy Type Theory combines the disciplines of Homotopy Theory and Dependent Type Theory by interpreting types in a homotopical fashion. Types are now to be seen as topological spaces; terms (programs, proofs) as points in such a space; and equality between them as paths, according to Table 1 (see Ch.

[3] For a more in-depth presentation of these types, see e.g. [6], Ch. 2.2 of [2] and Chs. 1.2, 1.5 and 1.7 of [8].
[4] See e.g. [6], Ch. 2.3 of [2] and Chs. 1.4 and 1.6 of [8]
[5] See e.g. [6], Ch. 2.4 of [2] and Chs. 1.12 and 2.11 of [8]

2.3 of [8] and Ch. 3.1 of [2] for the interpretation of Dependent Types). Although this paper has many pointers to other relevant sources, [8] is a core one, which can be supplemented by [2]; the work is mainly based on Licata's blog post in [5] (as well as the whole site, [1]), which also uses concepts from [11].

Type Theory	HoTT
Type A	space A
a : A	point a in A
A + B	coproduct of A and B
A × B	product space of A and B
A → B	function space of A and B
$x =_A y$	path space of A
$p, q : x =_A y$	path in A (with end points x and y)
$h, i : p =_{x=_A y} q$	paths between paths (homotopy)

Table 1. Homotopical interpretation of Martin-Löf's type theory.

Abstract Data Types and Views

Let us begin with an example, based on Licata's blogpost [5] : when talking about the naturals $\mathbb{N}$, one may use Peano arithmetic with zero and successor definitions, making it easier to reason inductively than it is to do so using the usual binary/decimal structure. Yet, when doing calculations, it is unreasonable to utilize the former and, in fact, we do utilize the latter. However, we are still talking about $\mathbb{N}$; how could we, then, allow for the cohabitation of these two approaches to the naturals?

Wadler [11] proposes a solution with the views mechanism which, in essence, ties the two representations through two functions. More broadly, we can think of $\mathbb{N}$ as an Abstract Data Type (ADT), which is a type characterized by its "abstractness", which consists, in practical contexts, in having its implementation hidden, so as to be "independent" of this implementation. Thus, one can "view" its implementation through this mechanism by constructing an appropriate function that "translates" one implementation into the other and vice-versa; whence we need, for the "view", two functions, accordingly called `in` and `out`.

This idea can be stretched by thinking about what could be described as "Abstract Modules", which can be thought of as a(n abstract) description of a type and some functions that express some expected behavior, without a specific implementation. This allows the module to be treated by itself, without any reference to the implementation details.

However, when reasoning about one of these modules, one needs to be sure that the implementation is, in fact, faithful to the ADT it represents, in the sense that, upon doing some proof, it would not "get stuck" within the chosen representation.

Again, the views mechanism can be used to resolve this issue: one could think of an abstract module for sequences, with some functions, such as `append` and `map`, being implemented, e.g. over reverse lists, which have a one-to-one correspondence with traditional, head-first lists, whence the `in` and `out` functions are naturally obtained. However, for a function such as `append` to work for the head-first implementation, one would have to define a specification function, i.e., a function that allows to "translate" `append` to said implementation, which would require the `in` and `out` functions; in fact, one would have to define such specifications for all of the module's functions (see Ch. 4.2 of [2]).

Note that these new specification functions are propositions that state that the sequence module is not stuck inside a specific implementation, as long as there are equivalent implementations (i.e. they have a one-to-one correspondence to the chosen one, which supplies the `in` and `out` functions), which is exactly the desired property, stated a few paragraphs before. This approach, then, can be seen as a sort of verification of the fact that the module is indeed "independent" of its implementation.

HoTT can provide an alternative compact solution to this issue; by first taking the abstract sequence module as a nested Σ-type, like so:

```
SEQ =    Σ Seq : (Type → Type) . (
         Σ empty : (Seq A) . (
         Σ single : (A → Seq A) . (
         Σ append : (Seq A → Seq A → Seq A) . (
         let map =
            ((A → B) → (Seq A → Seq B))
            in map
       ))))
```

HoTT can be used to work through this specification in a more compact fashion.

