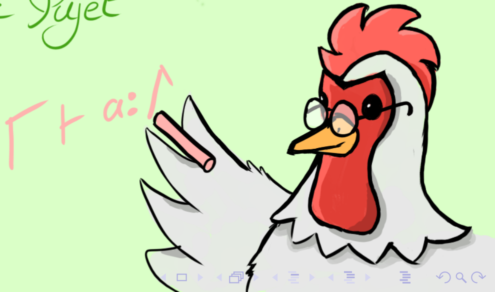
Mechanising reducibility proofs in Coq

Loïc Pujet



Plan

- Short discussion of reducibility proofs
- A Coq formalisation of decidability of type-checking

  Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, L.P.
- Continuity of MLTT functions

  Martin Baillon, Assia Mahboubi, Pierre-Marie Pédrot
- Internal computability of MLTT functions

  Martin Baillon, Yannick Forster, Assia Mahboubi, Kenji Maillard, Pierre-Marie Pédrot, L.P.

Reducibility Proofs

Common meta-theoretical properties

- Subject Reduction
- Consistency
- Canonicity
- Normalisation
- Decidability of conversion
- Decidability of type-checking
- ...

Reducibility Proofs

We cannot prove normalisation by a straightforward induction on the typing derivations:

$$\frac{\Gamma \vdash t : \Pi\,(x : A)\,.\,B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B[u/x]}$$

If we know that

- t normalises to $\lambda\, x\,.\,t'$
- u normalises to u'

we do not know how to get a normal form for $t\,u \equiv t'[u'/x]$

Reducibility Proofs

We cannot prove normalisation by a straightforward induction on the typing derivations:

$$\frac{\Gamma \vdash t : \Pi\,(x : A)\,.\,B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B[u/x]}$$

If we know that

- t normalises to $\lambda\,x\,.\,t'$
- u normalises to u'

we do not know how to get a normal form for $t\,u \equiv t'[u'/x]$

We need a stronger induction hypothesis.

Reducibility Proofs

Reducibility was designed by W. W. Tait to prove normalisation for Gödel's simply-typed system T.

The idea is to associate to every type A a predicate on terms $[\![A]\!]$, such that

$$[\![A \to B]\!]\ t := \forall x, [\![A]\!]\ x \to [\![B]\!]\ (t\ x)$$

Reducibility Proofs

Reducibility was designed by W. W. Tait to prove normalisation for Gödel's simply-typed system T.

The idea is to associate to every type A a predicate on terms $[\![A]\!]$, such that

$$[\![A \to B]\!]\ t := \forall x, [\![A]\!]\ x \to [\![B]\!]\ (t\ x)$$

Tait's method was subsequently extended to System F by Girard, with the introduction of reducibility candidates.
Nowadays, we have an extensive literature on reducibility proofs for all kinds of systems.

Taming Dependent Types

Handling dependent types, where computations occur inside of types too, involves quite a lot of bookkeeping.

That being said, there are ways to make this bookkeeping more manageable:

[1] Sterling. First steps in synthetic Tait computability

[2] Bocquet, Kaposi, Sattler. Relative induction principles for type theories

Taming Dependent Types

Handling dependent types, where computations occur inside of types too, involves quite a lot of bookkeeping.

That being said, there are ways to make this bookkeeping more manageable:

- conceptual frameworks that abstract away from the details of the proof such as Sterling's STC[1] or Bocquet, Kaposi and Sattler's relative induction principles[2]

[1] Sterling. First steps in synthetic Tait computability

[2] Bocquet, Kaposi, Sattler. Relative induction principles for type theories

Taming Dependent Types

Handling dependent types, where computations occur inside of types too, involves quite a lot of bookkeeping.

That being said, there are ways to make this bookkeeping more manageable:

- conceptual frameworks that abstract away from the details of the proof such as Sterling's STC[1] or Bocquet, Kaposi and Sattler's relative induction principles[2]
- or using a proof assistant to help you with verification and automation.

