

CSC324 Lecture 16

22 July 2020

Administrivia

The class voted for the 10th exercise to be included. So it will be!

However, changing the syllabus to drop the lowest exercise requires going through some administrative hoops, so while that was a good idea we unfortunately can't accommodate that suggestion this time around.

Because of even more fun university rules, I can't assign a due date into the assessment period, but i can grant an extension into the assessment period. As a result, ex9 and ex10 are **due on the last day of class (not the Saturday night)**, but you have an extension for ex10. See MarkUs for details.

Administrivia

Quiz 2 was scheduled for 3 August, but that turns out to be a holiday. Therefore, we'll have it on 10 August instead. (I'll likely adjust what material is covered in the quiz since it's later than I thought originally. Stay tuned.)

The plan for the final assessment: the specifics are still TBD, but I would like you to be able to write it during a three-hour block **of your choice** during the assessment period. Details to come.

Last time

- We introduced the notion of a type as a **collection of axioms and implications**, encoded as **logical propositions**

Today:

- We see how to typecheck a language of arithmetic and booleans
- *a menagerie of different kinds of types!*

Note about our use of theory...

... The goal here is to formalise notions that we already have built up intuitions for in this class, and to guide our thinking in a less ad-hoc way.!

As a result, some of the notation in this and future lectures may be "abbreviated" in order to skim over things that aren't of interest to us in this class, and I may play things fast and loose with notation depending on the given example.

The Pierce type theory textbook, linked to on the course homepage, is an excellent resource if you want a more formal treatment of all this.

Revisiting type of natural numbers

Recall this inductive definition of a number.

$$\text{Zero} : \text{Nat}$$

We can construct a well-typed representation of "the number 3" as:

$$\frac{t : \text{Nat}}{(\text{add1 } t) : \text{Nat}}$$
$$(\text{add1 } (\text{add1 } (\text{add1 } \text{Zero})))$$

But the only "constant valued" number in this type is zero. From the type system's perspective, all other natural numbers need to be "computed" through `add1`.

Inductive types

We will call a type that is defined in terms of:

- A base axiom
- A local implication that relates two terms to their type

an **inductive type**.

(The formal definition is broader than this, but this suits our purposes for now.)

$$\text{Zero} : \text{Nat}$$
$$\frac{t : \text{Nat}}{(\text{add1 } t) : \text{Nat}}$$

Type safety: progress + preservation

Definition: We say that typechecking can make **progress** if every subterm in a term has been fully reduced to a value (an axiom), or it can take a step according to the evaluation rules (an implication)

$$\text{Zero} : \text{Nat}$$
$$\frac{t : \text{Nat}}{(\text{add1 } t) : \text{Nat}}$$

In other words: if t is a well-typed term, either t is a value, or there exists a t' such that t' implies t .

(proof by induction on the typing rules.)

Type safety: progress + preservation

Definition: We say that typing is **preserved** if taking a typechecking step on a well-typed term produces a well-typed term.

(This might sound similar to progress (can we take a valid step or not?) but we'll see when we talk about subtyping and polymorphism that we might be able to use a typing rule that leads to a contradiction / invalid type, so just keep this in mind.)

$$\text{Zero} : \text{Nat}$$
$$\frac{t : \text{Nat}}{(\text{add1 } t) : \text{Nat}}$$

progress + preservation = canonical forms

If typechecking has progress and preservation, after a certain number of typechecking steps, we will end up with an irreducible value that is well-typed.

The canonical form of a type are all the "well-typed" values of that type.

Example: if t is a term of type `Bool`, its possible values are `true` and `false`, so hence so are its canonical forms.

