Basic Dependent Typing

Disclaimer: Like anything else I've talked about in this class, the system presented in this document is, at best, an incomplete fragment. Real implementations of real dependent type systems (such as Idris) will handle things slightly differently.

The World of Types and Values

What is the difference between a value, like `3`, and a type, like `int`? One answer that immediately comes to mind is that types are used to *describe* values, which are themselves primitive objects. These descriptions are then used to verify properties of a program, like making sure that we never try to add `"fish"` and `true`. In nearly any statically-typed programming language, "types" and "values" live in two different worlds, and don't overlap at all.

In many ways, a lot of fancy tricks and contructions used by late-stage functional programmers (phantom types, monad transformers, etc) are in fact ways to *lift* information from the value level to the type level, where the compiler can assist us in ensuring that code is correct.

A major limitation of this approach is that most languages don't have an expressive enough type-level language to encode "interesting" properties about a type. Recall the example of length-encoded lists in OCaml –

```
type z
type S of 'a

val cons: 'a * ('a, 'l) list -> ('a, 'l S) list
val append: ('a, 'l1) list -> ('a, 'l2) list -> ('a, ???) list
```

Even if we can encode the natural numbers into our typesystem, we have no way of writing inductive functions on these types! In fact, we can't do *anything* besides apply constructors, identity or constant functions! While there are other, less powerful tools we can add to our typesystem (typeclass elision, type families, etc), the logical extreme would be to *remove* the distinction between types and values entirely, so that type constructions can have the full expressive power of our value language!

Note that this is slightly different from "lifting" the entire value world into the

type world, which would imply that the value world is still separate. Rather, we are "dropping" all types and type constructors into the "value" world, making them first-class objects that can be freely interspersed with "regular" values. For example, we could write a function like this:

```
(* f: bool -> type *)
fun f true = int
  | f false = bool
```

Of course, merging the type-level and value-level languages *also* means that we can refer to these functions where a type is expected:

```
fun int_or_bool (b:bool) : f b =
  if b then 0 else false
```

Notice that, unlike what could be accomplished with a GADT, the input type is "normal", and it's the *output* type that's parameterised over the input.

This means that we can now properly refer to regular values in our types:

```
type _ vec =
  | Nil : ('a, 0) vec
  | Cons : 'a * ('a, n) vec -> ('a, n+1) vec
```

. . . as well as use regular functions on those values:

```
val append : ('a, n) vec -> ('a, m) vec -> ('a, n+m) vec
```

Values with types in the `vec` family are the first we'll see of the vaunted "dependent data", in which the type of the object *depends* on a value. Note that the `vec` type constructor itself can be given a proper type in this system:

```
val vec : type * nat -> type
```

which tells the typechecker that we really are allowed to use the `+` operator in the type of `append`.

What's in a proof?

Recall the Curry-Howard isomorphism, which states that "types are theorems and programs are proofs". For example, the type $A \times B$ is analogous to a proof of the theorem $\overline{A} \wedge \overline{B}$ (where $\overline{A}$ refers to the theorem represented by A) – if x is a proof of $\overline{A}$, and y is a proof of $\overline{B}$, then the tuple (x, y) (really, having both proofs) corresponds to having a proof of "$\overline{A}$ and $\overline{B}$". Similarly, sum types (disjoint unions) correspond to "or", and function arrows correspond to implication.

But these connectives (along with true and false) only correspond to a small fragment of logic. In particular, they allow us to encode a *propositional* logic over some other theory, which tends not to be very interesting – for one, pure propositional logic is decidable.

To upgrade into first-order logic, then, we need to add *binding* connectives, namely $\forall$ and $\exists$ ("forall" and "there exists"). What might proof terms for these look like?

Consider the arithmetic sentence $x > 2$. As written, with no other context, this isn't really a "proposition" in the sense that it can't really be true or false – we need to know what x is! This is a proposition with a "hole" in it, that gets filled by a *particular* value of x by some context.

