Brazilian type checking

Andrej Bauer
University of Ljubljana

Christopher Stone
Harvey Mudd College

Formalization of mathematics in proof assistants
Institut Henri Poincare
May 2014

1. Thanks to organizers for the invitation. This is promising to be a very exciting meeting.
2. The work I will present is being done jointly with Chris Stone, who is sitting somewhere in the audience. Hi Chris!

1. Brazil
2. TT

1. My talk is about the *design* of a proof checker, called "Brazil" for reasons that will become apparent. We have a working prototype which changes every day.
2. We are also working on an extension of Brazil, called "tt". This is a more ambitious piece of work. You can think of it as a special-purpose programming language for implementation of unification algorithms, tactics, search procedures, etc. Alternatively, you can think of it as a trusted kernel on top of which we can build ever more complex tools for formalization.
3. As tt is more experimental than Brazil, I will focus on Brazil and discuss tt only briefly at the end. But please talk to Chris and me, and we'll show you both prototypes in action. Even better, you can participate on GitHub.

1. Brazil

1. Brazil is based on a type theory proposed by VV, so in this sense it is closely related to HoTT. However, I think some ideas are relevant also to other kinds of foundation. VV's type theory is quite general and is likely able to accommodate special cases, such as classical mathematics.
2. Various aspects of Brazil in TT, in particular the programming language ideas, are independent of the foundation altogether. So you may be able to take some ideas from us and use them for your work. We'd be delighted.

Strict equality:

$$\frac{A\ \mathsf{type} \qquad a : A \qquad b : A}{\mathsf{Id}_A(a,b)\ \mathsf{type}} \qquad \frac{a : A}{\mathsf{refl}\, a : \mathsf{Id}_A(a,a)}$$

Paths:

$$\frac{A\ \mathsf{type} \qquad a : A \qquad b : A}{\mathsf{Paths}_A(a,b)\ \mathsf{type}} \qquad \frac{a : A}{\mathsf{idpath}\, a : \mathsf{Paths}_A(a,a)}$$

1. To motivate the setup, recall that there are *two* ways to introduce equality in type theory. They go by many different names. Let me call them *strict equality* and *paths*, because I want to use your topological intuitions.
2. If you are familiar with type theory you will recognize these by their elimination rules on the next slide. If not, think of the strict equality as the usual equality, presented as a set. If a and b are equal then $\mathsf{Id}_A(a,b)$ contains the element $\mathsf{refl}\, a$, and nothing else. If not, then $\mathsf{Id}_A(a,b)$ is empty.
3. Paths is the type of *paths* between a and b in a space A. Under this view, all types are "some sort of spaces". Here $\mathsf{idpath}\, a$ is the constant path at point a.
4. I am not going to give you pages upon pages of inference rules, you can look them up on our GitHub project.

Equality elimination:

$$\frac{\begin{array}{c} x : A \vdash P(x)\ \mathsf{type} \\ a : A \qquad b : A \qquad e : \mathsf{Id}_A(a,b) \qquad u : P(a) \end{array}}{u : P(b)}$$

Path elimination:

$$\frac{\begin{array}{c} x : A \vdash P(x)\ \mathsf{type} \\ a : A \qquad b : A \qquad p : \mathsf{Paths}_A(a,b) \qquad u : P(a) \end{array}}{\mathsf{transport}_A^{x.P}(a,b,p,u) : P(b)}$$

1. Here are the somwhat simplified versions of elimination rules. The fully dependent ones are more complex, and need not be considered here.
2. The strict version is just replacement of equals for equals. Since a and b are equal, we may use a in place of b.
3. The path version can be explained as follows: a and b may not be equal, but there is a path p between them. We can *transport* u along p from the fiber $P(a)$ to the fiber $P(b)$. We think of P as a topological fibration, i.e., a family of spaces indexed by A, with a path-lifting property. This is at the core of the HoTT.
4. The homotopical way of doing things has many benefits, among others the Univalence axiom. But it can also lead to complications.

