Computational Higher Type Theory (CHTT)

Robert Harper

Lecture Notes of Week 1 by Daniel Gratzer

1 Administrative Materials

- Participation in this course is measured by presenting papers to the seminar and taking notes on a weekly basis. There will be no credit assigned for the course.
- The course website is `http://cs.cmu.edu/~rwh/courses/chtt`.

2 Course Overview

This is a course dedicated to computational type theory. The term *computational type theory* (CTT) is due to Constable in conjunction with the Nuprl project [Constable et al., 1986], based on Martin-Löf's meaning explanation [Martin-Löf, 1982]. Possible topics of this course include, but are not limited to, the following:

- *Fundamentals* of computational type theory in the zero dimensional setting (as opposed to the higher dimensional setting).
- Explore the **RedPRL** [Team, 2018] proof assistant, a usable implementation of computational higher type theory.
- *Guarded computational type theory* (Sterling and Harper). Rough idea: talk about causality/time in type theory to discuss the consumption of infinite data through time.
- Higher type theory, i.e., *cubical type theory*, in the computational form [Angiuli and Harper, 2016a,b, Angiuli et al., 2017c, Cavallo and Harper, 2018]. Of particular concern in this setting are *univalence* and *inductive types*. Univalence is concerned with the identification of "equivalent" types. A computational interpretation of equivalence is roughly based on the idea of a line between types, allowing to map back and forth between them.
- Computational type theory as a *specification language* for program *verification*. Specifically the idea of using computational type theory to handle general references (Gratzer and Crary) and more generally that types classify computation, not merely logical propositions! This idea adopts the Brouwer-Heyting-Kolmogorov operational interpretation of intuitionistic logic that views type theory as a theory of *truth*, whereas the formal correspondence discovered by Curry and Howard views type theory as a theory of *proof*.
- A comparison of formal *higher type theories*, such as Cohen et al. [2016] (Kleene algebra form) and Angiuli et al. [2017a] (Cartesian form).

This week's lecture will be a more modest discussion of what defines the difference between *computational* type theory versus *formal* type theory. This discussion will be supplemented by investigating the practical differences between the formal perspective and the computational perspective on a simple type theory. This exercises will have two further benefits for our future study:

1. Provide a historical context for the development of critical ideas. In particular, the development of a relational interpretation of types.
2. Foreshadow some of the necessary technical tools for developing higher type theory [Angiuli et al., 2017b]. Higher type theory can roughly be understood as a *master theory of coercion* for the purpose of investigating the meaning of type equality. Both formal and computational higher type theory rely on two notions of a variable.

3 Computational Type Theory versus Formal Type Theory

We now position *formal type theory* (FTT) with *computational type theory* (CTT). In the beginning we focus on the zero dimensional setting and expand to the higher dimensional setting in later parts of this course. Formal type theory is also known as *structural*, *prescriptive*, or *axiomatic* type theory, whereas computational type theory is referred to as *behavioral*, *descriptive*, or *semantic*.

3.1 Formal Type Theory (FTT)

Formal type theory is a *formal logic*. It defines a *recursively-enumerable* set inductively by a collection of rules of judgments. The key judgments for formal type theory are:

- $\Gamma \vdash A \ \mathsf{type}$
- $\Gamma \vdash M : A$

These judgments are relative to the typing context Γ, which provides the typing of the free variables. A type theory is a collection of rules of such judgments, defining the precise meanings of "type" and "term". As a type theory becomes more complex, also necessary is a form of equality. Traditionally called *definitional equality* it is also defined by a pair of judgments:

- $\Gamma \vdash A \equiv B \ \mathsf{type}$
- $\Gamma \vdash M \equiv N : A$

These judgments are also used to classify *calculation.* All of the rules are expected to be stable under $\equiv$. A common way that this is ensured (at least for types) is the inclusion of the following rule:

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \equiv B \ \mathsf{type}}{\Gamma \vdash M : B}$$

These judgments link the type theory to a formal logic. A type, as defined by $\Gamma \vdash A \ \mathsf{type}$, can be treated as a proposition. Then a term can be understood as a proof. A proposition is true when there is a term that occupies the corresponding type, giving rise to the following formal correspondence:

$$\Gamma \vdash A \ \mathsf{type} \quad \sim \quad \Gamma \vdash A \ \mathsf{prop}$$

$$\Gamma \vdash M : A \quad \sim \quad \Gamma \vdash A \ \mathsf{true}$$

There is a slight mismatch in that there are types in type theory which seem utterly trivial from the perspective of logic. For instance, the natural numbers are just $\top$ under this interpretation, but we intuitively believe that the natural numbers have a more interesting structure than say, unit!

