Essential types

Towards the end of Layer 4 are definitions for the types used within the compiler. Here are the most important ones:

hoon. The Hoon ast. A Hoon program is thus parsed into a hoon value. (Notice the difference in capitalization.) Each rune in Hoon turns into a case of this large $%. For example ?& (“wutpad” in the deprecated pronunciation) parses into a %wtpd. The type of the + of each case can be a helpful reminder as to the abstract syntax. For example, noticing that %cnts takes a wing and a list of [wing hoon] may give us a sense of what kind of operation %= could be. Some of the non-runic cases correspond to the “primary expressions” of Hoon: for example, wing is for wing expressions, like foo.bar (or just foo), rock and sand are for atomic literals, knit is for tape literals, and tell and yell are for <foo> and >foo< respectively, whatever those are. Others do not arise from parsing, but rather are internal shorthands generated and consumed during the course of compilation: for example [%$ 6] is “get axis 6”, which could be expressed more cumbersomely as [%wing ~[%& 6]]. Finally, the autocons case, as in nock, means making a cell out of the results of the two expressions; i.e., it has the same semantics as %clhp.

wing. A wing is a Hoon expression like “a.+6.c”. Abstractly, a wing is a list of **limb**s. A limb is either a term (i.e. name, like “a”) or an axis (like “+6” in the concrete syntax). Axes are represented by the %& case. Names are represented by the %| case, with the p/@ud counting the number of ^s and the q/(unit term) containing the name, or ~ for “.” (i.e “current subject”). The type also provides an extra case, a raw term, which is handled equivalently to %| with no ^s and `that term. This is part of a recurring (mis)pattern of providing shorthands as duplicate possibilities in the data types, rather than as helper functions.

type. The object of type inference and type checking in Hoon. Learn more at https://urbit.org/docs/reference/hoon-expressions/advanced/, which you should probably read right now.

spec. The user-facing “type” language’s ast. When you make a tree of $ runes, or parse something in “type mode” the result is a spec. It is from specs, rather than types, that molds and bunts are generated. Specs contain more information and draw finer distinctions than types; the downconversion is more lossy than mere desugaring. (For example, if you manually specify a bunt value for your spec, this information will not be present in the corresponding type.) Incidentally, this conversion is never done directly; rather, one generates the code for the bunt from the spec, then type infers across that. Both steps of this process are cached: the first in the ~+ of example:ax, the second in the compiler jet for play:ut.

nock. Nock code.

skin. An attempt to introduce something like other languages’ notion of pattern into hoon, rather than reusing type for this.

port, palo, vein, opal. These types are used to report the result of name resolution. The top level result of name resolution is a port, which is either a palo (face (“leg”) or arm match) or a pair of type and nock, corresponding to a =*. A palo is a pair of vein and opal. The vein is a list of optional axes, reporting the path to get to the found name. The opal says what kind of match it is. These types are very hard to track and poorly laid out, and will be discussed again in the name resolution section below.

foot. A pair of ?(%wet %dry) and a hoon. Values of this unfortunately named type are used to talk about arm bodies in cores as entities in their own right.

tiki. An intermediate data structure used in parsing ? (wut) runes, which indicates whether the scrutinee (thing being matched) is direct (a wing) or indirect (a hoon). If indirect, it must be tislused before being tested, and in fact this is enforced by the types in $hoon, where e.g. the head of a ?- must be a wing. Additionally, if the user has applied a face at the top level of the scrutinee expression, this is factored out into the (unit term) of the tiki, and gives rise to a =* so the value is available under that name in the branches. The logic to generate the hoon, including with an extra =+ and/or =* if needed, is found in the ah core of Layer 5.

vase. This is not an internal compiler data structure but rather an end-user-facing abstraction that amounts to a very important entrypoint to the compiler. A vase is a pair of a type and a completely untyped noun. The expectation is that the noun fit the type; if not, the vase is said to be “evil.” A bunch of operations whose names begin with “sl” are provided in Layer 5 for manipulating these values; these operations amount to invocations of the compiler.

Synopsis of the compiler

We now provide a brief outline of how the compiler is structured into cores and arms in hoon.hoon. The goal here is not to understand what all of these parts are fully, but rather *where* they are. Some parts will be explained in more detail in a more logical order immediately following; others are summarized inline.