`true : Bool`

`false : Bool`

$t_1 : \text{Bool}$	$t_2 : T$	$t_3 : T$
<hr/>		
<code>if t_1 then t_2 else t_3 : T</code>		

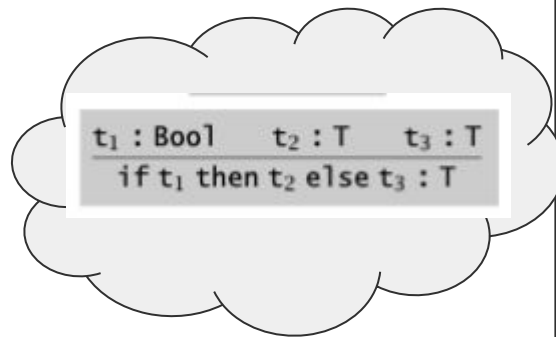
What is a type

Definition: We say that a term is well-typed if there exists some type T such that $t : T$. *Let's **derive the type** of this expression, or, demonstrate that it is not well-typed...*

```
if (zero? 0) then 0 else (add1 0)
```

What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.



```
if (zero? 0) then 0 else (add1 0)
```

What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.

$$\frac{t1: \text{Bool} \quad t2: T \quad t3: T}{\text{if (zero? } 0) \text{ then } 0 \text{ else (add1 } 0)}$$

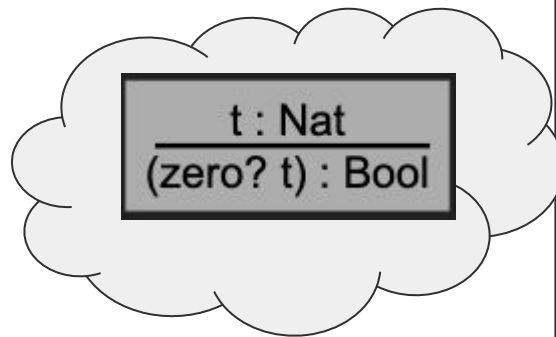
What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.

$$\frac{t1 = (\text{zero? } 0) \quad t2 = 0 \quad t3 = (\text{add1 } 0)}{\text{if } (\text{zero? } 0) \text{ then } 0 \text{ else } (\text{add1 } 0)}$$

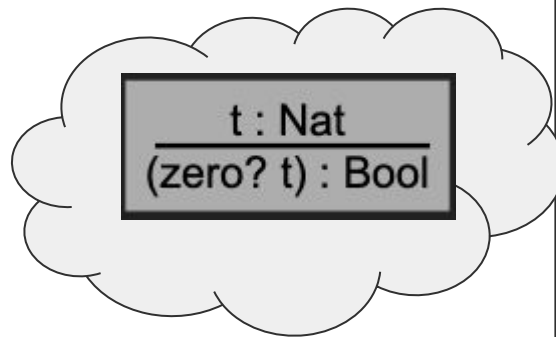
What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.


$$\frac{t1 = (\text{zero? } 0)}{\text{if } (\text{zero? } 0) \text{ then } 0 \text{ else } (\text{add1 } 0)}$$

What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.



$$\frac{(\text{zero? } 0)}{\text{if } (\text{zero? } 0) \text{ then } 0 \text{ else } (\text{add1 } 0)}$$

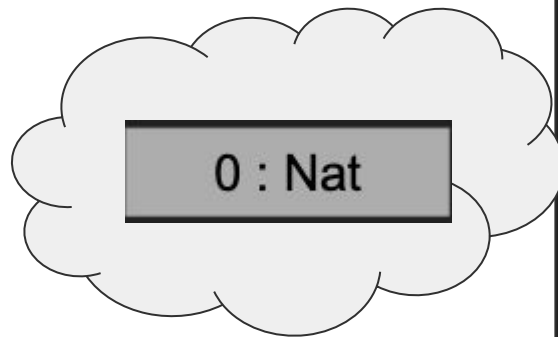
What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.

