We have the following domains

- `'const`: Any constant value. Can be put anywhere
- `'async`: A value with no associated clock, cannot be stored in registers
- `T`: a named domain. Two named domains cannot be mixed unless there is an explicit `T1 <: T2` constraint
- (t_1, t_2): a tuple. Things are a lot easier to reason about if we only have two-tuples so we'll have to do some conversion of `n`-tuples in the domain inferer
- $t_1 \to t_2$: Function from one domain to another

This is something we should account for when we do more complex pipelines

- `enabled(`t_1`)`: A domain with the same clock as t_1 but with a more restrictive enable.

In addition, we need to have constraints

- `!SyncReset`
- `!AsyncReset`
- `!Initial`
- `!Enable`

The default domain unless anything else is specified is {}. A unit or language construct can refine this with constraints

Subtyping rules

$$\frac{}{\texttt{'const} <: t_1}\ \text{const}$$

$$\frac{}{t_1 <: \texttt{'async}}\ \text{async}$$

$$\frac{\Gamma \vdash t_3 : \min(t_1, t_2)}{\Gamma \vdash (t_1, t_2) <: t_3}\ \text{tuple\_sub}$$

$$\frac{\Gamma \vdash t_1\{c_1 ... c_n\} \qquad \Gamma \vdash t_2\{c_1, ... c_n, d_1 ...\}}{\Gamma \vdash t_1 <: t_2}\ \text{constraints}$$

Normal rules

$$\frac{}{0 : \texttt{'const}}\ \text{literal}$$

$$\frac{}{0 \Rightarrow \texttt{'const}}\ \text{literal}$$

$$\frac{t_1 = \texttt{lookup}\ (\Gamma, x)}{\Gamma \vdash x : t_1}\ \text{name}$$

$$\frac{t_1 = \text{lookup}(\Gamma, x)}{\Gamma \vdash x \Rightarrow t}\ \text{name}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : (t_1, t_2)}\ \text{tuple\_union}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow t_1 \qquad \Gamma \vdash e_2 \Rightarrow t_2}{\Gamma \vdash (e_1, e_2) \Rightarrow (t_1, t_2)}\ \text{tuple\_union}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash \texttt{has\_clock(}\ t_1\texttt{)}}{\text{reg}\ x = e_2, \texttt{extend}\ (\Gamma, x \in t_1)}\ \text{clock\_requirement}$$

$$\frac{\Gamma e_1 \Rightarrow t_1 \qquad \Gamma \vdash x \Leftarrow \texttt{has\_clock}\ (t_1)}{\text{reg}\ x = e_1, \texttt{extend}\ (\Gamma, x \in t_1)}\ \text{clock\_requirement}$$

$$\dfrac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 \oplus e_2 : t_1}\ \text{binop}$$

$$\dfrac{\Gamma \vdash e_1 : t_2 \qquad \Gamma \vdash e_2 : t_u}{\Gamma \vdash \text{set } e_1 = e_2, t_2 <: t_1}\ \text{set}$$

The tuple issue

This should domaincheck

```
entity test<'a, 'b>(a: 'a T1, b: 'a T2) {
  reg x = (a, b);
}
```

and this

```
entity test<'a, 'b>(a: 'a T1, b: 'a T2) {
  reg x = (a, 0);
}
```

This should not

```
entity test<'a, 'b>(a: 'a T1, b: 'b T2) {
  reg x = (a, b);
}
```

Case one should be fairly simple, we want to show `x: 'a`

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{lookup}\ (\Gamma, a)}\ \text{name}}{a : {'a}} \qquad \dfrac{\dfrac{}{\texttt{lookup}\ (\Gamma, a)}\ \text{name}}{b : {'a}}}{\Gamma \vdash (e_1, e_2) : ({'a}, {'a})}\ \text{tuple\_union}}{(a, b) : {'a}}\ \text{tuple\_sub}$$

Case 2 `(a, 0)`

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{lookup}\ (\Gamma, a)}{\Gamma \vdash a : {'a}}\ \text{name} \qquad \dfrac{}{\Gamma \vdash 0 : \texttt{'const}}\ \text{const}}{\Gamma \vdash (a, b) : ({'a}, \texttt{'const})}\ \text{tuple\_union}}{\Gamma \vdash (a, 0) : {'a}}\ \text{tuple\_sub}$$

The final path fails for the `(a, b)` case, because `tuple_sub` requires the subtyping judgement

Set

`a: T, b: 'async; set a = b` is not allowed

$$\dfrac{\dfrac{}{a : T} \qquad \dfrac{}{b : \texttt{'async}}}{\Gamma \nvdash \text{set } a = b, \texttt{'async} \not<: T}\ \text{set}$$

$\square$

Issues

`has_clock` feels sketchy, essentially i'm imagining this:

```
match t {
  'async => false,
  // We could go with false here too, but we can fall back on the annonymous
  // later domain to allow `reg x = x + 1;` without specifying the domain
  'const => true,
  // Named types must have a HasClock constraint which is implicit unless otherwise
  // stated
  T => T.constraints.contains(HasClock),
  // Tuples have a clock if t1 and t2 are in compatible domains. For example
  // (a, 0) can be registered but its domain is (t_1, 'const). Therefore we need
  // a subtyping thing here.
  (t1, t2) => {
    let domain = min(t1, t2); // The smallest subdomain that contains both t1 and t2
    has_clock(domain)
  }
}
```