- Layer 4:
    - Types
- Layer 5:
    - Partial noun interpreter, operations on seminouns
    - “Smart constructors” for +$type, which lift voids (bool, cell, core, hint, face, fork)
    - Little code generation utilities and other such trifles
    - **ah**. Tiki engine: producing %wt hoons from the intermediate data structures emitted from the parser. Note that although you can say ?- (foo 2) %a !! %b !! == in the concrete syntax, in the abstract syntax of $hoon, the scrutinee (thing being tested) of ?- must be a wing. The foregoing expression must therefore be

converted into a =+ of (foo 2) followed by a ?- scrutinizing +2, and ah is the thing that does this transformation.
Sample: a tiki value.

- **ax.** Operations on specs
  Sample: a flag value ("fab," unused) and the spec being operated on.
  - **autoname.** The autonamer, which generates a face name which seems suitable for a spec. This is the thing which turns |= =term … into |= term=term …, effectively.
  - **example.** Generates the code for producing the bunt (default) value associated with the given spec.
  - **factory.** Generates the code for producing the mold (normalization) gate associated with the given spec.
- **ap.** Home of the desugarer.
  Sample: a flag value ("fab," unused) and a hoon.
  - Conversions between hoons and skins
  - **open.** The desugarer, which rewrites runes to more simple runes.
  - **reek.** Extract a wing from a hoon which is primarily about that wing (e.g. %wing, %cnts)
- **ut.** The main compiler core. Roughly, any operation that takes a type goes here. That type is factored (curried) into the sample of ut, where it is shared among all the arms.
  State pinned in context:
  - fan, rib. Used to track holds (%hold in type) as we expand them to make sure we don't get stuck in a loop
  - vet. A flag that, if yes, means we are in "vetting mode" and should error out in more circumstances
  - fab. Comments refer to this flag as demarcating a "fabrication mode" or "test mode," which can be turned off (or on?) by ^%. Although it is duly passed into ap and ax, the value is never read in live code, and has no effect on the operation of the compiler.
  Sample: a type ("sut"). For the Three Operations, this is the subject type we are operating with respect to.
  - **ar.** The "texture" (skin) engine. This core implements type checking and code generation for the new pattern matching subsystem.
  - The Three Operations: mint, play, mull, and their subroutines.
  - Various operations on types.
  - Name resolution.

The Three Operations — {mint, play, mull}:ut

Mint, play, and mull are the central operations of the compiler. They are central in the sense that they call pretty much everything else, and also in that they are at the core of what the compiler

“does.” Each of these operations is structured around a ?- scrutinizing the rune at the root of whatever hoon is being processed. Let’s understand them one-by-one.

Mint

You know how you were taught that the compiler is a function [type hoon] -> [type nock]? Well this is the thing that does that. It is therefore the principal entrypoint into the compiler. Let’s look at its signature:

```
++  ut
|_  sut/type
  ::  ..
  ++  mint
  |=  [gol/type gen/hoon]
  ^-  [p/type q/nock]
```

Sut and gen are the pair of subject type and program to compile. (Note how sut has been “factored out” into the sample of ut.) The result p is the inferred type of the product of running this code against the given subject type, and q is the generated code. So that’s how mint embodies [type hoon] -> [type nock].

But what’s with the extra gol/type? This “goal type” is a mandatory convenience which allows you to specify a type that q (the product type) must nest under when it is computed. Most external users specify %noun for this. For that matter, most internal recursions reset it to %noun as well. Honestly, I’m not sure why performing this check, if desired, isn’t the responsibility of the caller. Anyway, the “internal subroutine” +nice inside of mint is what’s responsible for asserting that the product nests under the goal.

Play

If you want to do type inference but not code generation, you call play. We certainly hope that p:(~(mint ut s) %noun h) is equal to (~(play ut s) h). Of course, because Hoon is a strict language, the former does much more work than the latter. In cases where mint doesn’t need the nock for a subexpression (e.g. the first clause of a ^+), it calls play rather than recursing.

