The Egal Manual

Chad E. Brown

September 30, 2014

Contents

1 Introduction **9**
1.1 Example 1: Socrates is Mortal . . . 10
1.2 Example 2: Nellie is Pink . . . 14
1.3 Example 3: Proof Scripts . . . 16

2 The Framework: Simple Type Theory **19**
2.1 Types . . . 19
2.2 Names . . . 20
2.3 Terms . . . 20
2.4 Proofs . . . 21
2.5 Contexts and Signatures . . . 22
2.6 Substitution and Conversion . . . 23
2.7 Nameless Representations and Global Identifiers . . . 24
2.8 Importing Previous Work (Databases and Signatures) . . . 29
2.9 Rules for Checking . . . 34

3 Syntax, Parsing and Printing **39**
3.1 Quick Guide to the Code . . . 41
3.2 Mathematical Expression Trees . . . 46
3.3 Tokens . . . 52
3.4 There Is No Grammar . . . 52
3.5 Semantic Parsing Relations . . . 54
3.6 Parser . . . 58
3.7 Correctness . . . 61
3.8 Conversion to Nameless Representation . . . 61

4 Documents **63**
4.1 Document Items . . . 63
4.2 Proof Tactics . . . 67

5 The Foundation: Tarski-Grothendieck Set Theory **71**

6 Examples: The 2014 Treasure Hunt 81
6.1 Basic Logic 81
6.2 Basic Higher Order Results 88
6.3 Zermelo's Well Ordering Theorem 94
6.4 Set Theory 95
6.5 Pairs and Functions 98

7 Example: Existence of God 101

8 Category Theory: The Final Treasures 107

9 Conclusion 113

Preface

Egal is the result of a failed project to support distributed formalization of mathematics. The code is being released as free and open source in September, 2014. I had hoped to write more documentation and, especially, to write the documentation more clearly. Unfortunately, I no longer have time or energy to devote to this project. As a result, this manual was hastily written and has barely been proofread. I'll simply ask readers (if there are any) to forgive me for writing informally and occasional lapses into pessimism and negativity. And *Doctor Who* references. I've found it impossible to write this in a traditional academic fashion, so instead I am exorcising demons. Nevertheless, I guarantee that both this manual and the Egal source code are worth every cent you paid for them.

The Egal proof checker is in many ways the culmination of two decades of my life. Much of the actual code was written during Winter 2013-2014. However, most of my life choices for the last 20 years have been the result of following a dream: the dream of formalizing mathematics using a computer program. I first became interested in this idea as a graduate student in mathematics in the early 1990's. By 1995 I decided to openly work in this area. The first theorem prover I looked at was McCune's first-order prover *Otter*. Quaife had formulated the axioms for von Neumann-Gödel-Bernays Set Theory in *Otter* and one could in principle use this as a formal foundation for mathematics [40]. I found working with the first-order Gödel set constructors too difficult (though Belinfante has done some work to make it more convenient since then). In 1997 I started working with Peter Andrews on his theorem prover Tps [5]. I found Church's simple type theory [16, 3] to be a far more realistic framework for formalizing mathematics. On the other hand, Tps was designed as an interactive and automated theorem prover, not as a tool for building a mathematical library. Other tools (mentioned below) were more appropriate for building a formal library of mathematics. I certainly learned a great deal from my time working on Tps, and I am grateful for that.

I'm obviously not the first person to work on formalizing mathematics. The underlying logical ideas go back to Frege [19, 20], Russell and Whitehead [47], Hilbert and Ackermann [26], Church [16], and so on. De Bruijn's AUTOMATH [13, 15] in the late 1960's was the first real proof checker. Since then there have been a number of successful theorem provers and proof checkers including Mizar [32], LCF [38], Isabelle [36], NuPRL [17], Coq [33] and members of the HOL family of provers [21, 43, 24]. I've used many of these and found things I liked and things I didn't. I liked the mathemat-

ical language of Mizar, but did not like that there were not independently checkable proof terms (something sometimes called the "de Bruijn criteria"). Similar objections apply theorem provers using the alternative LCF-approach (where correctness is guaranteed by abstraction in the metalanguage). In addition, I was never happy with the zoo of allegedly "simple" types one finds in Isabelle-HOL or the HOL provers. My work on higher-order automated theorem proving (first with TPS [5] and later with Satallax [12]) led me to having some prejudice against both polymorphism and type definitions.

While in the 1990s my preferences changed from first-order set theory to simple type theory, by the middle of the 2000s I was very dissatisfied with simple type theory as a foundation for a formalized library of mathematics. Mathematicians use set theory, and there are good reasons. At some point I read Wiedijk's wonderful paper *Is ZF a hack?* [48]. After having suffered through the complexities of Gödel-Bernays, it was a surprise to see how simple Zermelo-Fraenkel set theory is, if one writes it in an appropriate logical framework. This led me to develop Scunak [10], a program that supported set theory, dependent types, and proof irrelevance. Nothing came from the Scunak project, except a few publications and more experience. It also led me to read about work by Gordon [22, 2] about a combination of higher-order logic and set theory. Similar work was also done later by Obua [37]. Eventually I gravitated towards this approach: simple type theory as a framework and set theory as a foundation.

A modern theorem prover that (for the most part) satisfies the de Bruijn criteria is Coq. As part of my job in the past five years, I found I was using Coq quite a bit. Coq has an even larger zoo of types, but it at least includes the simple types and thus gives a language in which one can express set theory. For a long time I was a fan of dependent types (as Scunak witnesses), but dependent types lead to a number of annoying issues having to do with equality. In Coq, one can look up so-called "John Major equality" to get an idea of what I mean. In the end I decided that dependent types could be moved to the theory level. That is, dependent products and sums can be represented as sets and the typing rules could be lemmas. The work of Aczel [1], Werner [46], Barras [8] and others influenced me in this regard. My thinking was that what is "type checking" in Coq could still easily be automated if there were something like these types as predicates or sets. As a project stemming from these ideas a student Jonas Kaiser worked with me to formalize some of the semantics of type theory in a formal set theory in Coq [29]. Originally the idea was to do the development in a simply typed fragment of Coq. Due to some objections from people with more power than I had, it was decided that the project would not be restricted to the simply typed fragment of Coq. I ended up giving in, largely because I knew I could simply work on a simply typed version on my own, during my own time, without resistance.

Aside from my loss of faith in dependent types, another problem I had with Coq was the hidden implicit indices on type universes. Without making the indices explicit, it is easy to write a proposition for which one can prove P and its negation (see the Girard paradox [18] or its simplification as the Hurkens paradox [28]). This does not lead to a contradiction since if one tries to combine the two proofs to prove false, then

Coq signals a universe inconsistency. (On a surface level, it appears that modus ponens is unsound in Coq.)

It would probably be easy to make an alternative to Coq with explicit universe indices. Some aspects of this would still make me uncomfortable. Consider ordered pairs (x, y). Mathematically I tend to think this is an object which depends on two objects: x and y. Coq adherents have informed me on multiple occasions that this is not true. In fact it depends on the type of x, the type of y and then x and y. So the pairing constructor has four arguments, not two. If we make type indices explicit, the pairing operator will depend on five or six arguments: the levels of universes where the two types live (optionally one level for both or one level for each), then two types, then an object of each type.

I would sometimes bring up these kinds of objections to people, but the common response I'd get is: *Who cares?* I eventually started responding: *I care!* But it became clear I was the only one who did. In addition, after bringing up the same objection multiple times, I decided no one was even listening. One reaction to not being listened to is to shout. Another reaction is to shut down and stop talking. I can remember times in the past 10 years when I've done both, though not at the same time, obviously. I generally consider arguing to be a waste of time, even though I wasted plenty of time arguing. I considered a better strategy to be to just do what I wanted and then show my way is better. Or, as happened in this case, my way goes the way of Franz Reichelt.

Consistent with this strategy, I worked on my own at home on the code that ended up becoming Egal. I worked on it during vacations when I didn't need to argue with people about how I wanted to do things. (I used to get 6 weeks of vacation a year! What else was I going to do with that time?) I remember a horrible period in 2009 when I was trying to code the proof checker in Java to try to support proving in a web browser. What a nightmare. Eventually it was clear that 6 weeks a year weren't enough and I decided to take off roughly half the year (without pay) to just do what I wanted to do. That's what I did for 13 months of the past 2 years.

I wanted a theorem prover where I could specify a proposition P. Then anyone could submit a potential proof term D and a proof checker could check if D proves P. After we know P has been proven, the particular proof should not matter. People should be able to work on things in a distributed way and correctness and originality should be the only things that mattered. Correctness and originality should both be automatically checkable. To check originality one needs to be able to check if a proposition has been proven before.

I spent a significant amount of time during Winter 2012-2013 designing a protocol for multiple servers to agree on a "global identifier" for a (closed) proposition P (or, more generally, a closed term). I had the idea that these global identifiers could stand for the definitions which get reused. If two different people independently defined the same term (up to the names involved), then they would end up with the same "global identifier." My thought was that this would remove obvious duplication and could make it clear when someone had already made a definition or proven a theorem. Of course, this is quite brittle because someone could make trivial changes (e.g., commuting

conjuncts) which would yield different, but equivalent, definitions. Nevertheless, it seemed better than nothing.

After learning about Bitcoin in Spring 2013 I realized I had wasted a lot of time. There was an obvious way to get "global identifiers": use a cryptographic hash function on a serialization of a nameless representation of the term. I know this is obvious in retrospect, but I simply didn't know about cryptographic hash functions. I also realized I could use the same techinique on proof terms and then use the hash as a private key for bitcoin treasures. For the first time I saw a possible way I could both live my dream of formalizing mathematics and earn enough money to support myself. I could prove theorems, bury some bitcoins underneath the proof I had, and sell access to the unproven propositions. Why would anyone buy access? If they proved the theorem with the same proof I had, then they could pick up the corresponding bitcoin treasure. After doing it for the bitcoins, they'd keep doing it because it's fun. I hoped it would be like an educational lottery/video game. A set of problems would be released and around 50 people or so would buy access to the problems. Some of those 50 people would be able to get to the treasures before others and come out ahead, but everyone would have an inexpensive learning experience. And I would get to do what I want: stay home and prove theorems. On my own terms.

Of course, this isn't what happened at all. There was very little interest in the treasure hunt. (You can look in the bitcoin block chain from late March 2014 until late September 2014 to see the details.) I'm agnostic about the idea of open sourcing code. The main reason I'm releasing this code is because I don't see a reason not to release it. As a classically minded mathematician, not not releasing the code implies releasing the code.

I can't shake the feeling that the past 20 years of my life were wasted. Maybe if I release the code and explain how to use it someone else will make something useful or profitable out of it. It's open source. Do what you want with it. You don't need to ask my permission. In fact, I can't imagine I'll even want to talk about this project for a long time.

There's a wonderful scene in the third episode of *The Ribos Operation* (by Robert Holmes) in which a young conman from the stars on a primitive planet meets a down-on-his-luck old man who is derided as "Binro the Heretic." The conman asked Binro about his heresy and Binro explains his belief that the stars in the sky are suns and that these other suns might support life on other worlds. After listening, the conman told him he believed this as well, and continued:

I know it for a fact. You see I come from one of those other worlds. I thought I should tell you, because one day, even here, in the future, men will turn to each other and say: Binro was right.

In that spirit, there's one last thing I should to say to all those people who thought I was crazy, to all those people who dismissed my ideas over these many years:

You were right.

I was wrong.

Chapter 1

Introduction

If I were to tell you that the next thing I say will be true,
but that the last thing I said was a lie, would you believe me?
The Doctor, *The Green Death*, Episode Five (1973)

Egal is a proof checker for formalized mathematics based on simply typed set theory. It is designed to allow the distributed construction of a library of formalized mathematics. A description of a vision of and several motivations for a library of formalized mathematics constructed by a variety of participants was published as The QED Manifesto in 1994 [6]. Other motivations and information can be found in [49, 7]. I had planned to write more, but it all seems kind of pointless now.

In this document we will explain how to use Egal and work through a number of examples. The examples will also demonstrate how one can set up a treasure hunt. We will start in this introductory chapter with some very simple examples. The examples are intended to help readers get a quick start using Egal before needing to understand the full framework (simple type theory/higher-order logic) and the mathematical foundation (set theory). In Chapter 2 we will describe the framework. Chapter 3 explains the syntax for types, terms and proof terms as part of a fairly detailed look at a verified parser. Chapter 4 describes the structure of documents. In Chapter 5 we will describe the mathematical foundation given via the document `Foundation.mg`. Chapter 6 gives an overview of the 25 documents which were part of the 2014 mathgate.info bitcoin theorem proving treasure hunt. These documents construct many of the fundamental building blocks of mathematics. Chapter 7 gives information about a formalization of Gödel's proof of the existence of God in a modal logic. Chapter 8 provides information about a document about Category Theory. The Category Theory proofs are omitted and there is a final collection of treasures. Unlike the 2014 treasure hunt, in this case there is no deadline. I will never release the solutions.[1]

[1] In a sense it shouldn't matter if I never release the solutions. Voters have affirmed the idea that successful businesses were not built by the people who built them, but emerged as a byproduct of government infrastructure. In that same spirit, one could say about each theorem I proved: *I didn't prove that.* The proof emerged as a byproduct of government infrastructure. And now I stand back and allow others to be the vessels through which government infrastructure can produce proofs.

1.1 Example 1: Socrates is Mortal

For a first simple example, we will demonstrate Egal with the well-known argument concluding the mortality of Socrates. The code for this example can be found in a sequence of files named `Socrates`i`.mg` in the `formaldocs` subdirectory. The syntax Egal uses is very similar to Coq [33], so familiarity with Coq may be helpful.

We start by opening a section. Generally you do not need to open a section, but if you are using type variables you will need to open a section.

```
Section Socrates.
```

We then declare a simple type variable A.

```
Variable A:SType.
```

This type variable can be used until we end the section. I hate type variables and was reluctant to include them at all. I have only included them with tight restrictions (e.g., at most 3 type variables can be used at once). I have also purposefully made it annoying to use them in order to discourage their use. For example, types must always be explicitly given as arguments.

We next declare two predicates on A: Man and Mortal. To be more precise, we declare two variables both of which have the function type from A to prop. prop is the type of propositions.

```
Variable Man:A->prop.
Variable Mortal:A->prop.
```

Now if we have a term of type A we can obtain a proposition by either applying Man or Mortal to that term. At the moment we have no terms of type A, but we might have a local variable of type A. In fact, we will use such a local variable to represent the proposition that all men are mortal as follows:

```
forall x:A, Man x -> Mortal x.
```

Here we have written a proposition which is the ASCII representation of $\forall x : A.\mathsf{Man}\ x \rightarrow \mathsf{Mortal}\ x$. In words, for every x of type A, if x is a man, then x is mortal. The symbol $\rightarrow$ represents implication. The symbol $\forall$ is a binder (binding the variable x in this example) and represents universal quantification.

We do not wish to simply write this proposition, we want to assume it as a hypothesis. We can do this using the `Hypothesis` keyword and by giving a name to the hypothesis.

```
Hypothesis ManMortal: forall x:A, Man x -> Mortal x.
```

Now ManMortal is an assumed proof of the proposition $\forall x : A.\mathsf{Man}\ x \rightarrow \mathsf{Mortal}\ x$.

Next we need to represent the assumption that Socrates is a man. We first add a variable representing Socrates.

```
Variable Socrates:A.
```

We can now represent the proposition that Socrates is a man by applying the predicate Man to Socrates. We add the hypothesis to the context as before.

```
Hypothesis SocratesMan: Man Socrates.
```

We can now state the theorem that Socrates is mortal. We use the `Theorem` keyword.

```
Theorem SocratesMortal: Mortal Socrates.
```

We must now either give a proof of the theorem (ending with the keyword `Qed`) or leave the theorem unproven (by giving the keyword `Admitted`). In general one can always prove a theorem by using `exact` and then giving a "proof term". To learn more about "proof terms" you should look up information on the Curry-Howard-de Bruijn correspondence [27, 13]. In this case there is a very simple proof term: ManMortal Socrates SocratesMan. The reason why this is a proof term for the theorem is easy to explain. We have declared ManMortal to be a proof of $\forall x : A.\mathsf{Man}\ x \to \mathsf{Mortal}\ x$. We can also see ManMortal as a function which can be applied to a term t of type A and which yields a proof of Man $t \to$ Mortal t. In this case we apply it to Socrates to obtain a proof of Man Socrates $\to$ Mortal Socrates. We can also see this proof as a function which can be applied to a proof of Man Socrates and which yields a proof of Mortal Socrates. Here we apply it to the proof SocratesMan of Man Socrates. That is, ManMortal Socrates SocratesMan is the function ManMortal applied first to Socrates and then to SocratesMan. We could write ManMortal Socrates SocratesMan with parentheses as

$$((\mathsf{ManMortal}\ \mathsf{Socrates})\ \mathsf{SocratesMan})$$

but we omit parentheses whenever possible. By default the implicit binary operator represents function application and function application associates to the left. More details about the syntax will be explained in Chapter 3.

We give the proof in ASCII as follows:

```
exact ManMortal Socrates SocratesMan.
Qed.
```

We then close the section.

```
End Socrates.
```

Egal can be used to check this as follows:

```
egal Socrates1.mg
```

Of course, this is assuming `egal` is in your path and that you are using `egal` instead of `ocamlrun egal.bytecode` and so on. It also assumes you are in the directory with the file `Socrates1.mg`. Otherwise make appropriate modifications.

Calling `egal` on this checks everything and tells you nothing. If there were an error, it would report this error (and have an exit status other than 0). You can get more information by using `-v` to set a higher verbosity.

```
egal -v 10 Socrates1.mg
```

This results in the following cryptic output:

```
Proposition of Theorem SocratesMortal : (forall _:(?0 -> prop),
 (forall _:(?0 -> prop), ((forall _:?0, ((_2 _0) -> (_1 _0))) ->
 (forall _:?0, ((_2 _0) -> (_1 _0)))))) was assigned id
 MJ8kycoSfc9xwU3N5EeRy5PPwQq9aFNTfvss2nhcvzgjZsoW

Proof (fun _:(?0 -> prop) => (fun _:(?0 -> prop) => (fun _:(forall
 _:?0, ((_2 _0) -> (_1 _0))) => (fun _:?0 => (fun _:(_2 _0) => ((__1
 _0) __0))))))

 of SocratesMortal was assigned id
 MM2puxfNxZihV1Nd2o4C2otqwSm5gytPHn6YptXraSXVyLq7

*** With current salt, a treasure could be buried underneath the proof
 of SocratesMortal at 1CQpAnWKx8qKN5ciVLvahsDu7Lun19Wh88 to be
 redeemed with private key
 5JNBdm53bUjtv6MY9TnyiQmHrCNzMrCRccrWBMfPhdCGpG1Lki5.

(TREASURE "SocratesMortal" "1CQpAnWKx8qKN5ciVLvahsDu7Lun19Wh88"
 "5JNBdm53bUjtv6MY9TnyiQmHrCNzMrCRccrWBMfPhdCGpG1Lki5")
```

The first line tells you the global identifier

MJ8kycoSfc9xwU3N5EeRy5PPwQq9aFNTfvss2nhcvzgjZsoW

associated with the proposition proven as SocratesMortal. Superficially it may seem this proposition was Mortal Socrates, but this is not the case. The global identifier corresponds to the abstracted proposition. That is, all variables and hypotheses are abstracted with $\forall$ and $\rightarrow$ so that the proposition has no local dependencies. In this case, the global identifier corresponds to the proposition

$$\forall \mathsf{Man}\ \mathsf{Mortal} : A \rightarrow \mathsf{prop}.(\forall x : A.\mathsf{Man}\ x \rightarrow \mathsf{Mortal}\ x) \rightarrow \\ \forall \mathsf{Socrates} : A.\mathsf{Man}\ \mathsf{Socrates} \rightarrow \mathsf{Mortal}\ \mathsf{Socrates}.$$

Even this is not yet quite correct because there is the type variable A. Did I mention that I hate type variables? The global identifier "remembers" that there is one type variable.

To be more precise, Egal computes a nameless version (using de Bruijn indices for bound variables) of the proposition

$$\forall \mathsf{ManMortal} : A \rightarrow \mathsf{prop}.(\forall x : A.\mathsf{Man}\ x \rightarrow \mathsf{Mortal}\ x) \rightarrow \forall \mathsf{Socrates} : A.\mathsf{Man}\ \mathsf{Socrates} \rightarrow \mathsf{Mortal}\ \mathsf{Socrates}$$

and using ?0 for the type variable. This nameless version can be seen in the output:

```
Proposition of Theorem SocratesMortal : (forall _:(?0 -> prop),
 (forall _:(?0 -> prop), ((forall _:?0, ((_2 _0) -> (_1 _0))) ->
 (forall _:?0, ((_2 _0) -> (_1 _0)))))) was assigned id
```

This nameless term is paired with 1 (to remember there is one type variable) and serialized. This serialization is hashed using SHA256. The result is given two prefix bytes and a checksum and the result of this is written in base 58 giving the global identifier MJ8kycoSfc9xwU3N5EeRy5PPwQq9aFNTfvss2nhcvzgjZsoW. If anyone else considers this same proposition (up to the names), then they will compute the same global identifier.

The proof term is also converted to a nameless representation which is shown in the verbose output:

```
Proof (fun _:(?0 -> prop) => (fun _:(?0 -> prop) => (fun _:(forall
 _:?0, ((_2 _0) -> (_1 _0))) => (fun _:?0 => (fun _:(_2 _0) => ((__1
 _0) __0))))))
```

The proof is also serialized and hashed to give a global name for the proof:

MM2puxfNxZihV1Nd2o4C2otqwSm5gytPHn6YptXraSXVyLq7.

The hash value of the proof can also be used as a bitcoin private key. In general, we hash the proof along with a salt to obtain the private key. Since we have not declared a salt, no salt is used. Egal tells us the bitcoin private key and address in case we want to bury a treasure there.

```
*** With current salt, a treasure could be buried underneath the proof
 of SocratesMortal at 1CQpAnWKx8qKN5ciVLvahsDu7Lun19Wh88 to be
 redeemed with private key
 5JNBdm53bUjtv6MY9TnyiQmHrCNzMrCRccrWBMfPhdCGpG1Lki5.
```

The last line of the output repeats this treasure information as a lisp S-expression:

```
(TREASURE "SocratesMortal" "1CQpAnWKx8qKN5ciVLvahsDu7Lun19Wh88"
 "5JNBdm53bUjtv6MY9TnyiQmHrCNzMrCRccrWBMfPhdCGpG1Lki5")
```

This is simply because lisp is my preferred language. When I was setting up the treasure hunt I could do something like this:

```
egal -v 10 Socrates1.mg | grep TREASURE
```

This would give me S-expressions with treasure information that were easy for me to manipulate using a lisp interpreter.

Lisp has a bad reputation. Because something is fundamentally wrong with people.

If we wanted to bury a treasure under this proof, we could declare it just before the `Theorem` declaration as follows:

```
(* Treasure "1CQpAnWKx8qKN5ciVLvahsDu7Lun19Wh88" *)
```

In context it looks as follows:

```
(* Treasure "1CQpAnWKx8qKN5ciVLvahsDu7Lun19Wh88" *)
Theorem SocratesMortal: Mortal Socrates.
exact ManMortal Socrates SocratesMan.
Qed.
```

You can see this in the file `Socrates2.mg`.

The version you would give to someone who is supposed to find the proof would look as follows:

```
(* Treasure "1CQpAnWKx8qKN5ciVLvahsDu7Lun19Wh88" *)
Theorem SocratesMortal: Mortal Socrates.
Admitted.
```

You can see this in the file `Socrates3.mg`.

`(*` ... `*)` are for comments in Coq, but are used for these special declarations in Egal. Comments in Egal are written as

```
(** here are some comments **)
```

That is, the opening and closing delimiter must have at least two `*`'s. You cannot have a `*` inside the comment because I suck at writing lexers. So don't write `*` inside your comment.

Two other special commands can be used to give a title and to give authors. For example:

```
(* Title "Socrates Introductory Example" *)
(* Author "Chad E. Brown" *)
```

This information is used when generating html versions of the documents and when generating mysql data for the documents. I won't discuss these features of the code.

1.2 Example 2: Nellie is Pink

Today I watched Episode 4 of the 1979 Doctor Who serial *Destiny of the Daleks*. The serial is terrible. I'm not even sure I consider it canon. It's a mess from Romana's "regeneration" to the treatment of the Daleks as being "logical robots" (they are neither). Also, there was the unconvincing resurrection of Davros. Davros was a much better character when he was introduced in *Genesis of the Daleks*. I was surprised and fascinated to learn from Diamanda Hagan that Michael Wisher's original portrayal of Davros was based on Bertrand Russell.

In the final minutes of *Destiny of the Daleks*, the Doctor presents Davros with the following argument:

All elephants are pink. Nellie is an elephant, therefore Nellie is pink.

It should be obvious that this is precisely the same as the Socrates example except for the names. Since I'm essentially playing mind games with myself in order to write a document describing a failed project, I'll allow myself the pleasure of using the example from Doctor Who.

We can extend the file (see `Socrates4.mg`) to include this argument as well:

```
Section Nellie.

Variable A:SType.
Variable Elephant:A->prop.
Variable Pink:A->prop.
Hypothesis ElephantsPink: forall x:A, Elephant x -> Pink x.
Variable Nellie:A.
Hypothesis NellieElephant: Elephant Nellie.
(* Treasure "1CQpAnWKx8qKN5ciVLvahsDu7Lun19Wh88" *)
Theorem NelliePink: Pink Nellie.
exact ElephantsPink Nellie NellieElephant.
Qed.

End Nellie.
```

The treasure address and global names are exactly the same. The point is that the names don't matter.[2]

If we wanted to obscure the fact that the two proofs are the same (and have two distinct treasures) we could do this using salts (see `Socrates5.mg`).

Before the Socrates theorem we could declare:

```
(* Salt "Some Random Salt" *)
```

After the Socrates theorem but before the Nellie theorem we could declare

```
(* Salt "Some Other Random Salt" *)
```

Now the two treasure private keys and addresses will be different. If you try to run `egal` without changing the treasure declarations, it will fail:

```
egal Socrates5.mg

Failure at line 17 char 4: The proof of SocratesMortal does not
 correspond to the declared treasure.
```

You must update the treasure declarations to correspond to the salted versions (see `Socrates6.mg`).

[2]Yes, I know: you don't care that names don't matter. I guess I could call the fact that names don't matter "name irrelevance," and call the fact that no one cares "name irrelevance irrelevance."

1.3 Example 3: Proof Scripts

Very few proofs are given as exact proof terms. Instead proof tactics are used to construct the proof term. Proof tactics are common in theorem provers and the ones in Egal are similar to certain tactics in Coq. The tactic language in Coq is more modular and in fact users can define their own tactics. In Egal there are only a handful of tactics and they are hardcoded (as one can see in `mgcheck.ml`). I had intended to factor out the treatment of tactics, but instead I'm throwing my hands up in a fit of hopelessness and despair. It doesn't matter anyway. *Egal!*

Three tactics which are useful for reasoning with $\forall$ and $\rightarrow$ are `let`, `assume` and `apply`. We demonstrate them using the Nellie example (see `Socrates7.mg`).

This time we do not assume Nellie is an elephant. Instead we prove the proposition that *if* Nellie is an elephant, *then* Nellie is pink.

```
Section NellieWithApplyAndAssume.

Variable A:SType.
Variable Elephant:A->prop.
Variable Pink:A->prop.
Hypothesis ElephantsPink: forall x:A, Elephant x -> Pink x.
Variable Nellie:A.
(* Treasure "15wcUqiRHwb5t17iw6Lpafzm2iFhccRWwS" *)
Theorem NelliePink3: Elephant Nellie -> Pink Nellie.
```

The treasure address will be the same because we will construct the same (global) proof.

We start by assuming Nellie is an elephant using the `assume` tactic.

```
assume H: Elephant Nellie.
```

Now the goal is to prove Nellie is pink under this assumption. We can use the `apply` tactic with ElephantsPink to reduce to proving Nellie is an elephant.

```
apply ElephantsPink.
```

It is worth taking a moment to explain what has happened. We gave the proof term ElephantsPink to the `apply` tactic. The proof term ElephantsPink proves

$$\forall x : A.\mathsf{Elephant}\ x \rightarrow \mathsf{Pink}\ x.$$

Egal compares this to the current goal, Pink Nellie, and sees that it does not match. However, it might match for an appropriate instantiation for x. Next Egal replaces x with a variable X to be instantiatied, a so-called existential variable, and tries to match Elephant $X \rightarrow$ Pink X which again fails. Egal next looks to the right hand side of the implication Pink X and tries to match this with Pink X. This matches the current goal

of Pink Nellie if X is assigned to Nellie. Egal decides that the `apply` tactic succeeds and that the proof term will be of the form

ElephantsPink Nellie ?

where ? is a proof of Elephant Nellie yet to be constructed. This is the way that the `apply` tactic can generally reduce a goal to a number of subgoals.

The `prove` tactic can be used to make it clear what the current goal is. We can use it to make the proof more readable here.

```
prove Elephant Nellie.
```

The `prove` tactic is sometimes useful if one needs to expand definitions so that the goal is in a form that an instance of `apply` can succeed. Here it is only included for human readability of the proof.

Since this is proven by our first assumption we can complete the proof with `exact`.

```
exact H.
Qed.

End NellieWithApplyAssumeAndLet.
```

We next consider a variant of the Nellie example in which we have abstracted over Nellie.

```
Section NellieWithApplyAssumeAndLet.

Variable A:SType.
Variable Elephant:A->prop.
Variable Pink:A->prop.
Hypothesis ElephantsPink: forall x:A, Elephant x -> Pink x.
(* Treasure "15wcUqiRHwb5t17iw6Lpafzm2iFhccRWwS" *)
Theorem NelliePink4: forall n:A, Elephant n -> Pink n.
```

Here the goal is a universal quantifier. We use the `let` tactic with the name n to reduce the goal to proving the implication under the quantifier.

```
let n.
```

We now need to prove the implication. To do this we proceed as before.

```
assume H: Elephant n.
apply ElephantsPink.
prove Elephant n.
exact H.
Qed.
```

If you are familiar with Coq, note that the `let` and `assume` tactics are like the `intros` tactic in Coq. (In Coq implication is a special case of universal quantification.)

