Dependent Types

Chris Grossack
Hype4Types

April 11, 2019

1 Why Types?

As programmers, types allow us to statically check certain aspects of code correctness at compile time instead of at runtime. At the most basic level, types allow us to prevent nonsensical code like the following without having to test it:

```
val ohNo = 3 + "hi"
```

A slightly more realistic example might be something like the following:

```
fun createWorld (seed : int) : world = ...

fun getRandomNumber () : int = ...
fun getUserInput () : string = ...

val createWorldRandomly () : world = createWorld (getRandomNumber ())
val createWorldFromUserSeed () : world = createWorld (getUserInput ())
```

This is a real bug from python code that I wrote years and years ago,[1] which took me weeks to debug. Why? Because python was automatically coercing strings to ints by summing the ascii values of characters in the string. This made it so that if you wanted to replay a randomly generated world, you couldn't simply copy the seed. A strong type system would have stopped this bug before the code could even run, and would have saved me a lot of headaches.

Of course, type systems allow us to prove much stronger properties about our code than even this. Any decent type system will satisfy the following two theorems, which together say that well typed code can never fail horribly:

Theorem (Progress)**.** *If $e : \tau$, then either e is a value or e can step to e'*

Theorem (Preservation)**.** *If $e : \tau$ and e steps to e', then $e' : \tau$*

The Progress theorem says that if e is well typed, then it is either a value or can step. The Preservation theorem says that if it steps, say to e', then e' is also well typed. So it is either a value or can step, and so on until the expression is either evaluated or loops infinitely.[2]

These theorems are all well and good, but where does this power come from? The answer is that types correspond to a **logic** whose "truth values" (judgements) are not true and false, but instead are types!

1.1 Logic

A logic is at the most basic level, a set of symbols, a set of judgements we can make about those symbols, and a set of rules which say which judgements hold of which symbols. For us, code fragments are the symbols, types are the judgements, and inference rules give us the power to prove that certain code fragments have certain types. Consider the following language:

[1] before I knew better than to write in python...

[2] Those interested in these theorems, and others like them, should take 15-312

$$e ::= n \mid e + e \mid s \mid s \hat{} s \mid \texttt{len}(e)$$
$$\tau ::= \texttt{nat} \mid \texttt{str}$$

Here, our expresseions are natural numbers n, strings s, sums of expressions $e_1 + e_2$, concatenation of expressions $e_1 \hat{} e_2$, and length of expressions $\texttt{len}(e)$. Our types, which are the judgements we make of these expressions, are `nat` and `str`.

But how do we prove that a certain expression has a certain type? We use the following rules:

$$\frac{}{n \ : \ \texttt{nat}} \qquad \frac{}{s \ : \ \texttt{str}} \qquad \frac{e_1 \ : \ \texttt{nat} \qquad e_2 \ : \ \texttt{nat}}{e_1 + e_2 \ : \ \texttt{nat}} \qquad \frac{e_1 \ : \ \texttt{str} \qquad e_2 \ : \ \texttt{str}}{e_1 \hat{} e_2 \ : \ \texttt{str}} \qquad \frac{e \ : \ \texttt{str}}{\texttt{len}(e) \ : \ \texttt{nat}}$$

The statements above the bar are the assumptions which allow us to prove the statement below the bar. Rules which have no assumptions are called axioms.

Now we can derive that $\texttt{len}(\text{"hi"}) + 2 \ : \ \texttt{nat}$.

$$\frac{\dfrac{\dfrac{}{\text{"hi"} \ : \ \texttt{str}}}{\texttt{len}(\text{"hi"}) \ : \ \texttt{nat}} \qquad \dfrac{}{2 \ : \ \texttt{nat}}}{\texttt{len}(\text{"hi"}) + 2 \ : \ \texttt{nat}}$$

Of course, with a type system as simple as this, the type checker can simply check all possible (read: both) types for $\texttt{len}(\text{"hi"}) + 2$, and can reverse engineer what a proof must look like. Important to note, however, is that no matter how complicated the type system may be, it is *always* decidable (even efficient!) to check if a given proof is valid. For example, the following proof is not valid (why?):

$$\frac{\dfrac{}{2 \ : \ \texttt{nat}}}{\texttt{len}(2) \ : \ \texttt{nat}}$$

Now, if a type system is just a system of logic, what's stopping us from modeling everything we could possibly want in it? The answer, of course, is nothing at all.