Mull

Mull is used in typechecking wetness. (Wetness is Hoon’s quixotic attempt to provide something like parametric polymorphism.) Recall that a core type contains a formal payload type (q.q) and an actual payload type (p). The formal is the type the core was compiled against, and the actual is the type of the payload that we have in front of us, which may be different because we have edited parts. In any core, these edits must conform to the variance model. Additionally, for dry cores, the actual type must nest under the formal type; for wet cores, it must instead be the

case that the nock you'd get for the arm you're pulling (FIXME or all the arms? Yes. (but preserving this parenthetical so we can see the discussion)) would be the same whether compiled against the formal or actual type. But actually generating these nocks would be inefficient, so we instead "mull" the arm's body against the two types. To mull a hoon against two types is thus to confirm that compiling it against either type as subject would result in the same code, while also returning the two (possibly different) product types.

Now let's look at the signature of mull:

```
++  ut
|_  sut/type
  ::  ..
  ++  mull
  |=  [gol/type dox/type gen/hoon]
  ^-  [p/type q/type]
```

Sut is the actual type. Dox is the formal type. Gen is the hoon we are contemplating. P is the product type of the hoon when sut is taken to be the subject type. Likewise q is the product type for subject type dox. Finally, gol is a type that p must nest under (like in mint).

Note that whether mulling succeeds (i.e. comes to the conclusion that the generated nocks would be the same) is not captured in the return values (which are used for propagating information to recursive callers), but rather in whether the operation errors out. If it does, it pushes one of the infamous "mull-bonk" messages.

Mull has a single entrypoint, which is found towards the end of +fire. This use discards the returned pair of types and is made only for the effect of any mull-bonk errors that may occur. All other calls to mull are recursive calls (including from within mull's subroutines).

Mine and mile: mint and mull for |% *and* |@

Minting and mulling the introduction forms of cores (i.e. the |% and |@ runes) are sufficiently complex operations that they are factored out into two subroutines, mine:ut and mile:ut, respectively. These are called via the "grow" internal subroutines of mint and mull.

(Playing a core is not similarly complex; one need merely assemble core type from the immediately available data. In particular, remember that core types contain the parsed code of their arms (sort of like an incorporated hold type for each arm), both to make typechecking the mutual recursions easier, and to provide for wetness.)

Mile calles balk for the core's chapters, which calles bake for their feet (arms). Meanwhile, mine does all of its processing in-house.

There are two things to understand about this code. First, the ?-s with patterns like {~ * *} are constructing the battery shape based on the shape of the map-tree. Second, note that bake (again, ultimately a subroutine of mull) calls mull for dry cores but does nothing for wet cores. This is because if we are producing a dry core deep inside a wet arm, we must make sure that the core's contents agree on how they'd support the formal and actual types of the wetness. Meanwhile, if the produced core is, itself, wet this check should only occur at each consumer of that core, so it doesn't make sense to perform the check here. (In reality, should a three- or four-way check be performed instead for these?) This apparent reversal is lampshaded, but not explained, in a cheeky comment.

The et core: the Three Operations for %= (TODO: wordsmith this relationship better)

To handle the %cnts case, each of the Three immediately calls the corresponding arm of et:ut. For example, mull calls (~(mull et p.gen q.gen) gol dox). Let's look at the signature of the et core:

```
++  ut
|_  sut/type
  ::  ..
  ++  et
  |_  [hyp/wing rig/(list [wing hoon])]
    ++  play
      ^-  type
      ::  ..
    ++  mint
      |=  gol/type
      ^-  [type nock]
      ::  ..
    ++  mull
      |=  [gol/type dox/type]
      ^-  [type type]
      ::  ..
    --
```

(Pro tip: Any %=-related operation can be immediately sussed out by the fact that it takes a wing and a (list [wing hoon]).)

This should be pretty self-explanatory at this point. Each arm of et does name resolution for hyp in read mode (by calling find, discussed later), then assertion-wraps a call to a subroutine which contains the main part of the implementation. For {mint, play, mull}:et:ut, these are {elbo, etco, endo}:ut respectively. Note that among these subroutines' definitions is a listing for "ergo" which is an unused apparent alternative implementation of mint. The implementation actually used, etco, instead forwards into the complex oc and ad cores.