We should also expect that a standard formal type theory will define a *constructive* logic. The meaning of constructive is slightly ambiguous in its modern usage. There are at least two senses in which this can be interpreted.

1. A *computational* meaning: proofs are programs that run.
   This is a property not enjoyed by an arbitrary collection of rules and it also fails for interesting collections of sound rules. For instance, "book HoTT" is not constructive in this sense.
   There is a certain intrinsic motivation for this for a computer scientist; types classify terms which are thought of as programs so we expect them to run. Beyond aesthetic considerations, there are practical advantages to this. Upon calculating a particular term of type nat with some desired property, in a constructive type theory this term can be run to an actual numeral.
2. The *absence* of classical principles.
   That is, no proofs of the law of excluded middle, double negation elimination, or other

similar propositions. This view has been termed "not not constructive". The absence of these principles means the theory admits more models and correspondingly more logical principles, giving rise to axiomatic freedom. Another interesting observation is the fact that univalence is simply inconsistent in the presence of the law of excluded middle so it's only sensible to consider in a constructive theory.

Formal type theory also is recursively enumerable by definition. Many people also want the judgments to be *decidable*, as opposed to merely semidecidable. If the four judgments are decidable then, since type checking is proof checking, it's possible to validate whether a proof is correct. If you're using type theory for a foundation of mathematics centered around mechanization then perhaps this is a desirable property. Even in the most decidable theory the complexity of deciding judgments is "horrendous". This presents a slight practical issue for the above story.

Finally, the *computational* meaning of formal type theory is *extrinsically* imposed. If the type theory is constructive in the first sense then, after-the-fact, we can build such an interpretation.

3.2 Computational Type Theory (CTT)

In *computational type theory*, rather than working with computation after-the-fact, computation comes first. Moreover, rather than a collection of inference rules defining the type theory and bolting on models of these rules, the type theory stands alone. In particular, it does not depend on any other mathematical structures (e.g., sets) for its interpretation. Computational type theory can be seen as a foundational *theory of truth*, as opposed to a theory of proof. Computational type theory thus adopts the Brouwerian principle, in which truth is based on computation because the notion of computation (or of an algorithm) is a fundamental human faculty.

The start of computational type theory is then a system of computation, a programming language. The resulting deterministic transition system relies on two judgments

- $M \mapsto M'$, which signifies that the program M steps to the program M',
- M value, which signifies that M no longer steps and is a done computing.

The types are defined by a meaning explanation [Martin-Löf, 1982] in terms of computation. All of the objects of this meaning explanation are drawn from this programming language so that they have a notion of computation off the bat. This also means that types are just certain programs which avoids a sticking point in formal type theory, the precise relationship between terms and types.

In computational type theory, a type provides a specification of what values belong to it and is as such a program that runs to a value. A specification may look like

> nat specifies programs which evaluate to either 0 or $\mathrm{suc}(M)$ where M is a term of type nat.

This specification gives an obvious definition of what elements of a type are. An element of a type is simply a program which satisfies the specification of the type. We bundle up these ideas into two judgments:

- $\Gamma \gg A$ type, for when A evaluates to a value which names a specification,
- $\Gamma \gg M \in A$, for when M evaluates to a value that satisfies the specification of A.

These judgments are not enumerable or checkable in any sense. They are arbitrarily complex and are richer than the recursively enumerable set we encountered with formal type theory.

Types will also come equipped with a notion of *exact equality*, $\doteq$, specified at each type. This notion of equality can be bundled up into judgments:

$$\Gamma \gg A \doteq B \ \mathsf{type} \qquad\qquad \Gamma \gg M \doteq N \in A$$

Since these judgments are wildly undecidable there is a question of convenience. We can no longer rely on computers to aid us in the construction or checking of these judgments. In order to work with this, computational type theory can be paired with some proof theory (constructed after-the-fact) which provides a convenient way to establish these judgments. They cannot capture the full scope of the original judgments but by carving out useful subsets we can work with them. Since these proof theories are secondary and cannot possibly be canonical, it is supremely unimportant if they satisfy many classic proof theory properties. For instance, it's common to add explicit rules for cut or even that are not closed under computational equivalence.