In each of the proof scripts above we have constructed the same proof term using slightly different methods. The fact that it is the same proof term can be seen from the fact that the treasure addresses are the same. There is in fact a different, simpler proof of

$$\forall n : A.\mathsf{Elephant}\ n \to \mathsf{Pink}\ n$$

above. The fact that it is different is clear from the different treasure address.

```
(* Treasure "17irP55rL1WFyRtXy22BfpVhG6edW1syW1" *)
Theorem NelliePink5: forall n:A, Elephant n -> Pink n.
exact ElephantsPink.
Qed.

End NellieWithApplyAssumeAndLet.
```

Chapter 2

The Framework: Simple Type Theory

I think perhaps your logic is wearing a little thin.
The Doctor, *The Tomb of the Cybermen*, Episode Two (1967)

In this chapter we briefly introduce the framework of Egal. We will use the simply typed λ-calculus to represent objects and propositions. We will follow the Curry-Howard-de Bruijn correspondence [27, 13] to obtain λ-calculus representations for proof terms.

2.1 Types

Our mathematical objects will be partitioned using simple types. We will use α and β to range over simple types. Simple types all have one of four forms:

- α_0, α_1, α_2 are three type variables. I hate including them, so I'm happy if you ignore them.
- set: This is the type of all sets. There are common ways of encoding mathematical objects (e.g., numbers, pairs of sets, etc.) as sets. We will sometimes generically refer to elements of type set as the individuals of the mathematical universe.
- prop: This is the type of all propositions.
- $\alpha \to \beta$: If α and β are simple types, then $\alpha \to \beta$ is a simple type. This is the type of functions from elements of type α to elements of type β.

From now on we will usually say type instead of simple type. A type is *monomorphic* if it does not contain any type variables. We define $\mathcal{T}_n$ to be the types which only use the first n type variables. Hence $\mathcal{T}_0$ is the set of monomorphic types. The monomorphic types are the only real types. The ones with type variables are more like schemas for a set of types.

In Egal there is a way to represent types (as well as everything else) in ASCII. set is written as `set`, prop is written as `prop` and $\alpha \to \beta$ is written as α`->`β.

2.2 Names

We assume there are infinitely many names. We often use notation such as x and c to range over names. In practice during the formalization process, the name will be an alphanumeric identifier, possibly with underscores. A few examples are False, UnivOf and prop_ext.

2.3 Terms

We now define the set of terms. We use s and t to range over terms. The set of terms is generated from the following cases:

- $x\alpha_0 \cdots \alpha_n$ where $n < 3$. Every name applied to up to three types is a term. The usual case is when $n = 0$ and x itself is a term.[1]
- st: If s and t are terms, then st is a term. Assuming s is a function and t is an appropriate argument, the term st means s applied to t.
- $\lambda x : \alpha.s$: If x is a name, α is a type and s is a term, then $\lambda x : \alpha.s$ is a term. This corresponds to a function which takes an input of type α and returns a value determined by the form of s.
- $s \to t$: If s and t are terms, then $s \to t$ is a term. Assuming s and t are propositions, $s \to t$ is the proposition meaning s implies t.
- $\forall x : \alpha.s$: If x is a name, α is a type and s is a term, then $\forall x : \alpha.s$ is a term. Assuming s is a proposition when x has type α, this corresponds to the proposition which means s holds for all values of x. That is, this term represents universal quantification.

Fundamentally these are the only terms there are. Of course, Egal supports making definitions, but these definitions will be associated with a name, so they are included in the terms above. In addition, Egal supports declaring notation including infix operators, prefix operators, postfix operators and binders. More information will be given about this as needed. Much more information about notation will be given in Chapter 3.

[1] We do not allow general terms to be applied to types because I hate polymorphism and try to restrict it whenever possible. Well, to be honest if you look at the type `tm` in `syntax.ml` and `syntax.mli` you will find that there is a constructor `TpAp` which does allow general applications of terms to types. However, much of the code assumes that it is either a primitive or global name applied to the type. At the level I'm describing things here that's similar to saying it's a name applied to a type.

We will use parenthesis when necessary to remove ambiguity. When parentheses are omitted, application associates to the left and more strongly than implication. Implication associates to the right. For example, $s_1 s_2 s_3 \to t_1 t_2 t_3 \to s_1 s_2 s_3$ means the same as $(((s_1 s_2) s_3) \to (((t_1 t_2) t_3) \to ((s_1 s_2) s_3)))$. In general the scope of binders is as far to the right as is consistent with given parentheses. In particular this is true for the binders λ and $\forall$. We will often omit the type α from $\lambda x : \alpha.s$ Also, the scope of both binders λ and $\forall$ are and $\forall x : \alpha.s$, writing $\lambda x.s$ and $\forall x.s$ instead. The shorthand $\forall x.s$ will always mean $\forall x : \mathsf{set}.s$. Once we give the typing rules below, it will often be possible to infer the missing type in $\lambda x.s$. Also, we will often write $\lambda x_1 \cdots x_n : \alpha.s$ for $\lambda x_1 : \alpha \cdots \lambda x_n : \alpha.s$. Likewise, we will often write $\forall x_1 \cdots x_n : \alpha.s$ for $\forall x_1 : \alpha \cdots \forall x_n : \alpha.s$.

When writing terms in ASCII, we use the following conventions.

- s `->` t means $s \to t$.
- `fun` x `:` α `=>` s means $\lambda x : \alpha.s$.
- `forall` x `:` α `.` s means $\forall x : \alpha.s$.
- `fun` $x_1 \cdots x_n$ `:` α `=>` s means $\lambda x_1 \cdots x_n : \alpha.s$.
- `forall` $x_1 \cdots x_n$ `:` α `,` s means $\forall x_1 \cdots x_n : \alpha.s$.
- `fun` $x_1 \cdots x_n$ `=>` s means $\lambda x_1 \cdots x_n.s$.
- `forall` $x_1 \cdots x_n$ `,` s means $\forall x_1 \cdots x_n.s$.

2.4 Proofs

We now define the set of proof terms. We use $\mathcal{D}$ and $\mathcal{E}$ to range over proof terms. The set of proof terms is generated from the following cases:[2]

- $x\alpha_0 \cdots \alpha_n$ where $n < 3$. Every name applied to up to three types is a proof term. The usual case is when $n = 0$ and x itself is a proof term.[3]
- $\mathcal{D}t$: If $\mathcal{D}$ is a proof term and t is a term, then $\mathcal{D}t$ is a proof term.
- $\mathcal{D}\mathcal{E}$: If $\mathcal{D}$ and $\mathcal{E}$ are proof terms, then $\mathcal{D}\mathcal{E}$ is a proof term.
- $\lambda x : \alpha.\mathcal{D}$: If x is a name, α is a type and $\mathcal{D}$ is a proof term, then $\lambda x : \alpha.\mathcal{D}$ is a proof term.
- $\lambda x : s.\mathcal{D}$: If x is a name, s is a term and $\mathcal{D}$ is a proof term, then $\lambda x : s.\mathcal{D}$ is a proof term.

[2] Again, local `let` constructors are allowed in proof terms, but I'll ignore them because I'm tired and want to get on with my life.

[3] As with ordinary terms, we do not allow general proof terms to be applied to types.

The notational conventions for proof terms is similar to that for terms. In particular, annotations can be omitted from λ-abstractions if the proof checker can synthesize the annotation.

When writing terms in ASCII, we use the following conventions.

- `fun` x `:` α `=>` $\mathcal{D}$ means $\lambda x : \alpha.\mathcal{D}$.
- `fun` x `:` s `=>` $\mathcal{D}$ means $\lambda x : s.\mathcal{D}$.
- `fun` $x_1 \cdots x_n$ `=>` $\mathcal{D}$ means $\lambda x_1 \cdots x_n.\mathcal{D}$.

2.5 Contexts and Signatures

Note that names are not, in advance, associated with types or terms. Instead we associate names with types and terms using contexts and signatures. Contexts are used to associate variables with types or local definitions or associate hypotheses with assumed propositions. Signatures are used to associate names with constants or definitions (possibly polymorphic) or associate names with axioms (possibly polymorphic).

Contexts Γ are lists of name declarations of the following forms:

- $x : \alpha$: This means x is a term of type α.
- $x : \alpha := s$: This means x is the term s of type α.
- $x : s$: This means x is a proof term justifying s.

Signatures Σ are also lists of name declarations, but are at a higher level than contexts. The declarations on signatures are of the following forms:

- $c :_n^m \alpha$ where $n \in \{0, 1, 2, 3\}$ and all type variables in α are α_i with $i < n$. The m is a global identifier corresponding to c. (In practice m is a string derived from a 256-bit number, about which more will be said in Section 2.7.)
- $c :_n^m \alpha := s$ where $n \in \{0, 1, 2, 3\}$ and all type variables in α and s are α_i with $i < n$. Here m is a global identifier corresponding to (n, s).
- $c :_n^m s$ where $n \in \{0, 1, 2, 3\}$ and all type variables in s are α_i with $i < n$. Again, m is a global identifier corresponding to the term (n, s). In this case, we say (n, s) is *known* in Σ and c is the name of the known in Σ.

I would be happier if n were 0 every declaration in every signature. However, adding a choice operator (an ε-operator) at each type requires having some way of dealing with a family of names for each type. Also, a choice operator requires a corresponding polymorphic axiom. In addition, functional extensionality is an axiom parameterized over two types. Once the compromise to have enough type variables to handle each of these, it is difficult to justify not having polymorphic definitions for equality and

existential quantification. However, later when we consider importing defined objects without their definitions, we will only allow this for monomorphic definitions. In other words, polymorphic definitions cannot be made opaque. In addition, arguments to polymorphic names (names in a Σ where the corresponding n is positive) are always used when they are applied to the appropriate number of types. The types must be explicitly given. This follows my decision to support polymorphism as little as possible and to discourage its use. Polymorphism is evil.

The purpose of most Egal declarations is to build up contexts and signatures. The signature grows as the file is processed. The `Parameter` command puts a constant onto the current signature. The `Definition` command puts a definition onto the signature. The `Axiom` and `Theorem` commands put a name giving a proof term for a *known* onto the signature. The keywords `Lemma`, `Example`, `Fact`, `Remark`, `Corollary`, `Proposition` and `Property` are synonyms for `Theorem`.[4] Each `Theorem` declaration (regardless of the keyword used) requires either a correct proof script to be given (ended by `Qed`) or for the proof (or parts of the proof) to be *admitted* using `admit` (for local subgoals) and `Admitted` (for the rest of a proof).

For the most part, keywords (as well as much of the syntax) were chosen to be the same as Coq, in case this isn't obvious. Certainly I don't want to be accused of not acknowledging that I'm making many choices (on purpose) to be similar to Coq. Coq is a nice theorem prover, and I'm accustomed to using it.

The contexts can be grown within a section. A section can be begun with a `Section` command and ended with an `End` command. When a section ends, all the variables, local definitions and hypotheses of the section are popped from the context. The `Variable`, `Let`[5] and `Hypothesis` commands extend the current context.

2.6 Substitution and Conversion

When s and t are terms and x is a name, we write $s[x := t]$ for the result of substituting the term t for the name x in s. One may need to rename some bound variables in order to avoid capture, but we will assume this is done and not discuss details here.

There are four kinds of reducts that may occur in a term. Two of them are independent of contexts and signatures. One depends on a context and the other depends on a signature.

- β: When a term has the form $(\lambda x : \alpha.s)t$ we say it is a β-redex with reduct $s[x := t]$.

- η: When a term has the form $\lambda x : \alpha.sx$ and x is not free in s, we say it is an η-redex with reduct s.

[4] To find a list of all special symbols and keywords, see the file `lexer.mll` which is processed by ocamllex. Looking at it again, I notice there is a keyword `Conjecture` which is unused. The intention at one time was to allow users to register conjectures, but this intention was never realized.

[5] Warning: `Let` declarations have not been thoroughly tested.

- δ^Γ: If $x : \alpha := s \in \Gamma$, then x is a δ^Γ-redex with reduct s.
- δ^Σ: If $x :_n^m:= s \in \Sigma$, then $x\beta_0 \cdots \beta_{n-1}$ is a δ^Σ-redex with reduct $s[\alpha_0 := \beta_0, \ldots, \alpha_{n-1} := \beta_{n-1}]$. I guess I haven't defined this simultaneous type substitution notation, but probably you can figure out what it is. If you're not sure look at `tpsubst` and `tmtpsubst` in the file `syntax.ml`.[6]

We say that two terms s and t are convertible if they can be made the same via changing the names of bound variables (α-conversion) and any number of these reductions, relative to a Σ and Γ. If the signature and context are well-formed (to be defined later), then convertibility is decidable since reduction is confluent and terminates.[7]

2.7 Nameless Representations and Global Identifiers

There is still some distance between what is described above and what is implemented in Egal. Types, terms and proofs are input using local names. When Egal parses types, terms and proofs, they are all parsed as the same kind of preterm. Without getting into details about parsing here, they are parsed to the type `ltree` (for "layout tree", see `syntax.ml` and `syntax.mli`) which easily maps to an object of type `atree` (for "abstract tree", see `syntax.ml` and `syntax.mli`). See Chapter 3 for more information about parsing and printing.

Before type checking and proof checking is performed, these "abstract trees" are converted into a nameless representation using de Bruijn indices [14].

We take it for granted that you understand de Bruijn indices and simply remind you that a de Bruijn index indicates how many binders to pass through to get to the binder that binds the de Bruijn index. Just read de Bruijn's 1972 paper [14]. So stop reading this, go read de Bruijn's paper, and then come back.

Now that you've read it, I have something to say. De Bruijn was a mathematician who did a lot of work in a number of different areas. This "de Bruijn index" representation is frankly a minor thing developed as part of his AUTOMATH project and that has his name attached to it. (It's not even the only nameless reprentation he gave. There are also de Bruijn levels.) So please don't think of him as the "index" guy.

I will say more about the code in a moment, but first let me introduce nameless types, nameless terms and nameless proof terms as mathematical objects. I will use the same metavariables α, s and $\mathcal{D}$ for the nameless versions as I did for the named versions.

[6] Well, these implementations are for the nameless versions, but you should be able to figure out what's going on. It's just type substitution. There's not even the possibility of capture to worry about.

[7] Actually, in practice it doesn't matter that reduction terminates. It is easy to write down small terms that would take so long to terminate that they never actually would. There is a variable `beta_count` in `syntax.ml` which starts at a million and counts down. If your document requires more than a million β-reductions, it will not be accepted.

A nameless type is either 0, 1, 2 (3 nameless type variables), prop, set or $\alpha \to \beta$ where α and β are nameless types.

Nameless terms s, t are generated from the following grammar:

$$s, t ::= j|p\beta_0 \cdots \beta_{n-1}|m\beta_0 \cdots \beta_{n-1}|st|\lambda : \alpha.s|s \to t|\forall : \alpha.s$$

where j ranges over natural numbers,[8] p ranges over "primitive names"[9] and m ranges over "global identifiers".[10]

Nameless proof terms $\mathcal{D}, \mathcal{E}$ are generated from the following grammar:

$$\mathcal{D}, \mathcal{E} ::= j|m\beta_0 \cdots \beta_{n-1}|\mathcal{D}\mathcal{E}|\mathcal{D}t|\lambda : \alpha.\mathcal{D}|\lambda : s.\mathcal{D}$$

where j ranges over natural numbers,[11] and m ranges over "global identifiers".

The notions of β and η reduction have a natural counterpart for nameless terms.

Each closed term/proof can be transformed into a nameless term/proof. In fact, Egal never considers terms or proofs with names at all. There are "mathematical expressions" which are parsed and printed, but these are transformed into nameless types, nameless terms and nameless proofs in the process of type checking and proof checking. The printing and parsing of mathematical expressions will be the topic of Chapter 3.

We will nevertheless present the type checking and proof checking rules in Section 2.9 using a named representation, since this is more human readable.

For each nameless term s it is possible to obtain a "collision resistant" global identifier $s^\sharp$. By "collision resistant" we mean that if $s^\sharp = t^\sharp$ and $s = t$, then we will have found a collision in the SHA256 hashing function. At the moment, such collisions are thought to be impractical to obtain.

SHA256 is a cryptographic hashing algorithm. You can find a pdf description from NIST here:

`http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf`

I recommend downloading it and saving a local copy.[12] When there are funding problems with the U. S. Government the Democrats who work at NIST like to be jackasses and replace the pdfs with political statements that say things like:

Due to the government shutdown, we can't afford to host a pdf, but can only afford to host political messages like this instead. By the way, we're lying jackasses.

[8]These are the de Bruijn indices which will replace local variable names.

[9]These will be determined by the theory. In our case, there will be primitive names for the choice operator and a number of set theory primitives.

[10]These identifiers are used instead of the names in a signature Σ. Recall that global identifiers m were included as part of Σ. This is the reason they were included.

[11]These are the de Bruijn indices which will replace names of local hypotheses.

[12]If you do download the pdf, you should probably use Tor. Otherwise the government might hound you until you commit suicide like they did to Aaron Swartz.

That may not be an exact quote. I didn't save it. But they really did make the pdfs inaccessible and replace them with political messages against the government shutdown. And I find it much easier to believe they are lying jackasses than to believe there's a cost savings by removing access to the pdfs during a government shutdown. Google "shutdown theater" to learn more. There are some very nasty people in the world. And they think they're better than you.

Yes, I'm aware of the idea that the "shutdown" in 2013 was the fault of Republicans. The attribution of causality is notoriously difficult in general. If two parties cannot come to an agreement about whether to spend a lot or spend a lot more, then how does one decide which party is the one "refusing" to come to an agreement? Well, for the mainstream media (a.k.a, the MFM, as they are sometimes known at Ace of Spades) it's simple: it's the fault of Republicans. Why? Because roughly 90% of people in the media vote for Democrats. See how easy it is to assign blame?

Oh well. You know what they say. If you can't cut government spending, then create an alternative decentralized monetary system that can't be controlled by governments.

To have reasonable notation, we will use $s^\sharp$ for the global identifier for a monomorphic term s and use $(i,s)^\sharp$ for the global identifier for a term s with i type variables. The global identifier $s^\sharp$ is not simply the result of hashing a serialization of s. First we traverse s to see if there are any monomorphic closed proper subterms (other than global identifiers m). For each such proper subterm t, we compute $t^\sharp$ and replace the subterm with $t^\sharp$. After doing this we $\beta\eta$-normalize to obtain s'. (Note that we may have destroyed some β-redexes in the previous step. This was intentional.) The result of $\beta\eta$-normalizing may have created new monomorphic closed proper subterms, so we repeat the process of replacing such subterms t with $t^\sharp$. After this second time we will have a $\beta\eta$-normal nameless term for which every monomorphic closed proper subterm is a global identifier. After we obtain this modified term, we serialize it and hash it using SHA256. In addition, we add a prefix and a 16 bit checksum. The global identifier is then given as a base 58 string. The prefix is chosen so that all global identifiers start with `M`.

A similar process gives global identifiers $(i,s)^\sharp$ when we need to record the number of type variables i in s.

When we have a closed term (with names) which makes sense in a signature Σ, we can convert it to a nameless term by replacing each c with m whenever $c :_i^m \alpha \in \Sigma$ or $c :_i^m \alpha := s \in \Sigma$. The mapping of local variables to de Bruijn indices is well-known. You read about it in de Bruin's paper.

In addition to β and η reduction, we have the following notion of δ reduction: If $m = (n,s)^\sharp$, then $m\beta_0 \cdots \beta_{n-1}$ reduces to $s[\beta_0,\ldots,\beta_{n-1}]$. If we use δ reduction and know some (n,s) such that $m = (n,s)^\sharp$ for every m we encounter, then we can reduce a nameless term to a nameless term in which there are no occurrences of global identifiers.

Let's turn to the implementation.

The data types corresponding to the nameless representations can be found in

`syntax.ml` and `syntax.mli`.[13]

For types see the type `tp`.

```
type tp = TpVar of int | Prop | Set | Ar of tp * tp
```

Note that whatever you call the type variables you declare, they will internally be mapped to `TpVar` i where i is 0, 1 or 2. Do not declare more than 2 type variables. In fact, try not to declare any.

For terms see the type `tm`.

```
type tm =
  | DB of int
  | TmH of string
  | Prim of int
  | TpAp of tm * tp
  | Ap of tm * tm
  | Lam of tp * tm
  | Imp of tm * tm
  | All of tp * tm
```

The first three constructors probably need the most explanation. We say something about each of them.

- `DB`: `DB`(i) is de Bruijn index i. This stands for a locally bound variable. Since you read de Bruijn's paper, you know what this is. I guess I should say that I start counting from 0. A lot of people who use de Bruijn indices start counting from 1. It's not important of course, as long as you choose one and stick with it.

- `TmH`: `TmH`(m) Here m is the global identifier for a term, as described above. See `tm_id` and `ptm_id` if you want to see details.

- `Prim`: `Prim`(i) is the ith primitive. Only certain primitives are allowed and these will correspond to the choice operator and the set theory primitives. In the manual I'm calling these "primitive names" even though in the implementation they are numbers. Also, in the implementation certain names are hard coded to correspond to these primitives. In particular, `Prim`(0) is the polymorphic choice operator of type $(\alpha_0 \to \mathsf{prop}) \to \alpha_0$ referred to by Eps. `Prim`(1) is set membership and is referred to by In, `Prim`(2) is the empty set referred to by Empty, `Prim`(i) for i between 3 an 6 are other set operators referred to by the names Union, Power, Repl and UnivOf. `Prim`(i) for i greater than 6 is illegal. You can find the

[13]Some users complained about cryptic error messages. The problem was that the nameless representations look ugly went printed. One way around this would be to keep suggested names in some places and then use these to print a named representation. I've been told Coq does something like this. I considered doing it, but I'm certainly not going to do it now. Feel free to make this change yourself.

hard coded values along with their types by looking for the function `primname` in `syntax.ml`. More information is given in Chapter 5.[14]

- `TpAp`: `TpAp`(s, α) means applying the term s to the type α. In practice s must either be the primitive 0 (the choice operator Eps) or an appropriate global identifier. This might be different if Egal were modified to target a different theory.
- `Ap`: `Ap`(s, t) is ordinary term application st.
- `Lam`: `Lam`(α, s) corresponds to $\lambda : \alpha.s$ in the nameless representation, i.e., a term like $\lambda x : \alpha.s$ converted to the nameless representation. For example, $\lambda p : \mathsf{prop}.p$ would be represented as `Lam`$(\mathsf{prop}, \mathtt{DB}(0))$.
- `Imp`: `Imp`(s, t) is implication $s \to t$.
- `All`: `All`(α, s) corresponds to $\forall : \alpha.s$. For example, $\forall p : \mathsf{prop}.p$ would be represented as `All`$(\mathsf{prop}, \mathtt{DB}(0))$.

There is also a type which pairs a term with the number of type variables it has. This is important when computing a global id since the polymorphism level needs to be included before hashing.

```
type ptm = int * tm
```

$s^\sharp$ is computed using `tm_id`, which calls `tm_pack_full` to handle the three stage process described above. The function `tm_pack` handles replacing monomorphic closed proper subterms with global identifiers in the first and third phase, while the normalization in the second phase is handled by `tm_beta_eta_norm`.

Likewise $(i, s)^\sharp$ is handled by `ptm_id`. There is no `ptm_pack_full`; the three phases are inlined in this case. I think this is simply because the function `tm_pack_full` is called elsewhere (by functions that compute global identifiers for proof terms) whereas there was no need for a function `ptm_pack_full`.

Note that $(0, s)^\sharp$ (the result of calling with `ptm_id` $(0, s)$) is different from $s^\sharp$ (the result of calling `tm_id` with s) since $(0, s)^\sharp$ is explicit about the fact that it is monomorphic. This distinction may or may not be important. I don't remember.

For proof terms, there is the type `pf` and the type `ppf` remembering the number of type variables.

```
type pf =
  | Hyp of int
  | Known of string
  | PTpAp of pf * tp
  | PTmAp of pf * tm
  | PPfAp of pf * pf
```

[14] Of course, if you wanted to modify Egal to handle theories other than set theory, this could all be modified within the code.

```
  | PLam of tm * pf
  | TLam of tp * pf

type ppf = int * pf
```

If you understood the explanations of the constructors for `tm`, then you should have no trouble understanding most of the constructors for `pf`. The constructor `Hyp` plays the role of de Bruijn indices at the level of hypotheses (local assumptions).

The one constructor that requires some explanation is `Known`. This is similar to `TmH`, but differs in an important way. In a proof `Known`(m), m is the global id of a *proposition* (a term of type `prop`) rather than a proof. In other words, `Known`(m) simply asserts a proof of the proposition with global identifier m. In practice this is done because we already have a proof and it does not matter which one we use. For example, if we have a proposition s with global identifier m and a signature Σ of the form $c :_0^m s, d :_0^m s$, then both of the proof terms c and d will map to the nameless representation `Known`(m). Let me try to motivate this choice.

Egal is intended to build up a database of formalized mathematics. One starts from the primitives and axioms and makes definitions and proves theorems. If a theorem has been proven, its particular proof does not matter to someone wanting to use the theorem. For this reason Egal allows a user to "import" a previously proven fact as an axiom. As an axiom, one only knows the proposition. I could have designed Egal so that one needs to give some "global identifier" for the proof when importing it as an axiom, but this seems wrong. The particular proof is not important.

On the other hand, if a user wants to import a previously defined object, its particular definition *is* important. Previously made definitions can be made again and, indeed, polymorphic definitions can only be imported this way. Monomorphic definitions can be imported as a parameter. However, when importing a definition as a parameter in this way the user must explicitly give the global identifier corresponding to the definition. The alleged "definition" cannot be expanded if it is imported as a parameter (it will be opaque), but the global identifier is needed since it will be used as m in `TmH`(m) when the parameter is used.

I think these choices were key ingredients in what should have made Egal a good program for building a database of formalized mathematics. Now it's just obscure fact buried in a manual. Oh well. *Egal!*

2.8 Importing Previous Work (Databases and Signatures)

Egal allows users to import known objects (either opaquely as parameters or transparently as definitions) and known propositions (as axioms). For this to work propertly, there must be a way to make sure the imported known propositions are either axioms or previously proven theorems. A database plays this role.

A *database* Δ is a pair (Δ_0, Δ_1) where Δ_0 is a partial function from global identifiers to monomorphic types and Δ_1 is a set of global identifiers. The idea is that Δ_0 keeps up with the simple types of previously defined (monomorphic) objects. This allows users to import (monomorphic) objects as parameters instead of definitions by associating them with their global identifier. If the type of the parameter does not match the type of the global identifier, then there is an error, as one should expect. Similarly, Δ_1 keeps up with the global identifiers of (possibly polymorphic) propositions which were previously proven. This allows users to import previously proven results as axioms.

The way previous work can be imported is in signature files. A signature file imports previous work to build up the signature Σ which will be used to check the main document file. I have used the suffix `.mgs` for signature files and `.mg` for ordinary documents, but this isn't enforced. A signature file actually is a special case of an ordinary document. The only restriction is that no theorems can be proven. When Egal is called, the last argument is the main document to process. If the main document depends on some previously results, then the command line argument `-I` should be used to give signature files importing the previous results. For example, if you try use Egal to check the file `ExEq1.mg` without giving a signature file, then it will signal an error.

```
egal ExEq1.mg
```

```
Failure at line 10 char 54: Unknown term ~ -- it might be a proof in
 the position where a term is expected
```

In this case it complains that it does not know the notation for negation (which is more of a parsing issue), but in fact it does not know about negation at all. Negation is first defined in `PropLogic1.mg`. The work from `PropLogic1.mg` needed for `ExEq1.mg` is collected into the signature file `ExEq1Preamble.mg`. We use `-I` to specify this dependency, but now we get an error when processing the signature file.

```
egal -I ExEq1Preamble.mgs ExEq1.mg
```

```
Failure at line 12 char 50: The id
 MHhyQTtRNciKhL6BZWxm7bdikQQWiAg1DE2zEQQCJvp6jPbQ for the proposition
 for axiom andI is not indexed as previously known
```

If we look at the relevant line of `ExEq1Preamble.mgs` we see the following:

```
Axiom andI : forall A B : prop , A -> B -> A /\ B.
```

Egal complains because we have called it with an empty database, so it does not know this proposition is known. This is remedied by specifying an appropriate database index file with `-ind`. In this case, we use `IndexMar2014`, about which we will say a bit more below.

```
egal -ind IndexMar2014 -I ExEq1Preamble.mgs ExEq1.mg
```

Signature files can also import monomorphic definitions as parameters. To do this, one must explicitly give the appropriate global identifier. To see an example of this, look in the file `Nat1Preamble.mgs` and search for binunion. This represents the binary union of two sets. The definition for it can be found in `Set1b.mg`, but for `Nat1.mg` we only need its type and certain previously proven results about it. Hence in `Nat1Preamble.mgs` we give the following information:

```
(* Parameter binunion MLywQTtJcoNKgztgfTu69bhu4eUuLMxPDoLPPJ551y1WpprR *)
Parameter binunion : set -> set -> set.
(* Unicode :\/: "222a" *)
Infix :\/: 345 left  := binunion.
Axiom binunionI1 : forall X Y z : set , z :e X -> z :e X :\/: Y.
Axiom binunionI2 : forall X Y z : set , z :e Y -> z :e X :\/: Y.
Axiom binunionE : forall X Y z : set , z :e X :\/: Y -> z :e X \/ z :e Y.
```

If we ignore the infix notation of `:\/:` for binunion for the moment, we see that we have declared that binunion has the type $\mathsf{set} \to \mathsf{set} \to \mathsf{set}$ and global identifier

$$\mathsf{MLywQTtJcoNKgztgfTu69bhu4eUuLMxPDoLPPJ551y1WpprR}.$$

(This global identifier can be found by appropriately calling Egal on a document in which binunion is defined, e.g., `Set1b.mg`.) This is enough to put

$$\mathsf{binunion} :_0^{\mathsf{MLywQTtJcoNKgztgfTu69bhu4eUuLMxPDoLPPJ551y1WpprR}} \mathsf{set} \to \mathsf{set} \to \mathsf{set}$$

onto Σ. We then import three previously proven results as binunionI1, binunionI2 and binunionE so that we have enough facts about binunion to do proofs about it without knowing the definition.

