Homotopy Type Theory in Isabelle

Joshua Chen ✉ ⌂ ![ORCID]

University of Nottingham, UK

--- **Abstract** ---

This paper introduces Isabelle/HoTT, the first development of homotopy type theory in the Isabelle proof assistant. Building on earlier work by Paulson, I use Isabelle's existing logical framework infrastructure to implement essential automation, such as type checking and term elaboration, that is usually handled on the source code level of dependently typed systems. I also integrate the propositions-as-types paradigm with the declarative Isar proof language, providing an alternative to the tactic-based proofs of Coq and the proof terms of Agda. The infrastructure developed is then used to formalize foundational results from the Homotopy Type Theory book.

1 Introduction

Isabelle [15] is a simply typed proof assistant and logical framework. Of its multiple object logics Isabelle/HOL [12] is arguably the best known, however many other logics have been created since Isabelle's inception and are still bundled along with its distribution. Among these early logics is Isabelle/CTT (*constructive type theory*) [13], which is based on extensional Martin-Löf type theory but which has not been further developed. In light of considerable recent progress in the field of dependent type theory, it seems appropriate to revive support for this in Isabelle. This paper aims to do this by introducing *Isabelle/HoTT*, the first development of homotopy type theory in Isabelle.

Widely accepted folklore in the theorem proving community holds that a sufficiently strong logical framework can in principle be used to encode and work in any foundational theory of equal or lesser strength. However, a drawback to this approach is that one then has to implement the foundation-specific infrastructure on one's own, while working within the additional constraints imposed by the framework. This becomes particularly clear when the formalism of the framework logic is sufficiently different from that of the object logic, as in our current case.

The Isabelle/HoTT project may thus be viewed in three distinct but related ways:

- As the beginnings of an Isabelle formalization of homotopy type theory.
- As a dependently typed Isabelle object logic which improves on Isabelle/CTT with necessary supporting infrastructure for type checking, term elaboration, proof term abstraction and tactics.
- As a practical case study on the implementation issues discussed in the previous paragraph.

This is a short paper on ongoing work. Isabelle/HoTT currently lacks automation for function definitions, datatypes, and advanced features like higher inductive types (although these may be manually defined or postulated). Despite this, it is already able to formalize nontrivial results from the Homotopy Type Theory book [14]. In addition, although the logic presented here is formulated in the axiomatic style of the HoTT book, one could use the same approach to develop two-level type theory [1, 3] and cubical type theory [2, 8] in Isabelle.

Isabelle/HoTT is implemented as a library of Standard ML and Isabelle theory files. References to specific files are given as footnotes throughout this paper, and the source code is available online at `https://github.com/jaycech3n/Isabelle-HoTT/tree/ITP2021`.

Related Work

One of the earliest object logics for Isabelle was Paulson's Isabelle/CTT [13] for constructive type theory with extensional equality. Indeed, the fundamental ideas of using resolution to perform type checking and inference, and of discharging subgoals in order of increasing flexibility, already appear here. Isabelle/HoTT improves on this work by implementing universes, an intensional equality type, as well as better integration of type inference and implicit elaboration into the proof process.

Another recent study in developing homotopy type theory in a logical framework appears in work by Barras and Maestracci [5], where they present a partial embedding of de Morgan cubical type theory [8] using rewrite rules in the $\lambda\Pi$-calculus modulo logic of Dedukti [4]. Our encoding of axiomatic HoTT in simple type theory is more straightforward, allowing us to focus instead on issues arising from integrating the simply and dependently typed paradigms of the meta and object logics.

The largest computer developments of homotopy type theory are well known and use the Coq and Agda proof assistants [6, 7]. In these settings the theory is developed synthetically, and in the case of Coq the source code was directly modified in order to implement new features required by the theory. In our case the trusted prover code is untouched, and we simply extend Isabelle/Pure with new features using its existing logical framework facilities.