4 Case Study: The Simply Typed Lambda Calculus with Base Types

We now turn to trying to make the above discussion more concrete by applying it to a simple type theory. The language consists of the types

$$A, B ::= b \mid A \to B$$

and the terms

$$M, N ::= x \mid c \mid \lambda x{:}A.\, M \mid M\, N$$

The grammar for contexts is the completely standard version for non-dependent type theories.

$$\Gamma ::= \cdot \mid \Gamma, x : A \text{ (distinct variables)}$$

To express this language as a formal type theory we now define the usual judgments:

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B} \qquad \frac{}{\Gamma \vdash c_0 : b} \qquad \frac{}{\Gamma \vdash c_1 : b}$$

Here we have assumed the existence of two distinct constants c_0 and c_1 of type b. We want these constants to provide some way to distinguish terms and they can be thought of as the two different answers a program might provide. This does not, however, make the base type equivalent to `bool`. Firstly, we have not ensured that the base type includes no other constants besides these two distinguished points, so the base type may include arbitrarily many distinct members. Even if we did limit ourselves to just two such constants we have not internalized branching on the difference in our program. In other words, we have not included anything like an `if`. This is deliberate, we wish the base type to only serve as the answer that a program provides, and including `if` would lead to subtle complications for the upcoming proof of normalization.

Of course to work with this theory we need a notion of substitution: $[M/x]M'$. More generally, it will be necessary to have a notion of substitution between contexts to allow for simultaneous substitution: $\gamma : \Delta \to \Gamma$. This means that for each $x : A \in \Gamma$, we then have $\Delta \vdash \gamma(x) : A$. The notion of applying a substitution can be extended to a full term: $\hat{\gamma}(M)$. The details of this are elided because they're a bit low level.

There remains the question of the judgment $\equiv$. We would expect to have some notion of computation included in $\equiv$. More than this, however, we would also expect it to be the case that $\equiv$ is a reflexive, symmetric, transitive, compatible (congruent) relation. The first three qualifiers ensure that $\equiv$ behaves like some notion of equality and the last would allow us to replace equals by equals in terms. Two examples rules are:

$$\frac{\Gamma, x : A \vdash M \equiv N : B}{\Gamma \vdash \lambda x{:}A.\, M \equiv \lambda x{:}A.\, N : A \to B} \qquad \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x{:}A.\, M)\, N \equiv [N/x]M : B}$$

This concludes the essentials of a formal system here. We now turn to the computational understanding of this system. There are two ways to do this and we will briefly explore both.

- We study closed terms only[1] with a notion of closed evaluation.
- We can study open terms of any type paired with a notion of *open reduction*.

Of particular importance is the differences in the equalities that these two approaches produce. Foreshadowing: in the closed term model there are a number of equalities that hold on open terms because we know that variables will be replaced with closed terms of the right type. This permits reasoning principles that are simply invalid in an open reductions setting. For instance, in the closed setting only should $x : \mathsf{nat} \vdash x + 0 \equiv 0 + x : \mathsf{nat}$.

4.1 Closed Term Computation

In order to get started with the first computational interpretation we need to define the notion of closed evaluation that we're working with. This comes with two judgments,

$$M\ \mathsf{value} \qquad M \mapsto M'$$

On top of these judgments we can of course specify the reflexive transitive closure of $(- \mapsto -)$: $(- \mapsto^* -)$. We will also use the specialized version of $(- \mapsto^* -)$: $(- \Downarrow -)$. The rule is that $M \Downarrow N$ if $M \mapsto^* N$ and N value. The rules for stepping and value judgments are given below.

$$\frac{}{c\ \mathsf{value}} \qquad \frac{}{\lambda x{:}A.\, M\ \mathsf{value}} \qquad \frac{M \mapsto M'}{M\, N \mapsto M'\, N} \qquad \frac{}{(\lambda x{:}A.\, M)\, N \mapsto [N/x]M}$$

This could also have been phrased in terms of evaluation contexts. In this set up we define

$$E ::= \cdot \mid E\, M$$

Then we can define $E\{M\}$ as filling the hole with M. The main appeal of this approach is that our operational semantics collapse to the rule

$$M \mapsto M' \iff M = E\{P\} \wedge M' = E\{P'\} \wedge P \ \mathrm{contr}_\beta\ P'$$

Where $P\ \mathrm{contr}_\beta\ P'$ holds precisely when $P = (\lambda x{:}A.\, M)\, N$ and $P' = [N/x]M$ for some M and N.