$$\frac{\frac{0}{\text{(zero? 0)}}}{\text{if (zero? 0) then 0 else (add1 0)}}$$

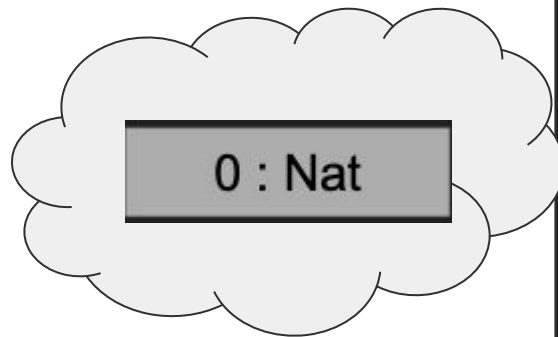
What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.

$$\frac{\frac{\frac{}{0}}{(\text{zero? } 0)}}{\text{if } (\text{zero? } 0) \text{ then } 0 \text{ else } (\text{add1 } 0)}$$


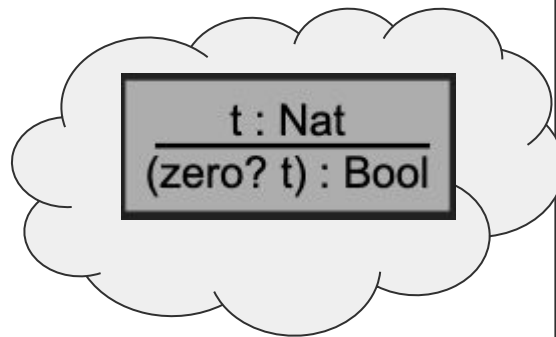
What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.

$$\frac{\frac{\frac{}{0 : \text{Nat}}}{(zero? \ 0)}}{\text{if } (zero? \ 0) \text{ then } 0 \text{ else } (add1 \ 0)}$$


What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.

$$\frac{\frac{\frac{}{0 : \text{Nat}}}{(zero? \ 0) : \text{Bool}}}{\text{if } (zero? \ 0) \text{ then } 0 \text{ else } (add1 \ 0)}}$$


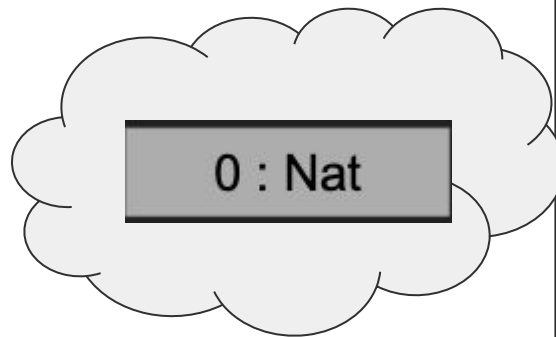
What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.

$$\frac{\frac{\frac{}{0 : \text{Nat}}}{(zero? \ 0) : \text{Bool}} \quad 0}{\text{if } (zero? \ 0) \text{ then } 0 \text{ else } (add1 \ 0)}}$$

What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.

$$\frac{\frac{}{0 : \text{Nat}}}{(\text{zero? } 0) : \text{Bool}} \quad 0 : \text{Nat}}{\text{if } (\text{zero? } 0) \text{ then } 0 \text{ else } (\text{add1 } 0)}$$


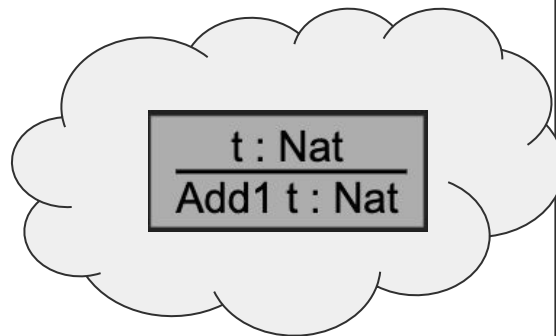
What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.

$$\frac{\frac{\frac{}{0 : \text{Nat}}}{(zero? \ 0) : \text{Bool}} \quad 0 : \text{Nat} \quad (add1 \ 0)}{\text{if } (zero? \ 0) \text{ then } 0 \text{ else } (add1 \ 0)}}$$

What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.