[1] Sterling. First steps in synthetic Tait computability

[2] Bocquet, Kaposi, Sattler. Relative induction principles for type theories

Type Theory in Type Theory

Abel, Öhman, Vezzosi. Decidability of Conversion for Type Theory in Type Theory (2018).

The authors build a reducibility proof for MLTT with one universe in the Agda proof assistant, without assuming any axiom.

They use their reducibility model to show that conversion is decidable, and as by-products they obtain subject reduction, injectivity of Π's, consistency, and canonicity.

Type Theory in Type Theory: Overview of the Proof

Abel, Öhman and Vezzosi build their reducibility model out of proof-irrelevant predicates.

This means that they cannot define a universe of types equipped with a reducibility structure. Instead, they define reducible types using induction-recursion:

$$
\begin{aligned}
&\Gamma \Vdash \mathsf{A} := \\
&\mid \Vdash_{\mathbb{N}} : \ (\Gamma \vdash \mathsf{A} \Rightarrow^* \mathbb{N}) \ \longrightarrow \ \Gamma \Vdash \mathsf{A} \\
&\mid \Vdash_{\mathsf{U}} : \ (\Gamma \vdash \mathsf{A} \Rightarrow^* \mathsf{U}) \ \longrightarrow \ \Gamma \Vdash \mathsf{A} \\
&\mid \Vdash_{\Pi} : \ (\Gamma \vdash \mathsf{A} \Rightarrow^* \Pi \mathsf{F} \mathsf{G}) \to (\Gamma \Vdash \mathsf{F}) \\
&\qquad\qquad \to ((\Gamma \Vdash \mathsf{a} : \mathsf{F}) \to \Gamma \Vdash \mathsf{G}[\mathsf{a}]) \ \longrightarrow \ \Gamma \Vdash \mathsf{A}
\end{aligned}
$$

with $\Gamma \Vdash \mathsf{a} : \mathsf{F}$ defined by recursion over a proof of $\Gamma \Vdash \mathsf{F}$.

Type Theory in Type Theory: Overview of the Proof

Of course, this is a simplification.

Type Theory in Type Theory: Overview of the Proof

Of course, this is a simplification.

```
record _⊩¹B⟨_⟩_ (Γ : Con Term ℓ) (W : BindingType) (A : Term ℓ) : Set where
  inductive
  constructor Bᵣ
  eta-equality
  field
    F : Term ℓ
    G : Term (1+ ℓ)
    D : Γ ⊢ A :⇒*: ⟦ W ⟧ F ▹ G
    ⊢F : Γ ⊢ F
    ⊢G : Γ ∙ F ⊢ G
    A≡A : Γ ⊢ ⟦ W ⟧ F ▹ G ≅ ⟦ W ⟧ F ▹ G
    [F] : ∀ {m} {ρ : Wk m ℓ} {Δ : Con Term m} → ρ ∷ Δ ⊆ Γ → ⊢ Δ → Δ ⊩¹ U.wk ρ F
    [G] : ∀ {m} {ρ : Wk m ℓ} {Δ : Con Term m} {a : Term m}
        → ([ρ] : ρ ∷ Δ ⊆ Γ) (⊢Δ : ⊢ Δ)
        → Δ ⊩¹ a ∷ U.wk ρ F / [F] [ρ] ⊢Δ
        → Δ ⊩¹ U.wk (lift ρ) G [ a ]
    G-ext : ∀ {m} {ρ : Wk m ℓ} {Δ : Con Term m} {a b}
          → ([ρ] : ρ ∷ Δ ⊆ Γ) (⊢Δ : ⊢ Δ)
          → ([a] : Δ ⊩¹ a ∷ U.wk ρ F / [F] [ρ] ⊢Δ)
          → ([b] : Δ ⊩¹ b ∷ U.wk ρ F / [F] [ρ] ⊢Δ)
          → Δ ⊩¹ a ≡ b ∷ U.wk ρ F / [F] [ρ] ⊢Δ
          → Δ ⊩¹ U.wk (lift ρ) G [ a ] ≡ U.wk (lift ρ) G [ b ] / [G] [ρ] ⊢Δ [a]
```