1.2 Curry-Howard Correspondence

The Curry-Howard Correspondence says *Types are Propositions and Programs are Proofs*. But what is meant by this? Simply put, we can think of a program $a : A$ as being a proof of A. Then we can prove $A \wedge B$ by providing a proof of A and a proof of B, so we represent this proposition by the type $A \times B$ of ordered pairs. Similarly, we can represent $A \vee B$ as $A + B$, the disjoint sum of A and B. A proof of $A \to B$ should take a proof of A and produce a proof of B, and indeed the type of functions $A \to B$ does this. We add a type $\mathbb{1}$ with exactly one value $* : \mathbb{1}$. This represents true, since we know it is inhabited. We also represent false with $\mathbb{0}$, a type with no values. With this in hand, we define $\neg A$ as $A \to \mathbb{0}$, as if A is false, then a proof of A is a contradiction.

This machinery is already powerful enough to prove propositional tautologies, such as

$$(A \to B \to C) \to (A \times B \to C)$$

and

$$A \to B \to A$$

$\lambda(f : A \to B \to C).\lambda((a, b) : A \times B).f \ a \ b$ is a program inhabiting the first type. Interpreted logically, this program is a proof of the claim given by the type. Similarly, $\lambda(a : A).\lambda(b : B).a$ inhabits the second type, and is thus a proof of that claim.

However these proofs seem very limited in scope. In particular, they have no quantifiers. We are proving propositional tautologies, but what if we want to prove something about, say $\mathbb{N}$? Also, how can we use these to prove complex properties about code?

2 Dependent Types

The big idea is to allow types to *depend* on values. In the same way that a proposition of the natural numbers $\varphi(x)$ might be true for some values of x and not others, we want a type $P(x)$ which might be inhabited for some values of x and not others! Thus, in the same way $\varphi(x)$ can be seen as a *family* of propositions, one for each x, we have $P(x)$ a family of types, one for each x.

As an example, consider the proposition $\varphi(x) = x > 5$. This doesn't have a truth value, because for some x it's true, and for some x it isn't. Thus, it makes no sense to try to prove such a claim. Given any particular x, say 7, it does make sense to try and prove $\varphi(7) = 7 > 5$ (we will, of course, succeed).

Similarly, we want a type system which is expressive enough to let us define a *type* $A(x) = x > 5$. Values inhabiting $A(7)$, then, will correspond to proofs of the fact that $7 > 5$. We say that $A(x)$ is a *dependent type* because its values *depend* on the x we choose. One way of viewing this is as a function $A : \mathbb{N} \to \texttt{Type}$.

Of course, the most important proposition is equality, and so we assume the existence of an Identity Type. Namely, if $x : A$, then $\texttt{refl}_x : x = x$. This says that reflexivity is a proof that $x = x$.

As an example of a type-valued function, consider the following:

```
notEmpty : [A] -> Type
notEmpty []       = 𝟘
notEmpty (x::xs)  = 𝟙
```

This is all well and good, but how do we use these dependent types? Just like $\varphi(x)$ is not very useful until we add quantifiers $\exists x.\varphi(x)$ and $\forall x.\varphi(x)$, we can start really using dependent types once we have quantifiers in our type system.