$$\frac{\frac{\frac{}{0 : \text{Nat}}}{(\text{zero? } 0) : \text{Bool}} \quad 0 : \text{Nat} \quad (\text{add1 } 0)}{\text{if } (\text{zero? } 0) \text{ then } 0 \text{ else } (\text{add1 } 0)}$$


What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.

$$\frac{\frac{\frac{}{0 : \text{Nat}}}{(\text{zero? } 0) : \text{Bool}} \quad 0 : \text{Nat} \quad \frac{\frac{}{0}}{(\text{add1 } 0)}}{\text{if } (\text{zero? } 0) \text{ then } 0 \text{ else } (\text{add1 } 0)}$$

What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.

$$\frac{\frac{\frac{}{0 : \text{Nat}}}{(\text{zero? } 0) : \text{Bool}} \quad 0 : \text{Nat} \quad \frac{\frac{}{0 : \text{Nat}}}{(\text{add1 } 0)}}{\text{if } (\text{zero? } 0) \text{ then } 0 \text{ else } (\text{add1 } 0)}$$

What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.

$$\frac{\frac{\frac{}{0 : \text{Nat}}}{(zero? \ 0) : \text{Bool}} \quad 0 : \text{Nat} \quad \frac{\frac{}{0 : \text{Nat}}}{(add1 \ 0) : \text{Nat}}}{\text{if } (zero? \ 0) \text{ then } 0 \text{ else } (add1 \ 0)}$$

What is a type

Definition: We say that a term is well-typed if there exists some type t such that $t : T$.

$$\frac{\frac{\frac{}{0 : \text{Nat}}}{(\text{zero? } 0) : \text{Bool}} \quad 0 : \text{Nat} \quad \frac{\frac{}{0 : \text{Nat}}}{(\text{add1 } 0) : \text{Nat}}}{\text{if } (\text{zero? } 0) \text{ then } 0 \text{ else } (\text{add1 } 0) : \text{Nat}}$$

What is a type

If we reach a stage where there is no implication rule to make further progress, we say that the term is **not well-typed**.

```
if (zero? 0) then 0 else (zero? 0)
```

What is a type

If we reach a stage where there is no implication rule to make further progress, we say that the term is **not well-typed**.

$$\frac{\text{pred: Bool} \quad \text{if-true: T} \quad \text{if-false: T}}{\text{if (zero? 0) then 0 else (zero? 0)}}$$

What is a type

If we reach a stage where there is no implication rule to make further progress, we say that the term is **not well-typed**.

$$\frac{\frac{\frac{}{0 : \text{Nat}}}{(zero? \ 0) : \text{Bool}} \quad 0 : \text{Nat} \quad (zero? \ 0)}{\text{if } (zero? \ 0) \text{ then } 0 \text{ else } (zero? \ 0)}}$$

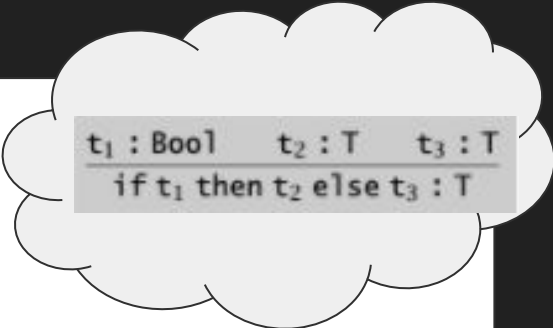
What is a type

If we reach a stage where there is no implication rule to make further progress, we say that the term is **not well-typed**.

$$\frac{\frac{\frac{}{0 : \text{Nat}}}{(zero? \ 0) : \text{Bool}} \quad 0 : \text{Nat} \quad \frac{\frac{}{0 : \text{Nat}}}{(zero? \ 0) : \text{Bool}}}{\text{if } (zero? \ 0) \text{ then } 0 \text{ else } (zero? \ 0)}$$

What is a type

If we reach a stage where there is no implication rule to make further progress, we say that the term is **not well-typed**.