In propositional logic, that's it – these statements are meaningless, unless we attach some external meaning to the symbol x. To use the language of logicians, we say that the variable x is *free*, or *unbound*, which is to say that it hasn't been given a meaning *yet*. If we have a given value for x, say 4, then we can transform (via substitution) the sentence $x > 2$ into the sentence $4 > 2$, which we can then evaluate for some truth value. We might say that a sentence with free variables is a function from "values" to "propositions" whose action is substitution. Alternatively, we can *bind* the variables via a quantifier like $\forall$ or $\exists$ – in a predicate like $\exists(x \in \mathbb{N}).x > 2$, the variable x *has* some meaning, namely that as a stand-in for some unknown value v bearing the property $v > 2$.

Higher-order proof terms

What does this mean in terms of promoting our Curry-Howard construction to first-order logic?

Remember that distinct terms (expressions, values) of some type τ represent distinct *proofs* of the proposition given by $\overline{\tau}$, and that propositions are defined to be *true* (in our system) when such a proof exists. Correspondingly, there exists a type for every proposition we might want to examine. Combined with the prior observation about free variables, we find that $x > 2$ acts as a function from *integers* to *types*! This suggests that this logic-/proof-based view has some similiarities with our "single world" type system above. In fact, they coincide completely!

Consider the predicate $\forall(x : \alpha).P(x)$ (where $P(x)$ is some proposition with x as a free variable). If this statement is true, then for any element x of type α, there exists a *proof* of the proposition $P(x)$. That is, we can generically produce a term of *type* $P(x)$, where x is arbitrary!

This should sound familiar – this is *almost* a description of a function! If we have a term $f : \alpha \to \beta$, it means that, given any $x : \alpha$, the term $f(x)$ has type β. The big difference is that we're allowing the proposition $P(x)$ to *depend* on the value x! This is a *dependent function* or *dependent product* type, and is denoted as

$$\prod_{x:\alpha} P(x)$$

which represents the predicate $\forall(x : \alpha).P(x)$ if $P : \alpha \to$ `type`.

As we've just noticed, a *value* of this term should be very similar to a regular function. In fact, if we choose $P(x)$ to be a constant function, say $P(x) = \beta$ for some type β, then we see that

$$\prod_{x:\alpha} P(x) = \prod_{x:\alpha} \beta$$

is a "function" that takes terms of type α to terms of type β. That's just a *regular* function! So we can generalize our typical λ abstraction to produce a *dependent* function type instead, where $\alpha \to \beta$ is shorthand for $\prod_{x:\alpha} \beta$.

What about the predicate $\exists(x : \alpha).P(x)$? In a constructive setting, proving this entails *producing* the value x, along with a *proof* that it satisfies the proposition $P(x)$. That is, you can think of it as a *value* coupled with a *proof*. But a *value* of some type is just *another proof*, of this other type! We already have an idea of what a "pair of proofs" looks like under Curry-Howard – that's just a tuple!

So the proposition $\exists(x : \alpha).P(x)$, given as the type

$$\sum_{x:\alpha} P(x)$$

has a *tuple* as its proof term, where the first element is x, and the second element has type $P(x)$. These are the so-called *dependent tuples*, or *dependent sums*. Just as before, we can recover our original product type by fixing $P(x) = \beta$, giving us that $\alpha \times \beta$ is shorthand for $\sum_{x:\alpha} \beta$.

You may wonder why tuples (typically products) are dependent sums, while functions (typically exponentials) are dependent products. One way to think of it is that a "product" is an iterated sum, and an exponent is an iterated product. Alternatively, you can think of binary sums and products as being indexed by the values `0` and `1` (for example, the "0^{th}" index of a tuple is its left element), and we are simply generalizing the index type. Finally, we can recover regular sums by parameterizing our sigma with the type $\not\models$ with elements `true` and `false` – then `inL x` is shorthand for `(false, x)`, and `inR y` is shorthand for `(true, y)` (or vice versa).