2   Logical Foundations

Judgments.   We begin[1], as usual, by declaring a meta type o of terms of the object logic, and a constructor `has_type` :: $o \rightarrow o \rightarrow prop$ (written as usual with an infix colon) to encode the typing assertion. Since we implement Russell-style universes, types are themselves terms and must have the same meta type, and in this way we will effectively have a set of untyped terms in higher order abstract syntax. Working with Tarski-style universes would allow us to maintain the syntactic type/term distinction with separate meta types, at the cost of having to introduce interpretation operators everywhere.

Judgmental equality of the type theory is shallowly embedded using the Isabelle/Pure equality $\equiv$. This forgets type information, but allows us to more easily reuse the simplifier to compute terms.

Universes.   We postulate a set of levels isomorphic to the standard natural numbers with their usual order, by declaring a meta type lvl and constants $\mathtt{O}$, $\mathtt{S}$ and $<$. Universes are formed by a constructor $\mathtt{U} :: lvl \rightarrow o$, and we axiomatize rules governing the ordering of levels, as well as the hierarchy and cumulativity of universes.

---

[1] `mltt/core/MLTT.thy`

Types and Terms. The constants for formers, constructors and eliminators for the Π, Σ and identity types are postulated using Church-style typing. Type families as well as function arguments to dependent eliminators are encoded using meta instead of object lambda terms. For example, in theoretical presentations the Σ-eliminator might be given by a term $\texttt{SigInd}(A, B, C, f)$ whose third and fourth arguments are, respectively, a type family $C\colon (\Sigma\, A\, B) \to U$ and a function $f\colon \Pi(x\colon A)(y\colon B(x)).\, C(x, y)$ defining the value of C on all pairs $(a, b)\colon \Sigma\, A\, B$. In the encoding, these are instead given as the simply typed meta functions $C :: o \to o$ and $f :: o \to o \to o$. However, after the Π-type has been encoded, Isabelle's implicit coercion mechanism (Section 12.3 of [17]) is used to coerce object functions into meta functions, which allows users to ignore this distinction most of the time.

Inference Rules. Following Jacobs and Melham [11], we define an encoding ε from the judgments of dependent type theory into Isabelle/Pure by sending $x_1\colon A_1, \ldots, x_n\colon A_n \vdash \mathcal{I}$ to the universally-quantified implication

$$\bigwedge x_1, \ldots, x_n.\ x_1\colon A_1 \Longrightarrow \cdots \Longrightarrow x_n\colon A_n \Longrightarrow \varepsilon(\mathcal{I}),$$

where $\varepsilon(t\colon T) := t\colon T$ and $\varepsilon(a \equiv b\colon T) := a \equiv b$ are the encodings of typing and equality discussed above. This encoding is recursively extended to inference rules by defining

$$\varepsilon\!\left(\frac{\mathcal{J}_1 \quad \cdots \quad \mathcal{J}_k}{\mathcal{J}}\right) := \big(\varepsilon(\mathcal{J}_1) \Longrightarrow \cdots \Longrightarrow \varepsilon(\mathcal{J}_k) \Longrightarrow \varepsilon(\mathcal{J})\big).$$

Note that entailment and derivability are both translated to Pure implication. The usual rules for formation, introduction, elimination, computation and congruence of Π, Σ and equality types (see e.g. [14]) can then be axiomatized.

More generally, a statement

$$\bigwedge \vec{x}.\ P_1(\vec{x}) \Longrightarrow \cdots \Longrightarrow P_k(\vec{x}) \Longrightarrow Q(\vec{x}) \tag{1}$$

in Isabelle/HoTT may be viewed as an extended form of type-theoretic judgment

$$\Gamma \vdash t\colon T \quad \text{or} \quad \Gamma \vdash t \equiv s\colon T \ (\text{where } T \text{ is implicit}),$$

where contexts $\Gamma = \big(P_1(\vec{x}), \ldots, P_k(\vec{x})\big)$ are also allowed to contain equality judgments, and where the metavariables $\vec{x} = \{x_1, \ldots, x_m\}$ may appear throughout. In particular, the x_i need not be explicitly typed by the context Γ. Such occurrences typically appear as unification variables in type checking and elaboration problems.