$$\frac{\frac{\frac{}{0 : \text{Nat}}}{(\text{zero? } 0) : \text{Bool}} \quad 0 : \text{Nat} \quad \frac{\frac{}{0 : \text{Nat}}}{(\text{zero? } 0) : \text{Bool}}}{\text{if } (\text{zero? } 0) \text{ then } 0 \text{ else } (\text{zero? } 0)}$$

$$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$

What is a type

Definition: The **bottom type** (\perp) is a special type that signals that a result, should this expression be evaluated cannot be returned / a result is undefined

$$\frac{\frac{\frac{}{\theta : \text{Nat}}}{(\text{zero? } \theta) : \text{Bool}} \quad \theta : \text{Nat} \quad \frac{\frac{}{\theta : \text{Nat}}}{(\text{zero? } \theta) : \text{Bool}}}{\text{if } (\text{zero? } \theta) \text{ then } \theta \text{ else } (\text{zero? } \theta) : \perp}$$

The cardinality of a type:

How many natural numbers are there in the universe?

(Countably) infinitely many, of course!

This is reflected in the typing rule for Add1: no matter how big a natural number is, you can always construct one that's one greater.

$$\text{Zero} : \text{Nat}$$
$$\frac{t : \text{Nat}}{\text{Add1 } t : \text{Nat}}$$

The cardinality of a type:

How many **zeros** are there in the universe?

Really, only one!

Any time I evaluate Zero, it produces a piece of data with exactly the same meaning.

This is in contrast to Add1, which means different things depending on what Nat it takes as an argument.

$$\text{Zero} : \text{Nat}$$
$$\frac{t : \text{Nat}}{\text{Add1 } t : \text{Nat}}$$

The cardinality of a type:

How many **zeros** are there in the universe?

Really, only one!

Two ways to think about Zero:

- It is a value constructor of no arguments
 - aka: a nullary function / thunk
- It is a singleton value

$$\text{Zero} : \text{Nat}$$
$$\frac{t : \text{Nat}}{\text{Add1 } t : \text{Nat}}$$

A type with one possible value

Thought experiment: What's the simplest type you can construct?

What about: what is the type with the smallest nonzero cardinality?

A type with one possible value

Thought experiment: What's the simplest type you can construct?

unit: Unit

What about: what is the type with the smallest nonzero cardinality?

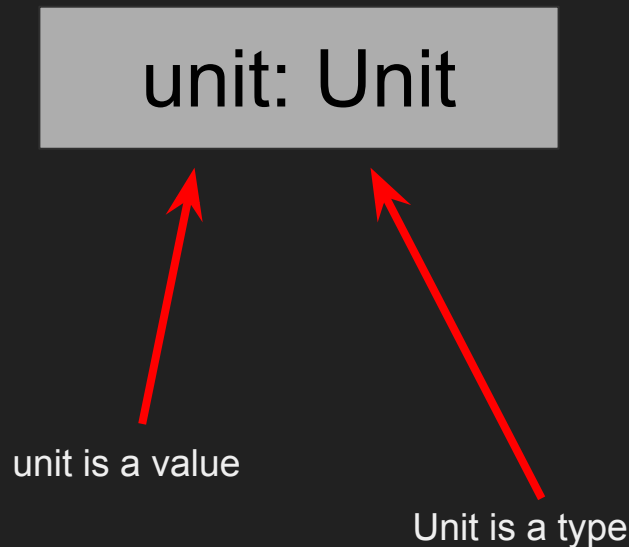
Unit is a type with one constant value: unit
(note the capitalisation)

Unit

Thought experiment: What's the simplest type you can construct?

What about: what is the type with the smallest nonzero cardinality?