With this machinery in hand, we are in a position to state the goal that we are after for the closed notion of computation.

> We want to show that all well-typed programs terminate. That is, if $\cdot \vdash M : A$ then there is a V so that $M \Downarrow V$ (shortened to $M \Downarrow$).

There is an analogous statement for open terms, which is that open terms "normalize". To formally state this, however, we need a notion of open reduction which we do not currently possess.

Attempt 1. *If $\cdot \vdash M : A$ then $M \Downarrow$.*

Proof Attempt. We proceed by induction on $\cdot \vdash M : A$ which is inductively defined after all.

- $\Gamma', x : A \vdash x : A$
  This is vacuous since we assumed $\Gamma = \cdot$.

[1]Perhaps we can restrict to even just closed terms of base type.

- $\dfrac{\cdot \vdash M : A \to B \qquad \cdot \vdash N : A}{\cdot \vdash M\,N : B}$
  In this case our induction hypothesis tells us that $M \Downarrow$ and $N \Downarrow$. So we can get that $M\,N$ runs to two values applied to each other.
  At this point, we're very stuck however. There's no intrinsic reason that values applied to each other should always terminate. In fact, this is only true if the application is well-typed. As an explicit instance of this, if the program was not required to be typed we would have to show that the following terminates.

$$(\lambda x{:}?.\, x\, x)\,(\lambda x{:}?.\, x\, x)$$

And thus our proof fails. X

4.1.1 Tait's Groundbreaking Development

This leads us to the conclusion that we should *strengthen* our induction hypothesis somehow so that we have leverage when we get to that case. In particular, we need something with better closure conditions. This motivates the switch from termination to *hereditary termination.* In order for this switch to be a good move, we need two facts to hold.

1. It needs to imply termination.
2. It needs to be strong enough to get the induction to go through.

Hereditary termination, in the style first proposed by Tait [Tait, 1967], written $\mathsf{HT}_A(-)$ will be defined by induction on the type, A.

$$\mathsf{HT}_b(M) \triangleq M \Downarrow$$
$$\mathsf{HT}_{A\to B}(M) \triangleq \forall N.\ \mathsf{HT}_A(N) \implies \mathsf{HT}_B(M\,N)$$

The key insight is that it's defined by induction on the structure of A. As a side note, this is the negative formulation of the clause for hereditary termination at function type since it is phrased in terms of an implication. This could also be formulated to ensure that functions are terms which run to lambdas and also satisfy the above condition. This positive formulation is slightly more robust in the case of empty types but will be equivalent for our system.

Let's reattempt our proof, but with a strengthened induction hypothesis.

Attempt 2. *If* $\cdot \vdash M : A$ *then* $\mathsf{HT}_A(M)$.

Proof Attempt. We proceed by induction on $\cdot \vdash M : A$ which is inductively defined after all.

- $\Gamma', x : A \vdash x : A$
  This is vacuous since we assumed $\Gamma = \cdot$.

- $\cdot \vdash c : b$
  This is immediate since we must show that $c \Downarrow$ but $c \mapsto^* c$ and c value holds.

- $\dfrac{\cdot \vdash M : A \to B \qquad \cdot \vdash N : A}{\cdot \vdash M\,N : B}$
  We have by assumption that $\mathsf{HT}_{A\to B}(M)$ and $\mathsf{HT}_A(N)$ and therefore, unfolding definitions we have $\mathsf{HT}_B(M\,N)$.

- $\dfrac{x : A \vdash M : B}{\cdot \vdash \lambda x{:}A.\, M : A \to B}$
  Now we must show that $\mathsf{HT}_{A\to B}(\lambda x{:}A.\, M)$. In order to show this, suppose that we have

some N so that $\mathsf{HT}_A(N)$ holds. We must now show that the following holds.

$$\mathsf{HT}_B((\lambda x{:}A.\, M)\, N)$$

And now we are stuck. We have no assumption about M since it's open. On the other hand, we do know that $(\lambda x{:}A.\, M)\, N \mapsto [N/x]M$.