3 Proof Infrastructure

Implicits and Elaboration. Implicit arguments and term elaboration are crucial to working in a dependently typed system. We declare constants `?` and `{}` representing, respectively, holes and implicit arguments, together with a theorem attribute `implicit` and an Isabelle syntax phase operation `make_holes`.[2] We can then use the usual definitional facilities together with the `implicit` attribute in the usual manner, e.g.

```
definition Id_i (infix "=" 110) where [implicit]: "x = y ≡ x ={} y"
```

---

[2] `mltt/core/implicits.ML`

where `x =A y` is the fully explicit notation for the equality type. The implicits `{}` in such definitions will be parsed by `make_holes` into holes `?`, which are then further converted into schematic variables (i.e. Isabelle/Pure metavariables) in goal statements.

The implementation of implicit arguments as schematic variables means that a general goal statement in Isabelle/HoTT is schematic. Such goals are not very well supported by the existing Isar commands, so we define new goal keywords `Lemma`, `Theorem`, etc. (replacing `lemma`, `theorem` etc.)[3] as well as a command `assuming` (replacing `assume`).[4] These call the type checker on assumptions to infer their implicit arguments and thus instantiate all metavariables before passing them to the regular context assumption mechanism.[5]

Proof Terms.   Consider the task of automatically abstracting proof terms into definitions. A theorem stated in a dependently typed system is given by a single type à la Curry-Howard. In contrast, in the LCF-style setting of Isabelle the assumptions of a theorem statement are typically available as facts in an Isabelle/Isar proof context, which are lifted to premises after the conclusion has been proved. In particular, these premises are *not* bound by the type of the theorem's conclusion. This distinction is exactly the isomorphism – given by the Π-introduction rule – between open terms with variables typed by a nonempty (type-theoretic) context, and closed terms of a Π-type (i.e. lambda terms).

Hence, in Isabelle, the proof term in a theorem's conclusion must be abstracted over all variables typed by the premises, in order to form a meta lambda term. This is then wrapped up into a definition. This functionality is available as a modifier `(def)` to the goal statement keywords discussed previously.

Induction/Elimination Rules.   In dependent type theory, given a predicate $C\colon A \to U$, the elimination rule for A is used to prove $C(a)$ for all $a\colon A$. Crucially, this requires that C encodes all the assumptions needed to prove its conclusion. As previously noted, in Isabelle these assumptions may instead appear out in the Isar context, and thus in order to be able to apply elimination rules correctly we must ensure that such "free-floating" assumptions are pushed into the type of the goal.

Concretely, this involves checking the conclusion Q of a goal (1) for variables that are typed by premises or Isar context facts P_i, and then using Π-formation to push these assumptions into the object logic predicate C. Furthermore, since the Isar context is unordered and there may be typing dependencies among these assumptions, we first topologically sort them by $\lesssim$, where $t_i\colon T_i \lesssim t_j\colon T_j$ if t_i is a subterm of T_j. This process is automated by infrastructure introducing the `elim` attribute and proof method.[6]

Propositional Equality and Calculational Reasoning.   The identity type $x =_A y$ is presented as an inductive family over its endpoints x, y. Its induction principle is subject to the same general considerations for elimination rules discussed above, and the proof method `eq` for reasoning with path induction is essentially a special case of the `elim` method.

Rewriting (aka *transport*) along propositional equalities is given by a method `rewr`.[7] We additionally adapt Isar's calculational reasoning (Sections 1.2 and 2.2.4 of [17]) to so-called *calculational* types, which are types $T\colon A \to A \to U$ that have a notion of composition

---

[3] `mltt/core/goals.ML`

[4] `mltt/core/elaborated_statement.ML`

[5] `mltt/core/elaboration.ML`

[6] `mltt/core/elimination.ML`, `mltt/core/tactics.ML`

[7] `hott/Identity.thy`

$\diamond\colon \Pi\{x, y, z\colon A\}.\ T(x, y) \to T(y, z) \to T(x, z)$ expressing transitivity of T. After declaring a calculational type and its transitivity rule with the `calc` keyword[8] and `trans` attribute we can then use the familiar idiom "`have...also have...finally show`" to construct transitive chains in proofs. By design, this technology is also general enough to support reasoning with chains of homotopies $f \sim g$.