`Unit` is a type with one constant value: `unit`
(note the capitalisation)

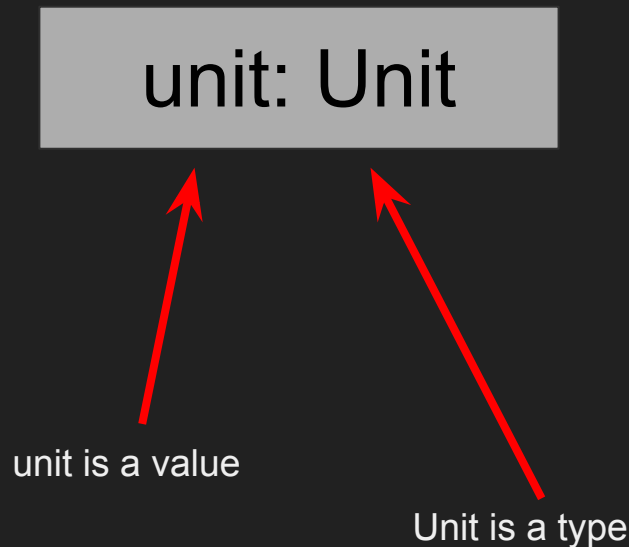


What's true about Unit?

Imagine we **typechecked** some **term** and saw that it would **evaluate** to an **expression** of **type** `Unit`

We know that the only possible value that that expression can evaluate to is `unit` !

`Unit` is an appropriate return type for functions with side effects (like returning `void`)



What about a type with zero possible values...

Such a type **is not constructible**; what would it ever evaluate to?

saying "this term typechecks as a type with zero possible values" means "this expression will not meaningfully evaluate to anything"

This is the **bottom type** (\perp) from last class!

(Bottom becomes interesting with subtyping; stay tuned...)

Enumerated types

Here's a type with seven data constructors, all of which take no arguments. Obviously, a Monday can't also be a Friday, so these are **disjoint** as well.

An **enumerated type** contains one or more disjoint value constructors of zero arguments.

```
Day.hs
1 data Day = Monday
2           | Tuesday
3           | Wednesday
4           | Thursday
5           | Friday
6           | Saturday
7           | Sunday deriving (Show)
```

The cardinality of an enumerated type

Within a given week, how many **Mondays** are there?

Just one! So, it makes sense that there is only one way to construct a Monday.

Same thing with Tuesday, Wednesday...

```
Day.hs
1 data Day = Monday
2           | Tuesday
3           | Wednesday
4           | Thursday
5           | Friday
6           | Saturday
7           | Sunday deriving (Show)
```

The cardinality of an enumerated type

Within a given week, how **many days of the week** are there in total? Seven, of course!

Therefore, the cardinality of an enumerated type is however many (nullary) value constructors it has.

```
Day.hs
1 data Day = Monday
2           | Tuesday
3           | Wednesday
4           | Thursday
5           | Friday
6           | Saturday
7           | Sunday deriving (Show)
```

The cardinality of an enumerated type

Within a given week, how **many days of the week** are there in total? Seven, of course!

Alternatively,

(the 1 way to construct a Monday)
+ (the 1 way to construct a Tuesday)
+ ...
+ (the 1 way to construct a Sunday)

```
Day.hs
1 data Day = Monday
2           | Tuesday
3           | Wednesday
4           | Thursday
5           | Friday
6           | Saturday
7           | Sunday deriving (Show)
```

The cardinality of an enumerated type

Within a given week, how **many days of the week** are there in total? Seven, of course!

Alternatively, the cardinality of an enumerated type is **the sum of the cardinalities of all its (nullary) data constructors**.

```
Day.hs
1 data Day = Monday
2           | Tuesday
3           | Wednesday
4           | Thursday
5           | Friday
6           | Saturday
7           | Sunday deriving (Show)
```

Enumerated types are sum types

This is why we call types that we describe informally as "either this kind of data OR this kind of data" as **sum types**.

```
Day.hs
1 data Day = Monday
2           | Tuesday
3           | Wednesday
4           | Thursday
5           | Friday
6           | Saturday
7           | Sunday deriving (Show)
```


Solving for the definition of product types

We have seen how a characteristic of a sum type of T_1 or T_2 has cardinality which is the sum of T_1 's and T_2 's.

