THE PROTEUS CHEAT-SHEET

Rational For Proteus

Notes I'll need for this section:

- Identities let it choose the best program structure so we don't have to think about it. This will also make it faster and more bug-proof.
- Models can be used in intensional contexts.
- Modeling is intuitive
- Models have structural info so can do part/whole inference
- Can work at high or low level
- function-like, but interactive.

Proteus Language Overview

Infons: "Pieces of information" or "Systems with state"
→ Types of infon: Integer, String, List, Description
→ Properties: Size, Value. Either can be "unknown"
→ Infons can be identical to other infons. Use '=' for identity.

Examples: Notice units for size and syntax for unknowns
E.g.: size of ints is measured in *states.* Unknown int is '_'.

Integers (states)	123, -123	_
Strings (chars)	"Hello", 'Hello'	$
Lists (items)	{1, 2, 3, 'hi' (4, 5} }	{…}, {1,2, ...}
Descriptions	#123, #'hi', #{1,_,$}	#_, #$, #{...}

Setting the size of an infon: Use * for size and + for value.
Examples: (Especially notice the last one)
→ 4 item list with last two items unknown: *4+{123, 456, …}
→ 16 state system in state 7: *16 + 7
→ String of length 8, value is unknown: *8+$
→ +{*3+$ *4+$}=+"CatDogs" // This parses to {'Cat', 'Dogs'}

Notes: A comma after an int means size defaults to 1
→ *{1, 2, 3} // three values with sizes of 1 state.*
A semicolon after an int means value defaults to _.
→ {1; 2; 3;} // three items of sizes 1 state, 2 states, 3 states.

Concatenating/Calculating: (…)
Lists in parenthesis are flattened by one layer. **Examples:**
({1, 2, 3} {4, 5, 6}) = {1,2,3,4,5,6}
("Hello" ' ' "World") = "Hello World"
(+3+7) = +10

Tags and Tag Chains
Tags let you assign a global name to an infon.
→ Tags can be used to create classes and named functions
Example: In world we define a tag: red={255,0,0}
→ then we can reference it: {red rectangle}
Note: that definition of red is a toy example. A useful definition would connect it to perception of photons and it would range over all the colors people are likely to call red.

Tag-*Chains* will make it easy to use multiple tags in an English-like way: "Big red bike" or "Very big red bike"

Unknowns and Parsing

Indexing & Selecting: \\\, \\^, %, &

Note about […, <>, ...]

Functions and Inverse Functions
Functions and their inverses are formed from identities.
→ A list in [ ] returns the last item*.
→ Putting :: <infon> after a list set the first item to <infon>
So: `[_, 7]::10` sets the _ to 10 then returns 7.
→ If, instead of 7, the last item was a map to other things, and to the first item, this will act like a function. The function body is in the [ ]. The arguments are after the ::. Multiple arguments can be passed as a list.

Repetitions: Map, Reduce, Filter, etc.: {… | ...}

Misc Notes
1. Candidates for syntactic sugar:

Formal Syntax

Getting Started with Proteus

Getting and Building the Proteus and Slip Engines

Testing the build

Normalizing your first Infon

Things to try

Proteus for C and Haskell Programmers

You can't do this with C or Haskell / Lisp:

A Tour of the Code

Source Files:
- **InfonIO.cpp:** Reads / writes Proteus files to / from infons
- **Proteus.h:** Defines constants, infon, agent, and macros
- **Proteus.cpp:** Implements normalize() and helpers.

The infon class

```
struct infon {
   UInt  flags;   // See below
   infon *size;  // The *-term: Size in states, chars, or items
   infon *value;  // Either uint, char* or infon* to first item.
   infon *next, *prev, *top, *pred;   // used for lists and such
   infon *spec1, *spec2; // Indexes, functions args, etc.
   infNode* wrkList;  // Pointers to infons identical to this. =
}

THE FLAG FIELD: 32 bits – bytes 0,1,2,3
→ Bytes 0 (for value) and 2 (for size) have similar structure.
→ The constant goSize shifts constants from byte 0 to byte 2.
Bytes 0 and 2: flags for value and size
* bit 0-1: flags&tType: 0=type unknown, 1=int, 2=string, 3=list
* bit 2: Field is reserved (04)   (notLast)
* bit 3: fConcat(8): This list uses (), not {}. Flatten it one level.
* bit 4: fLoop (10): Create a list by repeating a pattern. {...|...}
* bit 5: fInvert (20): For size, 'division', for value, 'minus'.   /, -
* bit 6: fIsComplete (40): Has missing pieces.
* bit 7: fUnknown (80): This item not given here. _, $, {…}, ?

Byte 1: Flags for status of whole infon.
* bit 0-2: flags&mRefMode // reference mode for infon
   → 0: toGiven; search/scan the literally given list in spec1. %
   → 1: toWorldCtxt: search agent's context then world.  &
   → 2: toHomePos: search 'this' starting from first element. \\
         → Scope is determined by how many '\'. rec'd in spec1
   → 3: fromHere: doHomePos but don't go Home.  ^
   → 4: asFunc: This is a function. spec1=args, spec2=body. :
   → 5: intersect: Return middle item instead of last. [..,<..>,..]
   → 6: asTag: This is a tag (spec1) to dereference. Red dog
   → 7: asNone: No special evaluation to be done.  *--- +---
* bit 3: mMode: function or search is inverted.          0x800
* bit 4: isNormed: Don't norm it again unless needed. 0x1000
* bit 5: asDesc: This is a description to evaluate later. 0x2000
* bit 6: toExec: Evaluate the description now.           0x4000
* bit 7: sizeIndef: Size is given, but can be overridden.0x8000

Byte 3: Flags for lists and misc.
* bits 0-3: This item: isFirst, isLast, isTop, isBottom.
* bit 4: hasAlts: Only one set of idents, not all, are asserted.
* bit 5: noMoreAlts: This is resolved, stop trying possibilities.
* bit 6: isTentative: This is set when the infon exists in one alternative but not others. When the alternative is resolved (in resolve() ) the infon is erased or isTentative is unset.
* bit 7: isVirtual: This infon will be filled in but it isn't yet. {..|..}

→ The next and prev fields are for list infons.
→ The pred field is like prev but spans multiple lists.
→ Top is either a pointer to the first item or its parent.
→ In non-list contexts, next and prev can have other uses.
```

The agent class

```
struct agent {
   agent(){world=World;};
   infon* normalize(infon* i, infon* firstID=0, bool doShortNorm=false);
   infon *world, context;
};
```

NOTE: The Slipstream code is not covered here.

Inferring Infon Structure from the Properties of Information

Notes I'll need for this section:

1. States=info: time/space
2. non-info is undetectable
3. representing via identity
4. Need patterns of * an d+
5. Normalize=find identities, substitute Identicals
6. Infons vs. numbers, infons vs. groups
7. Inferring from closed system.