Type Theory in Type Theory: Overview of the Proof

Of course, this is a simplification.

```
record _⊩¹B⟨_⟩_ (Γ : Con Term ℓ) (W : BindingType) (A : Term ℓ) : Set where
  inductive
  constructor Bᵣ
  eta-equality
  field
    F : Term ℓ
    G : Term (1+ ℓ)
    D : Γ ⊢ A :⇒*: ⟦ W ⟧ F ▹ G
    ⊢F : Γ ⊢ F
    ⊢G : Γ ∙ F ⊢ G
    A≡A : Γ ⊢ ⟦ W ⟧ F ▹ G ≅ ⟦ W ⟧ F ▹ G
    [F] : ∀ {m} {ρ : Wk m ℓ} {Δ : Con Term m} → ρ ∷ Δ ⊆ Γ → ⊢ Δ → Δ ⊩¹ U.wk ρ F
    [G] : ∀ {m} {ρ : Wk m ℓ} {Δ : Con Term m} {a : Term m}
        → ([ρ] : ρ ∷ Δ ⊆ Γ) (⊢Δ : ⊢ Δ)
        → Δ ⊩¹ a ∷ U.wk ρ F / [F] [ρ] ⊢Δ
        → Δ ⊩¹ U.wk (lift ρ) G [ a ]
    G-ext : ∀ {m} {ρ : Wk m ℓ} {Δ : Con Term m} {a b}
          → ([ρ] : ρ ∷ Δ ⊆ Γ) (⊢Δ : ⊢ Δ)
          → ([a] : Δ ⊩¹ a ∷ U.wk ρ F / [F] [ρ] ⊢Δ)
          → ([b] : Δ ⊩¹ b ∷ U.wk ρ F / [F] [ρ] ⊢Δ)
          → Δ ⊩¹ a ≡ b ∷ U.wk ρ F / [F] [ρ] ⊢Δ
          → Δ ⊩¹ U.wk (lift ρ) G [ a ] ≡ U.wk (lift ρ) G [ b ] / [G] [ρ] ⊢Δ [a]
```

Because all of the abstraction is unrolled, the definitions have plenty of side conditions, they have to use PERs to simulate quotients, etc.

A Formalisation of Decidability of Type Checking for MLTT in Coq

Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, L.P.

https://github.com/CoqHott/logrel-coq

Goals

- More automation: autosubst, tactics
- More robustness: extending the proof should be as easy as possible
- Less assumptions: induction-recursion is not actually necessary for this proof
- Finish the proof of decidability: show decidability of typing using a bidirectional algorithm.

Getting Rid of Induction-Recursion

In Coq, we do not have induction-recursion, so we need to get rid of it.

[3] Hancock, McBride, Ghani, Malatesta, Altenkirch. Small induction recursion

Getting Rid of Induction-Recursion

In Coq, we do not have induction-recursion, so we need to get rid of it.

The good news is that there is a particular kind of induction-recursion that we can encode with ordinary inductive types: small induction-recursion [3].

Inductive A : Type_i :=
| c1 : A
| c2 : (x : A) → A

elim : A → B
elim c1 = f1
elim (c2 x) = f2(elim x)

[3] Hancock, McBride, Ghani, Malatesta, Altenkirch. Small induction recursion

Getting Rid of Induction-Recursion

In Coq, we do not have induction-recursion, so we need to get rid of it.

The good news is that there is a particular kind of induction-recursion that we can encode with ordinary inductive types: small induction-recursion [3].

Inductive A : Type_i :=
| c1 : A
| c2 : $(x : A) \to A$

elim : $A \to B$
elim c1 = f1
elim $(c2\ x) = f2(\mathrm{elim}\ x)$

Inductive A : $B \to \mathrm{Type}_i$:=
| c1 : A f1
| c2 : $(b : B)(x : A\ b) \to A\ (f2(b))$

[3] Hancock, McBride, Ghani, Malatesta, Altenkirch. Small induction recursion

Getting Rid of Induction-Recursion

The bad news is, the definition of Abel et al. is not small induction recursion:
we are trying to define the reducibility of types, which is a Type_0, in parallel with the reducibility of terms, a Type_0-valued predicate.