Finally, note that beneath et are defined epla, emin, emul. These are presently unused. Historically, they made it easier for the Three Operations' jets to call into the et core.

Desugaring — open:ap

You may have noticed that the central ?-s of the Three Operations list only a small subset of the runes defined in $hoon. Instead of being handled directly, all of the other runes are ultimately translated (or *desugared*) into *basic runes* which are handled directly. This translation is done in open:ap.

At the end of each Operation's ?- is a * case. The corresponding code

- Passes the hoon being operated on into open:ap
- Checks to make sure the result is a different hoon. (It is an implementation error in the compiler if it is not, which we detect and zap out on.)
- Recurs with that new hoon into whichever Operation we were doing

The implementation inside open:ap is in some sense the least effortful one that, given the above dynamic, eventually results in a non-basic rune being translated properly. Instead of translating a whole tree of runes into a corresponding tree containing only basic runes, open translates just the root, leaving its subtrees unchanged to be subtrees of whatever combination of runes the root becomes. Moreover, a rune need not open directly into a basic rune. Many runes desugar into other non-basic runes. This is ok as long as a cycle is never formed, and it works because one of the things that can happen when you recur back into, say, mint, with your single-step-desugared hoon is that mint still doesn't recognize the root rune and has to open it again. For this reason, open is sometimes analogized to the macroexpand-1 operation of Lisps, and the non-basic runes to Lisp macros.

Tl;dr: If you want to understand what a rune is/does, look for it in mint/play. If you don't find it there, it's "syntactic sugar" and you should look for how it's translated in open:ap.

Typing ?: — {gain, lose, chip}:ut; ax:ut

Recall that in Hoon, ?: has the curious property that, for certain test conditions, the subject hypothesized for each branch is narrowed, or *refined*. For example, assuming a subject of type *, in

```
?:  ?=(@ .)
  <a...>
<b...>
```

<a…> will be compiled against subject type @, while <b…> will be compiled against type ^. (The product types will then be combined in a %fork, i.e. unioned together; see below.) Because

all of the conditionally branching ? runes desugar to ?:, this discipline affects them as well, and is a fundamental feature of the Hoon type system.

Let *t* be the first subhoon of a ?: — i.e. the subexpression that denotes the test to be performed. Let *S* be the subject type. Then the subject type for the then clause will be (gain:ut *t S*), while the subject type for the else clause will be (lose:ut *t S*). (Are you following along in play:ut?)

Looking at gain:ut and lose:ut, we see that they merely call chip:ut with & or | respectively. In chip, we see that the test conditions which give rise to refinement are few: ?= and ?# give rise to refinements on both branches, ?& gives rise to refinement only on the then branch (combining the refinements that would have occurred from each of the conjuncts), and ?| gives rise to refinement only on the else branch (likewise combining from the disjuncts). Finally, any unrecognized rune is desugared to see if that helps.

(The partial refinement on conjunction and disjunction is presumably to avoid some kind of combinatorial explosion, but is also rather unconvincing. This points at one of the several reasons why traditional PL considers the if-expression-centric aspect of this scheme dubious.)

?= and ?# correspond to two styles of “pattern matching” available in Hoon. (Note that the thing being tested in such a match must be a wing. This is so that we have a place in the subject to refine in the first place.) The first uses a type as the pattern, and the second uses a skin. Also, the second doesn’t work.

(Note that I said type, not spec. The concrete syntax will of course use a spec, but we don’t make any use of this extra data, instead immediately translating the spec to a type by playing across the code for generating the bunt (example:ax).)

Refining on a type match amounts to calling cool:ut, which does name resolution (find:ut) to get the axis path for the wing (if it is instead a =*, we do no refinement), then edits (take:ut) that part of the subject type by intersecting it with (fuse:ut) or subtracting from it (crop:ut) the type we’re using as our pattern. (The gates named in parentheses in the foregoing are explained in the later sections “The Algebra of Types” and “Axial Operations on Types.”)

Refining on a skin amounts to calling gain:ar or lose:ar for the then or else branches respectively.