A database can grow as documents are processed. By default Egal has an empty database, so that no previous objects or previously proven propositions can be imported.[15] Egal can optionally be given a database as an "index file" using the `-ind` command line option. For the treasure hunt files, the index file used was `IndexMar2014`. For this reason, almost all of the files are processed as

```
egal -ind IndexMar2014 ...
```

(or with modifications to the path if you are not in the `formaldocs` directory). The file `IndexMar2014` is human readable. It is a sequence of lines each of which is either a global identifier followed by a monomorphic type followed by a period (adding to Δ_0) or the keyword `Known` followed by a global identifier followed by a period (adding to Δ_1).

Such a database "index file" can be also created by Egal. To do this, use `-indout` to write out the database that results from processing the current document. If one uses both `-ind` and `-indout` as

[15] However, the primitive objects and axioms of the mathematical foundation (higher-order Tarski-Grothendieck set theory can always be declared. The corresponding global identifiers for the axioms are hard-coded into `mgcheck.ml`.

```
egal -ind Index1 -indout Index2 filetoprocess.mg
```

then the database will be initialized using `Index1`, the file `filetoprocess.mg` will be processed constructing a signature Σ and updating the database along the way, and the resulting database at the end will be written as index file `Index2`.

In this way the database (representing the essence of the mathematical library being constructed) can grow linearly, much like a block chain.[16]

The file `IndexMar2014` was created by starting with an empty file and growing the file by processing the treasure hunt documents. Here is (more or less) how it was constructed (giving the file `indextest1` to avoid overwritting the released `IndexMar2014`.

```
egal -indout indextest1 Foundation.mg
egal -ind indextest1 -indout indextest1 PropLogic1.mg
egal -ind indextest1 -indout indextest1 -I ExEq1Preamble.mgs ExEq1.mg
egal -ind indextest1 -indout indextest1 -I Set1aPreamble.mgs Set1a.mg
egal -ind indextest1 -indout indextest1 -I Preds1Preamble.mgs Preds1.mg
egal -ind indextest1 -indout indextest1 -I Relns1Preamble.mgs Relns1.mg
egal -ind indextest1 -indout indextest1 -I KnasterTarskiPreamble.mgs KnasterTarski.mg
egal -ind indextest1 -indout indextest1 -I Relns2Preamble.mgs Relns2.mg
egal -ind indextest1 -indout indextest1 -I Classical1Preamble.mgs Classical1.mg
egal -ind indextest1 -indout indextest1 -I ConditionalsPreamble.mgs Conditionals.mg
egal -ind indextest1 -indout indextest1 -I Set1bPreamble.mgs Set1b.mg
egal -ind indextest1 -indout indextest1 -I Exactly1ofPreamble.mgs Exactly1of.mg
egal -ind indextest1 -indout indextest1 -I EpsIndPreamble.mgs EpsInd.mg
egal -ind indextest1 -indout indextest1 -I Set1cPreamble.mgs Set1c.mg
egal -ind indextest1 -indout indextest1 -I Zermelo1908Preamble.mgs Zermelo1908.mg
egal -ind indextest1 -indout indextest1 -I Nat1Preamble.mgs Nat1.mg
egal -ind indextest1 -indout indextest1 -I Set2Preamble.mgs Set2.mg
egal -ind indextest1 -indout indextest1 -I QuotientsPreamble.mgs Quotients.mg
egal -ind indextest1 -indout indextest1 -I Nat2Preamble.mgs Nat2.mg
egal -ind indextest1 -indout indextest1 -I UnivInfPreamble.mgs UnivInf.mg
egal -ind indextest1 -indout indextest1 -I Ordinals1Preamble.mgs Ordinals1.mg
egal -ind indextest1 -indout indextest1 -I DisjointUnionsPreamble.mgs \\
                                                       DisjointUnions.mg
egal -ind indextest1 -indout indextest1 -I OrderedPairsPreamble.mgs OrderedPairs.mg
egal -ind indextest1 -indout indextest1 -I DepSumsPreamble.mgs DepSums.mg
egal -ind indextest1 -indout indextest1 -I FuncsSetsPreamble.mgs FuncsSets.mg
egal -ind indextest1 -indout indextest1 -I DepProdsPreamble.mgs DepProds.mg
```

Egal is also being released with another index file `IndexJune2014`. This file is an extension of `IndexMar2014` with some information about modal logic and gives the database one needs to start with to process the document giving Gödel's proof of the

[16]Well, to be like a block chain the next block should only contain the new entries in the database, but you probably get the idea.

existence of God. (I ported this proof to Egal starting from a Coq version by Benzmüller and Paleo [9].)

```
egal -ind IndexMar2014 GoedelGodPreamble.mgs

Failure at line 122 char 95: The id
 MLucBeuQfTjWvrBKTPNKycJoPHZSr6Q9qpJkhznXi7A1y8w2 for the proposition
 for axiom mp_dia is not indexed as previously known

egal -ind IndexJune2014 GoedelGodPreamble.mgs

egal -ind IndexJune2014 -I GoedelGodPreamble.mgs GoedelGod.mg
```

In the code Δ_0 corresponds to the hash table `indextms` and Δ_1 corresponds to the hash table `indexknowns`. We briefly look at some of the code to explain how knowns are added to Δ_1.

Examine the following code in `mgcheck.ml` which is called after a `Theorem` declaration is made (and before the proof is given):

```
let i = List.length !ctxtp in
let atm = check_tm a Prop !polytm sigtmof !sigtm !ctxtp !ctxtm in
let agtm = !tmallclos atm in
let ahv = ptm_id (i,agtm) sigtmof sigdelta in
```

The first line sets `i` to be the number of type variables in the current context. The second line calls `check_tm` on something that was parsed in order to check that it can be turned into a `tm` (nameless represetation) of type `prop`. The next line forms the closure of the `tm` so that it no longer depends on a context (except for type variables, if there are any). This closed term is assigned to `agtm`. Finally, a global identifier is computed for i paired with the closed `tm`. This is a global identifier for the (possibly) polymorphic closed proposition. This global identifier is assigned to `ahv`.

Later in the code (still handing a `Theorem` declaration) we see

```
proving := Some (x,i,agtm,ahv);
```

This means Egal will start interpreting what it reads as proof tactics which should ultimately construct a proof for the proposition `agtm` with `i` type variables and global identifier `ahv`. Here `x` is the name the document gives the theorem.

If the proof is successfully completed leaving no unproven subgoals, then the following code will be executed:

```
if not (Hashtbl.mem indexknowns gphv) then Hashtbl.add indexknowns gphv ();
```

While the variable is named `gphv` here, it has the same value as `ahv` above. Hence after the theorem has been proven Egal checks to see if the proposition's global identifier is already in Δ_1 and adds it to Δ_1 if not.

Let Σ be a signature. We say a database $\Delta = (\Delta_0, \Delta_1)$ *includes* Σ if the following three conditions hold:

- For every $c :_0^m \alpha \in \Sigma$, $\Delta_0(m) = \alpha$.
- For every $c :_0^m \alpha := s \in \Sigma$, $\Delta_0(m) = \alpha$.
- For every $c :_n^m s \in \Sigma$, $m \in \Delta_1$.

2.9 Rules for Checking

We now define rules for type checking and proof checking. We assume a fixed database $\Delta = (\Delta_0, \Delta_1)$.[17] We also assume a fixed set $\mathcal{P}$ of primitive names and corresponding types. That is, the elements of $\mathcal{P}$ are (p, i, α) where p is a primitive name, $i \in \{0, 1, 2, 3\}$ and $\alpha \in \mathcal{T}_i$. Finally we assume a set $\mathcal{A}$ of global identifiers of axioms. The sets $\mathcal{P}$ and $\mathcal{A}$ will be made precise in Chapter 5.

We give rules that define the following type checking and proof checking relations:

- $\Sigma\ \mathtt{ok}$
  This means the signature Σ is well-formed.
- $\Sigma \vdash_i \Gamma\ \mathtt{ok}$
  This means the context Γ is well-formed under the well-formed signature Σ and Γ only uses the first i type variables.
- $\Sigma; \Gamma \vdash_i s : \alpha$
  This means the term s has the type α in the well-formed signature Σ and well-formed context Γ and that Γ, s and α are restricted to using the first i type variables.
- $\Sigma; \Gamma \vdash_i \mathcal{D} : s$
  This means the term s has type prop and the proof term $\mathcal{D}$ proves s in the well-formed signature Σ and well-formed context Γ and that Γ, $\mathcal{D}$ and s are restricted to using the first i type variables.

The rules defining these relations are given in Figures 2.1 and 2.2.

As a simple example, the reader should check that in the empty signature and empty context, $\forall p : \mathsf{prop}.p$ has type prop. That is

$$\cdot; \cdot \vdash_0 \forall p : \mathsf{prop}.p : \mathsf{prop}$$

holds. The reader can also check that

$$\cdot; \cdot \vdash_0 \forall p : \mathsf{prop}.p \to p : \mathsf{prop}$$

holds. To try a proof checking example, the reader can verify that

$$\cdot; \cdot \vdash_0 \lambda p : \mathsf{prop}.\lambda u : p.u : \forall p : \mathsf{prop}.p \to p$$

$$\frac{}{\cdot\ \mathsf{ok}} \qquad \frac{\Sigma; \cdot \vdash_i s : \alpha}{\Sigma, c :_i^{(i,s)^\sharp} \alpha := s\ \mathsf{ok}}\ c \notin \mathsf{dom}\Sigma \qquad \frac{\Sigma\ \mathsf{ok}}{\Sigma, c :_0^m \alpha\ \mathsf{ok}}\ c \notin \mathsf{dom}\Sigma,\ \Delta_0(m) = \alpha$$

$$\frac{}{\Sigma, p :_i^{(i,p)^\sharp} \alpha\ \mathsf{ok}}\ p \notin \mathsf{dom}\Sigma,\ (p, i, \alpha) \in \mathcal{P} \qquad \frac{\Sigma; \cdot \vdash_i s : \mathsf{prop}}{\Sigma, c :_i^{(i,s)^\sharp} s\ \mathsf{ok}}\ c \notin \mathsf{dom}\Sigma,\ (i,s)^\sharp \in \Delta_1$$

$$\frac{\Sigma; \cdot \vdash_i \mathcal{D} : s}{\Sigma, c :_i^{(i,s)^\sharp} s\ \mathsf{ok}}\ c \notin \mathsf{dom}\Sigma \qquad \frac{\Sigma; \cdot \vdash_i s : \mathsf{prop}}{\Sigma, c :_i^{(i,s)^\sharp} s\ \mathsf{ok}}\ c \notin \mathsf{dom}\Sigma,\ (i,s)^\sharp \in \mathcal{A} \qquad \frac{\Sigma\ \mathsf{ok}}{\Sigma \vdash_i \cdot\ \mathsf{ok}}$$

$$\frac{\Sigma \vdash_i \Gamma\ \mathsf{ok}}{\Sigma \vdash_i \Gamma, x : \alpha\ \mathsf{ok}}\ x \notin \mathsf{dom}\Gamma,\ \alpha \in \mathcal{T}_i \qquad \frac{\Sigma \vdash_i \Gamma\ \mathsf{ok} \qquad \Sigma; \Gamma \vdash_i s : \alpha}{\Sigma \vdash_i \Gamma, x : \alpha := s\ \mathsf{ok}}\ x \notin \mathsf{dom}\Gamma$$

$$\frac{\Sigma \vdash_i \Gamma\ \mathsf{ok} \qquad \Sigma; \Gamma \vdash_i s : \mathsf{prop}}{\Sigma \vdash_i \Gamma, x : s\ \mathsf{ok}}\ x \notin \mathsf{dom}\Gamma \qquad \frac{\Sigma \vdash_i \Gamma\ \mathsf{ok}}{\Sigma; \Gamma \vdash_i x : \alpha}\ x : \alpha \in \Gamma$$

$$\frac{\Sigma \vdash_i \Gamma\ \mathsf{ok}}{\Sigma; \Gamma \vdash_i x : \alpha}\ x : \alpha := s \in \Gamma$$

$$\frac{\Sigma \vdash_i \Gamma\ \mathsf{ok}}{\Sigma; \Gamma \vdash_i c\beta_0 \cdots \beta_{n-1} : \alpha[\alpha_0 := \beta_0, \ldots, \alpha_{n-1} := \beta_{n-1}]}\ c :_n^m \alpha \in \Sigma,\ \beta_0, \ldots, \beta_{n-1} \in \mathcal{T}_i$$

$$\frac{\Sigma \vdash_i \Gamma\ \mathsf{ok}}{\Sigma; \Gamma \vdash_i c\beta_0 \cdots \beta_{n-1} : \alpha[\alpha_0 := \beta_0, \ldots, \alpha_{n-1} := \beta_{n-1}]}\ c :_n^m \alpha := s \in \Sigma,\ \beta_0, \ldots, \beta_{n-1} \in \mathcal{T}_i$$

$$\frac{\Sigma; \Gamma \vdash_i s : \alpha \to \beta \qquad \Sigma; \Gamma \vdash_i t : \alpha}{\Sigma; \Gamma \vdash_i st : \beta} \qquad \frac{\Sigma; \Gamma, x : \alpha \vdash_i s : \beta}{\Sigma; \Gamma \vdash_i \lambda x : \alpha.s : \alpha \to \beta}$$

$$\frac{\Sigma; \Gamma \vdash_i s : \mathsf{prop} \qquad \Sigma; \Gamma \vdash_i t : \mathsf{prop}}{\Sigma; \Gamma \vdash_i s \to t : \mathsf{prop}} \qquad \frac{\Sigma; \Gamma, x : \alpha \vdash_i s : \mathsf{prop}}{\Sigma; \Gamma \vdash_i \forall x : \alpha.s : \mathsf{prop}}$$

Figure 2.1: Type Checking Rules

$$\frac{\Sigma \vdash_i \Gamma\ \mathtt{ok}}{\Sigma; \Gamma \vdash_i c\beta_0 \cdots \beta_{n-1} : s[\alpha_0 := \beta_0, \ldots, \alpha_{n-1} := \beta_{n-1}]}\ c :^m_n s \in \Sigma,\ \beta_0, \ldots, \beta_{n-1} \in \mathcal{T}_i$$

$$\frac{\Sigma \vdash_i \Gamma\ \mathtt{ok}}{\Sigma; \Gamma \vdash_i x : s}\ x : s \in \Gamma \qquad \frac{\Sigma; \Gamma \vdash_i \mathcal{D} : s \to t \qquad \Sigma; \Gamma \vdash_i \mathcal{E} : s}{\Sigma; \Gamma \vdash_i \mathcal{D}\mathcal{E} : t}$$

$$\frac{\Sigma; \Gamma \vdash_i \mathcal{D} : \forall x : \alpha.s \qquad \Sigma; \Gamma \vdash_i t : \alpha}{\Sigma; \Gamma \vdash_i \mathcal{D}t : s[x := t]} \qquad \frac{\Sigma; \Gamma, x : s \vdash_i \mathcal{D} : t}{\Sigma; \Gamma \vdash_i \lambda x : s.\mathcal{D} : s \to t}$$

$$\frac{\Sigma; \Gamma, x : \alpha \vdash_i \mathcal{D} : t}{\Sigma; \Gamma \vdash_i \lambda x : \alpha.\mathcal{D} : \forall x : \alpha.s}$$

Figure 2.2: Proof Checking Rules

holds.

The relations are quite simple and it is possible for readers to use the rules to verify when the relation holds. More importantly, it is possible to write a computer program to check when these relations hold. Such a program is an example of a proof checker. In this case, Egal is such a proof checker.[18]

We say a database $\Delta = (\Delta_0, \Delta_1)$ is *valid* if the following conditions hold:

- For every $(m, \alpha) \in \Delta_0$ there is monomorphic closed term s such that $\cdot; \cdot \vdash_0 s : \alpha$ and $(0, s)^\sharp = m$.

- For every $m \in \Delta_1$ there exist i, s and $\mathcal{D}$ such that $\cdot; \cdot \vdash_i \mathcal{D} : s$ and $(i, s)^\sharp = m$.

We make a number of conjectures about the type checking and proof checking relations. These are mainly stated as conjectures because I haven't proven them, but the intention is that they are provable. (If they aren't, the rules should be changed.) I also state them here because I use them in arguments below.

Conjecture 2.9.1 *If s and t are convertible, $\Sigma; \Gamma \vdash s : \alpha$ and $\Sigma; \Gamma \vdash t : \beta$, then $\alpha = \beta$.*

Conjecture 2.9.2 *Suppose Δ is a valid database and Σ is a well-formed context (relative to Δ). If $\Sigma; \cdot \vdash_i s : \alpha$, then there is an s' such that $(i, s)^\sharp = (i, s')^\sharp$ and $\cdot; \cdot \vdash_i s' : \alpha$.*

[17] This diverges from the code a little because in the code Δ grows as declarations are made. However, the effect should be the same as if the database were not extended until the end.

[18] I'm covering something up. The rule that says it's OK to extend a signature by $c :_i^{(i,s)^\sharp} s$ if you have a proof of s isn't decidable. The way the algorithm actually works we do not need to ask if Σ is well-formed. It will be an invariant that Σ is well-formed.

Conjecture 2.9.3 *Suppose Δ is a valid database and Σ is a well-formed context (relative to Δ). If $\Sigma; \cdot \vdash_i \mathcal{D} : s$, then there exist $\mathcal{D}'$ and s' such that $(i, s)^\sharp = (i, s')^\sharp$ and $\cdot; \cdot \vdash_i \mathcal{D}' : s'$.*

We explain the idea for the proofs of Conjectures 2.9.2 and 2.9.3. Suppose $\Sigma; \cdot \vdash_i s : \alpha$. The process for computing $s^\sharp$ involves replacing the names from Σ with the corresponding global identifiers. Since Δ is valid and Σ is well-formed, for each global identifier there is a corresponding term. Hence we can form a nameless term s'' with no global identifiers by δ-expanding the global identifiers until there are none. Then s' would be some term which has s'' as its nameless representation.

Assume $\Delta = (\Delta_0, \Delta_1)$ is a valid database we start with before processing a document. After a document has been processed and a corresponding well-formed signature Σ has been created, we can extend Δ to be $\Delta' = (\Delta'_0, \Delta'_1)$ by taking Δ'_0 and Δ'_1 as follows:

$$\Delta'_0 := \Delta_0 \cup \{(m, \alpha) | c :_0^m \alpha := s \in \Sigma\}$$

and

$$\Delta'_1 := \Delta_1 \cup \{m | c :_i^m : s \in \Sigma\}.$$

We informally argue that Δ' is a valid database. The argument will not be a mathematical proof. It will will make use of the collision-resistance of cryptographic hashing functions. In addition, since we are arguing with named terms but the algorithms for obtaining global identifiers work with nameless terms, some level of imprecision is inevitable.

First we need to know Δ'_0 is a partial function. Assume not. That is, assume there exist m, α and β with $\alpha \neq \beta$, $(m, \alpha) \in \Delta'_0$ and $(m, \beta) \in \Delta'_0$. Since Δ_0 is a partial function, we cannot have both $(m, \alpha) \in \Delta_0$ and $(m, \beta) \in \Delta_0$. Without loss of generality, assume $(m, \alpha) \notin \Delta_0$ and so there must be some c and s with $c :_0^m \alpha := s \in \Sigma$. Since the signature is well-formed, we know $m = (0, s)^\sharp$.

We consider two cases. In the first case suppose $(m, \beta) \in \Delta_0$. Since Δ is valid, there is some monomorphic closed term t such that $\cdot; \cdot \vdash_0 t : \beta$ and $(0, t)^\sharp = m$. Since the global identifiers are determined from collision-resistant cryptographic hash functions, the only practical way we can have $(0, s)^\sharp = (0, t)^\sharp$ is if s and t are convertible.[19] By Conjecture 2.9.1 we conclude $\alpha = \beta$, a contradiction.

In the second case suppose $(m, \beta) \notin \Delta_0$. There must be some d and t with $d :_0^m \beta := t \in \Sigma$. Since the signature is well-formed, we know $m = (0, t)^\sharp$. Again using collision-resistance and Conjecture 2.9.1 we can almost certainly conclude $\alpha = \beta$, a contradiction.

Now we turn to proving Δ' is valid. Suppose $(m, \alpha) \in \Delta'_0$. If $(m, \alpha) \in \Delta_0$, then we know there is monomorphic closed term s such that $\cdot; \cdot \vdash_0 s : \alpha$ and $(0, s)^\sharp = m$

[19] Note that we are not saying all convertible terms have the same global identifier, which is certainly not true. We are saying that if the global identifiers are the same, then they resulted from hashing the serialization of the same terms. Since those terms went through some preprocessing (δ-contraction and $\beta\eta$-reduction), the original terms must be convertible.

by validity of Δ_0. Assume $(m, \alpha) \notin \Delta_0$. In this case there must be some c and s such that $c :_0^m \alpha := s \in \Sigma$ and $m = (0, s)^\sharp$. By Conjecture 2.9.2 there is an s' such that $m = (0, s')^\sharp$ and $\cdot; \cdot \vdash_i s' : \alpha$.

Finally suppose $m \in \Delta_1'$. If $m \in \Delta_1$, then there exist i, s and $\mathcal{D}$ such that $\cdot; \cdot \vdash_i \mathcal{D} : s$ and $(i, s)^\sharp = m$ by validity of Δ. Assume $m \notin \Delta_1$. There must be some $c :_i^m s \in \Sigma$. Since $m \notin \Delta_1$, this is only possible if there is some $\mathcal{D}$ such that $\Sigma; \cdot \vdash_i \mathcal{D} : s$. We obtain a $\mathcal{D}'$ and s' such that $\cdot; \cdot \vdash_i \mathcal{D}' : s'$ and $(i, s')^\sharp = m$ using Conjecture 2.9.3.

Chapter 3

Syntax, Parsing and Printing

There should have been another way.
The Doctor, *Warriors of the Deep*, Part Four (1984)

This is going to be painful. I hate writing parsers, and I always did it wrong. My parsers always had bugs. Even with parser generators I could never get rid of my shift-reduce conflicts. The version of the code I wrote during Winter 2012-2013 used Dijkstra's shunting yard algorithm to handle operators with priorities. I then extended it to handle binders. It worked most of the time, but I often found corner cases where it didn't work. Modifications would fix one thing but break another. Eventually I lost all hope.

Near the end of 2013 I decided to use Coq to specify a syntax for mathematical expressions which allows for binders, infix operators, and so on, write a pretty printer in Coq, write a parser in Coq and prove that the parser could correctly parse the output of the pretty printer. This would at least guarantees that every mathematical expression represented as an abstract tree can be linearized in a way that can be parsed back to the abstract tree. I'll refer to this as verifying correctness of the parser.

The devil is in the details of course, and it turned out there were a hell of a lot of details. It took months to complete and it takes Coq a long time to process the code. Probably some Coq wizards out there could have done it in a way Coq would be happier with. I found the experience so exhausting that I haven't wanted to revisit it, but to write this chapter I must. The main reason I did the Coq version is so that I would never need to debug a parser again.

Along with the source code for Egal I am releasing the Coq code verifying the parser. The parser for Egal is written in ocaml. The ocaml is not extracted from the Coq code so you would need to check by hand that the parser and printer for Egal correspond to what is proven correct in Coq.

In this chapter I will give the relevant syntax and the restrictions on the precedence levels for operators. Egal will signal an error if a user tries to assign an illegal precedence level. These restrictions are hypotheses in the Coq code (look for `Hypothesis` in the Coq file `ParsePrintFull3a.v`). Here are the restrictions:

1. Prefix operators and binders never have the same name.

2. No prefix operator has the same priority as a postfix operator or an infix operator.

3. No right associative infix operator has the same priority as a postfix operator or left associative or nonassociative infix operator.

4. No right associative infix operator has priority 500.

5. No prefix operator has priority 500.

6. All operators all have nonzero (positive) priority.

7. All operators have priority less than 1000.

At this point some of the restrictions seem arbitrary, but they are motivated by the following facts: the implicit infix operator (usually meaning application) is always left associative and has priority 0. Set membership $\in$ (in ASCII `:e` – following Mizar) and subset $\subseteq$ (in ASCII `c=` – following Mizar) both have priority 500.

Before continuing, I want to give a little example of a corner case. The mathematical expressions include the usual set theoretic notation. For example, we can express sets formed by separation as follows:

$$\{x \in X | \phi(x)\}.$$

This means the set of all elements x of X satisfying the condition $\phi(x)$. Similarly we can express sets formed by replacement as follows:

$$\{f(x) | x \in X\}.$$

This means the set of all $f(x)$ as x ranges over X.

Now, suppose $*$ is an infix operator and consider the expression

$$\{x \in X * a | y \in Y\}.$$

This has two possible parsings. The more natural one (to me) is to consider $*$ as an operation on sets (so a must be a set) and $\{x \in X * a | y \in Y\}$ corresponds to separation. In this case, the set is either $X * a$ if $y \in Y$ or empty if $y \notin Y$. However, it is equally legal for $*$ to be an operator which takes the proposition $x \in X$ and combines it with the value a to give a set. In that case, $\{x \in X * a | y \in Y\}$ corresponds to replacement and is the singleton set $\{x \in X * a\}$ if Y is nonempty or empty if Y is empty.

These two cases could be distinguished using typing information about the $*$ operation, but I decided trying to make the parser depend on the types of operators would complicate things even further.

The same goes for $\subseteq$ since I also have notation for

$$\{x \subseteq X | \phi(x)\}$$

and

$$\{f(x)|x \subseteq X\}.$$

Let me give some pointers as to how this ambiguity is resolved. The semantic parsing rule corresponding to replacement sets (see Figure 3.5) looks like this:

$$\frac{S_{1010}\,\bar{l}\,L \qquad S_{500}\,\overline{l'}\,L' \qquad S'_q\,\overline{r}\,(\widehat{\mathsf{Rep}}\,x\,\delta\,L\,L')\,M}{S_q(\mathtt{LCBRACE} :: \bar{l} + \mathtt{VBAR} :: \mathtt{NAM}(x) :: \delta :: \overline{l'} + \mathtt{RCBRACE} :: \overline{r})\,M}\ x \in \textsc{Nam},\ \delta \in \{\in, \subseteq\},\ L \notin \textsc{SetInfix}$$

The side condition that $L \notin \textsc{SetInfix}$ (L is not a set infix operator) exactly rules out these two cases:

$$\{. \in .|.\}$$

$$\{. \subseteq .|.\}$$

That is, if it looks like this, then it's separation, not replacement. I'm oversimplifying. If there is an infix operator $*$ with priority higher than 500, then

$$\{. \in . * .|.\}$$

would be seen as

$$\{(. \in .) * .|.\}$$

and must be replacement after all. Without going into more details (which I would need to dive into myself at this point), I will simply assert that the Coq proof is enough to know there isn't ambiguity. If you want to understand the verified parser in detail, look at the Coq code and try to figure out how this ambiguity is avoided.

There are, of course, differences between the Coq code and the ocaml code. In the ocaml code one is parsing a stream of tokens and one prints to a stream. In the Coq code I used a list of tokens. That is, the print function in Coq sends an abstract expression (or something like it) to a list of tokens. In Coq the parser maps a list of tokens to (optionally) an abstract expression and a remaining list of tokens to parse. Also, Coq only accepts structurally recursive functions. Tobias Tebbi (a former coworker of mine in Prof. Smolka's Lehrstuhl) had a techinique to make a function structurally recursive that doesn't appear to be structurally recursive. [1] My use of this idea was with a "Tebbi sublist" function named `tsubl` that can be found in the Coq file `Prelim2.v`. In the Coq implementation of the parser in the file `ParsePrintFull3a.v` I often needed to wrap arguments to recursive calls with calls to `tsubl` so that Coq would see that the function is structurally recursive. Of course, I don't do this in the ocaml code.

3.1 Quick Guide to the Code

There are two versions of the code: the Coq version and the ocaml version. The Coq code is in the `verifiedparser` subdirectory. Here is a guide to the Coq files:

[1] In the particular case of the gcd function, he used what we called a "Tebbi min function". I remember he posted this to the Coq club mailing list sometime in 2013, but I can't find it now.

- `Prelim2.v` imports parts of the Coq library and defines some generic functions for a type of tokens. The most important function is `tsubl`[2] which takes two lists of tokens l_1 and l_2 where l_2 should be a sublist of l_1 (i.e., l_2 can be obtained by popping enough elements from l_1) and returns l_2 by popping enough elements of l_1. While this seems to simply return l_2, it returns it in a way that Coq can see it is a structural subterm of l_1. Hence recursive calls using an argument `tsubl` $l_1\,l_2$ may be accepted by Coq where simply writing l_2 would not be accepted. Other than this definition, the file mainly consists of useful facts I didn't find in the Coq library. Many of them probably are in the Coq library. Sometimes if it's easier for me to prove something than it is to just state and prove it, I'll just state and prove it. I'm told this is bad form or something. Oh well. Maybe I'll get fired.

- `ParsePrintFull3a.v`: This imports `Prelim2` and is the file that defines the printer and parser, as well as a number of other things. The good news is that the file does have a lot of comments. I wrote them almost a year ago. Some of them are up-to-date and correct and some of them are not. If I were still able to distinguish between the correct ones and incorrect ones, I'd update them. Here are some of the main definitions and theorems in the file:

  - `SetInfixOp`: a type of two possible set theoretic infix operators: `InfMem` corresponding to $\in$ and `InfSubq` corresponding to $\subseteq$.
  - `InfixOp`: a type of infix operators which may either be one of the two set infix operators or an arbitrary name. (The fact that the name is of an infix operator will be in the "environment" which comes later.)
  - `AscKind`: an inductive type for three types of ascriptions for binders: `AscTp` for a colon (:), `AscSet` for $\in$ and `AscSubeq` for $\subseteq$.
  - `ATree`: This is the inductive type of "abstract trees". These are essentially the mathematical expressions in a tree form. More information is given in Section 3.2.
  - `BinderMid`: This is an inductive type of the only two kinds of separators for binders I support: `BinderMidComma` for a comma and `BinderMidDarr` for $\Rightarrow$.
  - `LTree` and `LTreeL`: These are mutually inductive types of "layout trees" and "lists of layout trees." This is very similar to abstract trees, but differs in a few respects. The way I think of it is as follows: layout trees make all their parentheses explicit and so there's no need to know priorities of operators. More information is given in Section 3.2.
  - `PICase`: This is an inductive case for the different kinds of postfix and infix operators. Prefix operators are handled differently from postfix and infix operators.