Refinement types

A form that many real dependently-typed languages use are "refinement types", which allow us to annotate the types of functions to restrict their inhabitants according to some proposition. For example, we might type a function like this:

```
('a -> bool) -> {v : ('a, n) vec} -> {v' : ('a, m) vec | m < n}
```

(it's worth noting that the `'a` and `n` in the type above are implicitly being quantified over)

This is known as a *refinement* type, and is the syntax used by Idris and Liquid Haskell, two type-based proof systems. In fact, when fully unrestricted, this is equivalent to the construction above. The exact conversion is a bit involved so we won't detail it here[1], but the type `{x:a | P(x)}` is effectively $\sum_{x:a} P(x)$, and function arrows `{x:a} -> P(x)` are $\prod_{x:a} P(x)$.

Reflection

We've made references to types like "the type corresponding to the proposition $5 < 2$", but we haven't specified what a proof of this may be! In general, 0th order propositions (those that don't use $\forall/\prod$ or $\exists/\sum$) don't contain any information content beyond their truth value. If "there exists some x such that $P(x)$" is true, then we should, in principle, be able to produce the value x (in a constructive setting). On the other hand, there is no "value" associated with the truthhood of $2 < 3$, not even an implicit one like a lambda abstraction.

In the past, we have used the "unit" type $\mathbb{1}$ containing only the value `()` to stand in for types that are inhabited but contain no meaningful information. In dependent type theory, we tend to instead call this value `Refl`, for "reflexivity". In particular, for any **value** $x : \alpha$, we say that $\texttt{Refl}_{\alpha}$ has type $x = x$. Note that, in this case, x is a *type-level* value, which means that $\texttt{Refl}_{\texttt{nat}}$ also serves as a witness to, say, `2+3 = 5`. We can generalize this to arbitrary propositions, where we can say that `Refl` inhabits the type *true* (really, `()` and $\mathbb{1}$), and that the type $false$ (*void*) has no inhabitants. So the type $2 < 3$ stepts to the type *true*, which is inhabited by `Refl` – so `Refl` is a proof of `2 < 3`. On the other hand, $3 < 2$ steps to $false$, which is a type with no inhabitants – so there cannot be a closed-form proof of $3 < 2$.

A note on decidability

Now that our *types* must also move towards a normal form, and that we must know the normal form of a given type in order to typecheck a dependently-typed program, a natural next question is whether typechecking is even possible in the general case.

The original intuitionistic type theory as devised by Martin-Löf actually sidesteps this issue in disallowing general recursion. Instead, the method of iteration was *induction*, in which a "recursive call" was not actually a call to a function, but instead was considered a parameter only available when defining an inductive function. In this way it could be enforced that any "recursive call" was only

[1] Care needs to be taken regarding the *parity* of types when converting between refinements and standard dependent types. Briefly, types to the left of a function arrow are considered to be in *negative* position, and types to the right are in *positive* position (and this composes as you'd expect), and the conversion is different when performed on positive vs negative types.

performed with parameters that were strictly structurally smaller, and that thus the computation halted.

Of course, programmers typically enjoy having Turing completeness available to them as a toole, and as such like to have general recursion as a tool. Languages like Idris will instead implement a *totality checker*, which will refuse to typecheck any code with type-level expressions that cannot be proven to halt.

Universality

One more issue remains unresolved – what is the type of `type`? If we allow `type: type`, then you can, without too much difficulty, encode Girard's Paradox (a type-theory equivalent of the classic Russell's Paradox) into the system, which demonstrates the unsoundness. Some languages, like Haskell[2], are fine with this – these languages don't style themselves as formal proof systems, and instead view dependent types as a way to show correctness for *programs*, of which nearly no real-world example will perform the contortions necessary to get an unsound result.