Equality elimination:

$$\frac{\begin{array}{c} x : A \vdash P(x)\ \mathsf{type} \\ a : A \qquad b : A \qquad e : \mathsf{Id}_A(a,b) \qquad u : P(a) \end{array}}{u : P(b)}$$

Path elimination:

$$\frac{\begin{array}{c} x : A \vdash P(x)\ \mathsf{type} \\ a : A \qquad b : A \qquad p : \mathsf{Paths}_A(a,b) \qquad u : P(a) \end{array}}{\mathsf{transport}_A^{x.P}(a,b,p,u) : P(b)}$$

1. One such complication is the construction of a model of HoTT inside HoTT. (Note that there is no problem with Gödelian phenomena because the inner model has fewer universe.)
2. We would like to perform the construction of semisimplicial types to create a model. However, even though the construction seems intuitively clear, using paths and transport makes everything horribly complicated because the type theory makes us write down all the transports, even though they are all trivial.
3. VV therefore proposed to have *both* equality types at the same time. Then we can use the strict equality where appropriate, to keep the construction manageable.

Fibered types:

$$A \text{ type} \qquad A \text{ fib} \qquad \frac{A \text{ fib}}{A \text{ type}}$$

Equality:

$$\frac{A \text{ type} \qquad a : A \qquad b : A}{\mathsf{Id}_A(a, b) \text{ type}}$$

$$\frac{A \text{ fib} \qquad a : A \qquad b : A}{\mathsf{Paths}_A(a, b) \text{ fib}}$$

1. We cannot just throw in both equalities – it would turn out that they coincide, and that would then contradict Univalence.
2. Instead, we should distinguish some of the types as *fibered*. These behave nicely with respect to paths: they have the path lifting property necessary to interpret transport.
3. Paths can only be formed and used on fibered types. Since strict equality is not fibered, we cannot show that path equality implies strict equality.
4. There are other complications. Let me show how crazy things get.

$$e : \mathsf{Id}_{\mathsf{U}}(\mathtt{nat} \to \mathtt{bool}, \mathtt{nat} \to \mathtt{nat}) \vdash (\lambda n : \mathtt{nat}\,.\, n + 3)\,\mathsf{42} : \mathtt{bool}$$

1. We may *assume* that the Baire space and the Cantor space are equal. Therefore $\lambda n : \mathtt{nat}\,.\, n + 3$ is map from `nat` to `bool`, so the application is a `bool`.
2. But we *cannot* β-reduce because on a careful reading of the β-rule we discover that it does not apply in this case. This destroys any hope for having normal forms.
3. This and similar examples force us to equip each application with explicit typing information. Luckily, all such annotations can be derived because they are unique, if they exist. Nevertheless, we need them.
4. By the way, the assumption that the Baire space and Cantor space are equal is *not* inconsistent. As homework, you should find *two* models which validate the assumption.

$$e : \mathsf{Id}_{\mathsf{U}}(D, D \to D)$$
$$\vdash (\lambda x : D \,.\, x\, x)(\lambda x : D \,.\, x\, x) : D$$

1. Worse, we may assume that a type is equal to its own function space.
2. The "paradoxical" λ-term which does not normalize becomes well-typed.
3. Therefore, it is going to be easy to get the proof checker into an infinite loop, as long as any sort of normalization is built in.
4. Is this a problem? Maybe theoretically, but certainly not in practice. The typical user is going to hit Ctrl-C well before the Sun becomes a red giant.

$$\frac{x : A \vdash P(x)\ \mathsf{type} \qquad a : A \qquad b : A \qquad e : \mathsf{Id}_A(a,b) \qquad u : P(a)}{u : P(b)}$$

1. We would like to send formalized mathematics to people in a faraway country, such as Brazil. They will check our development independently. [VV tried Bolivia but that didn't work.]
2. But how will they deal with strict equality elimination? If we send them the proof object u and $P(b)$, how can they ever guess what a and e are?
3. In fact, type checking in the presence of strict equality is undecidable. Without some divine help, Brazilians are doomed.
4. By the way, with path equality there is no problem because transport records a and the path from a to b.
5. Thus, we must also send to Brazil *additional hints* about equality, which however are not part of the proof object proper. These are *extra*.

1. Type theory with strict equality *and* paths.
2. A trusted proof checker.
3. *Practical* support for equality hints.
4. Sacrifice termination and completeness.
5. Preserve *soundness* guarantee.

1. Let us summarize what we want.
2. By "practical" we mean that hints should be used infrequently and need not be very large.
3. The soundness guarantee is: *if* the checker accepts a proof object (with hints), then there is a derivation that the proof object has the given type.
4. By now it is obvious that our proof checker is called "Brazil".