Given this, what can we say about a type whose cardinality is the product of T_1 and T_2 's?

Our only choice, really, is for this type to "contain" every possible T_1 and also "contain" every possible T_2 !

Typing rules for Pair

A pair holds a value of T1 and another value of T2.

This rule says: we can construct a product type of T1 and T2 if we have a value of both types.

$$\frac{t1 : T1 \quad t2 : T2}{\text{Pair } t1 \ t2 : T1 \times T2}$$

Typing rules for Pair

A pair holds a value of T1 and another value of T2.

These typing rules are **projections**: they indicate how to extract a T1 and T2 out of a T1xT2.

$$\frac{t1 : T1 \quad t2 : T2}{\text{Pair } t1 \ t2 : T1 \times T2}$$

$$\frac{p : T1 \times T2}{\text{fst } p : T1}$$

$$\frac{p : T1 \times T2}{\text{snd } p : T2}$$

Triple, and beyond...

This extends naturally to tuples of any length.

What needs to change for this to work for structure (record types)?

Only the names of the projections!

$$\frac{t1:T1 \quad t2:T2 \quad t3:T3}{\text{Triple } t1 \ t2 \ t3 : T1 \times T2 \times T3}$$

$$\frac{p : T1 \times T2 \times T3}{fst \ p : T1}$$

$$\frac{p : T1 \times T2 \times T3}{snd \ p : T2}$$

$$\frac{p : T1 \times T2 \times T3}{trd \ p : T3}$$

Triple, and beyond...

Note: a nice consequence of pattern matching is that often we don't need to think about the projection operators on a product type...

How can we access the `Float` attributes of the point values? Just as we define functions by pattern matching on primitive values and lists, we can also pattern match on *any value constructor*.⁹

```
1 distance :: Point -> Point -> Float
2 distance (Point x1 y1) (Point x2 y2) =
3     let dx = abs (x1 - x2)
4         dy = abs (y1 - y2)
5     in
6         sqrt (dx*dx + dy*dy)
```

A deep connection brewing...

We have seen how to determine that a conditional is well typed:

$$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad (\text{T-IF})$$

Determine the types of the three terms, and see if the logical implication holds.

A deep connection brewing...

We have seen how to evaluate a conditional:

```
Returns the value of the Rake expression under the given environment.  
|#  
(define (interpret env expr)  
  (match expr  
    ...  
    |((list 'when pred on-true on-false) (if (interpret env pred)  
                                              (interpret env on-true)  
                                              (interpret env on-false)))  
    ...))
```

The structure of **typechecking these terms** and **evaluating this expression** follow the same structural recursion!

A deep connection brewing...

Typechecking terms by recursively applying implication rules

feels structurally equivalent to

evaluating the expression by recursively evaluating the AST...

$$\frac{\frac{}{0: \text{Nat}} \quad \frac{}{0: \text{Nat}}}{\frac{(\text{zero? } 0): \text{Bool} \quad 0: \text{Nat} \quad (\text{add1 } 0): \text{Nat}}{\text{if } (\text{zero? } 0) \text{ then } 0 \text{ else } (\text{add1 } 0): \text{Nat}}}$$

```
(if (interpret env pred)
    (interpret env on-true)
    (interpret env on-false)))
```


A deep connection brewing...

...so a type system is a sort of programming language...

...**types** are constants in the "type program"...

...**typing rules** are expressions in the "type program"...

...**typechecking** is like evaluating the "type program"...

...evaluation of the expression produces a value that is a canonical form of the typechecked type

$\frac{}{0: \text{Nat}}$		$\frac{}{0: \text{Nat}}$
$\frac{}{(zero? \ 0): \text{Bool}}$	$0: \text{Nat}$	$\frac{}{(add1 \ 0): \text{Nat}}$
$\frac{}{\text{if } (zero? \ 0) \text{ then } 0 \text{ else } (add1 \ 0): \text{Nat}}$		

```
(if (interpret env pred)
    (interpret env on-true)
    (interpret env on-false)))
```