This approach, together with the fact that programs are now points of a topological space (since types are themselves topological spaces and programs are points of these types), means that two implementations (i.e., two instances of `SEQ` with, e.g., head-first lists and tail-first list implementations) will be two points in the same space (the abstract module `SEQ`) and equality between those points will be a path with them as endpoints. Moreover, since equality between pairs means equality between their components, this specific equality will mean two paths or, more precisely, several pair-nested equalities (see Chs. 2.2 and 2.3 of [2], App. A.2 of [8]).

Now, having the list and reverse list implementations, which instantiate the abstract module `SEQ`, like this

```
ListSeq : SEQ
ListSeq =
(List , ( [] , (lsingle , (lappend , lmap ))))

RListSeq : SEQ
RListSeq =
(RList , (rnull , (rsingle , (rappend , rmap ))))
```

only `in` and `out` functions between the two are needed; since these two structures have a one-to-one correspondence, an equivalence suffices. Note, however, that in HoTT an equivalence is not *only* a function with an inverse: an equivalence is such a function packed together with its inverse and proofs of such (i.e. two homotopies; see Ch. 3.2 of [2]).

With all this, the specification can be done in one line, which states:

$$\texttt{spec} : \texttt{RListSeq} = \texttt{ListSeq}$$

Note that instantiating `SEQ` with the reverse list implementation was not necessary; it becomes clear from working through the example that the only needed element is an equivalence between any given implementation associated with it and the list implementation (or the "viewing" implementation); more generally, this applies to any two equivalent implementations.

Concluding, in this case, Homotopy Type Theory provides a much more compact specification when compared to the use of a current dependently typed programming language, avoiding the (boring) need of giving a different specification for each component of the signature.

Acknowledgements Work supported by the European Union/Next Generation EU, through Programa de Recuperação e Resiliência (PRR) [Project Route 25 with Nr. C645463824-00000063] and Base Funding UIDB/00027/2020 of the Artificial Intelligence and Computer Science Laboratory – LIACC - funded by national funds through the FCT/MCTES (PIDDAC).

References

1. Homotopy Type Theory Blog, `https://homotopytypetheory.org/`
2. Cunha, P.: Homotopy Type Theory: a Comprehensive Survey. Master thesis, Mestrado em Ciência de Computadores, Faculdade de Ciências da Universidade do Porto (2023)
3. Grothendieck, A.: Esquisse d'un programme (with translation), french version: `https://webusers.imj-prg.fr/~leila.schneps/grothendieckcircle/EsquisseFr.pdf` and english translation: `http://matematicas.unex.es/~navarro/res/esquisseeng.pdf`
4. Hofmann, M., Streicher, T.: The groupoid interpretation of type theory. In: Twenty-five years of constructive type theory (Venice, 1995), Oxford Logic Guides, vol. 36, pp. 83–111. Oxford Univ. Press, New York (1998)
5. Licata, D.: Abstract types with isomorphic types. `https://homotopytypetheory.org/2012/11/12/abstract-types-with-isomorphic-types/`
6. Martin-Löf, P.: Intuitionistic Type Theory. In: Studies in proof theory (1984)
7. Martin-Löf, P.: An Intuitionistic Theory of Types: Predicative Part. In: Rose, H., Shepherdson, J. (eds.) Logic Colloquium '73, Studies in Logic and the Foundations of Mathematics, vol. 80, pp. 73–118. Elsevier (1975). https://doi.org/https://doi.org/10.1016/S0049-237X(08)71945-1
8. The Univalent Foundations Program: Homotopy Type Theory: Univalent Foundations of Mathematics. `https://homotopytypetheory.org/book`, Institute for Advanced Study (2013)

9. Voevodsky, V.A., Kapranov, M.M.: ∞-Groupoids as a model for a homotopy category. Uspekhi Mat. Nauk **45**, 183–184 (1990), http://dx.doi.org/10.1070/RM1990v045n05ABEH002673
10. Voevodsky, V.: A very short note on the homotopy λ-calculus. Unpublished note pp. 10–27 (2006), https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/2006_09_Hlambda_0.pdf
11. Wadler, P.: Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 307–313. ACM Press (1987). https://doi.org/10.1145/41625.41653, https://doi.org/10.1145/41625.41653