4 Type Checking

The type checker[9] is a key component integrated throughout the infrastructure described above. It is used by goal commands to perform implicit elaboration, hooked in to proof methods to automatically discharge ancillary typing conditions that arise throughout the course of a proof, and installed as an Isabelle simp-solver[10] to enable typed term reduction. It is also available as a standalone method `typechk`.

At its core is a tactic that recursively resolves goals against the type inference (i.e. formation, introduction and elimination) rules, suitable facts from the local Isar context, any additional rules declared with the `type` attribute, and the conversion rule. It is restricted to judgments $t\colon T$ where t is rigid (i.e. where the head of t is a constant) and T may be schematic. Combined with unification this yields a bidirectional type checking/inference algorithm, which is syntax-directed on the collection of type inference rules since every rule in this collection types a term with unique head. Nondeterminism is introduced when resolving against context facts and rules from the user-modifiable `type` theorem collection, and here backtracking allows the checker to try all possible options. If it fails to completely solve an inference problem, the type checker will return the goals on which it failed to the user for further refinement.

The conversion rule $\bigwedge a\, A\, A'.\ a\colon A \implies A \equiv A' \implies a\colon A'$ introduces normalization into the type checker; the simplifier is used to solve the second proof obligation. This is currently somewhat rudimentary since definitional unfolding is not yet implemented, but this is expected to be relatively straightforward to add.

As already noted by Paulson, the order in which subgoals are tackled in a type inference problem matters greatly, as the large number of metavariables – especially with implicit arguments – creates potentially many unification candidates and too large a search space if not resolved against the correct rule. He mitigates this by using a filter-and-repeat technique to attempt the subgoals with the fewest metavariables first; we achieve a similar effect by carefully ordering the premises of inference rules according to the criteria for bidirectional type systems set out by Dunfield and Krishnaswami [9].

5 Formalization

The object logic developed is used to formalize material from the first chapters of the Homotopy Type Theory book in Isabelle2020 [15], including results on equality, homotopies and equivalences, and more.[11] Figure 1 shows an example proof that the two ways one can define horizontal composition of equalities on a type A are equal[12], which is an intermediate result en route to the proof of the Eckmann-Hilton argument for $\Omega^2(A)$ (Theorem 2.1.6

---

[8] `mltt/core/calc.ML`
[9] `mltt/core/types.ML`
[10] An Isabelle simplifier component that solves subgoals arising from conditional simplification rules.
[11] `hott/*.thy`
[12] `hott/Identity.thy`.

```
locale horiz_pathcomposable = — ‹Conditions under which horizontal path-composition is defined.›
fixes i A a b c p q r s
assumes [type]: "A: U i" "a: A" "b: A" "c: A"
                "p: a =ₐ b" "q: a =ₐ b" "r: b =ₐ c" "s: b =ₐ c"
begin
  Lemma (def) horiz_pathcomp:
    assumes "α: p = q" "β: r = s" shows "p · r = q · s"
  proof (rule pathcomp)
    show "p · r = q · r" by right_whisker fact
    show ".. = q · s" by left_whisker fact
  qed typechk

  Lemma (def) horiz_pathcomp':
    assumes "α: p = q" "β: r = s" shows "p · r = q · s"
  proof (rule pathcomp)
    show "p · r = p · s" by left_whisker fact
    show ".. = q · s" by right_whisker fact
  qed typechk

  notation horiz_pathcomp (infix "⋆" 121)
  notation horiz_pathcomp' (infix "⋆'" 121)

  Lemma (def) horiz_pathcomp_eq_horiz_pathcomp':
    assumes "α: p = q" "β: r = s" shows "α ⋆ β = α ⋆' β"
    unfolding horiz_pathcomp_def horiz_pathcomp'_def
    proof (eq α, eq β)
      fix p q assuming "p: a = b" "q: b = c"
      show "refl p ·ᵣ q · (p ·ₗ refl q) = p ·ₗ refl q · (refl p ·ᵣ q)"
      proof (eq p)
        fix a r assuming "a: A" "r: a = c"
        show "refl (refl a) ·ᵣ r · (refl a ·ₗ refl r) = refl a ·ₗ refl r · (refl (refl a) ·ᵣ r)"
          by (eq r) (compute, refl)
      qed
    qed
end
```

Figure 1 Example: Horizontal composition.

of [14], also formalized in this work). This example demonstrates all the functionality described above in action: implicit elaboration of assumptions throughout all goal and proof statements, automatic definitions for the terms `horiz_pathcomp` and `horiz_pathcomp'` from their constructions via proofs, as well as path induction and calculational reasoning on equalities in the proof of the final lemma.