Contexts:

$$\begin{array}{rcll} \Gamma & ::= & \bullet & \text{empty context} \\ & | & \Gamma,\, x : T & \text{context extended with } x : T \end{array}$$

Equality hints:

$$\begin{array}{rcll} \mathcal{H} & ::= & \circ & \text{empty hints} \\ & | & \mathcal{H}, (e_1 \equiv_T e_2) & \text{extend hints with an equation} \\ & | & \mathcal{H}, (e_1 \rightsquigarrow_T e_2) & \text{extend hints with a rewrite} \end{array}$$

1. The typing contexts are as usual.
2. We also carry around a list of equality hints. These are of two kinds: ordinary equations and rewrite rules (directed equations) to be used during weak head normalization.

Types:

$$
\begin{array}{rll}
T, U ::= & \mathbb{U}_\alpha & \text{universe} \\
\mid & \mathsf{El}^{\alpha}\, e & \text{type named by } e \\
\mid & \mathsf{Unit} & \text{the unit type} \\
\mid & \prod_{(x:T)} U & \text{product} \\
\mid & \mathsf{Paths}_T(e_1, e_2) & \text{path type} \\
\mid & \mathsf{Id}_T(e_1, e_2) & \text{equality type}
\end{array}
$$

1. We use Tarski-style universes. With Russell-style universes and cummulativity it is not clear at all how to perform equality checks (at what universe – it matters!). I shall not say more about this, except that contrary to folk opinion, Tarski universes are *not* annoying to implement. And they are completely hidden from the user.
2. The magenta bits are type annotations. They must be present, but luckily it is easier to derive them than to require that they be provided. So, the magenta bits are not there in the concrete syntax. They get added during type checking and synthesis, we will see an example.

Terms:

$e ::=$	x	variable
	$\mathsf{equation}\ e_1 : e_2 \equiv_T e_3\ \mathsf{in}\ e_4$	use equality hint e_1 in e_4
	$\mathsf{rewrite}\ e_1 : e_2 \equiv_T e_3\ \mathsf{in}\ e_4$	use rewrite hint e_1 in e_4
	$e :: T$	ascribe type T to term e
	$\lambda x{:}T_1.T_2\,.\,e$	λ-abstraction
	$e_1\ @^{x:T_1.T_2}\ e_2$	application
	$\star$	the element of unit type
	$\mathsf{idpath}_T\ e$	identity path
	$\mathsf{J}_T([x\,y\,p\,.\,U], [z\,.\,e_1], e_2, e_3, e_4)$	path eliminator
	$\mathsf{refl}_T\ e$	reflexivity
	$\vdots$	$\vdots$

1. Here is the abstract syntax of terms, without universe names
2. Again, type annotations in magenta are derived automatically. I emphasize that this is *simpler* than having them provided and having to check that they are correct.
3. The "equation" and "rewrite" constructs are how hints are added to the system. This is a *local* construct, so you can tell Brazilians to use a specific hint only in a specific term.
4. When terms are compared for syntactic equality, the hints constructs are ignored. The hints are *not* part of the proof object proper.

Synthesize the type of e:

$$\Gamma \vdash e \Rightarrow T$$

Check that e has the given type T:

$$\Gamma \vdash e \Leftarrow T$$

Passage from synthesis to checking:

$$\frac{\Gamma \vdash e \Rightarrow T \qquad U \equiv T}{\Gamma \vdash e \Leftarrow U}$$

1. I cannot here explain all the technical details of what we did, but here is a broad outline of how we formulated VV's type theory in an algorithmic way.
2. For the experts: we use standard synthesis/checking distinction, type-directed equality checking, and we consult an equality hints database to overcome limitations of the equality checking algorithm.
3. Let me briefly comment on what that means. We split the typing judgment into two – synthesis and checking. Whether we should use synthesis or checking depends on the shape of the term e.
4. This way the type checking algorithm can be organized in such a way that type equality is invoked in precisely one rule. We thus have good control over where equality checking happens.