2.1 Sigma Types

$\Sigma_{x:A} P(x)$ is defined to be the disjoint union of $P(x)$ for each $x : A$. The notation comes from the equivalent definition:

$$\Sigma_{x:A} P(x) = \underbrace{P(x_1) + P(x_2) + P(x_3) \ldots}_{x_i : A}$$

Thus, when $P(x)$ is a constant function, say B, then

$$\Sigma_{x:A} B = \underbrace{B + B + B \ldots}_{x:A} = A \times B$$

This often causes confusion, as Sigma types are called "dependent products" by some, and "dependent sums" by others. This is purely because repeated addition looks like multiplication. The topic itself is not very difficult, even if the language makes it seem that it is. As an example, consider

```
P : Bool -> Type
P T = ℕ
P F = ℤ
```

A value inhabiting $\Sigma_{x:\texttt{Bool}} P(x)$ has two main ingredients. It keeps track of whether it is in $P(T)$ or $P(F)$, and it keeps track of which value it has in the associated type. So $(T, 3) : \Sigma_{x:\texttt{Bool}} P(x)$ is the $3 : \mathbb{N}$ living in $P(T)$. Similarly, $(F, -7) : \Sigma_{x:\texttt{Bool}} P(x)$ is the $-7 : \mathbb{Z}$ living in $P(F)$.

More generally, every value of $\Sigma_{x:A} P(x)$ is a tuple whose first element comes from A and whose second element comes from $P(x)$, where $x : A$ is the element in the first position. This, again, is what causes the confusion regarding dependent sums vs dependent products.

Now we know what Sigma types *are*, but we still aren't sure why they are useful. In fact, they correspond to two incredibly important ideas. First, and most obviously, $\Sigma_{x:A} P(x)$ corresponds to $\{x : A \mid P(x)\}$ in set theory. This is because the values of $\Sigma_{x:A} P(x)$ are of the form (x, p_x) where $x : A$ and $p_x : P(x)$ is a proof that $P(x)$ holds. Second, and equally usefully, $\Sigma_{x:A} P(x)$ corresponds to $\exists(x : A).P(x)$. Why? Because it is inhabited exactly when some (x, p_x) inhabits it. But then this x witnesses $\exists(x : A).P(x)$!

The correspondence between these two ideas is extremely deep, and is responsible for a lot of very powerful structure which exists in type theory, but which is absent in traditional set theory. This is one of the big selling points for replacing set theory with type theory as the foundations of mathematics.

2.2 Pi Types

Of course, where there is an existential quantifier there must be a universal quantifier not far away. They exist in the form of Pi Types, written $\Pi_{x:A}P(x)$. These are tuples indexed by A. Namely:

$$\Pi_{x:A}P(x) = \underbrace{P(x_1) \times P(x_2) \times P(x_3)\ldots}_{x_i:A}$$

Again, it is clear that if $P(x)$ is a constant function, say B, then:

$$\Pi_{x:A}B = \underbrace{B \times B \times B\ldots}_{x_i:A} = B^A = A \to B$$

Indeed, any tuple $(b_x)_{x:A}$ corresponds exactly to a function $\lambda(x:A).b_x$, and any function $f : A \to B$ corresponds to a tuple $(f\ x)_{x:A}$. Thus, $\Pi_{x:A}P(x)$ is the type of functions from A to $P(x)$, where the output type is allowed to depend on the input to the function. As an example, consider the following:

```
P : Bool -> Type
P T = ℕ
P F = Bool

f : Π_{x:Bool} P(x)
f T = 7
f F = F
```

Of course, these functions give us the power of universal quantification. If $f : \Pi_{x:A}P(x)$, then f assigns to each $x : A$ a proof $p_x : P(x)$. Thus, each $P(x)$ must be inhabited, and so $\forall(x:A).P(x)$!

3 Programming With Dependent Types

Now that we've developed this machinery, let's actually write some code!

```
(* we define > to be a type valued relation on ℕ *)
(* we are using the standard definition of ℕ as  *)
(* 0 or S n, where S is the successor function.  *)
> : ℕ -> ℕ -> Type
0   > 0   = 𝟘
S m > 0   = 𝟙
S m > S n = m > n

(* for completeness, we will also define a length function *)
len : [A] -> ℕ
len [] = 0
len x::xs = S (len xs)


(* get the nth element of a list of length > n *)
(* n is the index to retrieve, xs is the list  *)
(* to retrieve it from, and p is a proof that  *)
(* xs has length > n                           *)
nth : Π_{n:ℕ} Σ_{xs:[A]}(len(xs) > n) -> A
nth 0     (x::xs,p) = x
nth (S n) (x::xs,p) = nth n (xs,p)
```