Getting Rid of Induction-Recursion

The bad news is, the definition of Abel et al. is not small induction recursion:
we are trying to define the reducibility of types, which is a Type_0, in parallel with the reducibility of terms, a Type_0-valued predicate.

Fortunately, we can manage this with a layering strategy:

- Reducibility of small terms is a Type_0-valued predicate
- Reducibility of small types is a Type_1-valued predicate
- Reducibility of large terms is a Type_1-valued predicate
- Reducibility of very large types is a Type_2-valued predicate
- ...

Getting Rid of Induction-Recursion

The bad news is, the definition of Abel et al. is not small induction recursion:
we are trying to define the reducibility of types, which is a Type_0, in parallel with the reducibility of terms, a Type_0-valued predicate.

Fortunately, we can manage this with a layering strategy:

- Reducibility of small terms is a Type_0-valued predicate
- Reducibility of small types is a Type_1-valued predicate
- Reducibility of large terms is a Type_1-valued predicate
- Reducibility of very large types is a Type_2-valued predicate
- ...

We use the universe polymorphism of Coq to deal (more or less) transparently with all these levels.

Getting Rid of Induction-Recursion

In the end, we get a reducibility model for MLTT_n in MLTT_{n+4}!

Getting Rid of Induction-Recursion

In the end, we get a reducibility model for MLTT_n in MLTT_{n+4}!

This is not only aesthetically pleasing, it is also very useful to prove conservativity results:

Getting Rid of Induction-Recursion

In the end, we get a reducibility model for MLTT_n in MLTT_{n+4}!

This is not only aesthetically pleasing, it is also very useful to prove conservativity results:

For instance, we can extend this reducibility model to $\mathsf{CC}^{\mathsf{obs}}$.
Thus, given any well-typed term f of type $\mathbb{N} \to \mathbb{N}$ in $\mathsf{CC}^{\mathsf{obs}}$, we get a proof in MLTT that f is reducible, or in other words

$$(\mathsf{n} : \Lambda) \to \Vdash \mathsf{n} : \mathbb{N} \to \Vdash (\mathsf{f}\ \mathsf{n}) : \mathbb{N}$$

Getting Rid of Induction-Recursion

In the end, we get a reducibility model for MLTT_n in MLTT_{n+4}!

This is not only aesthetically pleasing, it is also very useful to prove conservativity results:

For instance, we can extend this reducibility model to CC^{obs}.
Thus, given any well-typed term f of type $\mathbb{N} \to \mathbb{N}$ in CC^{obs}, we get a proof in MLTT that f is reducible, or in other words

$$(\mathsf{n} : \Lambda) \to \Vdash \mathsf{n} : \mathbb{N} \to \Vdash (\mathsf{f}\ \mathsf{n}) : \mathbb{N}$$

From there, we can define an integer function f′ in MLTT that produces a proof of $\Vdash \mathsf{n} : \mathbb{N}$ and feeds it to the reducibility proof, from which it can extract the value of (f n).

Interlude: Gödel's incompleteness theorem

Wait a second...

We showed that MLTT_{n+4} proves the consistency of MLTT_n.
Thus, the full theory MLTT proves the consistency of MLTT_n for all n.

Wait a second...

We showed that MLTT_{n+4} proves the consistency of MLTT_n.
Thus, the full theory MLTT proves the consistency of MLTT_n for all n.

But remark that any proof of False in MLTT can only mention a finite number of universes.

Wait a second...

We showed that MLTT_{n+4} proves the consistency of MLTT_n.
Thus, the full theory MLTT proves the consistency of MLTT_n for all n.

But remark that any proof of False in MLTT can only mention a finite number of universes.