[2] This is the "Tebbi sublist" function mentioned above, the idea for which is due to Tobias Tebbi.

- `ParseEnv`: This is the type of a parsing environment. It maps names to a triple. The first component of the triple tells whether the name is a prefix operator and gives its priority if it is. The second component of the triple tells whether the name is either a postfix or infix operator (with some associativity) and gives its priority if it is. The last component tells if the name is a binder and gives whether the binder uses comma or ⇒ as a separator. Note that one name might be all three, in principle.[3] As a concrete example, consider the minus sign which can both be a prefix operator and an infix operator. That is the kind of case I wanted to make sure I support. A name cannot be both an infix and a postfix operator.
- `penv`: This is a variable of type `ParseEnv` and gives a fixed parsing environment for the remainder of the development.
- There are a number of hypotheses giving restrictions on the parsing environment. These restrictions were described earlier.
- `supp`: This is an inductive relation which defines when an abstract tree is *supported* by the parsing environment. If it is, then it can be laid out to a layout tree.
- `Binderishp`: This is a predicate on layout trees to check if the tree is *binderish* and essentially looks to see if the topmost visible thing on the layout tree is a binder (including let) or if-then-else after traversing through prefix operators and to the right of infix operators. More information is given in Section 3.2.
- `suppL` and `suppLL`: These are mutually defined inductive relations to check if the layout tree or list of layout trees is supported by a parsing environment with a given priority. Since the parentheses are already in layout trees, this relation essentially checks that if parentheses are needed then they're already there. More information is given in Section 3.2.
- `AL` : An inductive relation which essentially means a layout tree is a possible layout for an abstract tree. It means an abstract tree and a layout tree are more or less the same, up to parentheses.
- `L2A` : A function from layout trees to abstract trees which erases parentheses. This is an oversimplification. If the parentheses correspond to a tuple like (x, y, z), then it will map to a tuple abstract tree (x', y', z'). Layout trees do not distinguish between ordinary parentheses and tuples. Abstract trees only have tuples and do not have parentheses.
- `supp_L2A`: If a layout tree L is supported, then the abstract tree `L2A` L is supported.

[3] The type allows a name to be all three. One of the restrictions mentioned earlier is that a name cannot be both a prefix operator and a binder.

- AL_L2A: If a layout tree L is supported, then L is a possible layout for the abstract tree L2A L in the sense of AL.
- AL_A2L_exp: Every supported abstract tree A can be laid out as a layout tree L which is supported for some priority. In words, there's some way to put in enough parentheses.
- AL_A2L_allp: The previous result also holds for all priorities.
- AL_inj: Every layout tree corresponds to at most one abstract tree.
- TOKEN: This is the type of tokens.
- pReln_S'_, pReln_S_, pReln_TVs, pReln_N: These are four mutually defined relations which relate a list of tokens to other information. The idea is that the relations define when a list of tokens corresponds to one of the categories S', S, TVs or N and relates the list of tokens to its intended interpretation. A rough grammar for this is given in Section 3.4. The relations are given in Section 3.5.
- print_S and print_SL: This prints a layout tree/list of layout trees. It does not depend on the parsing environment because the parentheses have already been determined.
- print_S__thm: If a layout tree L is supported with priority p, then print_S applied to L returns a list of tokens which corresponds to the input according to pReln_S_ relative to every priority q with $q \geq p$.
- parse_Names: take a list of tokens and parse as many name tokens (that are not operators or binders) as possible and return the list of names and the remaining tokens.
- parse_S'_Infix: This is a helper function for parsing infix operators.
- parse_TVs_ascr: This is a helper function for parsing variables with ascriptions. This is used to handle things like

  ```
  forall (X Y : set) (x1 x2 :e X) (Y1 c= Y), body
  ```
- parse_Binder_ascr: This is a helper function to parse binders.
- parse_Let_ascr: This is a helper function to parse lets.
- parse_S_SetBraces: This is a helper function for parsing mathematical expressions like $\{x_1, \ldots, x_n\}$, $\{x \in X|p(x)\}$, $\{f(x)|x \in X\}$ and $\{f(x)|x \in X, p(x)\}$.
- parse_S'_: This is for parsing a list of tokens for the S' category relative to a priority. If it succeeds, it returns a layout tree and a list of leftover tokens. If it fails, it returns None.
- parse_S_: This is for parsing a list of tokens for the S category relative to a priority. If it succeeds, it returns a layout tree and a list of leftover tokens. If it fails, it returns None.

- parse_TVs: This is for parsing a list of tokens for the TVs category. If it succeeds, then it returns a list of lists of strings, possibly ascribed a layout tree (using one of the three ascription possibilities), along with the remaining tokens. If it fails, it returns None. To give the idea, calling it with input like this:

  ```
  (X Y : set) (x1 x2 :e X) (Y1 c= Y), body
  ```

  should result in returning something like this:

  $$\texttt{Some}\,([([X,Y],\texttt{Some}(\texttt{AscTp},set)),([x1,x2],\texttt{Some}(\texttt{AscSet},X)),$$

  $$([Y1],\texttt{Some}(\texttt{AscSubeq},Y))],[,body])$$

- parse_N: This parses list of tokens to obtain a list of layout trees, using commas as separators.

- ParsePrintFull3b.v, ParsePrintFull3bb.v, ParsePrintFull3bc.v, ParsePrintFull3bd.v, ParsePrintFull3be.v, ParsePrintFull3c.v, ParsePrintFull3cb.v, ParsePrintFull3d.v, ParsePrintFull3db.v: These files each contain one or more lemmas which is needed to prove the final correctness results. In principle these could all be in one file. In practice, Coq would die if I didn't separate them. As it is, many of them take a long time for Coq to compile. [4]

- ParsePrintFull3e.v: This has the final correctness results:

  - parse_S_lem uses mutual induction to prove that if the mutually inductive relations defined in ParsePrintFull3a.v hold and certain other conditions hold, then the parser returns the expected result. Almost every case of the big inductive proof is one of the lemmas from the previous files.

  - print_parse_S__id: This is a version of the main correctness result with an explicit priority. If L is a layout tree supported by priority q, then the parser called with priority q on the result of printing L yields L. It follows from the previous lemma.

  - print_parse_S_id: This is a version of correctness with the "highest" priority of 1023 which follows directly from the previous result.

In the ocaml code, one can look at syntax.ml, syntax.mli, parser.ml, parser.mli and lexer.mll.

Here are a few pointers into syntax.ml and syntax.mli:

- atree: This is the type of abstract trees.

[4] I don't know why this is true. Maybe someone could ask on the Coq club mailing list. I would, but I don't use email, and there's no option for anything other than email to communicate with the Coq club mailing list.

- `ltree`: This is the type of layout trees.
- `binderishp`: This is the test for whether a layout tree is binderish.
- `output_ltree`: This outputs a layout tree in an obvious way and corresponds to `print_S` in the Coq code.
- `ltree_to_atree`: This erases a layout tree to an abstract tree and corresponds to `L2A` in the Coq code.
- `picase`: This corresponds to the type `PICase` in the Coq code and simply gives the four cases for postfix and infix operators.

Here are a few pointers into `parser.ml` and `parser.mli`:

- `token` is the data type of tokens (which is used in `lexer.mll`) and corresponds to `TOKEN` in the Coq code.
- `parseenv` corresponds to `ParseEnv` in the Coq code.
- `tokenstream` is a datatype which helps me treat the stream of tokens more like a list.
- `destr_ts` is a helper function that splits the token stream into the head and the rest.
- `parse_S_` is the main parser function.
- `atree_to_ltree` maps abstract trees to layout trees and depends on the parsing environment. The parsing environment is implicit here and consists of a number of hash tables for looking up information about names as operators and binders.

3.2 Mathematical Expression Trees

Parsing maps a list or stream of tokens to an appropriate tree structure. In this section we define *abstract trees* as a target tree structure. In addition we define *layout trees*. The layout trees have additional information making them largely independent of the parsing environment. It is easy to erase this information to obtain an abstract tree from a layout tree. Conversely, an abstract tree can be "laid out" to a layout tree under reasonable conditions.

The following are *special symbols*: $=$ (`=`), $\neq$ (`<>`), $\notin$ (`/:e`), $\not\subseteq$ (`/c=`), $\neg$ (`~`), $+$, $*$, $\hat{}$, $-$, $!$, $\exists!$ (`exists!`), $\wedge$ (`/\`), $\vee$ (`\/`), $\wedge_{_}$ (`/_`), $\vee_{_}$ (`\/_`), $'$, $>$, $<$, $\leq$ (`<=`), $\geq$ (`>=`), $\rightarrow$ (`->`), $\leftarrow$ (`<-`), $\leftrightarrow$ (`<->`) and $\Leftrightarrow$ (`<=>`).

A *quoted symbol* is a sequence of characters from the characters `_+-*^~=<>/\` surrounded either by `'` characters or by `:`. For example, `'~*~'` and `:\/:` are quoted symbols.

A *general name* is either a special symbol, quoted symbol, or an alphanumeric identifier (possibly with underscores) beginning with a letter or an underscore.

We do all of this in the assumption that we have a parsing environment. A parsing environment determines if a general name is a prefix operator, infix operator (with associativity left, right or none), postfix operator, or a binder. It may be more than one of these or none of these. In the case of operators, the parsing environment also gives the corresponding priority p with $0 < p < 1000$. Using the parsing environment, we can define the following sets:

$$\textsc{Pre}_p := \{x \text{ general name} | x \text{ prefix operator with priority } p\}$$

$$\textsc{Inf}_p^{\mathrm{N}} := \{x \text{ general name} | x \text{ infix operator with no associativity and priority } p\}$$

$$\textsc{Inf}_p^{\mathrm{L}} := \{x \text{ general name} | x \text{ infix operator with left associativity and priority } p\}$$

$$\textsc{Inf}_p^{\mathrm{R}} := \{x \text{ general name} | x \text{ infix operator with right associativity and priority } p\}$$

$$\textsc{Post}_p := \{x \text{ general name} | x \text{ postfix operator with priority } p\}$$

$$\textsc{Bind}^{,} := \{x \text{ general name} | x \text{ binder expecting a comma to separate variables from the body}\}$$

$$\textsc{Bind}^{\Rightarrow} := \{x \text{ general name} | x \text{ binder expecting } \Rightarrow \text{ to separate variables from the body}\}$$

$$\textsc{Nam} := \{x \text{ general name} | x \notin \textsc{Bind}^{,} \cup \textsc{Bind}^{\Rightarrow} \cup (\bigcup_p (\textsc{Pre}_p \cup \textsc{Inf}_p^{\mathrm{N}} \cup \textsc{Inf}_p^{\mathrm{L}} \cup \textsc{Inf}_p^{\mathrm{R}}))\}$$

That is, $x \in \textsc{Nam}$ if x is a general name that is not an operator and not a binder.

In addition there are the two symbols $\in$ (`:e`) and $\subseteq$ (`c=`). These are not general names, but in certain contexts are considered infix operators of priority 500 with no associativity. To handle these operators, we define

$$\textsc{Inf}_{500}^{\mathrm{N}*} := \textsc{Inf}_{500}^{\mathrm{N}} \cup \{\in, \subseteq\}.$$

If $p \neq 500$, we define

$$\textsc{Inf}_p^{\mathrm{N}*} := \textsc{Inf}_p^{\mathrm{N}}$$

We now give a grammar defining the set of *abstract trees*. This corresponds to the inductive type `ATree` in the Coq code and the type `atree` in the ocaml code (`syntax.ml` and `syntax.mli`). We use A and B to range over abstract trees. We use x and y to range over general names. We use z to range over general names and $\in$ and $\subseteq$. We also use n to range over "numbers" (meant to represent many cases including 1, -2, 3.14, and so on, but numbers are handled the same say as names for parsing purposes, so I won't go into detail). We use $^?$ to indicate optional arguments. We use δ to range over $\{\in, \subseteq\}$ and use ε to range over $\{:, \in, \subseteq\}$.

$$
\begin{array}{rcl}
A, B & ::= & x \\
& | & n \\
& | & (\mathsf{Let}\, x\, (\varepsilon, A)^?\, A_1\, A_2) \\
& | & (\mathsf{Bi}\, x\, [(y_0, (\varepsilon_0, B_0)^?) \cdots (y_{n-1}, (\varepsilon_{n-1}, B_{n-1})^?)]\, A) \\
& | & (\mathsf{Pr}\, x\, A) \\
& | & (\mathsf{Po}\, x\, A) \\
& | & (\mathsf{Inf}\, z\, A_1\, A_2) \\
& | & (\mathsf{Implicit}\, A_1\, A_2) \\
& | & (\mathsf{Sep}\, x\, \delta\, A_1\, A_2) \\
& | & (\mathsf{Rep}\, x\, \delta\, A_1\, A_2) \\
& | & (\mathsf{SepRep}\, x\, \delta\, A_1\, A_2\, A_3) \\
& | & (\mathsf{SEnum}\, [A_0\, \cdots\, A_{n-1}]) \\
& | & (\mathsf{MTuple}\, A\, [A_0\, \cdots\, A_{n-1}]) \\
& | & (\mathsf{Tuple}\, A\, B\, [A_0\, \cdots\, A_{n-1}]) \\
& | & (\mathsf{If}\, A_0\, A_1\, A_2)
\end{array}
$$

Before continuing we give the intended meaning of each of these case.

- x: This is a general name which should satisfy $x \in \textsc{Nam}$ (that, it should not be an operator or binder).

- n: This is just a number. Ignore it.

- $(\mathsf{Let}\, x\, (\varepsilon, A)^?\, A_1\, A_2)$: This is a local let definition of x to be the value A_1 in the body A_2. Optionally x can be ascribed a "type" A. We should have $x \in \textsc{Nam}$. There are three possible ascriptions depending on $\varepsilon \in \{:, \in, \subseteq\}$. Depending on this, A might be a type or might be a set. I'm not sure the checking code supports any of this, but the parser does. Actually, I thought I had included the possibility that x has the form of a metatuple of variables so that it could act like a match for metatuples, but this doesn't appear in the Coq code. Ah, it is in the ocaml code. See `LeM` the constructor for `atree` in the ocaml code. I'll leave this out here because it's not in the Coq code.

- $(\mathsf{Bi}\, x\, [(y_0, (\varepsilon_0, B_0)^?) \cdots (y_{n-1}, (\varepsilon_{n-1}, B_{n-1})^?)]\, A)$ This is a binder and presumes $x \in \textsc{Bind}^{,} \cup \textsc{Bind}^{\Rightarrow}$. Each name y_i should be in $\textsc{Nam}$ and can optionally be ascribed a type or set. The body of the binder is A.

- $(\mathsf{Pr}\, x\, A)$: This is a prefix operator and presumes $x \in \textsc{Pre}_p$ for some p.

- $(\mathsf{Po}\, x\, A)$: This is a postfix operator and presumes $x \in \textsc{Post}_p$ for some p.

- $(\mathsf{Inf}\, z\, A_0\, A_1)$: This is an infix operator and presumes $z \in \textsc{Inf}_p^{\mathrm{N}*} \cup \textsc{Inf}_p^{\mathrm{L}} \cup \textsc{Inf}_p^{\mathrm{R}}$ for some p.

- $(\mathsf{Implicit}\, A_0\, A_1)$: This is the implicit (empty) infix operator. Usually this means A_0 applied to A_1, though there is some flexibility regarding this. If A_0 and A_1 are both interpreted to be sets, then the user can specify what the implicit infix

operation means. The implicit infix operator is always left associative and has priority 0 (meaning it necessarily binds more tightly than everything else).

- $(\mathsf{Sep}\, x\, \delta\, A_1\, A_2)$: This is a set formed by separation. It presumes $x \in$ NAM. In mathematical notation, if δ is $\in$, then we write

$$\{x \in A_1 | A_2\}$$

which is the set of all x in the set A_1 satisfying A_2. If δ is $\subseteq$, it means

$$\{x \subseteq A_1 | A_2\}.$$

- $(\mathsf{Rep}\, x\, \delta\, A_1\, A_2)$: This is a set formed by replacement. It presumes $x \in$ NAM. In mathematical notation, if δ is $\in$, then we write

$$\{A_1 | x \in A_2\}$$

meaning the set of all sets A_1 as x ranges over the set A_2. We also support where δ is $\subseteq$:

$$\{A_1 | x \subseteq A_2\}.$$

- $(\mathsf{SepRep}\, x\, \delta\, A_1\, A_2\, A_3)$: This is a combination of separation and replacement. It presumes $x \in$ NAM. In mathematical notation, if δ is $\in$, then we write

$$\{A_1 | x \in A_2, A_3\}$$

which is the set of all sets A_1 as x ranges over the set A_2 and x satisfies A_3. We also support where δ is $\subseteq$:

$$\{A_1 | x \subseteq A_2, A_3\}.$$

- $(\mathsf{SEnum}\, [A_0 \cdots A_{n-1}])$: This is a set formed as a finite enumeration of its elements. $\{A_0, \cdots, A_{n-1}\}$. If $n = 0$, it is $\{\}$ which is a way of writing the empty set.

- $(\mathsf{MTuple}\, A\, [A_0 \cdots A_{n-1}])$: This represents a "metatuple" with at least 1 component. We write this as $[A]$ or $[A, A_0]$ or $[A, A_0, A_1, A_2]$, and so on. I included metatuples for the case where all the components have the same simple type. In that case there is a known way to represent the tuple at a higher type so that β-reduction can play the role of projections. In the Coq version there's no nice way to get represent "projections" for metatuples, but in the ocaml code this is handled by lets with metatuples instead of variables (so one can write `let` $[x, y] :=$ $\cdots$ `in` $\cdots$ in Egal). This is the `LeM` case in the ocaml code, which we ignore here.

- $(\mathsf{Tuple}\, A\, B\, [A_0 \cdots A_{n-1}])$: This represents a tuple with at least 2 components. Mathematically we would write it as (A, B) or (A, B, A_0, A_1).

- $(\mathsf{If}\, A_0\, A_1\, A_2)$: This corresponds to if-then-else with A_0 as the condition, A_1 as the "then" clause and A_2 as the "else" clause.

In the Coq code (again, `ParsePrintFull3a.v`), there is an inductive predicate `supp` on abstract trees. This defines when an abstract tree is "supported" (by the parsing environment, which remains fixed here). Without giving details, the idea is that the predicate ensures that all the "presumptions" above are true. For example, $(\mathsf{Pr}\, x\, A)$ is supported if there is some p such that $x \in \textsc{Pre}_p$ and A is supported.

We now give a grammar defining the set of *layout trees*. These are very similar to abstract trees and one can think of them simply as the version of the abstract tree which is ready to be printed. The main difference between the two kinds of trees is that instead of Tuple nodes, layout trees have $\widehat{\mathsf{Paren}}$ nodes to indicate where parentheses should be. Where Tuple nodes must have two or more children, $\widehat{\mathsf{Paren}}$ nodes might only have one child. Another difference is that binder nodes in layout trees explicitly include whether the separator token should be a comma or $\Rightarrow$.

Layout trees correspond to the inductive type `LTree` in the Coq code and the type `ltree` in the ocaml code (`syntax.ml` and `syntax.mli`). In the Coq code the type is defined mutually with `LTreeL`, a type of lists of layout trees. There are different ways to define these kinds of types. I don't know why I did it this way or if it was a good decision.

We use L and M to range over layout trees. We use $\overline{L}$ and $\overline{M}$ to range over lists of layout trees. The conventions for x, y, z, n, δ and ε is the same as for abstract trees. We additionally use μ to range over the two symbols , (comma) and $\Rightarrow$. These are the two possible separators for binders.

$$
\begin{array}{rcl}
L, M & ::= & x \\
& | & n \\
& | & (\widehat{\mathsf{Let}}\, x\, (\varepsilon, L)^?\, L_1\, L_2) \\
& | & (\widehat{\mathsf{Bi}}\, x\, \mu\, [(y_0, (\varepsilon_0, M_0)^?) \cdots (y_{n-1}, (\varepsilon_{n-1}, M_{n-1})^?)]\, L) \\
& | & (\widehat{\mathsf{Pr}}\, x\, L) \\
& | & (\widehat{\mathsf{Po}}\, x\, L) \\
& | & (\widehat{\mathsf{Inf}}\, z\, L_1\, L_2) \\
& | & (\widehat{\mathsf{Implicit}}\, L_1\, L_2) \\
& | & (\widehat{\mathsf{Sep}}\, x\, \delta\, L_1\, L_2) \\
& | & (\widehat{\mathsf{Rep}}\, x\, \delta\, L_1\, L_2) \\
& | & (\widehat{\mathsf{SepRep}}\, x\, \delta\, L_1\, L_2\, L_3) \\
& | & (\widehat{\mathsf{SEnum}}\, [L_0\, \cdots\, L_{n-1}]) \\
& | & (\widehat{\mathsf{MTuple}}\, L\, [L_0\, \cdots\, L_{n-1}]) \\
& | & (\widehat{\mathsf{Paren}}\, L\, [L_0\, \cdots\, L_{n-1}]) \\
& | & (\widehat{\mathsf{If}}\, L_0\, L_1\, L_2)
\end{array}
$$

We define when layout tree is *binderish*. In the Coq code this is `Binderishp` and in the ocaml code this is `binderishp`. It is defined by recursion on layout trees, but the

idea is that a binder or let or if-then-else is visible by looking through prefix operators and to the right of infix operators.

- $\widehat{\mathsf{Bi}}$, $\widehat{\mathsf{Let}}$ and $\widehat{\mathsf{If}}$ nodes are binderish.
- $(\widehat{\mathsf{Pr}}\, x\, L)$ is binderish if L is binderish.
- $(\widehat{\mathsf{Inf}}\, x\, L\, M)$ is binderish if M is binderish.

Note that $\widehat{\mathsf{Paren}}$-nodes are not binderish. Sometimes algorithms add a $\widehat{\mathsf{Paren}}$-node with one child just to prevent the child from being binderish.

There is also a notion of a layout tree being "supported". In this case, there is more to check than for an abstract tree since the priorities play an important role. The Coq relation is `suppL` and depends on a natural number (the priority) and a layout tree. We say L is *supported with priority* q if this relation holds. The reader can look at the Coq code to see the full definition. To give the idea, we show three cases here:

- $(\widehat{\mathsf{Bi}}\, x\, [(y_0, (\in, L_0))]\, \mu\, L)$ is supported with priority q if $q > 0$, $x \in \textsc{Bind}^{\mu}$, $y_0 \in \textsc{Nam}$ and L_0 and L are supported with priority 1010.
- $(\widehat{\mathsf{Pr}}\, x\, L)$ is supported with priority q if there is some $p < q$ such that $x \in \textsc{Pre}_p$ and L is supported with priority $p + 1$.
- $(\widehat{\mathsf{Po}}\, x\, L)$ is supported with priority q if there is some $p < q$ such that $x \in \textsc{Post}_p$, L is not binderish and L is supported with priority $p + 1$.

If L is supported with priority p, then it is supported with priority q for all $q \geq p$. See `suppL_leq` in the Coq code.

One can define a relation between abstract trees and layout trees which essentially says L is a *possible layout for* A. In the Coq code this is defined as an inductive relation `AL`.

There is an obvious recursive algorithm for mapping layout trees into abstract trees. See `L2A` in the Coq code and `ltree_to_atree` in the ocaml code (in `syntax.ml`). For $\widehat{\mathsf{Bi}}$-nodes, the algorithm creates a Bi-node by forgetting μ. For $\widehat{\mathsf{Paren}}$-nodes, the node is dropped if there is one child and the node becomes a Tuple-node if there is more than one. It is easy to prove that if the layout tree was supported with a priority, then the abstract tree will be supported. See the lemma `supp_L2A` in the Coq code. Also, the layout tree L is a possible layout for the abstract tree `L2A` L. See the lemma `AL_L2A` in the Coq code. Every layout tree is a possible layout for at most one abstract tree. See the result `AL_inj` in the Coq code.

Printing a layout tree is trivial because we have already decided how to lay it out (e.g., where all the parentheses will be). The main printing work consists of "laying out" an abstract tree.

In the ocaml code this is the function `atree_to_ltree` in the files `parser.ml` and `parser.mli`. The reason it is in the parser code instead of the syntax code is because

the parsing environment is required to layout an abstract tree. For example, we must look up priorities of operators to determine if a parenthesis is required.

There is no corresponding function in the Coq code. Instead there are existential proofs which essentially contain the algorithms in their proofs. One auxiliary lemma which is helpful is `AL_A2L_aux` which says that if A is an abstract tree, L is a layout tree supported with priority p and L is a possible layout for A, then forall q there is a layout tree L' supported with priority q where L' is also a possible layout for A. The proof is simple: Use L if $q \geq p$ and use $(\widehat{\mathsf{Paren}}\, L\, [])$ otherwise.

The two main results are are `AL_A2L_exp` and `AL_A2L_allp`. The meat of the layout algorithm is in the proof of `AL_A2L_exp`. The statement of `AL_A2L_exp` says that if A is supported, then there is some priority p and some layout tree L such that L is supported with priority p and L is a possible layout for A. This is proven by induction, relying on `AL_A2L_aux` to add parentheses when needed. The `AL_A2L_allp` says there is a layout for every priority p, which easily follows from `AL_A2L_exp` and `AL_A2L_aux`.

3.3 Tokens

Figure 3.1 displays a list of tokens is given in ASCII, in mathematical notation, as a named token in the ocaml code (see the data type `token` in `parser.ml` and `parser.mli` and its use in `lexer.mll`) and as a named token in the Coq code (see the inductive type `TOKEN` in the Coq file `ParsePrintFull3a.v`). There are more tokens in the ocaml code to handle commands and so on. Here we're just looking at mathematical expressions, since this is where the nontrivial parsing happens.

As mentioned before, printing a layout tree is easy. We recursively traverse the layout tree and generate the appropriate outputs. By "appropriate output" we mean print ASCII in the ocaml code and returning a list of tokens in the Coq code. The ocaml code is `output_ltree` in `syntax.ml` and `syntax.mli`. (There is also a function `output_ltree_html` if you want an html version. That's how I made the content for the mathgate.info web pages.) In the Coq code the function is `print_S` which takes a layout tree and returns a list of tokens. Here is a case to give the idea:

$$\texttt{print_S}(\widehat{\mathsf{Bi}}\, x\, \mu\, [[y], \texttt{None}]\, L) = \texttt{NAM}(x) :: \texttt{NAM}(y) :: \mu' :: (\texttt{print_S}\, L)$$

Here μ' is the token for comma if μ is a comma or the token for $\Rightarrow$ if μ is $\Rightarrow$.

3.4 There Is No Grammar

During the development process I wrote many grammars. Here I will give two which correspond, more or less, to the implemented parser. Looking at the Coq code, however, it is clear that there actually is no grammar. A grammar would correspond to a collection of inductive predicates on lists of tokens (one predicate for each category). Instead of having such a collection of predicates, there is a collection of inductive relations which relate lists of tokens to layout trees (more or less – more details will come

description	ASCII	math	ocaml	Coq
special symbol or alphanumeric name starting with _ or a letter	x	x	NAM(x)	NAM(x)
number maybe negative maybe with a decimal maybe with an exponent	*[-]n[.m][Ek]*	$[-]n[.m][Ek]$	NUM(neg?,n,?m,?k)	NUM(neg?,n,?m) (not in Coq version)
	(	(	LPAREN	LPAREN
	)	)	RPAREN	RPAREN
	,	,	COMMA	COMMA
	:	:	COLON	COLON
	=>	$\Rightarrow$	DARR	DARR
	let	let	LET	LET
	:=	:=	DEQ	DEQ
	in	in	IN	IN
	:e	$\in$	MEM	MEM
	c=	$\subseteq$	SUBEQ	SUBEQ
	|	|	VBAR	VBAR
	[	[	LBRACK	LBRACK
	]	]	RBRACK	RBRACK
	{	{	LCBRACE	LCBRACE
	}	}	RCBRACE	RCBRACE
	if	if	IF	IF_
	then	then	THEN	THEN
	else	else	ELSE	ELSE

Figure 3.1: Tokens

later). In some cases the layout trees are used in the definition of the inductive relation. Hence there is no obvious way to "erase" the layout trees from the definition and simply obtain a traditional grammar.

Having said that, we give these grammars anyway. They might make things clearer.

We first give a simple category U which is a list of names which are not binders or operators.

$$U ::= (empty)|xU(\text{where } x \in \textsc{Nam})$$

We first approximate the grammar we target in Figure 3.2 using three categories, one of which depends on a priority: S_q, TVs and N. A priority q is a number between 0 and 1023.

We can remove left recursion by adding another category $S'(q)$ and modifying the category S_q. This is shown in Figure 3.3 and is a grammar closer to what is implemented.

3.5 Semantic Parsing Relations

Instead of having a grammar, there are relations between lists of tokens and other information for each category. The relations say that if one is trying to parse the tokens for a category, possibly with some auxiliary information, then it should succeed and the relation gives the appropriate value the parser should return.

We start with something simple. Recall the category U:

$$U ::= (empty)|xU(\text{where } x \in \textsc{Nam})$$

We will use $\bar{l}$ and $\bar{r}$ to range over lists of tokens. We use $\overline{x}$ to range over lists of names. We can define the following corresponding relation $U\,\bar{l}\,\overline{x}$ with two rules:

$$\frac{}{U\,[]\,[]} \qquad\qquad \frac{U\,\bar{l}\,\overline{x}}{U\,(\texttt{NAM}(x) :: \bar{l})(x :: \overline{x})}\ x \in \textsc{Nam}$$

In the Coq code, this is the inductive relation `pReln_Names`.

The lemma `pReln_Names_map` ensures that when we print a list of names from Nam, then the resulting list of tokens is U-related to the list of names. That is,

$$U\,(\texttt{map}\,\texttt{NAM}\,\overline{y})\,\overline{y}$$

assuming every $y \in \overline{y}$ is in Nam.

A corresponding parsing function is `parse_Names` which takes a list $\bar{l}$ of tokens and returns a pair $(\overline{x}, \bar{r})$ where $\overline{x}$ is a list of names from Nam and $\bar{r}$ is a list of tokens (a sublist of $\bar{l}$, the leftover tokens). This function simply traverses $\bar{l}$ as long as the head of the list is a name from Nam and when it no longer is it returns the names collected and the leftover tokens.