Fishing: Code Generation for ?= and ?#

Testing whether a value matches a given type or skin is called *fishing for* that type or skin. The nock that actually executes these tests is generated by fish:ut and fish:ar:ut, respectively. If you understand the types of type and skin and are fluent in nock, you should be able to read and make sense of these functions directly, and I encourage you to. It will help to know that flan, flor, and flip are small code generation utilities for doing boolean conjunction, disjunction, and

negation respectively, all of which amount to nock 6s. Also note how we track holds to make sure we don't get stuck in an infinite loop.

The Algebra of Types — {nest, fuse, crop}:ut

A set of operations which manipulate types in and of themselves, without relation to other notions, forms a logical core of the compiler. The first of these are: the subtyping predicate (nest), and the intersection (fuse) and subtraction (crop) of types. I like to call these an algebra because, together with the %fork constructor of $type, they look like the operations of a Boolean algebra. Unfortunately, however, the Boolean algebra laws are not followed, most notably in the case of crop.

Nest

The Hoon type system is based on subtyping, and the nest:ut gate implements the subtyping check. A type *a* is a subtype of *b*, morally (i.e., this is how it would operate in an ideal world), if and only if every value of *a* is also a value of *b*. For example, 1 is an atom, but it is also a noun; in fact every atom is a noun, so atom is subtype of noun. We sometimes write $a \leq b$ for this relation. If we think in terms of sets (which is not technically correct, but is a useful intuition pump), subtyping is like the subset relation. In a language with functions, if a function takes a *b*, then you can pass it an *a*; indeed the actual argument types that are permitted are precisely those that nest in the formal argument type. Of course, Hoon doesn't have functions, so this statement has to be modified into one about the formal (q.q) and actual (p) payload types found in the core type at the time one of the arms is pulled.

Now let's look at the signature of nest:

```
++  ut
|_  sut/type
  ::  ..
  ++  nest
  |=  [tel/? ref/type]
  ^-  ?
```

The return value is true if a subtyping relationship is found to exist. The extra argument tel, if set to &, means that instead of returning false to indicate nest failure, we error out with the familiar "nest-fail" message. Finally, the order of arguments is such that testing $a \leq b$ means passing *b* for sut and *a* for ref.

There are two things to note about the high-level structure of nest. First, like many other operations that act on types, nest needs to make sure it doesn't go into an infinite loop. Infinite looping can happen (or rather could, if not for the intervention about to be described), for example, if a hold (a pair of type and hoon), expands (by playing the hoon against the type) into

the exact same hold type. It can also happen for carefully constructed core types, which, instead of having the return types of the arms embedded within, have the arms' hoons, and likewise require playing. To thwart this threat, we maintain state tracking the types we've seen at these problem points. This state is pinned at the top of nest and is therefore implicitly passed to all of the subroutines. The seg set tracks holds in sut that we're in the process of expanding. Whenever sut is a hold, we first check if it's already in seg; otherwise we add it, play the hold (repo:ut does this for us) and recurse. Likewise reg tracks holds in ref. Finally, gil tracks pairs of [sut ref] that have already been considered, such that if they're considered again we have a problem. This is checked and added to whenever sut or ref is a hold, or both are cores.

Second, nest is organized into a number of "internal subroutines" packaged together into a core. Two of these, dext and sint, are an example of a common naming pattern: in operations defined in terms of a dext and sint, dext (after *dexter*, the Latin word for "right") involves a ?- on one argument, while sint (after *sinister*, for "left") involves a ?- on the other. (Unfortunately, in (~(nest ut sut) | ref), dext scrutinizes sut and sint scrutinizes ref. Oh well.) Additionally, dext is the main entrypoint and typically handles the "meaty" or "complex" cases while sint handles simple cases or cases that need a little bit of preprocessing before they can be shunted back to dext.

If you look intently at dext and sint, hopefully you can see that pretty much everything is simple and corresponds to the behavior you'd expect: void and noun are bottom and top, atoms nest according to the aura matching rules (as implemented by fitz of Layer 4; but note that the aura rules you're familiar with mean, absurdly, that nest is not transitive), cells nest if their heads and tails nest, faces get stripped, holds get expanded, and forks mean that any or all of the disjuncts must nest (depending on which side of the test they're on). But cores are not simple at all; nest-checking these involves a cumbersome and rather suspicious amount of checks: first we check moisture, then we wrangle the formal and actual argument types (which are handled differently in sut and ref, for no reason I can tell), then we verify variance compatibility (in deem), and finally we confirm that the batteries have exactly the same names and shape (so unfortunately no structural polymorphism / "duck typing"), and the arms, when played against the formal argument types, nest pairwise.