To resolve this, however, we need to introduce a similar hierarchy as proposed by Russell himself with his Theory of Types (which is actually unrelated to the type theory discussed here). Instead of keeping a single type `type` to contain all types, we actually produce a hierarchy $\texttt{type}_n$, where $\texttt{type}_{-1}$ is the type of values, $\texttt{type}_0$ is the type of "regular" types, and $\texttt{type}_{n+1}$ contains the type $\texttt{type}_n$. This stratification is often defined such that $\texttt{type}_{n+1}$ is a strict supertype of $\texttt{type}_n$, containing all "smaller" types but also some "larger" ones. In doing so, we can then simply use the shorthand `type` to refer to the smallest $\texttt{type}_n$ such that everything in question has this type.

Note that there does *not* exist a function mapping n to $\texttt{type}_n$ for all natural numbers n – there is no type "large" enough to serve as its codomain. In fact, these universes of type theory are not truly indexed by natural numbers at all, but the specifics of this are beyond the scope of this paper.

Examples

The following examples and explanations are not written in any real-world dependently-typed programming language known to the author. However, the syntax and approach should be adaptable to most dependent languages without much issue.

In the formal theory, any induction term must use the *inductor* (or "elimination form") of a given type to reduce it to strictly smaller syntactic components.

[2]Haskell isn't actually dependent (yet), but you can lift a lot of expression-level properties into the type-level language (with great difficulty) via language extensions, the typeclass system and type families.

However, we will elide these, and instead use the more-familiar syntax of recursion and pattern-matching. Then we can write

```
fun f 0 = e1
  | f (s(n)) = e2
```

instead of the much more intimidating $ind_{\texttt{nat}}(\_, e_1, \lambda k, r.e_2)$.

We'll also use the following definitions:

```
fun (+) 0      m = m
  | (+) (s(n)) m = s(n+m)

fun (<=) 0      m      = true
  | (<=) (s(n)) 0      = false
  | (<=) (s(n)) (s(m)) = n <= m
```

Theorem proving

It is well known that, if $n <= m$, then there exists some k such that $n + k = m$; this is commonly known as "subtraction". We will show that this is true for all naturals n and m by producing a term of the appropriate type.

What is that type? The theorem might be phrased in first-order logic like this:

$$\forall(n : \texttt{nat}).\forall(m : \texttt{nat}).((n < m) \implies \exists(k : \texttt{nat}).n + k = m)$$

Translating into the language of types, we then get

$$\prod_{n:\texttt{nat}} \prod_{m:\texttt{nat}} ((n < m) \to \sum_{k:\texttt{nat}} (n + k = m))$$

Now we need to write a type of this term. The outermost two connectives are both $\prod$, so we start with two lambda abstractions:

```
fun subtraction_proof n m = _todo
```

Now, we need to produce a value of type $(n < m) \to \sum_{k:\texttt{nat}}(n + k = m)$. In fact, this is another function, so we add the next parameter:

```
fun subtraction_proof n m p = _todo
```

Now what? Let us outline the proof using natural language:

> We proceed by induction.
>
> If `n = 0`, then choose `k = m`. Then `0 + m = m` by definition of `(+)`.
>
> Otherwise, `n = s(x)` for some `x`.
>
> If `m = 0`, then we have that `s(x) <= 0`, a contradiction. So we must have `m = s(y)` for some `y`. By definition of `(<=)`, we have `x <= y`

from `s(x) <= s(y)`, so there exists some `k'` such that `x+k' = y` by the inductive hypothesis.

Choose `k = k'`. Then `s(x) + k' = s(x+k') = s(y)` by definition of `(+)`

Let's translate this into a formal proof using dependent types.