Let $+$ denote the operation of appending lists. It is easy to see that if $U\,\bar{l}\,\overline{x}$ and $\bar{r}$ is not a list with some $x \in \textsc{Nam}$ at the head, then `parse_Names` $(\bar{l} + \bar{r}) = (\overline{x}, \bar{r})$. This is

$$
\begin{array}{lcll}
S_q & ::= & x & (x \in \textsc{Nam}) \\
 & \mid & n & (n \text{ is a number}) \\
 & \mid & x\, S_{p+1} & (x \in \textsc{Pre}_p,\ p < q) \\
 & \mid & S_{p+1}\, x & (x \in \textsc{Post}_p,\ p < q) \\
 & \mid & S_p\, z\, S_p & (z \in \textsc{Inf}_p^{\mathrm{N}*},\ p < q) \\
 & \mid & S_{p+1}\, x\, S_p & (x \in \textsc{Inf}_p^{\mathrm{L}},\ p < q) \\
 & \mid & S_p\, x\, S_{p+1} & (x \in \textsc{Inf}_p^{\mathrm{R}},\ p < q) \\
 & \mid & S_1\, S_0 & (q > 0) \\
 & \mid & x\, y\, U\, \mu\, S_{1010} & (q > 0,\ y \in \textsc{Nam},\ \mu \text{ is , or } \Rightarrow \text{ and } x \in \textsc{Bind}^{\mu}) \\
 & \mid & x\, TVs\, \mu\, S_{1010} & (q > 0,\ \mu \text{ is , or } \Rightarrow \text{ and } x \in \textsc{Bind}^{\mu}) \\
 & \mid & \texttt{let}\, x\, \varepsilon\, S_{1010} := S_{1010}\, \texttt{in}\, S_{1010} & (q > 0,\ \varepsilon \in \{:, \in, \subseteq\} \text{ and } x \in \textsc{Nam}) \\
 & \mid & \texttt{let}\, x := S_{1010}\, \texttt{in}\, S_{1010} & (q > 0 \text{ and } x \in \textsc{Nam}) \\
 & \mid & \{x \delta S_{500} | S_{1010}\} & (x \in \textsc{Nam},\ \delta \in \{\in, \subseteq\}) \\
 & \mid & \{S_{1010} | x \delta S_{500}\} & (x \in \textsc{Nam},\ \delta \in \{\in, \subseteq\}) \\
 & \mid & \{S_{1010} | x \delta S_{500}, S_{1010}\} & (x \in \textsc{Nam},\ \delta \in \{\in, \subseteq\}) \\
 & \mid & \{\} & \\
 & \mid & \{S_{1023} N\} & \\
 & \mid & [S_{1023} N] & \\
 & \mid & (S_{1023} N) & \\
 & \mid & \texttt{if}\, S_{1010}\, \texttt{then}\, S_{1010}\, \texttt{else}\, S_{1010} & \\
TVs & ::= & \text{(empty)} & \\
 & \mid & (x\, U)\, TVs & (x \in \textsc{Nam}) \\
 & \mid & (x\, U\, \varepsilon\, S_{1010})\, TVs & (x \in \textsc{Nam},\ \varepsilon \in \{:, \in, \subseteq\}) \\
N & ::= & \text{(empty)} & \\
 & \mid & ,\ S_{1023}\ N &
\end{array}
$$

Figure 3.2: Not the grammar

S'_q	$::=$	(empty)	
	$\mid$	$x\, S'_q$	$(x \in \textsc{Post}_p,\, p < q)$
	$\mid$	$z\, S_p\, S'_q$	$(z \in \textsc{Inf}^{\mathrm{N}*}_p \cup \textsc{Inf}^{\mathrm{L}}_p,\, p < q)$
	$\mid$	$x\, S_{p+1}$	$(x \in \textsc{Inf}^{\mathrm{R}}_p,\, p < q)$
	$\mid$	$S_0\, S'_q$	$(q > 0)$
S_q	$::=$	$x\, S'_q$	$(x \in \textsc{Nam})$
	$\mid$	$n\, S'_q$	(n is a number)
	$\mid$	$x\, S_{p+1}\, S'_q$	(x is prefix with priority $p < q$)
	$\mid$	$(S_{1023}\, N)\, S'_q$	
	$\mid$	$x\, y\, U\, \mu\, S_{1010}$	($q > 0$, $y \in \textsc{Nam}$, μ is , or $\Rightarrow$ and $x \in \textsc{Bind}^{\mu}$)
	$\mid$	$x\, TVs\, \mu\, S_{1010}$	($q > 0$, μ is , or $\Rightarrow$ and $x \in \textsc{Bind}^{\mu}$)
	$\mid$	$\mathtt{let}\, x\, \varepsilon\, S_{1010} := S_{1010}\, \mathtt{in}\, S_{1010}$	$(q > 0,\, x \in \textsc{Nam})$
	$\mid$	$\mathtt{let}\, x := S_{1010}\, \mathtt{in}\, S_{1010}$	$(q > 0,\, x \in \textsc{Nam})$
	$\mid$	$\{x \delta S_{500} \mid S_{1010}\}\, S'_q$	$(x \in \textsc{Nam},\, \delta \in \{\in, \subseteq\})$
	$\mid$	$\{S_{1010} \mid x \delta S_{500}\}\, S'_q$	$(x \in \textsc{Nam},\, \delta \in \{\in, \subseteq\})$
	$\mid$	$\{S_{1010} \mid x \delta S_{500}, S_{1010}\}\, S'_q$	$(x \in \textsc{Nam},\, \delta \in \{\in, \subseteq\})$
	$\mid$	$\{\}\, S'_q$	
	$\mid$	$\{S_{1023} N\}\, S'_q$	
	$\mid$	$[S_{1023} N]\, S'_q$	
	$\mid$	$(S_{1023} N)\, S'_q$	
	$\mid$	$\mathtt{if}\, S_{1010}\, \mathtt{then}\, S_{1010}\, \mathtt{else}\, S_{1010}$	
TVs	$::=$	(empty)	
	$\mid$	$(x\, U)\, TVs$	$(x \in \textsc{Nam})$
	$\mid$	$(x\, U\, \varepsilon\, S_{1010})\, TVs$	$(x \in \textsc{Nam},\, \varepsilon \in \{:, \in, \subseteq\})$
N	$::=$	(empty)	
	$\mid$	$,\ S_{1023}\ N$	

Figure 3.3: Also not the grammar

the lemma parse_Names_lem in the Coq code. This is the essential "correctness" lemma we need for the parser for U. Actually, I think correctness should be the following:

$$\texttt{parse_Names}\,(\texttt{map}\,\texttt{NAM}\,\overline{x}) = (\overline{x}, [])$$

Of course, there would need to be an assumption that every $x \in \overline{x}$ is in NAM. I don't seem to have an explicit result stating this version of correctness, but it follows from the lemma using $\texttt{map}\,\texttt{NAM}\,\overline{x}$ for $\bar{l}$ and nil for $\bar{r}$.

The example of the category U should indicate where we are going. For each category there will be a relation relating a list of tokens to its intended parsing. We will then write a parsing function for that category and prove that if something is in the relation, then the parser will obtain the intended result.

At a high level, this is what happens in the Coq code. The relations are pReln_S'_, pReln_S_, pReln_TVs and pReln_N are defined by mutual induction in ParsePrintFull3a.v. In the cases of pReln_S'_ and pReln_S_ the first argument is a priority q. We write the q as a subscript in these two cases and so write the relations here as S'_q, S_q, TVs and N. Here is more information:

- $S'_q\,\bar{l}\,L\,M$ relates a list of tokens $\bar{l}$ to two layout trees L and M. The idea is that we assume L represents what has been parsed so far. Building on L, $\bar{l}$ should be parsed to give M.

- $S_q\,\bar{l}\,L$ relates a list of tokens $\bar{l}$ to a layout tree L. The idea is that $\bar{l}$ should parse to give L.

- $TVs\,\bar{l}\,[(\overline{x_0}, (\varepsilon_0, L_0)^?), \cdots, (\overline{x_{n-1}}, (\varepsilon_{n-1}, L_{n-1})^?)]$ means $\bar{l}$ is a list of tokens specifying a list of (bound) variables possibly with ascriptions. In practice, this is what may follow a binder.

- $N\,\bar{l}\,\overline{L}$ means $\bar{l}$ parses to a list of layout trees, each of which is separated by a comma in $\bar{l}$.

There are two new auxiliary definition used in the definition: a predicate E_q on lists of tokens and a set SETINFIX of certain layout trees.

We define SETINFIX to be the set of all $\widehat{\mathsf{Inf}}$-nodes where the infix operator is $\in$ or $\subseteq$.

$$\text{SETINFIX} := \{(\widehat{\mathsf{Inf}}\,\delta\,L\,M) | \delta \in \{\in, \subseteq\}\}$$

E_q is a test to see if we should stop parsing $\bar{l}$ for S'_q. We call it $E_q(\bar{l})$. It is S'__endp in the Coq code. Looking at it now, I think it's easier to say when $E_q(\bar{l})$ is false (i.e., when we should continue parsing $\bar{l}$ for S'_q). $E_q(\bar{l})$ does *not* hold if $q > 0$, $\bar{l}$ is nonempty and one of the following holds:

- The head of $\bar{l}$ is NAM(x) where either $x \in$ NAM or $x \in \text{INF}^{\text{N}}_p \cup \text{INF}^{\text{L}}_p \cup \text{INF}^{\text{R}}_p$ with $p < q$.

- The head of $\bar{l}$ is a number.
- The head of $\bar{l}$ is a left parenthesis, left curly brace or left bracket (i.e., one of `({[`).
- The $q > 500$ and the head of $\bar{l}$ is $\in$ or $\subseteq$.

The rules defining the four relations are given in Figures 3.4 and 3.5. In the rules we use the ocaml notation to specify the tokens except for the following: If μ is comma or $\Rightarrow$, we write μ as a token to mean `COMMA` or `DARR`, respectively. If $\delta \in \{\in, \subseteq\}$, we write δ as a token to mean `MEM` or `SUBEQ`. If $\varepsilon \in \{:, \in, \subseteq\}$, we write ε as a token to mean `COLON`, `MEM` or `SUBEQ`.

The following results (among others) are proven in `ParsePrintFull3a.v`:

- `pReln_S'_S'_leq`: If $q \leq q'$ and $S'_q\, \bar{l}\, L\, M$, then $S'_{q'}\, \bar{l}\, L\, M$.
- `pReln_S_S_leq`: If $q \leq q'$ and $S_q\, \bar{l}\, L$, then $S_{q'}\, \bar{l}\, L$.
- `pReln_S'__S'__app`: If $S'_q\, \bar{l}\, L\, M$, M is not binderish, $q' \geq q - 1$, $E_{q'}(\bar{r})$ and there is no right associative infix operator of priority q', then $S'_{q'+1}\, \bar{r}\, M\, M'$ implies $S'_{q'+1}\, (\bar{l} + \bar{r})\, L\, M'$.
- `pReln_S__S'__app`: If $S_q\, \bar{l}\, L$, L is not binderish, $q' \geq q - 1$, $E_{q'}(\bar{r})$ and there is no right associative infix operator of priority q', then $S'_{q'+1}\, \bar{r}\, L\, M$ implies $S_{q'+1}\, (\bar{l} + \bar{r})\, M$.
- `pReln_S_InfixRight`: If $x \in \textsc{Inf}^{\mathrm{R}}_p$, $S_p\, \bar{l}\, L$, $S_{p+1}\, \bar{r}\, M$ and L is not binderish, then $S_{p+1}\, (\bar{l} + \mathtt{NAM}(x)\, \bar{r})\, (\widehat{\mathsf{Inf}}\, x\, L\, M)$.
- `print_S_Bindvars`: Printing an appropriate list of bound variable info gives something TVs-related to the input.
- `print_S__main`: If L is a p supported layout tree, then $S_p\, (\mathtt{print_S}\, L)\, L$. The lemma also gives information about the N-relation as lists of 1023 supported layout trees.
- `print_S__thm`: If L is a p supported layout tree and $p \leq q$, then $S_q\, (\mathtt{print_S}\, L)\, L$.

3.6 Parser

The parser is defined in the Coq code near the end of `ParsePrintFull3a.v`. The main functions are `parse_S'_`, `parse_S_`, `parse_TVs` and `parse_N` which are defined by mutual recursion. The definition is preceeded by a number of definitions to handle special cases. These helper functions are also in the ocaml code in `parser.ml`.

In the code you find many arguments of the form $(\mathtt{tsubl}\, \bar{l}\, \bar{r})$. These are the uses of the Tebbi sublist function so that Coq can tell the parser is structurally recursive. The value of $(\mathtt{tsubl}\, \bar{l}\, \bar{r})$ is the same as $\bar{r}$.

$$\frac{}{S'_q\,[]\,L\,L} \qquad \frac{S'_q\,\bar{l}\,(\widehat{\mathsf{Po}}\,x\,L)\,M}{S'_q\,(\mathtt{NAM}(x) :: \bar{l})\,L\,M}\;x \in \textsc{Post}_p,\; q < p$$

$$\frac{S_p\,\bar{l}\,L' \qquad S'_q\,\bar{r}\,(\widehat{\mathsf{Inf}}\,z\,L\,L')\,M}{S'_q\,(\mathtt{NAM}(x) :: \bar{l} + \bar{r})LM}\;z \in \textsc{Inf}^{\mathrm{N}*}_p \cup \textsc{Inf}^{\mathrm{L}}_p,\; p < q,\; L' \text{ not binderish},\; E_p(\bar{r})$$

$$\frac{S_p\,\bar{l}\,M}{S'_q\,(\mathtt{NAM}(x) :: \bar{l})\,L\,(\widehat{\mathsf{Inf}}\,x\,L\,M)}\;x \in \textsc{Inf}^{\mathrm{R}}_p,\; p < q,\; M \text{ binderish}$$

$$\frac{S_0\,\bar{l}\,L' \qquad S'_q\,\bar{r}\,(\widehat{\mathsf{Implicit}}\,LL')\,M}{S'_q\,(\bar{l} + \bar{r})\,L\,M}\;q > 0 \qquad \frac{S'_q\,\bar{l}\,x\,L}{S_q\,(\mathtt{NAM}(x) :: \bar{l})\,L}\;x \in \textsc{Nam}$$

$$\frac{S'_q\,\bar{l}\,n\,L}{S_q\,(\mathtt{NUM}(n) :: \bar{l})\,L} \qquad \frac{S_{p+1}\,\bar{l}\,L}{S_q\,(\mathtt{NAM}(x) :: \bar{l})\,(\widehat{\mathsf{Pr}}\,x\,L)}\;x \in \textsc{Pre}_p,\; p < q,\; L \text{ binderish}$$

$$\frac{S_{p+q}\,\bar{l}\,L \qquad S'_q\,\bar{r}\,(\widehat{\mathsf{Pr}}\,x\,L)\,M}{S_q\,(\mathtt{NAM}(x) :: \bar{l} + \bar{r})\,M}\;x \in \textsc{Pre}_p,\; p < q,\; L \text{ not binderish},\; E_p(\bar{r})$$

$$\frac{S_{1023}\,\bar{l}\,L \qquad N\,\overline{l'}\,\overline{L'} \qquad S'_q\,\bar{r}\,(\widehat{\mathsf{Paren}}\,L\overline{L'})}{S_q\,(\mathtt{LPAREN} :: \bar{l} + \overline{l'} + \mathtt{RPAREN} :: \bar{r})\,M}$$

$$\frac{U\,\bar{l}\,(y :: \bar{y}) \qquad S_{1010}\,\bar{r}\,L}{S_q\,(\mathtt{NAM}(x) :: \bar{l} + \mu :: \bar{r})\,(\widehat{\mathsf{Bi}}\,x\,\mu\,[(y :: \bar{y}, \mathtt{None})]\,L)}\;q > 0,\; x \in \textsc{Bind}^{\mu}$$

$$\frac{U\,\bar{l}\,\bar{y} \qquad S_{1010}\,\overline{l'}\,L \qquad S_{1010}\,\bar{r}\,M}{S_q\,(\mathtt{NAM}(x) :: \bar{l} + \mu :: \overline{l'} + \bar{r})\,(\widehat{\mathsf{Bi}}\,x\,\mu\,[(\bar{y}, (\mu, L))]\,M)}\;q > 0,\; x \in \textsc{Bind}^{\mu}$$

$$\frac{TV\,s\,\bar{l}\,\overline{V} \qquad S_{1010}\,\bar{r}\,L}{S_q\,(\mathtt{NAM}(x) :: \bar{l} + \mu :: \bar{r})\,(\widehat{\mathsf{Bi}}\,x\,\mu\,\overline{V}\,L)}\;q > 0,\; x \in \textsc{Bind}^{\mu}$$

$$\frac{S_{1010}\,\bar{l}\,L \qquad S_{1010}\,\bar{r}\,M}{S_q\,(\mathtt{LET} :: \mathtt{NAM}(x) :: \mathtt{DEQ} :: \bar{l} + \mathtt{IN} :: \bar{r})\,(\widehat{\mathsf{Let}}\,x\,\mathtt{None}\,L\,M)}\;q > 0,\; x \in \textsc{Nam}$$

$$\frac{S_{1010}\,\overline{l'}\,L' \qquad S_{1010}\,\bar{l}\,L \qquad S_{1010}\,\bar{r}\,M}{S_q\,(\mathtt{LET} :: \mathtt{NAM}(x) :: \mathtt{DEQ} :: \bar{l} + \mathtt{IN} :: \bar{r})\,(\widehat{\mathsf{Let}}\,x\,(\varepsilon, L')\,L\,M)}\;q > 0,\; \varepsilon \in \{:, \in, \subseteq\},\; x \in \textsc{Nam}$$

Figure 3.4: Semantic Parsing Relation Rules (1)

$$\frac{S_{500}\,\overline{l}\,L \qquad S_{1010}\,\overline{l'}\,L' \qquad S'_q\,\overline{r}\,(\widehat{\mathsf{Sep}}\,x\,\delta\,L\,L')\,M}{S_q(\texttt{LCBRACE} :: \texttt{NAM}(x) :: \delta :: \overline{l} + \texttt{VBAR} :: \overline{l'} + \texttt{RCBRACE} :: \overline{r})\,M}\; x \in \textsc{Nam},\ \delta \in \{\in, \subseteq\}$$

$$\frac{S_{1010}\,\overline{l}\,L \qquad S_{500}\,\overline{l'}\,L' \qquad S'_q\,\overline{r}\,(\widehat{\mathsf{Rep}}\,x\,\delta\,L\,L')\,M}{S_q(\texttt{LCBRACE} :: \overline{l} + \texttt{VBAR} :: \texttt{NAM}(x) :: \delta :: \overline{l'} + \texttt{RCBRACE} :: \overline{r})\,M}\; x \in \textsc{Nam},\ \delta \in \{\in, \subseteq\},\ L \notin \textsc{SetInfix}$$

$$\frac{S_{1010}\,\overline{l}\,L \qquad S_{500}\,\overline{l'}\,L' \qquad S_{1010}\,\overline{l''}\,L'' \qquad S'_q\,\overline{r}\,(\widehat{\mathsf{SepRep}}\,x\,\delta\,L\,L'\,L'')\,M}{S_q(\texttt{LCBRACE} :: \overline{l} + \texttt{VBAR} :: \texttt{NAM}(x) :: \delta :: \overline{l'} + \texttt{COMMA} :: \overline{l''} + \texttt{RCBRACE} :: \overline{r})\,M}\; x \in \textsc{Nam},\ \delta \in \{\in, \subseteq\},\ L \notin \textsc{SetInfix}$$

$$\frac{S'_q\,\overline{r}\,(\widehat{\mathsf{SEnum}}\,[])\,M}{S_q\,(\texttt{LCBRACE} :: \texttt{RCBRACE} :: \overline{r})\,M} \qquad \frac{S_1023\,\overline{l}\,L \qquad N\,\overline{l'}\,\overline{L'} \qquad S'_q\,\overline{r}\,(\widehat{\mathsf{SEnum}}\,(L :: \overline{L'}))\,M}{S_q\,(\texttt{LCBRACE} :: \overline{l} + \overline{l'} + \texttt{RCBRACE} :: \overline{r})\,M}$$

$$\frac{S_1023\,\overline{l}\,L \qquad N\,\overline{l'}\,\overline{L'} \qquad S'_q\,\overline{r}\,(\widehat{\mathsf{MTuple}}\,(L :: \overline{L'}))\,M}{S_q\,(\texttt{LBRACK} :: \overline{l} + \overline{l'} + \texttt{RBRACK} :: \overline{r})\,M}$$

$$\frac{S_{1010}\,\overline{l}\,L \qquad S_{1010}\,\overline{l'}\,L' \qquad S_{1010}\,\overline{l''}\,L''}{S_q\,(\texttt{IF} :: \overline{l} + \texttt{THEN} :: \overline{l'} + \texttt{ELSE} :: \overline{l''})\,(\widehat{\mathsf{If}}\,L\,L'\,L'')}\; q > 0 \qquad \frac{}{TVs\,[]\,[]}$$

$$\frac{U\,\overline{l}\,\overline{x} \qquad TVs\,\overline{r}\,\overline{V}}{TVs\,(\texttt{LPAREN} :: \overline{l} + \texttt{RPAREN} :: \overline{r})\,((\overline{x}, \texttt{None}) :: \overline{V})}$$

$$\frac{U\,\overline{l}\,\overline{x} \qquad S_{1010}\,\overline{l'}\,L \qquad TVs\,\overline{r}\,\overline{V}}{TVs\,(\texttt{LPAREN} :: \overline{l} + \varepsilon :: \overline{l'} + \texttt{RPAREN} :: \overline{r})\,((\overline{x}, (\varepsilon, L)) :: \overline{V})}\; \varepsilon \in \{:, \in, \subseteq\} \qquad \frac{}{N\,[]\,[]}$$

$$\frac{S_{1023}\,\overline{l}\,L \qquad N\,\overline{l'}\,\overline{L'}}{N\,(\texttt{COMMA} :: \overline{l} + \overline{l'})\,(L :: \overline{L'})}$$

Figure 3.5: Semantic Parsing Relation Rules (2)

The ocaml code is in `parser.ml`. The functions are also named `parse_S'_`, `parse_S_`, `parse_TVs` and `parse_N`. There is no need to use the Tebbi sublist function here.

The reader can check by hand that the Coq version and ocaml version correspond. That was the intention. I wrote the ocaml code following the Coq code and it's, of course, possible I made a mistake. I'm not going to check now.

Only the function `parse_S_` is provided in the interface file `parser.mli`.

You can get a good idea of what the parser does by examining the rules in Figures 3.4 and 3.5. These essentially define the functions as their graph. The parser functions return values of option type, where `None` means the list of tokens does not parse. If it returns a value, it also returns a list of remaining tokens to parse.

3.7 Correctness

The rest of the Coq code consists of the correctness proof. There are a few small helper definitions and lemmas at the end of `ParsePrintFull3a.v` after the parser is defined. For the most part though, the correctness proof boils down to proving `parse_S_lem` in `ParsePrintFull3e.v`. This lemma takes effort to state, so I won't try. It consists of four conjuncts, one for each of S'_q, S_q, TVs and N. It is proven by mutual induction and has a case for each rule in Figures 3.4 and 3.5. Most of the cases correspond to a lemma like `parse_S_lem_1`, `parse_S_lem_2`, and so on, each of which is stated and proven in one of the files `ParsePrintFull3b.v` `ParsePrintFull3bb.v` `ParsePrintFull3bc.v` `ParsePrintFull3bd.v` `ParsePrintFull3be.v` `ParsePrintFull3c.v` `ParsePrintFull3cb.v` `ParsePrintFull3d.v` and `ParsePrintFull3db.v`. The proofs for these lemmas are long and tedious. I don't want to look at them again.

After one has the main lemma above, one can use it to prove `print_parse_S__id` which says if L is q-supported, then

$$\texttt{parse_S_}\,q\,(\texttt{print_S}\,L) \;=\; \texttt{Some}(L, [])$$

This is the main thing I am calling "correctness of the parser," though probably "parser-printer coherence" would be a more appropriate name. Note that it doesn't mention the semantic parsing relations. Those relations were, of course, used to prove the result. The final result is `print_parse_S_id` which makes the result specific to $q = 1023$.

3.8 Conversion to Nameless Representation

After obtain a layout tree from the parser, we can easily convert it to an abstract tree (using `ltree_to_atree` in the Egal ocaml code). We then usually need to interpret this abstract tree as either a simple type, a simply typed term or a proof of a simply typed proposition. The code for doing this is in `interpret.ml`.

- `extract_tp`: Given an abstract tree and a list of names for type variables, return a nameless simple type or raise a `Failure` exception.

- `extract_tm`: Given an abstract tree along with signature and context information, return a nameless well-typed term and its nameless type. Otherwise, raise a `Failure` (or maybe `Not_found`).
- `check_tm`: Given an abstract tree and a nameless type, along with signature and context information, return a nameless well-typed term of the given type. Otherwise, raise a `Failure` (or maybe `Not_found`).
- `extract_pf`: Given an abstract tree along with signature and context information, return a nameless proof term and a nameless proposition (nameless term of type `prop`) that the proof term proves. Otherwise, raise a `Failure` (or maybe `Not_found`).
- `check_pf`: Given an abstract tree and nameless proposition, along with signature and context information, return a nameless proof term that proves the nameless proposition. Otherwise, raise a `Failure` (or maybe `Not_found`).

Chapter 4

Documents

For you, this is the beginning. Chapter one, paragraph one, as they say.
The Doctor, *The Trial of a Time Lord*, Part Four (1986)

Now that we have a way to specify types, terms and proofs, we can organize the presentation of them using documents. Documents consist of comments, document items and proof tactics. Comments were mentioned earlier: Text (without `*`) surrounded by `(**` and `**)` is skipped as comments. Content outside proofs are document items. Content inside proofs are proof tactics. The code to read and process documents is in the main file `mgcheck.ml`.

Again, many of the document items and proof tactics are very similar to those in Coq [33]. I even wrapped some of the document items in `(*` and `*)` so that Coq could (sometimes) read the same code.

In the descriptions we write L to refer to a layout tree. In practice it is given as ASCII which can be lexed to be a list of tokens $\bar{l}$ such that $S_{1023}\,\bar{l}\,L$, as defined in Chapter 3. We also use M to refer to layout trees. We use x and y for names. We use s and s' for strings. Strings are specified by wrapping text in `"`.

4.1 Document Items

Each possible *document item* is described below. Document items are parsed with the function `parse_docitem` in `parser.ml`.

- `(* Parameter` $x\,y$ `*)`
  This associates the name x with the global identifier y. This must be used before importing a previously defined object. It may also be used before giving a definition if you want to make sure the definition is the same as the one you intend (up to the names used).

- `Parameter` x `:` L`.`
  Declare the name x to be a constant of type L (i.e., the nameless type extracted

from L). If x is one of the primitive names (Eps, In, Empty, Union, Power, Repl, UnivOf), then it is an error to assign it anything other than its expected type (see Chapter 5). If x is not a primitive name, then it must have already been declared to correspond (via a global identifier) to an object previously defined (according to the database Δ, see Section 2.8). The database ensures that the same simple type is assigned to the name x as the global identifier has in the database. The information is put onto the signature Σ. (If there is a context Γ, then the actual type of x is the result of popping the types of variables from Γ, but throughout the rest of the section the name x will be used as if it is applied to the variables in Γ. This is similar to the way AUTOMATH and Coq behave.)

- `Definition` x `:=` M `.`
  This declares x to be defined by M. That is, a typed term t and its simple type α is extracted from M. If there are no type variables and no variables on Γ, then $x :_0^{(0,t)^\sharp} \alpha := t$ is added to Σ. If Γ is empty but there are i type variables, then $x :_i^{(i,t)^\sharp} \alpha := t$ is added to Σ. Suppose there are no type variables but Γ consists of one variable $y : \beta$. In this case what is added to Σ is

$$x :_0^{(0,\lambda:\beta.t)^\sharp} \beta \to \alpha := \lambda : \beta.t.$$

  In general, if the number of type variables are recorded and the variables from Γ are abstracted away before the declaration is added to Σ. Nevertheless, as long as these variables are in context, you can use the name x without explicitly applying it to the necessary type variables or typed variables, as with parameters.

- `Definition` $x : L$ `:=` M `.`
  In this case the intended type is made explicit. That is, a type α is extracted from L, a term t is extracted from M and checked to have type α. (Knowing the intended type α allows one to omit type annotations on initial λ-abstractions.) After this, Σ is extended as described above.

- `Axiom` $x : L$ `.`
  A proposition t is extracted from L. A proposition $\tilde{t}$ is constructed to be independent of the context Γ by

  1. popping each assumption $y : t'$ from Γ and changing the proposition t to $t' \to t$,
  2. popping each variable $y : \alpha$ from Γ and changing the proposition t to $\forall y : \alpha.t$ and
  3. doing something reasonable with $y : \alpha := t'$. (I think it makes a β redex like $(\lambda y : \alpha.t)t'$.)

  Egal checks if the axiom is allowed. It does this by computing $(i, \tilde{t})^\sharp$ where i is the number of type variables and checking if $(i, \tilde{t})^\sharp$ is either an axiom of the theory

(see Chapter 5) or is a previously proven result (according to the database Δ, see Section 2.8). In this case, $x :_i^{(i,\tilde{t})^\sharp} \tilde{t}$ is added to Σ.

- `Theorem` $x : L$ `.`
  A proposition t is extracted from L. Egal then switches into proof mode and begins reading proof tactics to construct a proof term. When the proof mode begins, it has a single goal which is to prove t in the context Γ. The proof mode remembers the name x as well as the version $\tilde{t}$ of t independent of Γ (computed as described for `Axiom`) and the corresponding global identifier m. After the proof is finished, $x :_i^m \tilde{t}$ is added to Σ and m can be added to the database. Even if the proof is admitted, $x :_i^m \tilde{t}$ is added to Σ. The database only contains fully proven theorems. The following are synonyms for `Theorem`: `Lemma`, `Example`, `Fact`, `Remark`, `Corollary`, `Proposition` and `Property`. They all behave the same way, but are rendered using the synonym in the html versions.

- `Section` x `.`
  This begins a section named x. Within a section you can declare local variables an hypotheses which are put onto Γ and are popped from Γ when the section ends.

- `End` x `.`
  This ends the current section, which should be named x.