A reasonable person might wonder why p works as a proof of both $\texttt{len}(\texttt{x} :: \texttt{xs}) > \texttt{S n}$ and $\texttt{len}(\texttt{xs}) > \texttt{n}$. The answer, it turns out, is fairly simple. Our types can now have expressions, just like our programs can have expressions. Because of this, we occasionally have to evaluate our types! Notice that, by definition,

$$\texttt{len}(\texttt{x} :: \texttt{xs}) > \texttt{S n} \equiv \texttt{S}(\texttt{len}(\texttt{xs})) > \texttt{S n} \equiv \texttt{len}(\texttt{xs}) > \texttt{n}$$

Here we are using $\equiv$ to denote definitional equality, which captures the meta-logical notion of "evaluation", to distinguish it from $=$, which we use for the equality type inside the logic itself. The first equivalence follows from the definition of `len`, and the second from the definition of $>$. Notation aside, however, this means the two types are definitionally the same. In particular, they have the same values! Thus p indeed works for both. Then we also know that when we completely evaluate the type[3] $p : \mathbb{1}$, and so $p \equiv *$.

Since we know what p must be, then, can we trick the type system? What happens if we call `nth 0` $([], *)$? Thankfully, $([], *)$ does not have type $\Sigma_{xs:[A]}\texttt{len}(\texttt{xs}) > \texttt{0}$, since

$$\texttt{len}([]) > 0 \equiv 0 > 0 \equiv \mathbb{0}$$

but $*$ does not have type $\mathbb{0}$. Thus the code won't typecheck, and we can sleep soundly knowing our `nth` function is safe.

In this system, we have the power to truly enforce the correctness of our code in the type system. It would be extremely tedious, but finally we have the ability to statically guarantee that if it compiles, it must be correct!

4 Doing Math With Dependent Types

Of course, it seems like such a waste using this powerful a logical system purely for programming... Earlier on, we even alluded to the fact that some people want to use type theory[4] as an alternate foundation for mathematics. The reasoning for this is dual to that of programming in this type theory. Since our proofs are programs, and typechecking them is decidable, we can have a computer guarantee that a proof is correct! As mathematics progresses, important proofs become progressively harder to understand, and humans make mistakes. There are countless papers which have been published, only to have somebody find a counterexample which was due to a subtle reasoning error which we humans overlooked. Thankfully, computers (and importantly, typecheckers) are very good at refusing to overlook subtle errors, as anyone who has tried to write a substantial code base in SML will tell you.

There are other reasons to do math in type theory. In set theory, we are assumed to be working in a logical system whose purpose is to act on totally distinct objects called "sets". In type theory, however, the set-like objects and the logic-like objects are both types! This correspondence allows us great power and flexibility which we don't have in a set theoretic universe. Additionally, there are proof tools, such as path induction and univalence, which are outright *false* in a set theoretic universe. These tools can often be helpful in simplifying proofs from set theory, or even in proving new results altogether!

Let's take a quick moment to see what doing math in Type Theory is all about. We'll cover this topic in much greater depth during the Homotopy Type Theory lecture, but it's good to see some examples here.

Every type comes equipped with an induction principle which lets us prove something about every value of that type. For example:

$$\begin{aligned}
\texttt{ind}_{\mathbb{1}} &: P(*) \to \Pi_{x:\mathbb{1}} P(x) \\
\texttt{ind}_{\mathbb{0}} &: \Pi_{x:\mathbb{0}} P(x) \\
\texttt{ind}_{\mathbb{N}} &: P(0) \to \Pi_{n:\mathbb{N}} \left(P(n) \to P(Sn)\right) \to \Pi_{x:\mathbb{N}} P(x) \\
\texttt{ind}_{=} &: P(x, x, \texttt{refl}_x) \to \Pi_{y:A} \Pi_{\alpha:x=y} P(x, y, \alpha)
\end{aligned}$$

$\texttt{ind}_{\mathbb{1}}$ says that if you can prove $P(*)$, then you've proven $P(x)$ for every $x : \mathbb{1}$. Similarly, $\texttt{ind}_{\mathbb{0}}$ says that every property holds of every value of the empty type. Finally, $\texttt{ind}_{\mathbb{N}}$ is the familiar principle of induction on $\mathbb{N}$. The last induction principle says how to prove properties about equality types. Since the only constructor is `refl`, it suffices to prove a property holds of `refl` for the same reason it suffices to prove a property holds of just $*$ when using $\texttt{ind}_{\mathbb{1}}$. Using these inductive tools, we can prove properties about any type we introduce. As examples, let's prove a basic property about equality, and a basic property of $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$.