Fuse

The fuse:ut gate implements "type intersection." Morally, a value is in (~(fuse ut *a*) *b*) iff it is in *a* and in *b*. I use the term "intersection" because this operation is analogous to the one by that name in set theory.

The code here does more or less what you'd expect, except when it comes to cores, where it shrugs and, rather unconvincingly, reinterprets them as cells with noun head (via repo:ut). Note that because intersection is a commutative operation, we don't need to separate the code into dext and sint, and can instead recur with arguments swapped.

Crop

The crop:ut gate implements “type subtraction.” Morally, a value is in (~(crop ut *a*) *b*) iff it is in *a* and not in *b*. Accordingly, the union of (~(crop ut *a*) *b*) and *b* ought to be *a*, and the intersection of *a* and (~(crop ut %noun) *b*) ought to be (~(crop ut *a*) *b*). (Unfortunately, in Hoon, as will be seen, many things that should be are not.) Thus crop is like the “set difference” operator.

We see that like nest, crop maintains a set, bix, of observed holds, and follows a dext-sint structure. As in fuse, the rules for cores are particularly unpersuasive. Subtracting an atom or cell type from a core type yields that same core type, even if the subtrahend is [%cell %noun %noun]. Meanwhile, subtracting a core from a core yields the first core, even if the first is a proper subtype of the second. It is with cells, however, that the most notable problem occurs. If you think about it algebraically, (~(crop ut %noun) *a*) is like the “set complement” of *a*, !*a*, and we’d expect ![*a b*] = union([!*a* !*b*], [*a* !*b*], [!*a b*]). Unfortunately, this behavior would result in a combinatorial explosion as the length of the tuple increases. So crop instead answers [%noun !*b*] for this case. This behavior means that iterated pattern matching works for patterns like [%foo *], but not for patterns like [[%foo *] [%bar *]], analogous to the oft used (Foo _, Bar _) in Haskell. Ironically, we see the cost of this immediately in the code for crop: with tuple-like pattern matching, we wouldn’t need the cumbersome dext-sint architecture!

%fork

When we take the intersection or subtraction of two types, we have to run a computation that traverses these types and builds up a new type. The union of types, on the other hand, is accomplished rather differently in this system. There is an actual case of $type, namely %fork, to represent the union of types. If *a* and *b* are types, then their union is the “bigger” type [%fork (sy *a b* ~)]. (Recall that sy makes a set containing its arguments. A convenience gate, or “smart constructor,” fork (of the start of Layer 5) is provided to streamline the process of constructing these %forks, so that the foregoing can be obtained by saying (fork *a b* ~).) The cost of handling unions is thus moved from the places where they are made to the places where they are consumed, which must iterate over these forks and do something sensible with them.

Note that this %fork construction is rife with representation problems. There are many ways to represent what are conceptually the same type in different ways with nestings of %fork or by combining forks with other constructions. For example [?(%a %b) ?(%c %d)] is the same as ?([%a %c] [%a %d] [%b %c] [%b %d]). In addition, an empty fork is the same as void. The fork smart constructor detects some of these and replaces them with a canonical form, but cannot catch them all. The %fork type also lets you construct “indiscriminable unions,” types which, in the worst cases, essentially have no way for their values to be used. A more sensible, but more difficult, thing to have done would have been to represent only discriminable unions in $type, and have a +fork:ut to combine types into these. But it’s unclear if this can be made workable given e.g. $^ or the encoding of both recursion and wetness as %hold.

Historical note: Apparently, crop and fuse were originally free constructions %crop and %fuse in $type as well. These had to be removed because they easily lead to combinatorial explosions. So the choice to make %fork rather than fork:ut was not an attempt at optimization.