Thus, a proof of False in MLTT must really be a proof of False in MLTT_n for some integer n – but we proved that these cannot exist.
We just proved that MLTT is consistent inside of MLTT?!

Wait, what?

Not so fast!

There is a catch:

- It is true that given an actual integer $n = 3, 6, 23...$ we know how to build a proof of consistency of $MLTT_n$.
- However, we cannot do it from an abstract integer n. In other words, we cannot prove $\Pi(n : \mathbb{N})$. consistent$(MLTT_n)$.

Not so fast!

There is a catch:

- It is true that given an actual integer $n = 3, 6, 23\ldots$ we know how to build a proof of consistency of MLTT_n.
- However, we cannot do it from an abstract integer n. In other words, we cannot prove $\Pi(n : \mathbb{N})\ .\ \mathsf{consistent}(\mathsf{MLTT}_n)$.

This is not too difficult to see:

We need n+4 universes to prove consistency of MLTT_n. Thus, we would need an infinite number of universes to prove $\Pi(n : \mathbb{N})\ .\ \mathsf{consistent}(\mathsf{MLTT}_n)$, but a proof term can only contain finitely many universes.

Not so fast!

There is a catch:

- It is true that given an actual integer $n = 3, 6, 23\ldots$ we know how to build a proof of consistency of MLTT_n.
- However, we cannot do it from an abstract integer n. In other words, we cannot prove $\Pi(n : \mathbb{N}) \,.\, \mathsf{consistent}(\mathsf{MLTT}_n)$.

This is not too difficult to see:

We need n+4 universes to prove consistency of MLTT_n. Thus, we would need an infinite number of universes to prove $\Pi(n : \mathbb{N}) \,.\, \mathsf{consistent}(\mathsf{MLTT}_n)$, but a proof term can only contain finitely many universes.

This means we cannot do our proof of consistency of MLTT after all. All is well!

Not so fast!

And actually, this sort of behaviour is already present in classical set theory: ZFC can prove the consistency of any finite fragment of ZFC.

But it cannot prove this uniformly (as long as ZFC is consistent!)

End of interlude

Bidirectional Type-Checking

Abel et al. only show decidability of conversion. While this is the most complicated part of a type-checking algorithm, going from there to the decidability of typing is non-trivial.

Indeed, Abel et al. use a theory without annotations on binders, for which conversion is decidable but typing is not.

Bidirectional Type-Checking

In our development, we show decidability of typing, by extending the conversion checking algorithm to a full account of algorithmic typing, defined in a bidirectional fashion.

Bidirectional Type-Checking

In our development, we show decidability of typing, by extending the conversion checking algorithm to a full account of algorithmic typing, defined in a bidirectional fashion.

In order to show the equivalence of the bidirectional presentation of MLTT with the declarative presentation, we actually do three logical relations in one:
our entire model is parameterized with a generic typing interface, which is instantiated three times.

Bidirectional Type-Checking

In our development, we show decidability of typing, by extending the conversion checking algorithm to a full account of algorithmic typing, defined in a bidirectional fashion.

In order to show the equivalence of the bidirectional presentation of MLTT with the declarative presentation, we actually do three logical relations in one:
our entire model is parameterized with a generic typing interface, which is instantiated three times.

- once with the declarative typing and declarative conversion
- once with the declarative typing and algorithmic conversion
- once with the bidirectional typing and algorithmic conversion.

Bidirectional Type-Checking

At each step, we use the reducibility model to more properties on the system:

the first pass gives us enough properties to instanciate the generic interface with the mixed system, and the second pass gives us enough to instanciate it with the fully algorithmic system.

Bidirectional Type-Checking

At each step, we use the reducibility model to more properties on the system:

the first pass gives us enough properties to instanciate the generic interface with the mixed system, and the second pass gives us enough to instanciate it with the fully algorithmic system.

From there, we get a complete proof of decidability of the type-checking, without having to duplicate the reducibility model.