Application (traditional):

$$\textsc{term-app}\quad \frac{\Gamma \vdash e_1 \,:\, \prod_{(x:T)} U \qquad \Gamma \vdash e_2 \,:\, T}{\Gamma \vdash e_1 \,@^{x:T.U}\, e_2 \,:\, U[e_2/x]}$$

Application (synthesis & checking):

$$\textsc{syn-app}\quad \frac{\Gamma\,;\,\mathcal{H} \vdash e_1 \Rightarrow T_1 \qquad \Gamma\,;\,\mathcal{H} \vdash T_1 \rightsquigarrow^* \prod_{(x:T)} U \not\rightsquigarrow \qquad \Gamma\,;\,\mathcal{H} \vdash e_2 \Leftarrow T}{\Gamma\,;\,\mathcal{H} \vdash e_1 \,@^{x:T.U}\, e_2 \Rightarrow U[e_2/x]}$$

1. For example, consider the traditional application rule and how it is done by synthesis and checking.
2. Application is a synthesizing form. Notice that we need to weak head normalize in order to verify that e_1 has the correct type. Also, the magenta annotations are read off the synthesized parts. This is the case with all magenta annotations.

Install a new hint:

$$\textsc{chk-equation-hint}\quad \frac{\Gamma\,;\,\mathcal{H} \vdash e_1 \Rightarrow U' \qquad \Gamma\,;\,\mathcal{H} \vdash U' \rightsquigarrow^* \mathsf{Id}_U(e_2, e_3) \not\rightsquigarrow \qquad \Gamma\,;\,(\mathcal{H}, (e_2 \equiv_U e_3)) \vdash e_4 \Leftarrow T}{\Gamma\,;\,\mathcal{H} \vdash (\mathsf{equation}\ e_1 : e_2 \equiv_U e_3\ \mathsf{in}\ e_4) \Leftarrow T}$$

Using a hint:

$$\textsc{chk-eq-hint}\quad \frac{(e_1 \equiv_T e_2) \in \mathcal{H}}{\Gamma\,;\,\mathcal{H} \vdash e_1 \approx e_2 \Leftarrow T}$$

1. Next, equality checking is done by an adaptation of an algorithm developed by Bob Harper and Chris Stone. The algorithm has function extensionality and various η-rules built in.
2. Our version consults the hints. For example, to check that e_1 and e_2 are equal at type T, we first check whether they are syntactically equal, then we consult the hints, then we proceed with the type-directed equality algorithm (which calls itself recursively at a smaller type).
3. At a base type the algorithm uses weak-head normalization to compare terms. There we use rewrite hints.

Symmetry of equality:

```
Definition sym :=
fun (A : Type) (a b : A) (p : a == b) =>
  equation p in (refl a :: (b == a)).
```

Strict equality eliminator:

```
Definition G :=
fun (A : Universe 0)
  (P : forall (x y : A), x == y -> Universe 0)
  (r : forall z : A, P z z (refl z))
  (x y : A) (p : x == y)
  =>
    equation p in (r x :: P x y p).
```


1. The basic facts about equality can be derived using hints.

```
Definition Type := Universe f0.

Parameter bool : Type.
Parameter true false : bool.

Parameter bool_ind :
  forall (P : bool -> Type) (b : bool),
    P true -> P false -> P b.

Parameter bool_ind_true :
  forall (P : bool -> Type)
         (x : P true) (y : P false),
    bool_ind P true x y == x.

Rewrite bool_ind_true.
```


1. Here we have an example using rewrites. Essentially, we can *define* booleans with the usual conversion rules. Of course this idea is not limited just to Booleans.
2. One thing to note: we are installing a *universal* hint.

1. Brazil

1. Brazil is an *experiment*. One may worry about various parts of it: Do the type annotations get large? Is it complete and in what sense? How do we incorporate universe management?
2. These are all valid concerns, and will have to be addressed. We are happy to discuss them with you.

2. TT

1. In conclusion let me say a few words about TT.
2. We heard yesterday that there is a distinction between *functional* or *declarative* style of programming, and *procedural* programming.
3. In the theory of programming languages procedural programming is just one case of a more general concept known as *computational effect*. All sort of things are computational effects: state, I/O, non-determinism, exceptions, etc. The typical operations of a proof assistant may be viewed as computational effects: unification, context manipulation, backtracking, etc.
4. In fact, Brazilian equality hints are a very simple form of computatinal effect, too.

2. TT

1. We are developing a language based on a general theory of so-called algebraic effects and handlers that is tailored to the needs of a proof assistant. The handlers are a powerful programming concepts, a generalization of exception handlers and delimited continuations. They allow us to write purely functional programs in a procedural style (and more).
2. For example, it will be possible in TT to implement meta-variables and unification as a *derived* concept. At the same time, tt will have a soundness guarantee: it will refuse to generate invalid Brazilian terms.
3. But it is too early to tell how successful the experiment will be. Ask us again on Friday, Chris will be done by then.