Remark on the purpose of crop and fuse in Hoon, for people familiar with other subtyping-based languages

The algebra of types is used for a different purpose in Hoon than in other languages that have a similar structure. In these languages, we union together the types coming out of the branches of an if expression to get the type for the whole. If the types coming out of the branches are function types, taking the union of these means taking the union of the return types and the intersection of the argument types — at contravariant positions in a type, the union operation "flips" into being an intersection. Indeed, this is the only way type intersection arises in such a language.

In Hoon, as in the other languages, we build a %fork for the result type of each ?:. But taking the union is just combining two things into a data structure, not performing a recursive traversal of the type trees. So type intersection does not arise from union in Hoon, but instead plays a completely different role. Fusing (and cropping) are used solely for the typing rules of the ?: ?= combo: as discussed above, the type being matched with is intersected with the subject for the then branch, and subtracted from it for the else branch.

Axial Operations on Types — {peek, take, tack}:ut

Peek

Consider the type corresponding to the surface syntax [* ^ @]. What is the type of its +3? The answer is [^ @]. Taking some axis of a type to get a type that models a subtree is called "axial projection" and is implemented in peek:ut. Look at the signature:

```
++  ut
|_  sut/type
  ::  ..
  ++  peek
  |=  [way/?(%read %rite %both %free) axe/axis]
  ^-  type
```

In addition to the arguments we expect, there is an extra argument "way" which describes the access pattern we're asking for, analogous to the "mode" argument of a file open operation in unix. This "way" comes into play only when our access path takes us into the payload of a core; whereupon it is checked against the core's variance model using the peel:ut subroutine.

The function opens with some axis operations

```
?:  =(1 axe)
  sut
=+  [now=(cap axe) lat=(mas axe)]
```

Cap and mas are functions which, given an axis >1, return the “head axis” (2 or 3) and “tail axis” (axis of the thing in the left or right subtree) respectively.

We then pin an empty set of type to make sure we don’t go into an infinite loop when expanding holds (if we do, rather than erroring we return %void). Afterwards, we discriminate on the tag of the type. The only interesting cases are *, i.e. %face or %hint, which calls repo:ut to strip these things off, and %core.

Looking at the code for %core, we see that the battery of a core is considered to have type %noun, which is an understandable choice, but is yet another way in which cores don’t feel like first-class citizens of the type system. If the path takes us into the payload, we peel to determine whether we are able to access the sample and whether we are able to access the context. The variable tow is pinned; it is 1 if we are accessing the whole payload, 2 if we’re accessing something in the sample, and 3 if we’re accessing something in the context. This determines which of the flags coming out of peel need to be true. If the all relevant access checks pass, we recur into the actual payload type (as opposed to the formal payload type).

At this point, the code takes a very strange turn. You might think that if you’re not allowed access to something, and you write code trying to access it, the result would be a compiler error. And you’d be right, for modes other than %read. But the compiler will, for some reason which is beyond me, allow you to read something which you don’t have access to; it’ll just strip away the type information so you’ll see it as a %noun. This is analogous to the (similarly strange) situation with batteries, although those you can access as %noun in any mode!

Take (and Tack)

Related to axial projection is axial editing. This allows us to take a value and replace a subtree with something type-incompatible. The new value will have a modified type, and computing what type it is is the job of take:ut. Witness the signature:

```
++  ut
|_  sut/type
  ::  ..
  ++  take
  |=  [vit/vein duz/$-(type type)]
  ^-  [axis type]
```

A vein is a “search trace” as returned by the name resolution system, a list of possibly missing axes. The interpretation of this as a single axis is provided by the tend:ut subroutine; the nonnull axes are traversed from the root in *reverse* order. The first return value of take is the single axis corresponding to this traversal. (Why take takes a vein while peek takes a single axis, and why take feels the need to helpfully return this single axis even though doing so is conceptually unrelated to the operation being performed, is just one of those mysteries, I suppose.) The second return value is, of course, the edited type.

The execution proceeds by reversing the search trace, then walking along it step by step. The null elements correspond to staying in place. The actual axes we handle with an inner recursion that breaks each axis into a sequence of left and right steps using cap and mas, as in peek. When we get to the end of the list, we apply duz to the current type. Thus the replacement is allowed to depend on the original value of the part of the type being replaced. This capability is taken advantage of only in cool:ut, discussed above. The tack:ut subsidiary operation is a convenience function for when the replacement doesn’t depend on the original; it calls take with duz a constant function and is used in the implementation of %=.