And thus our proof fails. X

We've failed, but we failed a step further which is progress. Our issue seems to be that we need a bit of information about open terms because we need to know about the body of the function. The definition of hereditary termination is going to make us apply it after all. The key move here to switch from closed terms to working with open terms and proving something about all well-behaved closing substitutions applied to these open terms. In order to write down our strengthened claim let us define an extension of $\mathsf{HT}_{-}(-)$ to substitutions:

$$\mathsf{HT}_\Gamma(\gamma) \triangleq \forall x : A \in \Gamma.\ \mathsf{HT}_A(\gamma(x))$$

Theorem 3. *If $\Gamma \vdash M : A$ and $\gamma : \cdot \to \Gamma$ so that $\mathsf{HT}_\Gamma(\gamma)$ then $\mathsf{HT}_A(\hat{\gamma}(M))$.*

Proof. We proceed by induction on $\cdot \vdash M : A$ which is inductively defined after all.

- $\Gamma', x : A \vdash x : A$
  In this case, we must show that $\mathsf{HT}_A(\hat{\gamma}(x))$ but this is immediate since $x : A$ is in the context and $\mathsf{HT}_\Gamma(\gamma)$.

- $\Gamma \vdash c : b$
  This is immediate since we must show that $\hat{\gamma}(c) = c \Downarrow$ but $c \mapsto^* c$ and $c\ \mathsf{value}$ holds.

- $\dfrac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}$
  We have by assumption that $\mathsf{HT}_{A\to B}(\hat{\gamma}(M))$ and $\mathsf{HT}_A(\hat{\gamma}(N))$ and therefore, unfolding definitions we have $\mathsf{HT}_B(\hat{\gamma}(M\, N))$.

- $\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : A \to B}$
  Now we must show that $\mathsf{HT}_{A\to B}(\hat{\gamma}(\lambda x{:}A.\, M))$. In order to show this, suppose that we have some N so that $\mathsf{HT}_A(N)$ holds. We must now show that the following holds.

  $$\mathsf{HT}_B(\hat{\gamma}(\lambda x{:}A.\, M)\, N)$$

  It is obvious that it suffices to show that $\mathsf{HT}_B([N/x]\hat{\gamma}(M))$. We can then define the substitution $\gamma' = \gamma[x \mapsto N]$. Then, by the assumptions that $\mathsf{HT}_\Gamma(\gamma)$ and $\mathsf{HT}_A(N)$ we have that $\mathsf{HT}_{\Gamma,x:A}(\gamma')$. Our goal, though, can be rewritten:

  $$\mathsf{HT}_B([N/x]\hat{\gamma}(M)) \iff \mathsf{HT}_B(\hat{\gamma}'(M))$$

  What was our induction hypothesis for M again? It states precisely that since $\mathsf{HT}_{\Gamma,x:A}(\gamma')$ then $\mathsf{HT}_B(\hat{\gamma}'(M))$ holds so we're done. □

In the above, I have claimed that it was *obvious* that if a term steps to another one and this new term is hereditarily terminating the first term is terminating. This can be formally stated:

Lemma 4. *If $\mathsf{HT}_A(M')$ and $M \mapsto M'$ then $\mathsf{HT}_A(M)$.*

This is a result of the determinacy of $\mapsto$ and the fact that hereditary termination is purely behavioral: it relies only on the evaluation behavior of M.

References

Carlo Angiuli and Robert Harper. Computational higher type theory I: Abstract cubical realizability. `https://arxiv.org/abs/1604.08873v2`, 2016a.

Carlo Angiuli and Robert Harper. Computational higher type theory II: Dependent cubical realizability. `https://arxiv.org/abs/1606.09638v2`, 2016b.

Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Kuen-Bang Hou (Favonia), Robert Harper, and Daniel R Licata. Cartesian cubical type theory. `http://www.cs.cmu.edu/~rwh/papers/uniform/uniform.pdf`, 2017a.

Carlo Angiuli, Robert Harper, and Todd Wilson. Computational higher-dimensional type theory. POPL 2017, 2017b.

Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. Computational higher type theory III: Univalent universes and exact equality. `https://arxiv.org/abs/1712.01800`, 2017c.

Evan Cavallo and Robert Harper. Computational higher type theory IV: Inductive types. `https://arxiv.org/abs/1801.01568`, 2018.

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. `https://arxiv.org/abs/1611.02108`, 2016.

Robert L. Constable, Stuart F. Allen, H. M. Bromley, Walter Rance Cleaveland, J. F. Cremer, Robert W. Harper, Doug J. Howe, Todd B. Knoblock, Nax P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. 1986.

Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI, Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science, Hannover 1979*. 1982.

William W. Tait. Intensional Interpretations of Functionals of Finite Type I. *Journal of Symbolic Logic*, 1967.

The RedPRL Development Team. RedPRL – the People's Refinement Logic. `http://www.redprl.org/`, 2018.