- `Variable` $\overline{x} : L$. If L is `SType`, then this declares each variable in x to be a type variable. Otherwise, this declares each variable in x to be of the type extracted from L, adding each to Γ. Note: Instead of `:` you can use `:e` ($\in$) or `c=` ($\subseteq$) and it will parse. I never wrote the code to support these kinds of ascriptions though. It would have added a variable and a hypothesis to Γ.)

- `Hypothesis` $x : L$ `.`
  A proposition t is extracted from L and $x : t$ is added to Γ. After this, x can be used as a proof term proving t.

- `Let` x `:=` M `.`
  A typed term t of type α is extracted from M and $x : \alpha := t$ is added to Γ. Local definitions like this have not been thoroughly tested.

- `Let` $x : L$ `:=` M `.`
  In this case, a simple type α is extracted from L and this is used to extract a term t of type α from M. Then $x : \alpha := t$ is added to Γ. Again, using `:e` ($\in$) or `c=` ($\subseteq$) will parse, but the code to interpret it has not been written.

- `Infix` $x\, p$ `:=` L `.`
  This declares the name x to be an infix operator with priority p and no associativity which expands to be the term extracted from L.

- `Infix` x p `left :=` L `.`
  This declares the name x to be an infix operator with priority p and left associativity which expands to be the term extracted from L.

- `Infix` x p `right :=` L `.`
  This declares the name x to be an infix operator with priority p and right associativity which expands to be the term extracted from L.

- `Postfix` x p `:=` L `.`
  This declares the name x to be a postfix operator with priority p which expands to be the term extracted from L.

- `Prefix` x p `:=` L `.`
  This declares the name x to be a prefix operator with priority p which expands to be the term extracted from L.

- `Binder` x μ `:=` L `.`
  This declares the name x to be a binder with separator μ (which must be `,` or `=>`). The meaning of the binder is given by the term extracted from L.

- `Binder` x μ `:=` L `;` M `.`
  This declares the name x to be a binder with separator μ (which must be `,` or `=>`). The meaning of the binder is given by the terms extracted from L and M.

- `Binder+` x μ `:=` L `.`
  This declares the name x to be a binder which can bind several variables at once and uses separator μ (which must be `,` or `=>`). The meaning of the binder is given by the term extracted from L.

- `Binder+` x μ `:=` L `;` M `.`
  This declares the name x to be a binder which can bind several variables at once and uses separator μ (which must be `,` or `=>`). The meaning of the binder is given by the terms extracted from L and M.

- `Notation` x $\overline{y}$ `.`
  This handles a few special cases for notation: `IfThenElse`, `Repl`, `Sep`, `ReplSep`, `SetEnum`, `SetEnum0`, `SetEnum1`, `SetEnum2`, `Nat`, `SetLam` and `SetImplicitOp`. I'm not motivated to describe all of these. An example of each of them occurs in the treasure documents under the `formaldocs` directory, and you can probably determine what they mean by how they're used.

- `(* Unicode` x s_1 `...` s_n `*)`
  This associates the name x with a sequence of unicode symbols. I used this for creating html versions that I thought looked nice. It was probably a waste of time, so I'll delete it from the examples in this manual and not say anything more about it.

- (* Subscript x *)
  Wrap html `sub` tags around the html presentation of the name so it will be a subscript.

- (* Superscript x *)
  Wrap html `sup` tags around the html presentation of the name so it will be a superscript.

- (* Author s $\overline{s'}$ *)
  Specify the authors of the document.

- (* Title s *)
  Specify the title of the document.

- (* Treasure s *)
  Specify the bitcoin treasure address for the next proof. If it doesn't match the given proof (if there is one), Egal will tell you. If it does match the given proof, Egal will tell you the private key.

- (* Salt s *)
  Set a salt string which is used for computing bitcoin private keys and addresses.

- (* ShowProofTerms *)
  (* HideProofTerms *)
  These were supposed to turn on and off whether proof terms are shown in the html versions, but the variable they toggle (`showproofterms`) doesn't seem to be used.

The details of the semantics for `Binder` and `Binder+` are complicated. If you want to know more, look at the definition of `declare_binder` in `parser.ml` and the usage of `bindersem` in `interpret.ml`. There are comments in the code in `interpret.ml` that describe the 7 cases for declared binders.

4.2 Proof Tactics

While Egal is in proof mode, it reads proof tactics, proof braces and proof bullets. Proof braces are `{` and `}`. When the ending proof brace is encountered, a proof of precisely the subgoal which existed when the opening proof brace was encountered should be completed. Proof bullets are `-`, `+` and `*` and are used to keep up with which subgoal is being proven. Proof braces and proof bullets are intended to behave like they do in Coq, and as far as I know they do behave in Egal like they do in Coq.

The code to parse items in proof mode is found in `parse_pftacitem` in `parser.ml`.

Egal will leave proof mode in one of two ways:

- `Qed.`
  This means the proof is finished. If there were no admits in the proof, a complete proof term will be constructed and double checked. Nothing should go wrong. Famous last words.

- `Admitted.`
  This means you're not going to finish the proof now. All remaining subgoals are forgotten and the theorem is treated as if it were proven for the rest of the document.

The proof tactics are similar to ones in Coq. Some of them have the same name (e.g., `exact`, `apply` and `rewrite`). The tactics `let` and `assume` are both like `intros` in Coq. The tactic `prove` is like `change` in Coq. The tactic `claim` is like `assert` in Coq. The tactic `witness` is like `exists` in Coq.

Here are the proof tactics.

- `exact` L`.`
  Assume the current goal is t in context Γ. Egal tries to extract a proof term proving t from L. If it succeeds then this goal is popped off the stack of subgoals.

- `apply` L`.`
  Assume the current goal is t in context Γ. Egal extracts a proof term and a proposition t' from L. Egal tries to match t' with t, progressively going underneat universal quantifiers and implications to do so. If there is a match, then the current subgoal is removed from the stack and a subgoal is added to the stack for each of the antecedents of the implications in t' which were progressed through. If `apply` fails and you think it should succeed, it might mean you need to use the `prove` tactic first so that the current goal has a form similar to the proposition proven by the proof term given to `apply`.

- `let` $\overline{x}$ `:` L`.`
  Assume $\overline{x}$ consists of n names and a type α is extracted from L. This will only succeed if the current goal has the form

  $$\underbrace{\forall : \alpha \cdots \forall : \alpha}_{n} .t$$

  in a context Γ. The result will be to replace the current goal with t in context $\Gamma, \overline{x} : \alpha$.

- `let` $x_1 \, \ldots \, x_n$`.`
  In this case no type is given, so it can succeed more generally. This will succeed if the current goal has the form

  $$\forall : \beta_1 \cdots \forall : \beta_n .t$$

  in a context Γ. The result will be to replace the current goal with t in context $\Gamma, x_1 : \beta_1, \ldots, x_n : \beta_n$.

- `assume` $\overline{x}$ `:` L `.`
  Assume $\overline{x}$ consists of n names and a proposition t' is extracted from L. This will only succeed if the current goal has the form

$$\underbrace{t' \to \ldots t' \to}_{n} t$$

  in a context Γ. The result will be to replace the current goal with t in context $\Gamma, \overline{x} : t'$. Almost all the time, n will be 1 if `assume` is used this way.

- `assume` $x_1 \ldots x_n$ `.`
  In this case no proposition is given, so it can succeed more generally. This will succeed if the current goal has the form

$$t_1 \to \cdots t_n \to t$$

  in a context Γ. The result will be to replace the current goal with t in context $\Gamma, x_1 : t_1, \ldots, x_n : t_n$.

- `prove` L `.`
  This extracts a proposition t' from L. If the current goal is t, then Egal checks if t and t' are the same up to conversion. If they are, then the current goal is replaced by t'.

- `claim` x `:` L `.`
  Assume the current goal has context Γ. A proposition t is extracted from L. The context of the current goal is changed to be $\Gamma, x : t$. A new goal of proving t under Γ is pushed onto the goal stack. The effect of this is that you need to prove t under Γ and afterwards you can use t as a new hypothesis.

- `witness` L `.`
  If the current goal is existential, then a term is extracted from L and this term is used as the existential witness. This only works after a certain existential introduction lemma is known.

- `rewrite` L `.`
  Assuming L is the proof of a Leibniz equation (possibly universally quantified), use the equation to rewrite the current goal form right to left. This only works after symmetry of equality is known.

- `rewrite` L `at` $\overline{n}$ `.`
  Assuming L is the proof of a Leibniz equation (possibly universally quantified), use the equation to rewrite the current goal form right to left at the given positions. The positions count the subterms of the goal that are matched by the equation. This only works after symmetry of equality is known.

- `rewrite <-` L `.`
  Assuming L is the proof of a Leibniz equation (possibly universally quantified), use the equation to rewrite the current goal form left to right. Ironically, this works even before symmetry of equality is known.

- `rewrite <-` L `at` $\overline{n}$ `.`
  Assuming L is the proof of a Leibniz equation (possibly universally quantified), use the equation to rewrite the current goal form right to left at the given positions. The positions count the subterms of the goal that are matched by the equation. Again, this works even before symmetry of equality is known.

- `set` x `:=` M `.`
  This adds a local definition the current goal's context. **Warning:** It hasn't been used much, and may be buggy.

- `set` x `:` L `:=` M `.`
  This adds a local definition the current goal's context. **Warning:** It hasn't been used much, and may be buggy.

- `prove` L_0 `,` L_1 `,` $\cdots$ L_n `.`
  This parses, but the code to interpret it was never written. The idea is to split the current subgoal into several (explicitly given) subgoals, assuming Egal can figure out how to make the decomposition.

- `cases` L `:` ....
  This was never written beyond the parser. The idea was to figure out that L proves a disjunction (or something like a disjunction) and then use that disjunction to have several cases.

Chapter 5

The Foundation: Tarski-Grothendieck Set Theory

The whole of creation is very delicately balanced in cosmic terms, Jo.
The Doctor, *The Time Monster*, Episode Six (1972)

We can now turn to the theory we use as a foundation for mathematics. The theory is Tarski-Grothendieck set theory [44, 23]. This is the same foundational theory used in the Mizar Mathematical Library [45], although the set of theorems for Egal is larger since Egal is higher order and Mizar is (more or less) first-order. Like everything else, this doesn't really matter because no one is ever going to even notice that Egal exists.

The Egal document giving the foundation is `Foundation.mg` and can be found in the directory `formaldocs`. The only reason certain parameters and axioms can be included here even though the database is empty is because they are hard-coded into Egal as primitive names and axioms. For example, the primitive names for parameters can be found in the function `primname` in `syntax.ml`. There it associates the following names with an internal number for the "primitive" used in the `tm` type, the number of type variables they expect and their type. Here is this information.

- Eps: This corresponds to the nameless term $\mathtt{Prim}(0)$. It is the only polymorphic primitive and depends on one type. Its nameless type is $(0 \to \mathsf{prop}) \to 0$. Using the primitive name, a term of type $(\alpha \to \mathsf{prop}) \to \alpha$ can be written as the layout tree $\mathsf{Eps}\, L_\alpha$ where L_α is a layout tree corresponding to α. Usually, however, a binder notation is declared for Eps so that terms of the form $\mathsf{Eps}\, \alpha\, (\lambda\, x : \alpha.s)$ can be laid out as $\mathsf{some}\, x : L_\alpha, L_s$ where L_α corresponds to α and L_s corresponds to s.

- In: This corresponds to the nameless term $\mathtt{Prim}(1)$ and has type $\mathsf{set} \to \mathsf{set} \to \mathsf{prop}$. It is a priori associated with the infix notation `:e` (ASCII for $\in$). Likewise, the notation `c=` (ASCII for $\subseteq$) is a priori associated with $\lambda : \mathsf{set}.\lambda : \mathsf{set}.\forall : \mathsf{set}.0 \in 2 \to 0 \in 1$. (Actually, `c=` can also be used for any R and S of type $\beta_1 \to \cdots \to \beta_n \to \mathsf{prop}$.) These notations are hard coded because $\in$ and $\subseteq$ are used in several special ways in the syntax.

- Empty: This corresponds to the nameless term Prim(2) and has type set. It means the empty set, obviously, which is usually written as $\emptyset$.
- Union: This corresponds to the nameless term Prim(3) and has type set $\to$ set. Applying this function to a set X yields what is usually written as $\bigcup X$.
- Power: This corresponds to the nameless term Prim(4) and has type set $\to$ set. It corresponds to power sets. Applying this function to a set X yields what is usually written as $\wp X$.
- Repl: This corresponds to the nameless term Prim(5) and has type set $\to$ (set $\to$ set) $\to$ set. This corresponds to sets formed by replacement. A term like (Repl $s\,(\lambda x.t)$) corresponds to what is usually written as $\{t|x \in s\}$.
- UnivOf: This corresponds to the nameless term Prim(6) and has type set $\to$ set. The idea is that this takes a set and gives the least Grothendieck universe for which the set is a member.

Likewise, `mgcheck.mg` explicitly contains the global identifier for a collection of (sometimes polymorphic) propositions in the variable `indexknowns`.[1] If a proposition has one of the global identifiers, then it is allowed to be assumed as an axiom.

We walk through most of the document `Foundation.mg` here.

First we define the logical connectives and quantifiers we will need along with notation. More information about these are given in Section 6.1. We need them here in order to state axioms in a convenient way.

```
Definition False : prop := forall P:prop, P.

Definition not : prop -> prop := fun A:prop => A -> False.

Prefix ~ 700 := not.

Definition and : prop -> prop -> prop :=
 fun A B:prop => forall P:prop, (A -> B -> P) -> P.

Infix /\ 780 left := and.

Definition or : prop -> prop -> prop :=
 fun (A B : prop) => forall P:prop, (A -> P) -> (B -> P) -> P.

Infix \/ 785 left := or.

Definition iff : prop -> prop -> prop :=
```

[1] This is equivalent to always ensuring these knowns are in the database Δ.

```
 fun (A B:prop) => (A -> B) /\ (B -> A).

Infix <-> 805 := iff.

Section Eq.

Variable A:SType.

Definition eq : A->A->prop :=
 fun x y:A => forall Q:A->prop, Q x -> Q y.

End Eq.

Infix = 502 := eq.

Section Ex.

Variable A:SType.

Definition ex : (A->prop)->prop :=
 fun Q:A->prop => forall P:prop, (forall x:A, Q x -> P) -> P.

End Ex.

Binder+ exists , := ex.
```

We next give two extensionality principles as axioms. These are accepted as axioms because their global identifiers are hard coded into the Egal source code. The first axiom is propositional extensionality which states that two propositions are equal if they are equivalent. The second axiom is functional extensionality. Since it depends on two horrible type variables, one can also think of functional extensionality as a scheme of infinitely many axioms. It says that if α and β are types, and f and g are functions from α to β, then $f = g$ if $fx = gx$ for every x of type α. That is, two functions are equal if they give the same value for every input.

```
Axiom prop_ext : forall A B:prop, (A <-> B) -> A = B.

Section FE.

Variable A B:SType.

Axiom func_ext : forall (f g : A -> B), (forall x:A, f x = g x) -> f = g.

End FE.
```

We next turn to the choice operator. In this case we have a primitive Eps which takes a type α and gives a term of type $(\alpha \to \mathsf{prop}) \to \alpha$. The corresponding axiom (or axiom scheme) says that for each type α, if P is a predicate on α and x is an element of type alpha satisfying P, then Eps gives an element of type α satisfying P. The x is simply a witness that P is not the empty predicate; the value Eps gives does not depend on the witness x. Note that when the axiom is given the name Eps is not applied to the type A. The reason is because we are still in the section where the type variable A was declared. After the section is closed, every time Eps is used it must be explicitly given the type α as its first argument. It would not be hard to modify Egal to infer the type α in many cases, but whenever I faced such a choice I always chose the route of making polymorphism inconvenient to keep the code simpler and to discourage the use of polymorphism, which is evil.

```
Section Choice.

Variable A:SType.

Parameter Eps : (A->prop)->A.

Axiom EpsR : forall P:A->prop, forall x:A, P x -> P (Eps (fun x => P x)).

End Choice.
```

Next we define the binder notation so that we can write some $x : L_\alpha, L_s$ as mentioned above. In mathematical notation we write $\varepsilon x : \alpha.s$. Recall that the `,` means that we separate the specification of the variable(s) bound from the body using a comma. The only other alternative is `=>`. In the unlikely event that you are reading this manual linearly, you read about this in Section 4.1.

```
Binder some , := Eps.
```

The axioms above are at the level of the simple type theory, and are often included in versions of higher-order logic. Without these axioms, by default the simple type theory described in Chapter 2 is intuitionistic. The combination of extensionality and choice allow one to prove the logic is classical (see Section 6.2).

We next turn to the primitives and the axioms of set theory (Tarski-Grothendieck). The good news is that no more polymorphism is required.

First we declare the set membership relation using the primitive name In.

```
Parameter In:set->set->prop.
```

Notationally the infix operator `:e` (with priority 500 and no associativity) is automatically registered for this primitive as soon as it is declared. We do not explicitly give it in the code. (In fact, technically there is no way to declare it, since `:e` is, unlike most infix operators, lexically not a "name".) Again, `:e` comes from Mizar, and is ASCII for

$\in$. It's probably obvious, but assuming L_s and L_t are layout trees for terms s and t of type set, respectively, then L_s :e L_t is a layout tree for $\mathsf{In}\, s\, t$.[2]

We also define Subq for the subset relation, using the infix notation built into the code for set membership.

```
Definition Subq : set->set->prop :=
fun X Y => forall x:set, x :e X -> x :e Y.
```

The infix notation c= (corresponding to $\subseteq$) is hard-coded to correspond to this term when it is used with two terms of type set.

The first set theory axiom is set extensionality which states that two sets are equal if they have the same elements. Or, as it is actually stated here, if $X \subseteq Y$ and $Y \subseteq X$, then $X = Y$.

```
Axiom set_ext : forall X Y:set, X c= Y -> Y c= X -> X = Y.
```

The next axiom is an $\in$-induction principle. This corresponds to what is often called the axiom of foundation in set theory. It is a higher-order axiom since it involves quantification over a predicate P on sets, and these predicates P may also depend on quantification over predicates. There is a classically equivalent first-order presentation called regularity, which is given as a theorem in a the document EpsInd.mg. The induction principle says that if you want to prove P holds for all sets X, it is enough to prove it holds for a generic X under the assumption of an inductive hypothesis that P holds for every member of X. This axiom corresponds to thinking of the sets as being constructed iteratively, where a set is formed from elements constructed at a previous stage.

```
Axiom In_ind : forall P:set->prop,
 (forall X:set, (forall x:set, x :e X -> P x) -> P X)
 -> forall X:set, P X.
```

We now turn to set contructors. First there is the empty set. There is a corresponding axiom saying the empty set has no members.

```
Parameter Empty : set.

Axiom EmptyAx : ~exists x:set, x :e Empty.
```

Next there is the union operator. The corresponding axiom says that $x \in \bigcup X$ if and only if there is some Y such that $x \in Y$ and $Y \in X$.

[2] In some ways it would be preferable to reverse the arguments so that $x \in X$ would be $\mathsf{In}\, X\, x$ and so the η-short form of the predicate (or "class") corresponding to the set X would be $\mathsf{In}\, X$. I did this in Scunak. It was cool, but for some reason I decided not to do this in Egal. So the predicate corresponding to the set X is the ugly $\lambda x : \mathsf{set}.\mathsf{In}\, x\, X$.

```
Parameter Union : set->set.

Axiom UnionEq : forall X:set, forall x:set,
 x :e Union X <-> exists Y:set, x :e Y /\ Y :e X.
```

Next there is the power set operator. The corresponding axiom says $Y \in \wp X$ if and only if $Y \subseteq X$. That is, $\wp X$ is the set of all subsets of Y.

```
Parameter Power : set->set.

Axiom PowerEq : forall X Y:set, Y :e Power X <-> Y c= X.
```

Now we turn to the replacement operator which will be used to form sets like $\{s|x \in X\}$. The primitive name Repl is used to declare the operator with the appropriate type.

```
Parameter Repl : set->(set->set)->set.
```

We next use a special `Notation` declaration so that Egal knows how to interpret layout trees of the form $\{L_s|x \in L_t\}$. Literally we say to interpret the replacement notation using the parameter Repl. It looks like we're stuttering, but we're not. The first `Repl` says what kind of special notation we're declaring.

```
Notation Repl Repl.
```

Now that we have this notation, we can state the corresponding axiom in what I hoped was a readable way. It says $y \in \{F\,x|x \in X\}$ if and only if there is some $x \in X$ such that $y = Fx$.

```
Axiom ReplEq :
forall X:set, forall F:set->set, forall y:set,
 y :e {F x|x :e X} <-> exists x:set, x :e X /\ y = F x.
```

Finally we turn to Grothendieck universes. This is a strong form of the axiom of infinity that one finds in Zermelo-Fraenkel set theory.

We first define the notation of a transitive set. A set U is *transitive* if every member is a subset.

```
Definition TransSet : set->prop := fun U:set => forall X:set, X :e U -> X c= U.
```

A Grothendieck universe is a transitive set which is closed under the operators given above. We first define three definitions for what it means for U to be closed under each of these operators, and then conjoin them into the single definition ZF_closed.

```
Definition Union_closed : set->prop
 := fun U:set => forall X:set, X :e U -> Union X :e U.
Definition Power_closed : set->prop
```

```
 := fun U:set => forall X:set, X :e U -> Power X :e U.
Definition Repl_closed : set->prop
 := fun U:set => forall X:set, X :e U -> forall F:set->set,
   (forall x:set, x :e X -> F x :e U) -> {F x|x :e X} :e U.
Definition ZF_closed : set->prop := fun U:set =>
   Union_closed U
/\ Power_closed U
/\ Repl_closed U.
```

We now give the last primitive name: UnivOf. This is an operator which takes a set N and returns a Grothendieck universe U such that $N \in U$. The first axiom says $N \in$ UnivOf N. The second axiom says UnivOf N is transitive. The third axiom says UnivOf N is closed under the operations.

```
Parameter UnivOf : set->set.

Axiom UnivOf_In : forall N:set, N :e UnivOf N.

Axiom UnivOf_TransSet : forall N:set, TransSet (UnivOf N).

Axiom UnivOf_ZF_closed : forall N:set, ZF_closed (UnivOf N).
```

Finally we have one last axiom which states that Univ_Of N is the *least* Grothendieck universe containing N. That is, if U is a Grothendieck universe with $N \in U$, then UnivOf N is a subset of U. I don't think I used this axiom in any of the treasure hunt documents, but I remember using it in other developments. For example, one can use it to prove that UnivOf Empty is the set of hereditarily finite sets.

```
Axiom UnivOf_Min : forall N U:set, N :e U
  -> TransSet U
  -> ZF_closed U
  -> UnivOf N c= U.
```

And this all the material one needs to build up all the usual mathematical universe and even a bit more. I sometimes used the cool picture in Figure 5.1 to illustrate the mathematical universe. The cone represents the iterative conception of the set theoretic universe. Starting from the empty set, the set constructors allow one to fill up the first Grothendieck universe with all the hereditarily finite sets. After that one can fill up the next Grothendieck universe with all the sets one expects to find in Zermelo-Fraenkel set theory. And so on.

I don't know the young lady in the photo. That part of the photo is from Shutterstock. Well, I think the picture looks cool, but no one else has ever told me that. And certainly I've found over the years that hardly anyone shares my tastes in, well, much of anything.

At one time I had planned to help produce a series of math related images working with a model/artist I met in Poland in 2011. We lost touch and it never happened.[3] Hopefully she's involved in bigger and better projects and finding a lot of success. She was very talented.

[3] I lost touch with almost everyone from my past in 2012 after I realized I'm not human and have essentially nothing in common with the humans I used to know.

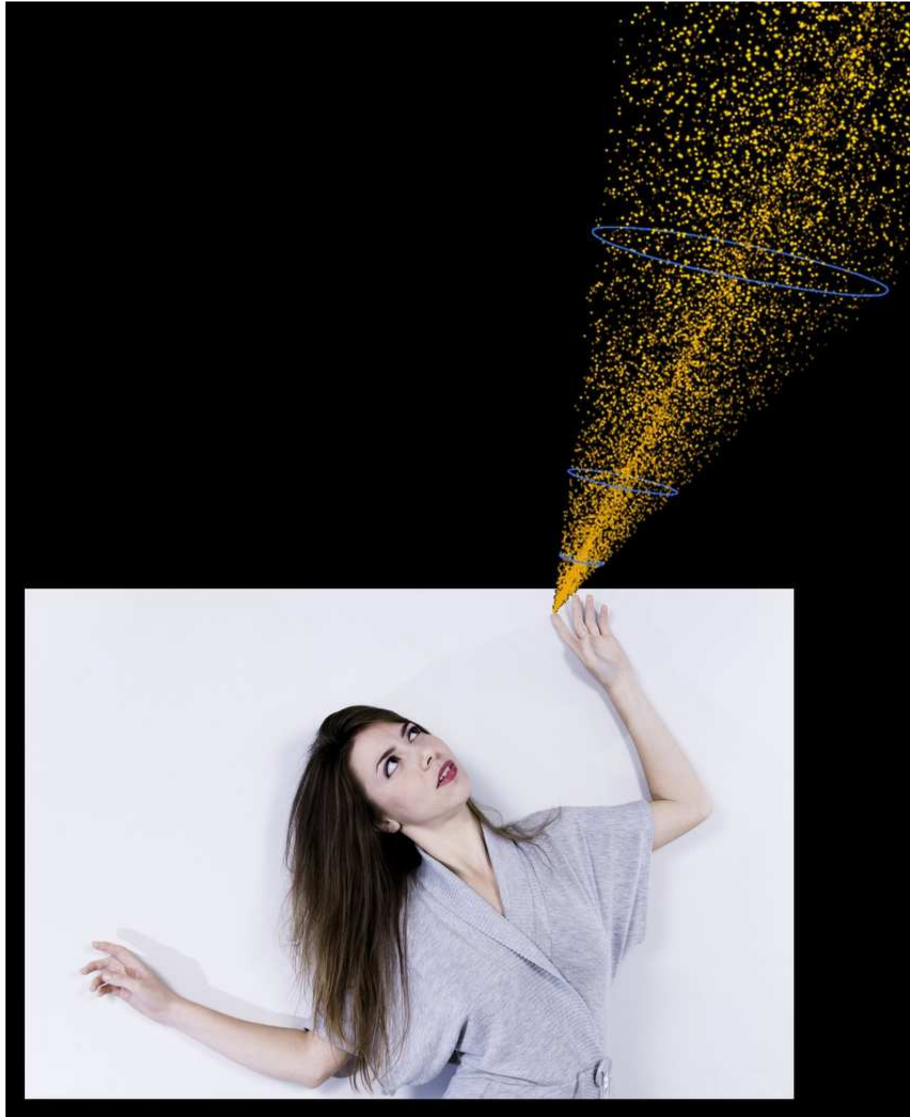

Figure 5.1: The Mathematical Universe

Chapter 6

Examples: The 2014 Treasure Hunt

Well done! I'm very glad to hear that you're bringing logic into your guesses.
The Doctor, *The Final Test* (*The Celestial Toymaker*, Episode 4) (1966)

The bitcoin theorem proving treasure hunt at mathgate.info took place from March 2014 until September 2014. There were 25 documents, each of which had bitcoin treasures which could be claimed by giving a certain proof of a theorem. For all but the first document there was also a preamble document importing previous objects and results for use in the new document. All these documents can be found in the `formaldocs` directory.

The purpose of the 25 documents is to provide many of the basic building blocks of mathematics, ultimately ending with pairs, functions, sets of pairs and sets of functions. We survey the 25 documents in this chapter pointing out what was supposed to be important. In some cases we present them in a different order than they were released in the treasure hunt.

6.1 Basic Logic

The two documents `PropLogic1.mg` and `ExEq1.mg` define the basic logical connectives and quantifiers, as well as equality. The definition of equality is often called Leibniz equality, because the idea that two things are equal if they share all the same properties is attributed to Leibniz. The definitions of conjunction, disjunction and existential quantification are sometimes called the Russell-Prawitz definitions. They are definitely in Prawitz [39] (in 1965, though the publication date here is given as 2006). If you look at the English in specific passages carefully enough you can find some of them in Russell [42] (in 1903).

We start with the file `PropLogic1.mg`. Since this file has no preamble signature file and imports nothing, it can be checked by calling Egal as follows:

```
egal PropLogic1.mg
```

Here are a few examples.

```
Definition False : prop := forall P:prop, P.
```

Here we have defined False by $\forall P : \mathsf{prop}.P$. That is, False holds if every proposition holds. In other words, False is the "least likely" proposition to be provable. We can make this precise by proving a lemma corresponding to an elimination principle for False: If we have a proposition P and a proof of False, then we also have a proof of P. In mathematical notation, we write $\bot$ for False and so the propostion we want to prove is

$$\forall P : \mathsf{prop}.\bot \to P.$$

The proof term is easy:

$$\lambda P : \mathsf{prop}.\lambda H : \bot.HP.$$

The reader should examine the rules in Figure 2.2 to be convinced that the proof term proves the proposition. Egal can determine the ascription of prop for P and $\bot$ for H and so we can specify the proof term more concisely as $\lambda PH.HP$. Here is the Egal code.

```
Theorem FalseE : forall P : prop, False -> P.
exact (fun P H => H P).
Qed.
```

The definition of True is similar, except True is provable.

```
Definition True : prop := forall P:prop, P -> P.

Theorem TrueI : True.
exact (fun P p => p).
Qed.
```

We next define negation. In mathematical notation we write $\neg A$ for the negation of A when A is a proposition. By definition $\neg A$ is $A \to \bot$. In ASCII the $\neg$ prefix operator is ~.

```
Definition not : prop->prop := fun A:prop => A -> False.

Prefix ~ 700 := not.
```

We next define conjunction (and). If A and B are propositions, then $A \wedge B$ is notation for and $A\,B$. In ASCII we write /\ for $\wedge$. This infix operator is left associative.[1] By definition $A \wedge B$ is

$$\forall P : \mathsf{prop}.(A \to B \to P) \to P.$$

That is, if we know $A \wedge B$ and want to prove P, then it is enough to prove P under the assumptions of A and B. This is essentially defining conjunction based on its elimination principle.