The first step is to case on whether `n = 0`. That sounds like a pattern match...

```
fun subtraction_proof 0 m p = _todo_z
  | subtraction_proof (s(x)) m p = _todo_s
```

`_todo_z` should have type $\sum_{k:\texttt{nat}}(0+k=m)$, which is a tuple. In the proof, we said "choose `k = m`, which suggests

```
fun subtraction_proof 0 m p = (m, _todo_proof)
  | subtraction_proof (s(x)) m p = _todo_s
```

Finally, `_todo_proof` should have `0+m = m`. Just our luck, this is exactly the first clause of the definition of `(+)`, so we can conclude this via `Refl`.

```
fun subtraction_proof 0 m p = (m, Refl : (0+m=m))
  | subtraction_proof (s(x)) m p = _todo_s
```

(where we explicitly annotated the `Refl` type used for clarity)

Next up is `_todo_s`. Once again, we checked whether `m` was `0`...

```
fun subtraction_proof 0 m p = (m, Refl : (0+m=m))
  | subtraction_proof (s(x)) 0 p = _todo_m_z
  | subtraction_proof (s(x)) (s(y)) p = _todo_m_s
```

If `m = 0`, then we said that this is a contradiction. How do we express that with dependent types? Well, remember that we take in a *proof* that `n <= m`, which in this case is the value `p : s(x) <= 0`. But as `s(x) <= 0` evaluates to `false` by definition of `(<=)`, the *type* `s(x) <= 0` is actually empty! So `p` is actually an element of the *void* type, which we can eliminate via the `abort` (or `absurd`) function that crashes the program when given a value of type `void`.

```
fun subtraction_proof 0 m p = (m, Refl : (0+m=m))
  | subtraction_proof (s(x)) 0 p = abort p
  | subtraction_proof (s(x)) (s(y)) p = _todo_m_s
```

Now for the tricky inductive step. We cited the inductive hypothesis on `x` and `y`, which corresponds to a recursive call of `subtraction_proof`. But what `p` do we use in this call?

Well, by definition, `s(x) <= s(y)` steps to `x <= y`, so they are actually the same type! This means we can use the same `p`:

```
fun subtraction_proof 0 m p = (m, Refl : (0+m=m))
  | subtraction_proof (s(x)) 0 p = abort p
  | subtraction_proof (s(x)) (s(y)) p =
```

```
      let (k, pf) = subtraction_proof x y p in
      _todo_sum
```

Finally, we said that we can use the same `k` as our witness for this case, giving

```
fun subtraction_proof 0 m p = (m, Refl : (0+m=m))
  | subtraction_proof (s(x)) 0 p = abort p
  | subtraction_proof (s(x)) (s(y)) p =
      let (k, pf) = subtraction_proof x y p in
      (k, _todo_proof)
```

What can we use for `_todo_proof`? It should have type `s(x) + k = s(y)`, which, by definition of `(+)`, is the same type as `x + k = y`, so we can use the same `pf` returned to us by the inductive hypothesis –

```
fun subtraction_proof 0 m p = (m, Refl : (0+m=m))
  | subtraction_proof (s(x)) 0 p = abort p
  | subtraction_proof (s(x)) (s(y)) p =
      let (k, pf) = subtraction_proof x y p in
      (k, pf)
```

Dependent Types as Contracts

Coming soon! The above should be enough to complete HW13.

References

Dependent Type Theory — Theorem Proving in Lean 3.4.0 documentation. (2017). Retrieved December 2, 2019, from Github.io website: https://leanprover.github.io/theorem_proving_in_lean/dependent_type_theory.html

Lafont, Y. (2011). Introduction to dependent type theory. Retrieved from l'INSTITUT DE MATHÉMATIQUES DE MARSEILLE website: http://iml.univ-mrs.fr/~lafont/HETT/coquand1.pdf

The Idris Tutorial — Idris 1.3.1 documentation. (2017). Retrieved December 2, 2019, from Idris-lang.org website: http://docs.idris-lang.org/en/latest/tutorial/

The Univalent Foundations Program, & Institute For Advanced Study (Princeton, N.J. (n.d.). Homotopy type theory : univalent foundations of mathematics.