[3] that's a strange sentence...

[4] precisely, Martin-Löf dependent type theory, which is (conveniently) what we're studying here

4.1 Example Proofs

Let's start by showing $\Pi_{f:A\to B}\Pi_{x:A}\Pi_{y:A}\ x = y \to fx = fy$, that is, if $x = y$, then $fx = fy$.

Let $P(x, y, \alpha) \equiv fx = fy$.

If we can prove $p_0 : P(x, x, \mathtt{refl}_x)$, then $\mathtt{ind}_{x=}\ p_0 : \Pi_{y:A}\Pi_{\alpha:x=y}P(x, y, \alpha)$. But $P(x, x, \mathtt{refl}_x) \equiv fx = fx$, and $\mathtt{refl}_{fx}$ has this type![5] Thus $\mathtt{ind}_{x=}\ \mathtt{refl}_{fx} : \Pi_{y:A}\Pi_{\alpha:x=y}P(x, y, \alpha)$. Of course, since P doesn't actually depend on α, this is the same type as $\Pi_{y:A}\ x = y \to P(x, y, \alpha)$. Then

$$\lambda f.\lambda x.\mathtt{ind}_{x=}\ \mathtt{refl}_{fx} : \Pi_{f:A\to B}\Pi_{x:A}\Pi_{y:A}x = y \to fx = fy$$

as desired. For convenience, let's call this function $\mathtt{ap}_f\ x\ y\ \alpha$.

Now, a definition:

$$0 + n = n$$
$$(Sm) + n = S(m + n)$$

Let's prove $\Pi_{m:\mathbb{N}}\ m + 0 = m$. Notice this is not necessarily clear from the definition, even though we know it to be true.

Informally, we induct on m. If $m = 0$, then by definition $m + 0 = 0 + 0 = 0 = m$. If $m = Sm'$, then $m + 0 = (Sm') + 0 = S(m' + 0) = Sm' = m$, where the third equality follows from the inductive hypothesis. Let's *formalize* this argument, by finding a program witnessing the truth of this proposition.

Let $P(m) \equiv m + 0 = m$. Then $\mathtt{ind}_{\mathbb{N}}$ will return a value of type $\Pi_{m:\mathbb{N}}P(m)$ if we can provide it a value $p_0 : P(0)$ and a value $p_{is} : \Pi_{n:\mathbb{N}}\,(P(n) \to P(Sn))$.

Now, $P(0) \equiv 0 + 0 = 0 \equiv 0 = 0$ by computation, and we know $\mathtt{refl}_0 : 0 = 0$, so $p_0 = \mathtt{refl}_0$ will work. As for the inductive step, we want a function p_{is} which takes in an n, and takes in a proof $p_n : P(n)$, and then returns a value $p_{Sn} : P(Sn)$. Such a function starts $\lambda n.\lambda p_n.$, but what should we return? Well, it should have type $P(Sn) \equiv (Sn) + 0 = (Sn) \equiv S(n + 0) = Sn$, but $p_n : n + 0 = n$, and so $\mathtt{ap}_S\ (n + 0)\ n\ p_n : (S(n + 0) = Sn)$.

So $p_{is} \equiv \lambda n.\lambda p_n.\mathtt{ap}_S\ (n + 0)\ n\ p_n$ works, and $\mathtt{ind}_{\mathbb{N}}\ p_0\ p_{is} : \Pi_{n:\mathbb{N}}P(n)$, equivalently, $\Pi_{n:\mathbb{N}}n + 0 = n$. As desired.

[5] and now you see why we use $\equiv$ to denote definitional equality