[1] This is different from Coq, where conjunction is right associative. I had a reason for doing things this way, but I don't remember it now.

```
Definition and : prop->prop->prop :=
 fun A B:prop => forall P:prop, (A -> B -> P) -> P.

Infix /\ 780 left := and.
```

The first thing we prove is a lemma andI giving an introduction principle. Again, the exact proof term is given, and again it is left to the reader to check the proof using the rules in Figure 2.2.

```
Theorem andI : forall (A B : prop), A -> B -> A /\ B.
exact (fun A B a b P H => H a b).
Qed.
```

We next prove lemmas for left elimination and right elimination. The idea is that if we know $A \wedge B$, then we know A by one lemma and B by the other lemma. In practice, these two lemmas are not used very often. Usually, we simply apply the proof of the conjunction $A \wedge B$ and as a result are able to assume both A and B. Nevertheless, one of the two lemmas can be used if only one side of the conjunct is required.

```
Theorem andEL : forall (A B : prop), A /\ B -> A.
exact (fun A B H => H A (fun a b => a)).
Qed.

Theorem andER : forall (A B : prop), A /\ B -> B.
exact (fun A B H => H B (fun a b => b)).
Qed.
```

Disjunction (or) is also defined using an elimination principle. In this case we write $A \vee B$ for the disjunction of A and B. We write \/ for $\vee$. If we know $A \vee B$ and want to prove P, it is enough to prove two cases: (1) P holds if A holds and (2) P holds if B holds. Two introduction principles are proven: $A \vee B$ follows from A and $A \vee B$ follows from B.

```
Definition or : prop->prop->prop :=
 fun (A B : prop) => forall P:prop, (A -> P) -> (B -> P) -> P.

Infix \/ 785 left := or.

Theorem orIL : forall (A B : prop), A -> A \/ B.
exact (fun A B a P H1 H2 => H1 a).
Qed.

Theorem orIR : forall (A B : prop), B -> A \/ B.
exact (fun A B b P H1 H2 => H2 b).
Qed.
```

Equivalence (iff) is defined using conjunction and implication. We write $\leftrightarrow$ (in ASCII `<->`) as infix notation for equivalence.

```
Definition iff : prop->prop->prop := fun (A B:prop) => (A -> B) /\ (B -> A).

Infix <-> 805 := iff.
```

The file `PropLogic1.mg` also includes a number of lemmas for working with several conjuncts or several disjuncts. Here are the cases for 3. These are sometimes imported as axioms for use in later files.

```
Section PropN.

Variable P1 P2 P3:prop.

Theorem and3I : P1 -> P2 -> P3 -> P1 /\ P2 /\ P3.
exact (fun H1 H2 H3 => andI (P1 /\ P2) P3 (andI P1 P2 H1 H2) H3).
Qed.

Theorem and3E : P1 /\ P2 /\ P3 -> (forall p:prop, (P1 -> P2 -> P3 -> p) -> p).
exact (fun u p H => u p (fun u u3 => u p (fun u1 u2 => H u1 u2 u3))).
Qed.

Theorem or3I1 : P1 -> P1 \/ P2 \/ P3.
exact (fun u => orIL (P1 \/ P2) P3 (orIL P1 P2 u)).
Qed.

Theorem or3I2 : P2 -> P1 \/ P2 \/ P3.
exact (fun u => orIL (P1 \/ P2) P3 (orIR P1 P2 u)).
Qed.

Theorem or3I3 : P3 -> P1 \/ P2 \/ P3.
exact (orIR (P1 \/ P2) P3).
Qed.

Theorem or3E : P1 \/ P2 \/ P3 ->
 (forall p:prop, (P1 -> p) -> (P2 -> p) -> (P3 -> p) -> p).
exact (fun u p H1 H2 H3 => u p (fun u => u p H1 H2) H3).
Qed.

End PropN.
```

We next turn to the file `ExEq1.mg`. First note that it depends on the preamble `ExEq1Preamble.mgs`. It can be checked by calling Egal as follows:

```
egal -ind IndexMar2014 -I ExEq1Preamble.mgs ExEq1.mg
```

The preamble file repeats the definitions of the logical connectives from `PropLogic1.mg` and states some of the theorems from `PropLogic1.mg` as axioms. The index file `IndexMar2014` is needed so that the database knows these theorems were proven at some point.

We first define equality using Leibniz equality, giving infix notation =, as expected. At the same time we define its negation (disequality) and give infix notation `<>` for disequality (mathematically written $\neq$).

```
Section Poly1_eq.
Variable A:SType.

Definition eq : A -> A -> prop := fun x y => forall Q:A -> prop, Q x -> Q y.
Definition neq : A -> A -> prop := fun x y => ~eq x y.
End Poly1_eq.

Infix = 502 := eq.

Infix <> 502 := neq.
```

There are a number of proofs about equality. Below we show eqE for an elimination principle, eqI for an introduction principle (reflexivity of equality), eq_sym for symmetry of equality and eq_trans for transitivity of equality.

```
Section Poly1_eqthms.
Variable A:SType.

Theorem eqE : forall x y:A, forall q:A->prop, q x -> x = y -> q y.
exact (fun x y q H1 H2 => H2 q H1).
Qed.

Theorem eqI : forall x:A, x = x.
exact (fun x q H => H).
Qed.

Theorem eq_sym : forall x y:A, x = y -> y = x.
exact (fun x y H => H (fun y => y = x) (eqI x)).
Qed.

Theorem eq_trans : forall x y z:A, x = y -> y = z -> x = z.
exact (fun x y z H1 H2 => H2 (fun z => x = z) H1).
Qed.
```

Whenever the current goal is to prove $t = t$ where t has type α, we can complete the proof of the goal with the `apply` tactic:

apply eql L_α
where L_α is a layout tree corresponding to α.[2]

As soon as we have defined equality, the tactic `rewrite` with `<-` (to rewrite from right to left) works immediately. After `eq_sym` has been proven, then we can use `rewrite` without `<-` (to rewrite from left to right).[3] Here are three examples demonstrating the `rewrite` tactic.

```
Section Poly1_eq_Examples.
Variable A:SType.

Example rewrite_example_1 :
 forall P:A->A->prop, forall x y:A, x = y -> P x x -> P y y.
let P x y.
assume H1: x = y.
assume H2: P x x.
prove P y y.
rewrite <- H1.
prove P x x.
exact H2.
Qed.

Example rewrite_example_2 :
 forall P:A->A->prop, forall x y:A, x = y -> P y x -> P x y.
let P x y.
assume H1: x = y.
assume H2: P y x.
prove P x y.
rewrite H1.
prove P y y.
rewrite <- H1 at 2.
prove P y x.
exact H2.
Qed.

Example rewrite_example_3 :
 forall f:A->A->A, (forall x y:A, f x y = f y x)
   -> forall x y:A, f (f x y) y = f y (f y x).
let f.
assume H1: forall x y:A, f x y = f y x.
```

[2]Yes, you really have to give the α. I may have mentioned how much I hate type variables. If it bothers you, then prove enough monomorphic instances like eql_set and eql_set_prop and use these. I'm not sure why I didn't do this. It sounds like something I would have done.

[3]Of course, only the global identifier of the proposition of `eq_sym` matters, not the name `eq_sym`.

```
let x y.
prove f (f x y) y = f y (f y x).
rewrite H1.
prove f y (f x y) = f y (f y x).
rewrite H1 at 2.
prove f y (f y x) = f y (f y x).
apply (eqI A).
Qed.

End Poly1_eq_Examples.
```

Next we define existential quantification. Again we are guided by the elimination principle. Let α be a type and $Q : \alpha \to \textsf{prop}$ be a predicate on α. We want to define the proposition $\exists x : \alpha . Q\, x$. This in fact will be binder notation for $\textsf{ex}\, \alpha\, (\lambda x : \alpha . Qx)$. So what we really want to define is ex which is (unfortunately) polymorphic (again). The idea for the definition is to say that if we know $\exists x : \alpha . Q\, x$ and we want to prove P, then it is enough to prove P under the assumption that we have an x such that $Q\, x$. The way this is realized is by defining $\textsf{ex}\, \alpha$ to be

$$\lambda Q : \alpha \to \textsf{prop}. \forall P : \textsf{prop}. (\forall x : \alpha . Q\, x \to P) \to P.$$

In Egal we do this as follows:

```
Section Poly1_exdef.
Variable A:SType.
Variable Q:A->prop.

Definition ex : prop := forall P:prop, (forall x:A, Q x -> P) -> P.
End Poly1_exdef.
```

We next declare the ASCII name `exists` to be binder notation for ex, as indicated above. In ASCII, we use a comma to precede the body. In addition, for the first time we are declaring a binder that is allowed to bind many variables at once. This corresponds to applying ex to a corresponding λ-abstraction for each of the variables. We indicate this by using `Binder+` instead of `Binder`. The Egal code figures out how to make sense of this.

```
Binder+ exists , := ex.
```

We prove an elimination principle (which is rarely, if ever, used) and an introduction principle.

```
Section Poly1_exthms.
Variable A:SType.

Theorem exE : forall P:A->prop, forall Q:prop, (forall x:A, P x -> Q)
```

```
 -> (exists x:A, P x) -> Q.
exact (fun P Q H1 H2 => H2 Q H1).
Qed.

Theorem exI : forall P:A->prop, forall x:A, P x -> exists x:A, P x.
exact (fun P x H1 Q H2 => H2 x H1).
Qed.
```

After the introduction principle has been proven, the `witness` tactic can be used to prove existential goals. If you want to prove $\exists x : \alpha.s$, then you can use `witness` with a layout term specifying a term t of type α. Egal will apply exI with this term t and reduce the goal to proving $s[x := t]$. Here is a simple example.

```
Example witness_example : forall x:A, exists y:A, x = y.
let x.
prove exists y:A, x = y.
witness x.
prove x = x.
exact (eqI A x).
Qed.
```

No new tactics are introduced after this point.

6.2 Basic Higher Order Results

In this section we will briefly discuss the contents of the files `Preds1.mg`, `Relns1.mg`, `KnasterTarski.mg`, `Relns2.mg`, `Classical1.mg`, `Conditionals.mg`, `Exactly1of.mg` and `Quotients.mg`. These can be checked with Egal as follows:

```
egal -ind IndexMar2014 -I Preds1Preamble.mgs Preds1.mg
egal -ind IndexMar2014 -I Relns1Preamble.mgs Relns1.mg
egal -ind IndexMar2014 -I KnasterTarskiPreamble.mgs KnasterTarski.mg
egal -ind IndexMar2014 -I Relns2Preamble.mgs Relns2.mg
egal -ind IndexMar2014 -I Classical1Preamble.mgs Classical1.mg
egal -ind IndexMar2014 -I ConditionalsPreamble.mgs Conditionals.mg
egal -ind IndexMar2014 -I Exactly1ofPreamble.mgs Exactly1of.mg
egal -ind IndexMar2014 -I QuotientsPreamble.mgs Quotients.mg
```

In `Preds1.mg` a number of polymorphic operators for taking complements, unions, intersections of predicates over a type α. In this file the `c=` notation is used for sub-predicates. (As mentioned before it can be used as the $\subseteq$ on two sets or for the obvious relation on two relations $\beta_1 \rightarrow \cdots \beta_n \rightarrow \mathsf{prop}$.) In mathematical notation, for $P, Q : \alpha \rightarrow \mathsf{prop}$, we write $P \subseteq Q$ to mean $\forall x : \alpha.P, x \rightarrow Q\, x$. A number of results are proven. Eventually we want to prove results like predicate extensionality (pred_ext): If $P \subseteq Q$ and $Q \subseteq P$, then $P = Q$.

```
Section PE.
Variable A:SType.

Theorem pred_ext : forall (P Q : A -> prop), P c= Q -> Q c= P -> P = Q.
```

It is not possible to prove predicate extensionality without having some extensionality principles as axioms. This is the first use of propositional extensionality and functional extensionality (as introduced in Chapter 5) and motivates their inclusion as axioms.

In `Relns1.mg` there are some basic definitions and results about binary relations of type $\alpha \to \alpha \to \mathsf{prop}$. For example, the properties of being an equivalence relation, of being a partial equivalence relation (per) and of being a partial order are defined here. A number of examples are given and proven to have the properties.

In `KnasterTarski.mg` a number of versions of the Knaster-Tarski Fixed Point Theorem are proven. A simple way to state a version of the Knaster-Tarski Fixed Point Theorem is as follows: If $F : \wp(X) \to \wp(X)$ is a monotone operator, then there is a set $Y \in \wp(X)$ such that $FY = Y$. In this document, we work with predicates and relations on types instead of sets and prove Knaster-Tarski results as theorems of higher-order logic.[4]

These are the most complicated proofs to this point. They are truly higher-order as the least fixed point (lfp) is defined as the least prefixed point via an impredicative universal quantification. Also, the greatest fixed point (gfp) is defined as the greatest postfixed point via an impredicative existential quantification.

```
Section Poly1_fpdef.
Variable A:SType.

Variable F:((A->prop)->A->prop).
Hypothesis Fmon:forall p q:A->prop, p c= q -> F p c= F q.

Definition lfp : A->prop :=
fun z => forall p:A -> prop, F p c= p -> p z.

Definition gfp : A->prop :=
fun z => exists p:A -> prop, p c= F p /\ p z.
```

Note that although the hypothesis Fmon is in context when the definitions lfp and gfp are made, the definitions do not depend on Fmon. In general, definitions never depend on hypotheses because the framework is simple type theory.

In addition to the versions for unary predicates, there are versions for binary predicates. Here are the definitions of the least and greatest fixed points for these.

```
Section Poly2_fpdef.
Variable A B:SType.
```

[4] These provide some early examples of higher-order theorems in TPS, see THM2 in the TPS library.

```
Variable F : (A->B->prop) -> (A->B->prop).
Hypothesis Fmon:forall R S:A->B->prop, R c= S -> F R c= F S.

Definition lfp2 : A->B->prop :=
  fun z1 z2 => forall p:A->B->prop, F p c= p -> p z1 z2.

Definition gfp2 : A->B->prop :=
  fun z1 z2 => exists p:A->B->prop, p c= F p /\ p z1 z2.
```

As an example, the last half of the file uses the least fixed point of a certain relation to obtain the transitive closure of a relation.

In `Relns2.mg` there are definitions about binary relations of type $\alpha \to \beta \to \mathsf{prop}$ such as whether the relation is functional, injective, total or surjective. If f is of type $\alpha \to \beta$, then the relation $\lambda x : \alpha.\lambda y : \beta.f\,x = y$ is functional and total. A function f of type $\alpha \to \beta$ may be injective and/or surjective. If f is an injective function, then $\lambda x : \alpha.\lambda y : \beta.f\,x = y$ is an injective relation. If f is a surjective function, then $\lambda x : \alpha.\lambda y : \beta.f\,x = y$ is a surjective relation. Mathematically, one might expect that if $R : \alpha \to \beta \to \mathsf{prop}$ is a total functional relation, then there is some function $f : \alpha \to \beta$ such that R is the same as $\lambda x : \alpha.\lambda y : \beta.f\,x = y$. Without new axioms, this cannot be proven. In order to obtain this it would be enough to include a description operator, but we will include a stronger choice operator, as first discussed in Chapter 5.

```
Section Choice.
Variable A:SType.
Parameter Eps : (A->prop)->A.
Axiom EpsR : forall P:A->prop, forall x:A, P x -> P (Eps P).
End Choice.

Binder some , := Eps.
```

Once we have a choice operator, we can prove that for every total relation $R : \alpha \to \beta \to \mathsf{prop}$, there is a function $f : \alpha \to \beta$ such that $\forall x : \alpha.R\,x\,(f\,x)$ (see Skolem). We can also prove that if $f : \alpha \to \beta$ is surjective, then there is a $g : \beta \to \alpha$ such that $f(gy) = y$ for every y. Likewise, if f is injective, then there is a g such that $g(fx) = x$ for every x. The file `Relns2.mg` with two versions of Cantor's Theorem. Cantor's Theorem informally says $\wp(X)$ has greater cardinality than X. At the level of type theory, we can modify this to say the type $\alpha \to \mathsf{prop}$ has greater cardinality than α. This can be made precise in many ways. Two of the ways are as follows: The Surjective Cantor Theorem (SurjectiveCantor) says there is no surjective function from α to $\alpha \to \mathsf{prop}$. This is proven by an easy diagonalization argument.

```
Theorem SurjectiveCantor : ~exists g:A->A->prop, surjective A (A->prop) g.
assume H: exists g:A->A->prop, surjective A (A->prop) g.
apply H.
```

```
let g.
assume H1: forall f:A->prop, exists x:A, g x = f.
apply (H1 (fun x:A => ~g x x)).
let x.
assume H2: g x = fun x:A => ~g x x.
claim L1: g x x <-> ~g x x.
{
  rewrite H2 at 1.
  prove ~g x x <-> ~g x x.
  exact (iff_refl (~g x x)).
}
exact (iff_circuit (g x x) L1).
Qed.
```

Another formulation is the Injective Cantor Theorem (InjectiveCantor): there is no injection from $\alpha \to$ prop to α. There is also a diagonalization argument to directly prove this theorem [4], but here we can give a simpler proof using the previously proven result that every injection has a one-sided inverse. This one-sided inverse would be a surjection from α to $\alpha \to$ prop, contradicting the surjective version of the theorem.

```
Theorem InjectiveCantor : forall h:(A->prop)->A, ~injective (A->prop) A h.
let h.
assume H1: injective (A->prop) A h.
apply (inj_inverse_ex (A->prop) A h H1).
let g.
assume H2: forall p:A->prop, g (h p) = p.
claim L1: surjective A (A->prop) g.
{
  let p. witness (h p). exact (H2 p).
}
apply SurjectiveCantor.
witness g.
exact L1.
Qed.
```

Once we have propositional and functional extensionality as well as a choice operator, we can prove our logic is classical. This is known as Diaconescu's Theorem [41, 34]. This is the first result in `Classical1.mg` which we include here with its proof. The proof is interesting and it gives you a chance to see if you can read an interesting Egal proof. They are meant to be readable.

```
Theorem classic : forall P:prop, P \/ ~ P.
let P:prop.
set p1 := fun x : prop => x \/ P.
set p2 := fun x : prop => ~x \/ P.
```

```
claim L1:p1 True.
{ prove (True \/ P). apply orIL. exact TrueI. }
claim L2:(some x:prop, p1 x) \/ P.
{ exact (EpsR prop p1 True L1). }
claim L3:p2 False.
{ prove ~False \/ P. apply orIL. assume H. exact H. }
claim L4:~(some x:prop, p2 x) \/ P.
{ exact (EpsR prop p2 False L3). }
apply L2.
- assume H1:(some x:prop, p1 x).
  apply L4.
  + assume H2:~(some x:prop, p2 x).
    prove P \/ ~ P.
    apply orIR.
    prove ~ P.
    assume H3 : P.
    claim L5:p1 = p2.
    {
      apply (pred_ext prop).
      - prove (p1 c= p2).
        let x. assume H4.
        prove (~x \/ P).
        apply orIR.
prove P.
        exact H3.
      - prove (p2 c= p1).
        let x. assume H4.
        prove (x \/ P).
        apply orIR.
prove P.
        exact H3.
    }
    apply H2. rewrite <- L5. exact H1.
  + assume H2:P.
    prove P \/ ~ P.
    apply orIL.
    prove P.
    exact H2.
- assume H1:P.
    prove P \/ ~ P.
    apply orIL.
    prove P.
    exact H1.
```

```
Qed.
```

The rest of `Classical1.mg` consists of consequences of the logic being classical. For example we have the following theorems:

```
Theorem NNPP : forall p:prop, ~ ~ p -> p.
Theorem prop_deg : forall p:prop, p = True \/ p = False.
Theorem not_and_or_demorgan : forall A B:prop, ~(A /\ B) -> ~A \/ ~B.
Theorem not_all_ex_demorgan : forall P:A->prop, ~(forall x:A, P x)
  -> exists x:A, ~P x.
```

There are also a number of identities proven. One could use these to rewrite some logical operators in favor of others.

```
Theorem eq_true : True = (~False).
Theorem eq_and_nor : and = (fun (x y:prop) => ~(~x \/ ~y)).
Theorem eq_or_nand : or = (fun (x y:prop) => ~(~x /\ ~y)).
Theorem eq_or_imp : or = (fun (x y:prop) => ~ x -> y).
Theorem eq_imp_or : (fun (x y:prop) => (x -> y)) = (fun (x y:prop) => (~x \/ y)).
```

The last example in the file is the so-called Drinker problem. This is sometimes explained as follows: If you walk into a bar, there is always someone you can point to so that if that person drinks then everyone drinks. It seems obviously wrong, but it's actually correct. We begin by stating it as a proposition.

```
Example Drinker : forall d:A->prop, exists x:A, d x -> forall x:A, d x.
```

The type α is for the people in the bar and the predicate $d : \alpha \to \mathsf{prop}$ identifies the drinkers in the bar. If someone in the bar isn't drinking, then use them as the witness. If they drink then everyone drinks simply because they aren't drinking. If everyone is drinking, then just pick someone as it doesn't matter because everyone is drinking anyway.[5] The actual proof in `Classical1.mg` is a little different, and we leave it to the reader to examine it.

The purpose of `Conditionals.mg` is to define the polymorphic if-then-else operation (If) and introduce the special if-then-else notation for it. The choice operator is used to do this.

```
Definition If : prop->A->A->A :=
 (fun p x y => some z:A, p /\ z = x \/ ~p /\ z = y).

Notation IfThenElse If.
```

Here are two useful theorems:

[5] By the way, this example might give a hint why some people reject classical logic.

```
Theorem If_0 : forall p:prop, forall x y:A,
~ p -> (if p then x else y) = y.

Theorem If_1 : forall p:prop, forall x y:A,
p -> (if p then x else y) = x.
```

Note that the way the syntax for if-then-else is arranged, if we omitted the parentheses above and wrote, for example,

```
if p then x else y = x
```

then it would be parsed as if it were

```
if p then x else (y = x)
```

so be careful. This is because the part after an `else` always has scope as far to the right as possible, like a binder. Maybe this was a bad decision because I often forgot about it, but there is no way I'm reopening development of the parser.

The file `Exactly1of.mg` defines two logical operations exactly1of2 and exactly1of3 and proves results about them. exactly1of2 is just exclusive or, but it was mainly included to help define exactly1of3. exactly1of3 is included because some mathematical properties and theorems state that exactly one of three possibilities hold (e.g., trichotomy properties). If someone is lazy and formalizes the property using disjunction, it will be wrong. Including exactly1of3 makes it easy for lazy people not to be wrong.

Finally the file `Quotients.mg` provides the infrastructure for forming quotients of partial equivalence relations using canonical elements. Canonical elements are provided by the choice operator. There are actually two developments in the file. The second one assumes there is a default function d which can try to choose the canonical element. The default canonical value is used if it can be and the choice funciton is used otherwise.

6.3 Zermelo's Well Ordering Theorem

In this section we include a few remarks about the file `Zermelo1908.mg` which can be checked in Egal as follows:

```
egal -ind IndexMar2014 -I Zermelo1908Preamble.mgs Zermelo1908.mg
```

This is a formalization of Zermelo's second proof of his Well-Ordering Theorem, published in 1908 [50]. I'm releasing Egal with a pdf explaining Zermelo's proof in terms that probably roughly correspond the contents of `Zermelo1908.mg`. I managed to do this formalization in Coq briefly before the treasure hunt started and found it interesting enough that I did this Egal version and included it in the treasure hunt. It was one of the documents with a 1 bitcoin treasure (which no one found before the 4 week deadline). This 1 bitcoin treasure was for the lemma C_lin that took me quite a long time to prove on paper. It's basically a double induction. If you look in Zermelo's

paper, you'll see one or two pages where he essentially explains the proof of this lemma without being explicit about the fact that this is what he's proving. I went through a lot of variants before I managed to formulate the lemma in such a way that it went through. In the end, though, I think it came out very nice. Have a look.

6.4 Set Theory

We now consider a few files developing some basic set theory. The files are `Set1a.mg`, `Set1b.mg`, `Set1c.mg`, `Set2.mg`, `EpsInd.mg`, `Nat1.mg`, `Nat2.mg`, `UnivInf.mg` and `Ordinals1.mg`. These can be checked as follows.

```
egal -ind IndexMar2014 -I Set1aPreamble.mgs Set1a.mg
egal -ind IndexMar2014 -I Set1bPreamble.mgs Set1b.mg
egal -ind IndexMar2014 -I Set1cPreamble.mgs Set1c.mg
egal -ind IndexMar2014 -I Set2Preamble.mgs Set2.mg
egal -ind IndexMar2014 -I EpsIndPreamble.mgs EpsInd.mg
egal -ind IndexMar2014 -I Nat1Preamble.mgs Nat1.mg
egal -ind IndexMar2014 -I Nat2Preamble.mgs Nat2.mg
egal -ind IndexMar2014 -I UnivInfPreamble.mgs UnivInf.mg
egal -ind IndexMar2014 -I Ordinals1Preamble.mgs Ordinals1.mg
```

The file `Set1a.mg` introduces the foundational set constructors (other than the one for universes) and proves a few easy results about them. These constructors were discussed already in Chapter 5.

The file `Set1b.mg` defines unordered pairs $\{x, y\}$ using if-then-else, replacement and the set $\wp(\wp(\emptyset))$. More details about this construction can be found in [11], the pdf of which is included with the Egal distribution. The history of the construction is also explained in [11]; it isn't original to me. From unordered pairs it is easy to define singletons $\{x\}$. Also, binary unions $X \cup Y$ are defined as $\bigcup\{X, Y\}$. An operation $\mathsf{SetAdjoin}\, X\, y$ is defined to be $X \cup \{y\}$. After this the general special `SetEnum` notation is declared:

```
Notation SetEnum Empty Sing UPair SetAdjoin.
```

At this point Egal can interpret $\{\}$ as be $\emptyset$, $\{x\}$ to be the singleton set, $\{x, y\}$ to be the unordered pair, and $\{x, y, z_1, \ldots, z_n\}$ to be the result of using $\mathsf{SetAdjoin}$ n times starting from $\{x, y\}$. The last definition made is of unions of families. If $X : \mathsf{set}$ and $F : \mathsf{set} \to \mathsf{set}$, then $\mathsf{famunion}\, X\, F$ corresponds to what is mathematically written as $\bigcup_{x \in X} Fx$. There is also corresponding binder notation `\/_`.

In `Set1c.mg` a number of results are proven and then the definition Sep is made.

```
Section SepSec.

Variable X:set.
```

```
Variable P:set->prop.
Let z : set := some z :e X, P z.
Let F:set->set := fun x => if P x then x else z.

Definition Sep:set
:= if (exists z :e X, P z) then {F x|x :e X} else Empty.

End SepSec.
```

The definition Sep supports forming sets by separating out the elements from a set that satisfy a given property.[6] Special notation is declared.

```
Notation Sep Sep.
```

After this, Egal can interpret notation like $\{x \in s|t\}$.

In addition a combination of replacement and separation is defined and declared as special notation.

```
Definition ReplSep : set->(set->prop)->(set->set)->set :=
 fun X P F => {F x|x :e {z :e X|P z}}.

Notation ReplSep ReplSep.
```

After this, Egal can interpret notation like $\{t|x \in s, t'\}$ to mean the set of all t as x ranges over the elements of s satisfying t'.

In `Set2.mg` binary intersection and set complement are defined. The basic boolean properties are proven.

The development in the file `EpsInd.mg` uses the $\in$-induction axiom to prove a few results (e.g., no set is a member of itself). In addition, a notation of definition by $\in$-recursion is developed (see In_rec). More information about this can be found in [11].

In `Nat1.mg` the natural numbers are represented by the finite ordinals. We start by defining ordinal successor:

```
Definition ordsucc : set->set := fun x:set => x :\/: {x}.
```

Then there is special notation so that Egal can interpret natural numbers n as ordsucc applied n times to $\emptyset$.

[6]Hidden history: Zermelo used an old German word *Aussonderung* for this. Germans always seem to be a little uncomfortable with the word. It literally means something like "sorting out," but it seems to have a connotation similar to the word *segregation* does in English. To be fair to Zermelo, he was around during the time of the Nazis and, unlike some other German logicians like Gentzen, was not a Nazi himself. Stories are sometimes told that Zermelo refused to do the *Heil Hitler* salute and consequently lost his professorship at Freiburg. I've read in other sources that he actually would do it because it was legally required (and he seems to have suffered under the common mistaken belief that he was morally required, as a "citizen," to do what was legally required of him), but it was clear he didn't want to do it and didn't do it with enough gusto. That was considered bad enough by some of his colleagues.

```
Notation Nat Empty ordsucc.
```

First a number of basic results about 0, 1 and 2 are proven. For example, $0 \in 1$, $0 \neq 1$, $0 \neq 2$, $1 = \{0\}$ and $2 = \{0, 1\}$. A predicate nat_p is defined which carves out the natural numbers.

```
Definition nat_p : set->prop :=
 fun n:set => forall p:set->prop,
   p 0 -> (forall x:set, p x -> p (ordsucc x)) -> p n.
```

A few induction and inversion principles are proven.

```
Lemma nat_ind : forall p:set->prop,
 p 0 -> (forall n, nat_p n -> p n -> p (ordsucc n))
  -> forall n, nat_p n -> p n.

Lemma nat_inv : forall n, nat_p n
 -> n = 0 \/ exists x, nat_p x /\ n = ordsucc x.

Lemma nat_complete_ind : forall p:set->prop,
  (forall n, nat_p n -> (forall m :e n, p m) -> p n)
 -> forall n, nat_p n -> p n.
```

We can use $\bigcup$ to obtain the predecessor of a number:

```
Lemma Union_ordsucc_eq : forall n, nat_p n -> Union (ordsucc n) = n.
```

The development of the natural numbers is continued in `Nat2.mg`. Here $\in$-recursion is specialized to primitive recursion on the natural numbers: nat_primrec. This is used to define addition and multiplication and the usual properties are proven. The last example is $1 + 1 = 2$.

In `UnivInf.mg` the constructor and axioms for Grothendieck universes are used. An infinite set (InfiniteSet) is defined to be a set X for which there is an injective function from X to X which is not surjective. A set is finite if it is not infinite. The universe UnivOf Empty, which I will denote by $\mathcal{U}_\emptyset$, is proven to be infinite by taking the ordinal successor to be the injective function which is not surjective. All natural numbers are in $\mathcal{U}_\emptyset$.[7] The set ω is defined to be $\{n \in \mathcal{U}_\emptyset | \mathsf{nat_p}\, n\}$. (Usually, in ZF set theory an axiom of infinity is required to prove ω is a set. In this case, the universe axioms play this role.) A ZF_model is defined to be a set which is transitive, ZF_closed and has ω as a member. The set UnivOf ω satisfies these properties.