6   Discussion and Future Work

Isabelle/HoTT and its accompanying formalization show that Isabelle's simply typed logical framework infrastructure is feasibly able to provide strong support for modern-day developments of dependent type theory. However, many improvements are still possible, and future work aims to implement inductive and higher inductive types, as well as to explore how the techniques presented in this paper may be used to implement cubical type theory [2, 8] and two-level type theory [3, 16].

It would be productive to attempt to formalize the notion of a semisimplicial type [10] in Isabelle/HoTT. Internalizing the full definition of such an object in homotopy type theory is a well known open problem, with the current state-of-the-art requiring a two-level type theory in order to have a strict equality and natural number type on the outer level [1]. In principle, Isabelle's logical framework can easily provide these. The main hurdles would again be in implementing enough features on the object logic level, for example to support mutually inductive datatypes. In this way, the goal of formalizing semisimplicial types could provide further impetus to the development of homotopy type theory in Isabelle.

References

1   Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending Homotopy Type Theory with Strict Equality. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:17, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.CSL.2016.21`.

2   Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. In Dan Ghica and Achim Jung, editors, *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*, volume 119 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.CSL.2018.6`.

3   Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications, 2019. `arXiv:1705.03307`.

4   Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a logical framework based on the $\lambda\pi$-calculus modulo theory, 2016. URL: `http://www.lsv.fr/~dowek/Publi/expressing.pdf`.

5   Bruno Barras and Valentin Maestracci. Implementation of two layers type theory in Dedukti and application to cubical type theory. *Electronic Proceedings in Theoretical Computer Science*, 332:54–67, January 2021. `doi:10.4204/eptcs.332.4`.

6   Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The HoTT library: A formalization of homotopy type theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, page 164–172, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3018610.3018615`.

7   Guillaume Brunerie, Kuen-Bang Hou (Favonia), Evan Cavallo, Tim Baumann, Eric Finster, Jesper Cockx, Christian Sattler, Chris Jeris, Michael Shulman, et al. Homotopy type theory in Agda, 2021. URL: `https://github.com/HoTT/HoTT-Agda`.

8   Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.TYPES.2015.5`.

9   Jana Dunfield and Neel Krishnaswami. Bidirectional typing, 2020. `arXiv:1908.05839`.

10 Hugo Herbelin. A dependently-typed construction of semi-simplicial types. *Mathematical Structures in Computer Science*, 25(5):1116–1131, 2015. `doi:10.1017/S0960129514000528`.

11 Bart Jacobs and Thomas F. Melham. Translating dependent type theory into higher order logic. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, TLCA '93, page 209–229, Berlin, Heidelberg, 1993. Springer-Verlag.

12 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, April 2020. URL: `https://isabelle.in.tum.de/website-Isabelle2020/dist/Isabelle2020/doc/tutorial.pdf`.

13 Lawrence C. Paulson. *Constructive Type Theory*, April 2020. URL: `https://isabelle.in.tum.de/website-Isabelle2020/dist/Isabelle2020/doc/logics.pdf`.

14 The Univalent Foundations Program and Institute for Advanced Study. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, 1st edition, 2013.

15 University of Cambridge, Technische Universität München, and Contributors. Isabelle2020, April 2020. URL: `https://isabelle.in.tum.de/website-Isabelle2020`.

16   Vladimir Voevodsky. A simple type system with two identity types, February 2013. Unpublished note, available online at `https://www.math.ias.edu/vladimir/Lectures`. URL: `https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS.pdf`.

17   Makarius Wenzel. *The Isabelle/Isar Reference Manual*, April 2020. URL: `https://isabelle.in.tum.de/website-Isabelle2020/dist/Isabelle2020/doc/isar-ref.pdf`.