Continuity of Functionals in Martin-Löf Type Theory

Martin Baillon, Assia Mahboubi, Pierre-Marie Pédrot

Constructive Math and Continuity

Usually in constructive mathematics, every function f that can be defined from the Cantor space $\mathbb{N} \to \mathbb{B}$ to the natural numbers $\mathbb{N}$ is uniformly continuous:
we only need finitely many digits of the input to compute the output.

Constructive Math and Continuity

Usually in constructive mathematics, every function f that can be defined from the Cantor space $\mathbb{N} \to \mathbb{B}$ to the natural numbers $\mathbb{N}$ is uniformly continuous:
we only need finitely many digits of the input to compute the output.

$$f(01101110000011110001\ldots) = 3$$
$$f(11110011100100110000\ldots) = 4$$
$$f(10111000111000010101\ldots) = 0$$
$$\ldots$$

Constructive Math and Continuity

Usually in constructive mathematics, every function f that can be defined from the Cantor space $\mathbb{N} \to \mathbb{B}$ to the natural numbers $\mathbb{N}$ is uniformly continuous:
we only need finitely many digits of the input to compute the output.

$$f(011011\,10000011110001\ldots) = 3$$
$$f(111100\,11100100110000\ldots) = 4$$
$$f(101110\,00111000010101\ldots) = 0$$
$$\ldots$$

Martin-Löf Type Theory

Martin-Löf Type Theory was originally designed as a framework for constructive mathematics.

Does it mean that all functions $(\mathbb{N} \to \mathbb{B}) \to \mathbb{N}$ are continuous in MLTT?

Martin-Löf Type Theory

Martin-Löf Type Theory was originally designed as a framework for constructive mathematics.
Does it mean that all functions $(\mathbb{N} \to \mathbb{B}) \to \mathbb{N}$ are continuous in MLTT?

The internal statement of continuity is not provable:
there is no term of type $\Pi\, \mathsf{f}\, .\, \Sigma\, \mathsf{n}\, .\, \mathsf{uniformly\_continuous}(\mathsf{f}, \mathsf{n})$.

Martin-Löf Type Theory

Martin-Löf Type Theory was originally designed as a framework for constructive mathematics.
Does it mean that all functions $(\mathbb{N} \to \mathbb{B}) \to \mathbb{N}$ are continuous in MLTT?

The internal statement of continuity is not provable:
there is no term of type Π f . Σ n . uniformly_continuous(f, n).

(indeed, this statement is false in the usual set-theoretic model)

External Continuity

Nevertheless, Coquand and Jaber used sheaf-valued logical relations to show that all functions from the Cantor space to $\mathbb{N}$ are uniformly continuous[4].

[4] Coquand and Jaber, A Note on Forcing and Type Theory

External Continuity

Nevertheless, Coquand and Jaber used sheaf-valued logical relations to show that all functions from the Cantor space to $\mathbb{N}$ are uniformly continuous[4].

Their proof is external: it is done by induction on typing derivations in some meta-theory.

[4] Coquand and Jaber, A Note on Forcing and Type Theory

Continuity of MLTT in MLTT

Martin Baillon, Assia Mahboubi and Pierre-Marie Pédrot are working on an extended version of this argument in our Coq framework for logical relations.

Continuity of MLTT in MLTT

Martin Baillon, Assia Mahboubi and Pierre-Marie Pédrot are working on an extended version of this argument in our Coq framework for logical relations.

- Their model supports large elimination of inductive types, which makes the logical relation more complex (as types needs to be sheafified too)

Continuity of MLTT in MLTT

Martin Baillon, Assia Mahboubi and Pierre-Marie Pédrot are working on an extended version of this argument in our Coq framework for logical relations.

- Their model supports large elimination of inductive types, which makes the logical relation more complex (as types needs to be sheafified too)
- The argument can be carried out in MLTT itself. This is as close as we can get to an internal proof of continuity: we get an inhabitant of

  $$\Pi\, f\,.\ (\mathsf{MLTT} \vdash f : \mathbb{B}^{\mathbb{N}} \to \mathbb{N}) \to \Sigma\, n\,.\ \mathsf{uniformly\_continuous}(f, n)$$

  but of course, we do not have internally that all functions are well-typed.