The file `Ordinals1.mg` develops a bit of theory about the ordinals. An ordinal is defined to be a transitive set all of whose members are transitive. The empty set is an ordinal (ordinal_Empty). Members of ordinals are always ordinals (ordinal_Hered). We have the following induction principle on ordinals, which is essentially a special case of $\in$-induction:

[7] With some more work, it could be proven that $\mathcal{U}_\emptyset$ consists of precisely the hereditarily finite sets.

```
Theorem ordinal_ind : forall p:set->prop,
  (forall alpha, ordinal alpha -> (forall beta :e alpha, p beta) -> p alpha)
  ->
  forall alpha, ordinal alpha -> p alpha.
```

The ordinal successor of an ordinal is an ordinal (ordinal_ordsucc). Also, ω is an ordinal (omega_ordinal). Ordinals satisfy a trichotomy property: exactly one of three of the following hold for ordinals α and β: $\alpha \in \beta$, $\alpha = \beta$, $\beta \in \alpha$. The lazy version of this property formulated with disjunction is proven first (ordinal_trichotomy_or). The proof is interesting and involves a double $\in$-induction. Using the lazy version one can prove the real version (ordinal_trichotomy) formulated using exactly1of3.

6.5 Pairs and Functions

We finally consider the last five files from the treasure hunt. The purpose of these developments is to give a nonstandard representation of pairs, functions and, in the end, tuples. They have a number of unusual properties. One that I like to show off is $X \times X = X^2$. In other words, ordered pairs are the same thing as functions with domain 2. I wrote an article about the representations last year [11], and the pdf of this article is included with Egal. I think this is probably the coolest thing I've ever done. And no one will ever know. *Egal!*

The five files are `DisjointUnions.mg`, `OrderedPairs.mg`, `DepSums.mg`, `FuncsSets.mg` and `DepProds.mg`. They can be checked as follows:

```
egal -ind IndexMar2014 -I DisjointUnionsPreamble.mgs DisjointUnions.mg
egal -ind IndexMar2014 -I OrderedPairsPreamble.mgs OrderedPairs.mg
egal -ind IndexMar2014 -I DepSumsPreamble.mgs DepSums.mg
egal -ind IndexMar2014 -I FuncsSetsPreamble.mgs FuncsSets.mg
egal -ind IndexMar2014 -I DepProdsPreamble.mgs DepProds.mg
```

In `DisjointUnions.mg` two injections are defined on the set theoretic universe. The first one Inj1 is defined by $\in$-recursion as follows:

$$\mathsf{Inj1}\,X = \{\emptyset\} \cup \{\mathsf{Inj1}\,x | x \in X\}$$

The second one is Inj0 which is simply defined by

$$\mathsf{Inj0}\,X = \{\mathsf{Inj1}\,x | x \in X\}.$$

Both are proven injective and it is proven that they never give the same value. I can tell you the secret now: in the end $\mathsf{Inj0}\,X$ will be the same as the ordered pair $(0, X)$ and $\mathsf{Inj1}\,X$ will be the same as the ordered pair $(1, X)$. Using these two injections, disjoint unions are defined as setsum, and we can write $X + Y$ (in ASCII we use `:+:` for this $+$) for this disjoint union.

In `OrderedPairs.mg` we define an ordered pair constructor `pair` which takes X and Y and returns $X+Y$. Yes, the ordered pair is simply the disjoint sum. There is no notation like (X,Y) in this file because this notation is reserved for n-tuples which come later.

In `DepSums.mg` we define dependent sums $\Sigma x \in X.Y$ and simple products $X \times Y$ as certain sets of ordered pairs.

In `FuncsSets.mg` we define $\lambda x \in X.Y$ (a set representation of this function) simply as $\Sigma x \in X.Y$. Yes, λ is the same as Σ. The representation comes from Aczel [1], though I don't recall him explicitly noting that this is Σ. But of course it is. What's funny is that in AUTOMATH there was no distinction between λ and $\forall/\Pi$, so it was like λ and Π are the same. This seems a bit bizarre now. And now I'm going down the road of saying λ is Σ. And humans think I'm crazy. But they're wrong.[8] Of course λ is Σ. Why? Because it has beautiful consequences.

Recall that $2 = \{0,1\}$. Consider $\lambda x \in 2.f(x)$. This is the same as $\Sigma x \in \{0,1\}.f(x)$ which is the same as

$$\{(0,y)|y \in f(0)\} \cup \{(1,z)|z \in f(1)\}$$

which is the same as $f(0) + f(1)$ which is just the ordered pair of $f(0)$ and $f(1)$. How can anyone believe in a representation where beautiful things like that don't happen?

Finally in `DepProds.mg` dependent products $(\Pi x \in X.Y)$ and simple exponents (Y^X) are defined. This is where we prove $X \times X = X^2$.

[8] I know I'm contradicting the Preface. I'm obviously of two minds on these matters.

Chapter 7

Example: Existence of God

Logic, my dear Zoe, merely enables one to be wrong with authority.
The Doctor, *The Wheel In Space*, Episode Three (1968)

Benzmüller and Paleo [9] have recently considered several interactive and automated proofs corresponding to Gödel's proof of the existence of an entity which has all positive properties. Gödel defines an entity which has all positive properties to be a god, and so one can say Gödel has given an existence proof for a god. (Such a god would also be unique, so we would be justified in simply writing "God.") Of course, there are a number of hypotheses on which the argument is based, so the reader can simply reject at least one of the hypotheses and feel secure in remaining an atheist, an agnostic or a polytheist. We include this example as a fun way to demonstrate that theorem proving applies beyond mathematics. The Egal version we discuss here is heavily based on the Coq version given by Benzmüller and Paleo [9].

The argument is made in a modal logic[1] which can be embedded into higher-order logic. In our case, we embedded into the set theory. The reason is so that if someone wanted to prove something is *not* a theorem of modal logic, they would have a reasonable chance to construct an appropriate Kripke model in the set theory.

The file describing the embedding of modal logic is `Modal.mg` and depends on some things imported via `ModalPreamble.mg`

```
egal -ind IndexMar2014 -I ModalPreamble.mgs Modal.mg
```

To give a sample, here are some logical operations and notation given at the level of operations on predicates over sets:

```
Definition mnot : (set->prop) -> (set->prop) := fun p w => ~ p w.
Prefix :~: 700 := mnot.

Definition mand : (set->prop) -> (set->prop) -> (set->prop) :=
 fun p q w => p w /\ q w.
```

[1] It's in a first-order version of S5, I think, look up the references if you want to know.

```
Infix :/\: 780 left := mand.

Definition mor : (set->prop) -> (set->prop) -> (set->prop) :=
 fun p q w => p w \/ q w.
Infix :\/: 785 left := mor.

Definition mimplies : (set->prop) -> (set->prop) -> (set->prop) :=
 fun p q w => p w -> q w.
Infix :->: 810 right := mimplies.
```

In addition to the usual logical operations, there are two modal operators: the box for necessity and the diamond for possibility. Each of them depend on an accessibility relation R. For the Gödel argument, the relation R is assumed to be an equivalence relation on some set of "possible worlds." I do things a little differently here. Instead of having a set of possible worlds, I assume R is a per and consider the w to be a possible world if $R\,w\,w$. This choice shows up in the definition of valid.

```
Section ModalOps.

Variable R : set -> set -> prop.

Definition box_ : (set->prop) -> set->prop := fun p w => forall u, R w u -> p u.
Definition dia_ : (set->prop) -> set->prop := fun p w => exists u, R w u /\ p u.

Definition Valid_ : (set->prop) -> prop := fun p => forall w, R w w -> p w.

End ModalOps.
```

We also define quantifiers.

```
Section ModalQuants.
Variable A : SType.
Definition ModalAll : (A->set->prop) -> set->prop := fun p w => forall x:A, p x w.
Definition ModalEx : (A->set->prop) -> set->prop := fun p w => exists x:A, p x w.
End ModalQuants.
Binder+ mforall , := ModalAll.
Binder+ mexists , := ModalEx.
```

Then there are some theorems. Some of them depend on R being a per, and others don't.

A new index file `IndexJune2014` was created using `Modal.mg` and consequently the results in `Modal.mg` and be imported via `GoedelGodPreamble.mgs`. This means the main file `GoedelGod.mg` is checked as follows:

```
egal -ind IndexJune2014 -I GoedelGodPreamble.mgs GoedelGod.mg
```

We briefly describe some of the main file `GoedelGod.mg`. We assume a relation R is given and give local definitions for box, dia and Valid with this R fixed.

```
Variable R:set->set->prop.

Let box := box_ R.
Let dia := dia_ R.
Let Valid := Valid_ R.
```

We assume R is a per. There are a few simple lemmas proven I will skip here.

```
Hypothesis Rper: per set R.
```

At this point, keep in mind that we think of "modal propositions" as being of type set $\to$ prop. Often when you see set $\to$ prop, you should think of it as a proposition.

We assume there is a notion of a property being positive.

```
Variable Positive : (set->set->prop) -> set->prop.
```

A property here is something that takes an object (a set) and returns a modal proposition. Note that a property might, in principle, be positive at one possible world and not at another.

We next begin to assume hypotheses corresponding to Gödel's axioms used in his argument.

First, every property is either positive or its negation is positive. We state this as two axioms which basically say the negation of a property p is positive if and only if the property p itself is not positive.[2]

```
Hypothesis Axiom1a : Valid
 (mforall p:set->set->prop, Positive (fun x => :~: p x) :->: :~: Positive p).
Hypothesis Axiom1b : Valid
 (mforall p:set->set->prop, :~: Positive p :->: Positive (fun x => :~: p x)).
```

Next, if p is positive and it is necessary that q follows from p, then q is positive.

```
Hypothesis Axiom2 : Valid
 (mforall p q:set->set->prop,
  Positive p :/\: box (mforall x, p x :->: q x) :->: Positive q).
```

Then we have the following theorem. I'll leave out the proof, but it is in `GoedelGod.mg`. The theorem says that if a property is positive, then it must be possible for something to exist satisfying the property.

```
Theorem Thm1 : Valid
 (mforall p:set->set->prop, Positive p :->: dia (mexists x, p x)).
```

[2]This already seems highly questionable to me, but I'm just repeating it.

Next we define what it means to be a god. Something has the property of being a god if it has all positive properties.

```
Definition G : set->set->prop :=
 fun x => mforall p:set->set->prop, Positive p :->: p x.
```

The next hypothesis is that the property of being a god is positive.[3]

```
Hypothesis Axiom3 : Valid (Positive G).
```

It is possible that a god exists. The proof is short and is in the file.

```
Theorem Cor1 : Valid (dia (mexists x, G x)).
```

The next hypothesis states that every positive property is necessarily positive.[4]

```
Hypothesis Axiom4 : Valid
 (mforall p:set->set->prop, Positive p :->: box (Positive p)).
```

The next definition is of the "essence" of a property. Given a property, the "essence" of a property is another property. Something x satisfies "essence" of p if it satisfies p and every property q that x satisfies is necessarily implied by p. Probably a better way to say this is in natural language is "x is the essence of p." That is, x in some way codes up the essential nature of the property p. Of course, I don't really know what I'm talking about. Think of it however you want. The logic is what matters.

```
Definition Essence : (set->set->prop) -> set->set->prop :=
 fun p x => p x :/\:
     mforall q:set->set->prop, q x :->: box (mforall y, p y :->: q y).
Infix ess 69 := Essence.
```

The infix notation allows us to write $p \,\mathsf{ess}\, x$ for x is the essence of p.

The next theorem states that if x is a god, then x is the essence of being a god.

```
Theorem Thm2 : Valid (mforall x, G x :->: G ess x).
```

Now there is a definition of a property called NE. An x has this property if whenever x is the essence of a property p, then something necessarily exists that satisfies the property p.

```
Definition NE : set->set->prop :=
 fun x => mforall p:set->set->prop, p ess x :->: box (mexists y, p y).
```

The last hypothesis is that the property NE is positive.

```
Hypothesis Axiom5 : Valid (Positive NE).
```

---

[3] Again, this is open to debate.

[4] Yet again, this is debatable.

It is now possible to prove a lemma that says that if there exists a god, then there necessarily exists a god.

```
Lemma Lem1 : Valid ((mexists x, G x) :->: box (mexists x, G x)).
```

Combining the results, we can prove that a god necessarily exists.

```
Theorem Thm3: Valid (box (mexists x, G x)).
```

As an easy consequence, there is a god. I'll even show the proof this time.

```
Theorem Cor2 : Valid (mexists x, G x).
let w.
assume Hw: R w w.
exact Thm3 w Hw w Hw.
Qed.
```

So God exists after all. Take that Hitchens![5]

[5]Sorry. Too soon. I do miss having Christopher Hitchens around. If only all the humans were as interesting as he was.

Chapter 8

Category Theory: The Final Treasures

It means the collapse of the universe has started and nothing can stop it.
The Doctor, *The Two Doctors*, Part Two (1985)

One of my motivations for choosing simply typed Tarski-Grothendieck set theory as a foundation is because I wanted something that could reasonably handle Category Theory [31, 30]. The trouble is that some categories are too big to be considered sets. One solution is to have universes so that one can always relativize to the universe in which one is working and obtain a category as a set in this way. I did an Egal development in which I gave three different definitions of a category and proves a few results. The development uses a number of objects and results defined and proven in the treasure hunt documents. The requirements are imported via `CategoryPreamble.mgs`. A version of the development without the proofs is under `formaldocs` as `CategoryProblems.mgs`. To check this version do this:

```
egal -ind IndexMar2014 -I CategoryPreamble.mgs CategoryProblems.mg
```

I buried some final bitcoin treasures beneath my proofs. To get the treasures, replace `Admitted` with a proof and modify the proof until it corresponds to the treasure. I doubt I'll ever release the solutions, so feel free to take your time. One caveat: If the price of bitcoins rises a lot (again), I might reclaim the treasures myself and replace them with versions that are closer to the value now (in September, 2014). The value of all the current treasures in this document in terms of fiat is approximately 600 U.S. Dollars or 500 Euros.

The first important definition is of a *metacategory* and is at the level of the type theory. There are predicates and relations carving out objects from a type α and arrows from a type β. In addition, functions giving identity arrows and compositions are given. This information specifies a metacategory if certain "typing" properties hold and the identity laws and associativity of composition hold. We give the details in the syntax of Egal.

```
Section MetaCat.

Variable A B: SType.

Variable Obj: A -> prop.
Variable Hom: A -> A -> B -> prop.
Variable id: A -> B.
Variable comp: A -> A -> A -> B -> B -> B.

Definition idT : prop := forall X:A, Obj X -> Hom X X (id X).

Definition compT : prop :=
 forall X Y Z:A, forall f g:B,
 Obj X -> Obj Y -> Obj Z ->
 Hom X Y f -> Hom Y Z g ->
 Hom X Z (comp X Y Z g f).

Definition idL : prop :=
 forall X Y:A, forall f:B,
 Obj X -> Obj Y -> Hom X Y f -> comp X X Y f (id X) = f.

Definition idR : prop :=
 forall X Y:A, forall f:B,
 Obj X -> Obj Y -> Hom X Y f -> comp X Y Y (id Y) f = f.

Definition compAssoc : prop :=
 forall X Y Z W:A, forall f g h:B,
 Obj X -> Obj Y -> Obj Z -> Obj W ->
 Hom X Y f -> Hom Y Z g -> Hom Z W h ->
 comp X Y W (comp Y Z W h g) f = comp X Z W h (comp X Y Z g f).

Definition MetaCat : prop
 := (idT /\ compT) /\ (idL /\ idR) /\ compAssoc.
```

I proved an introduction and an elimination lemma, which are now left to the reader.

Metacategories are potentially large and there is no way in general to encode them as sets. Nevertheless, I expect the level of metacategories would be the easiest level at which to work if one were proving results about categories.

One example of a metacategory is considered. We take a type α and consider every predicate over α to be an object. Given two objects X and Y of type $\alpha \to$ prop, the arrows from X to Y are the functions $\alpha \to \alpha$ mapping elements of X to elements of Y, modulo a certain partial equivalence relation. That is, we make use of quotients, as defined during the treasure hunt and imported in `CategoryPreamble.mgs`. The proofs

that everything works are left to the reader.

Next *locally small categories* are defined. In this case, the objects are specified by a predicate on sets. The arrows are in Hom-sets. Also, identity and composition functions must be given. These specify a locally small category if the corresponding conditions hold. Again, we give the details in Egal syntax.

```
Section LocallySmallCat.

Variable Obj: set -> prop.
Variable Hom: set -> set -> set.
Variable id: set -> set.
Variable comp: set -> set -> set -> set -> set -> set.

Definition idT' : prop := forall X:set, Obj X -> id X :e Hom X X.
Definition compT' : prop :=
 forall X Y Z:set, forall f g:set,
 Obj X -> Obj Y -> Obj Z ->
 f :e Hom X Y -> g :e Hom Y Z ->
 comp X Y Z g f :e Hom X Z.

Definition idL' : prop :=
 forall X Y:set, forall f:set,
 Obj X -> Obj Y -> f :e Hom X Y -> comp X X Y f (id X) = f.

Definition idR' : prop :=
 forall X Y:set, forall f:set,
 Obj X -> Obj Y -> f :e Hom X Y -> comp X Y Y (id Y) f = f.

Definition compAssoc' : prop :=
 forall X Y Z W:set, forall f g h:set,
 Obj X -> Obj Y -> Obj Z -> Obj W ->
 f :e Hom X Y -> g :e Hom Y Z -> h :e Hom Z W ->
 comp X Y W (comp Y Z W h g) f = comp X Z W h (comp X Y Z g f).

Definition LocallySmallCat : prop
 := (idT' /\ compT') /\ (idL' /\ idR') /\ compAssoc'.
```

Every locally small category induces a metacategory, and this is part of the development.

The universe of all sets forms a locally small category, taking Hom-sets to be given by the function spaces (using the Aczel representation of functions [1]) defined during the treasure hunt (see also [11]) and imported via `CategoryPreamble.mgs`.

Finally *small categories* are defined. A small category is given by a set of objects and Hom-sets giving arrows. Every small category induces a locally small category and this is part of the development.

```
Section SmallCat.

Variable Obj: set.
Variable Hom: set -> set -> set.
Variable id: set -> set.
Variable comp: set -> set -> set -> set -> set -> set.

Definition idT'' : prop := forall X:set, X :e Obj -> id X :e Hom X X.
Definition compT'' : prop :=
 forall X Y Z:set, forall f g:set,
 X :e Obj -> Y :e Obj -> Z :e Obj ->
 f :e Hom X Y -> g :e Hom Y Z ->
 comp X Y Z g f :e Hom X Z.

Definition idL'' : prop :=
 forall X Y:set, forall f:set,
 X :e Obj -> Y :e Obj -> f :e Hom X Y -> comp X X Y f (id X) = f.

Definition idR'' : prop :=
 forall X Y:set, forall f:set,
 X :e Obj -> Y :e Obj -> f :e Hom X Y -> comp X Y Y (id Y) f = f.

Definition compAssoc'' : prop :=
 forall X Y Z W:set, forall f g h:set,
 X :e Obj -> Y :e Obj -> Z :e Obj -> W :e Obj ->
 f :e Hom X Y -> g :e Hom Y Z -> h :e Hom Z W ->
 comp X Y W (comp Y Z W h g) f = comp X Z W h (comp X Y Z g f).

Definition SmallCat : prop
 := (idT'' /\ compT'') /\ (idL'' /\ idR'') /\ compAssoc''.
```

A small category induces both a locally small category and a metacategory.

A small category can be encoded as an object in the set theoretic universe by forming a 4-tuple. The 4-tuple codes up the set of objects, the function giving Hom-sets, the function giving identity arrows and the function giving compositions.

```
Definition SmallCatAsObject : set :=
(Obj,
 (fun X :e Obj => fun Y :e Obj => Hom X Y),
 (fun X :e Obj => id X),
 (fun X :e Obj => fun Y :e Obj => fun Z :e Obj =>
  fun g :e Hom Y Z => fun f :e Hom X Y => comp X Y Z g f)).

End SmallCat.
```

In order to prove this works as expected, a number of results about tuples and functions are imported. These were defined and prove in the treasure hunt documents. A few new obvious lemmas were also needed. For example, there are the following very simple facts about numbers:

```
Lemma In_0_3 : 0 :e 3.
Lemma In_1_3 : 1 :e 3.
Lemma In_2_3 : 2 :e 3.
Lemma In_0_4 : 0 :e 4.
Lemma In_1_4 : 1 :e 4.
Lemma In_2_4 : 2 :e 4.
Lemma In_3_4 : 3 :e 4.
Lemma neq_3_0 : 3 <> 0.
Lemma neq_3_1 : 3 <> 1.
Lemma neq_3_2 : 3 <> 2.
```

Also, there are the following facts about 4-tuples:

```
Lemma tuple_4_0_eq : (x,y,z,w) 0 = x.
Lemma tuple_4_1_eq : (x,y,z,w) 1 = y.
Lemma tuple_4_2_eq : (x,y,z,w) 2 = z.
Lemma tuple_4_3_eq : (x,y,z,w) 3 = w.
```

These say that if the 4-tuple (x_0, x_1, x_2, x_3) is applied (as a set theoretic function) to the number $i \in \{0, 1, 2, 3\}$, then the result is x_i.

Chapter 9

Conclusion

Is your name really "Disruptive Influence"?
The Doctor, *The Caretaker* (2014)

With no power comes no responsibility.

Bibliography

[1] Aczel, P.: On relating type theories and set theories. In: T. Altenkirch, W. Naraschewski, B. Reus (eds.) TYPES, *Lecture Notes in Computer Science*, vol. 1657, pp. 1–18. Springer (1998)

[2] Agerholm, S., Gordon, M.: Experiments with ZF Set Theory in HOL and Isabelle. In: in Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications, LNCS, pp. 32–45. Springer-Verlag (1995)

[3] Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof, 2nd edn. Kluwer Academic Publishers (2002)

[4] Andrews, P.B., Bishop, M., Brown, C.E.: System description: TPS: A theorem proving system for type theory. In: D. McAllester (ed.) Proceedings of the 17th International Conference on Automated Deduction, *Lecture Notes in Artificial Intelligence*, vol. 1831, pp. 164–169. Springer-Verlag, Pittsburgh, PA, USA (2000)

[5] Andrews, P.B., Brown, C.E.: TPS: A hybrid automatic-interactive system for developing proofs. Journal of Applied Logic **4**(4), 367–395 (2006)

[6] Anonymous: The QED Manifesto. In: A. Bundy (ed.) CADE, *Lecture Notes in Computer Science*, vol. 814, pp. 238–251. Springer (1994)

[7] Barendregt, H., Wiedijk, F.: The challenge of computer mathematics. Transactions A of the Royal Society **363**, 2351–2375 (2005)

[8] Barras, B.: Sets in Coq, Coq in Sets. Journal of Formalized Reasoning **3**(1) (2010). URL http://jfr.unibo.it/article/view/1695

[9] Benzmüller, C., Paleo, B.W.: Gödel's proof of God's existence. In: J.Y. Beziau, K. Gan-Krzywoszynska (eds.) Handbook of the World Congress on the Square of Opposition IV, pp. 22–23 (2014). URL http://www.stoqatpul.org/lat/materials/hand_square2014.pdf

[10] Brown, C.E.: Combining type theory and untyped set theory. In: Automated Reasoning – Third International Joint Conference, IJCAR 2006, *Lecture Notes in Computer Science*, vol. 4130, pp. 205–219 (2006)

[11] Brown, C.E.: Reconsidering pairs and functions as sets. Tech. rep., Saarland University (2013). Technical Report of Article Submitted to Journal of Automated Reasoning

[12] Brown, C.E.: Reducing higher-order theorem proving to a sequence of sat problems. Journal of Automated Reasoning pp. 57–77 (2013)

[13] de Bruijn, N.: AUTOMATH, a language for mathematics. Tech. Rep. 68-WSK-05, T.H.-Reports, Eindhoven University of Technology (1968)

[14] de Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae (Proceedings) **34**(5), 381–392 (1972)

[15] de Bruijn, N.: Reflections on Automath. Eindhoven University of Technology (1990). Also in [35], pp 201–228

[16] Church, A.: A formulation of the simple theory of types. The Journal of Symbolic Logic **5**, 56–68 (1940)

[17] Constable, R.L., Allen, S.F., Bromley, M., Cleaveland, R., Cremer, J.F., Harper, R.W., Howe, D.J., Knoblock, T.B., Mendler, P., Panangaden, P., Sasaki, J.T., Smith, S.F.: Implementing Mathematics with the NuPRL Proof Development System. Prentice-Hall, Englewood Cliffs, NJ (1986)

[18] Coquand, T.: An analysis of Girard's paradox. In: Proceedings of the Symposium on Logic in Computing Science. IEEE, Cambridge, Massachusetts (1986)

[19] Frege, G.: Begriffschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens. Nebert, Halle (1879). Also in [25], pages 1–82

[20] Frege, G.: Grundlagen der Arithmetik. , Breslau (1884)

[21] Gordon, M., Melham, T.: Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic. Cambridge University Press (1993)

[22] Gordon, M.J.C.: Set theory, higher order logic or both? In: J. von Wright, J. Grundy, J. Harrison (eds.) Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, Proceedings, *Lecture Notes in Computer Science*, vol. 1125, pp. 191–201. Springer (1996). DOI 10.1007/BFb0105405. URL http://dx.doi.org/10.1007/BFb0105405

[23] Grothendieck, A., Verdier, J.L.: Théorie des topos et cohomologie étale des schémas - (SGA 4) - vol. 1, *Lecture notes in mathematics*, vol. 269. Springer-Verlag (1972)

[24] Harrison, J.: HOL light: A tutorial introduction. In: M. Srivas, A. Camilleri (eds.) Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96), *Lecture Notes in Computer Science*, vol. 1166, pp. 265–269. Springer-Verlag (1996)

[25] van Heijenoort, J.: From Frege to Gödel. A Source Book in Mathematical Logic 1879–1931. Harvard University Press, Cambridge, Massachusetts (1967)

[26] Hilbert, D., Ackermann, W.: Grundzüge der Theoretischen Logik, first edn. Die Grundlehren der Mathematischen Wissenschaften in Einzeldars tellungen, Band XXVII. Springer Verlag, Berlin (1928)

[27] Howard, W.: The formulas-as-types notion of construction. In: J. Seldin, J. Hindley (eds.) To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 479–490. Academic Press, New York (1980)

[28] Hurkens, A.J.C.: A simplification of Girard's paradox. In: Typed Lambda Calculus and Applications, pp. 266–278 (1995)

[29] Kaiser, J.: Formal Construction of a Set Theory in Coq. Master's thesis, Universität des Saarlandes (2012)

[30] Lambek, J., Scott, P.: Introduction to higher order categorical logic. Cambridge University Press, Cambridge, UK (1986)

[31] MacLane, S., Moerdijk, I.: Sheaves in Geometry and Logic: A First Introduction to Topos Theory. Springer-Verlag (1992)

[32] Matuszewski, R., Zalewska, A. (eds.): From Insight to Proof - Festschrift in Honour of Andrzej Trybulec, *Studies in Logic, Grammar, and Rhetoric*, vol. 10(23). The University of Bialystok, Polen (2007)

[33] The Coq development team: The Coq proof assistant reference manual. LogiCal Project (2004). URL http://coq.inria.fr. Version 8.0

[34] N. D. Goodman and J. Myhill: Choice Implies Excluded Middle. Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik **24**, 461 (1978)

[35] Nederpelt, R., Geuvers, J., Vrijer, R.d. (eds.): Selected Papers on Automath. Studies in Logic and the Foundations of Mathematics **133**. North-Holland, Amsterdam (1994)

[36] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)

[37] Obua, S.: Partizan Games in Isabelle/HOLZF. In: K. Barkaoui, A. Cavalcanti, A. Cerone (eds.) ICTAC, *Lecture Notes in Computer Science*, vol. 4281, pp. 272–286. Springer (2006)

[38] Plotkin, G.D.: LCF considered as a programming language. Theoretical Computer Science **5**, 223–255 (1977)

[39] Prawitz, D.: Natural deduction: a proof-theoretical study. Dover (2006)

[40] Quaife, A.: Automated deduction in von Neumann-Bernays-Gödel set theory. Journal of Automated Reasoning **8**(1), 91–147 (1992). DOI 10.1007/BF00263451. URL http://dx.doi.org/10.1007/BF00263451

[41] R. Diaconescu: Axiom of choice and complementation. Proceedings of the American Mathematical Society **51**, 176–178 (1975)

[42] Russell, B.: The Principles of Mathematics. Cambridge University Press (1903)

[43] Slind, K., Norrish, M.: A brief overview of HOL4. In: Theorem Proving in Higher Order Logics, pp. 28–32. Springer (2008). DOI 10.1007/978-3-540-71067-7_6. URL http://dx.doi.org/10.1007/978-3-540-71067-7_6

[44] Tarski, A.: Über unerreichbare Kardinalzahlen. Fundamenta Mathematicae **30**, 68–89 (1938)

[45] Trybulec, A.: Tarski Grothendieck set theory. Journal of Formalized Mathematics **Axiomatics** (2002). Released 1989

[46] Werner, B.: Sets in types, types in sets. In: M. Abadi, T. Ito (eds.) TACS, *Lecture Notes in Computer Science*, vol. 1281, pp. 530–346. Springer (1997)

[47] Whitehead, A., Russell, B.: Principia Mathematica. Cambridge University Press (1910^1, 1927^2)

[48] Wiedijk, F.: Is ZF a hack?: Comparing the complexity of some (formalist interpretations of) foundational systems for mathematics. J. Applied Logic **4**(4), 622–645 (2006). DOI 10.1016/j.jal.2005.10.011. URL http://dx.doi.org/10.1016/j.jal.2005.10.011

[49] Wiedijk, F.: Statistics on digital libraries of mathematics. In: Computer Reconstruction of the Body of Mathematics, *Studies in Logic Grammar and Rhetoric*, vol. 18(31). The University of Białystok, Poland (2009)

[50] Zermelo, E.: Neuer Beweis für die Möglichkeit einer Wohlordnung. Mathematische Annalen **65**, 107–128 (1908). English translation, “The Possibility of a Well-Ordering” in [25], pages 183–198