Minor Operations on Types — {repo, wrap}:ut

These operations on types don’t really fit into the other categories.

Repo

The purpose of repo:ut is to take a type and produce a slightly simpler type which is sort of equivalent to it. The transformation affects only the root of the type data structure being passed in, so in this way it is similar to hoon desugaring, though the analogy between these two operations does not go much further. When another operation of the compiler cannot or doesn’t want to handle a particular case of type, it will very often call repo to have that case translated into one it does handle. So hints and faces are stripped, core becomes a cell of %noun (yet another callous treatment of batteries in the type system) and the actual payload type, %noun is expanded using its definition into fork of atom or cell of nouns (this behavior is taken advantage of, for example, in crop). Finally, holds are passed to the rest:ut subroutine, which expands them by calling play. This subroutine is notable because it is the thing that mutates and uses the value of fan, a set pinned in the context of ut. Fan tracks what holds we’ve seen so far, so if in the process of playing to expand a hold, we wind up repoing that same hold, we’ll know we’ve gotten in an infinite loop.

Wrap

You know how you can use ^?, ^&, and ^| to change the variance of a type? This is the function that implements that.

Name Resolution — find:ut

When you say a.b in Hoon, the compiler needs to find a.b in the subject. The operation responsible is find:ut, which has signature:

```
++  ut
|_  sut/type
  ::  ..
  ++  find
    |=  [way/?(%read %rite %both %free) hyp/wing]
    ^-  port
```

This gate is a thin wrapper around the main logic, which is in fond:ut. Fond has the same signature, except instead of returning a port, it returns a “pony.” These types are some of the most confusing and worst laid out in the system. Let’s take a look:

```
+$  port  (each palo (pair type nock))              successful match
+$  palo  (pair vein opal)                          wing trace, match
+$  vein  (list (unit axis))                        search trace
+$  opal                                            limb match
          $%  [%& p/type]                           leg
              [%| p/axis q/(set [p/type q/foot])]   arm
          ==
```

```
+$  pony                      raw match
          $@  ~               void
          %+  each            natural/abnormal
            palo              arm or leg
          %+  each            abnormal
            @ud               unmatched
          (pair type nock)    synthetic
```

Pony has two extra not-found cases compared to port, which find zaps out on if it encounters — the ~ and the [%& %| @ud] —, and one duplicate (!?!) case of (pair type nock), which find moves to the other case. The palo case of the pony is returned unmolested.

The extra not-found cases are used as intermediate signals when doing recursive tree search. The ~ case seems to mean “give up entirely.” The [%& %| n] case means “I didn’t find it in this subtree, but there are n ^s remaining.” So for example, if we’re looking for ‘^^^a,’ and get a [%& %| 1] when we look in the left subtree, this means we found 2 ‘a’s in the left and should now proceed to look for ‘^a’. On the other hand, if the left search had returned ~, we would not attempt a right search.

If we expand out port (recalling that (pair a b) is [p/a q/b] and (each a b) is $%([%& a] [%| b])), we get:

```
+$  port
  $%  $:  %&
          (list (unit axis))
          $%  [%&  p/type]
              [%|  p/axis q/(set [p/type q/foot])]
      ==  ==
      [%| p/type q/nock]
  ==
```

There are three cases here, corresponding to the three possible ways a name can resolve:

1. To a face (sometimes called a “leg,” encoded %& %&)
2. To an arm of a core (encoded %& %|)
3. To an alias, as from =* (encoded %|).

Additionally, the first two cases have associated with them a “search trace,” or vein, which if filtered for nonnulls, reversed, and pegged together, is the axis of the matching face or of the core within which the matching arm was found. So, for example, [`2 ~ `3] means, start at the root, go to +3, stay put, go to +2 — +6 in other words.

The body of fond is divided into two parts. The second, starting with the => that pins axe, lon, and heg, is the code for processing a single limb (the current limb being heg). The first is the recursion that executes the second for each limb of the wing, in reverse order.