The Church-Turing Thesis in Martin-Löf Type Theory

Martin Baillon, Yannick Forster, Assia Mahboubi, Kenji Maillard, Pierre-Marie Pédrot, L.P.

Constructive Mathematics and Computability

Perhaps more fundamentally than continuity, constructive maths is supposed to enforce effective computability:
any constructively defined integer function f should come with some effective process that takes an integer n and outputs $f(n)$.

Constructive Mathematics and Computability

Perhaps more fundamentally than continuity, constructive maths is supposed to enforce effective computability:
any constructively defined integer function f should come with some effective process that takes an integer n and outputs $f(n)$.

According to the Church–Turing thesis, this is the same as saying that constructively defined functions can be computed by a Turing machine, or lambda-calculus.

In Martin-Löf Type Theory

As with continuity, the internal statement of computability is not provable

$$\Pi\ (\mathsf{f} : \mathbb{N} \to \mathbb{N})\ .\ \Sigma\ (\mathsf{t} : \Lambda)\ .\ \mathsf{computes\_function}(\mathsf{f}, \mathsf{t})$$

In Martin-Löf Type Theory

As with continuity, the internal statement of computability is not provable

$$\Pi\ (\mathsf{f} : \mathbb{N} \to \mathbb{N})\ .\ \Sigma\ (\mathsf{t} : \Lambda)\ .\ \mathsf{computes\_function}(\mathsf{f}, \mathsf{t})$$

On the other hand, we already proved the external statement of computability: our reducibility proof contains a notion of reduction, and we proved that it implies conversion and always terminates.

In fact, this proof extends the computability to all types (not only integer functions), and even to open terms.

Internalising Computability

Take one step further: can we add a computability axiom to MLTT? It might not be provable, but is is consistent to postulate it?

$$\Pi\ (\mathsf{f} : \mathbb{N} \to \mathbb{N})\ .\ \Sigma\ (\mathsf{t} : \Lambda)\ .\ \mathsf{computes\_function}(\mathsf{f}, \mathsf{t})$$

This axiom can be separated into two components:

$\mathsf{quote} : (\mathsf{f} : \mathbb{N} \to \mathbb{N}) \to \Lambda$

$\mathsf{eval} : (\mathsf{f} : \mathbb{N} \to \mathbb{N})(\mathsf{n} : \mathbb{N}) \to \mathsf{computes\_to}((\mathsf{quote}\ \mathsf{f})\ @\ \lceil \mathsf{n} \rceil, \lceil \mathsf{f}\ \mathsf{n} \rceil)$

In other words, we can recover the code of any integer function, and it should compute said function.

¬funext

Quote and eval are incompatible with the extensionality of functions.

$\neg$funext

Quote and eval are incompatible with the extensionality of functions.

Proof:

- funext implies that two extensionally equal functions have equal codes
- since the equality between codes is decidable, we can use quote to decide whether an integer function has the same code as the zero function
- thus, we can decide whether an integer function is identically zero
- this implies that we can decide the halting problem with an integer function
- and then, we can recover the code of this integer function, which is a program that decides the halting problem. Contradiction.

A Strategy to Prove Consistency?

In fact, it is not completely clear how to prove that quote and eval are not inconsistent.

A Strategy to Prove Consistency?

In fact, it is not completely clear how to prove that quote and eval are not inconsistent.

Possible lead: equip them with a reduction strategy, and use a reducibility model to show normalisation and thus consistency.

A Strategy to Prove Consistency?

In fact, it is not completely clear how to prove that quote and eval are not inconsistent.

Possible lead: equip them with a reduction strategy, and use a reducibility model to show normalisation and thus consistency.

This intuitively makes sense, because normalisation models are quite close in spirit to realisability models, except that they do not enforce funext:

while two functions f, g are equal in the model when they send equal inputs to equal outputs, the presence of neutral terms means that equality of f and g in the model implies that they have the same normal form, i.e. the same code.

Thank you!