Watt: A Self-Sustaining Functional Language

C. Guy Yarvin
cgyarvin@gmail.com

Abstract

Watt is a typed functional language, defined by a 3000-line kernel which compiles itself to Nock. Nock is a simple non-lambda automaton, defined by a 250-word spec. Nock and Watt are unpatented and in the public domain.

1. Introductions

Watt has three: scientific, practical, and mythical.

1.1 Scientific

As CS research, Watt is a response to the call of Kay and others ([3], [5]) for a "Maxwell's equations of software": a small model of computing which defines a self-sustaining programming language. Nock is the model; Watt is the language.

Previous approaches to Maxwellian programming, generally divided into Lisp (from McCarthy [9] to Graham [4]) and Smalltalk (from Kay [7] to Piumarta [11]) families, have focused on interpreters (lambda calculus, object-message models) which could be characterized as "one-layer": the transformation from source to execution model is relatively straightforward. In a one-layer interpreter, the compiler is a mere parser or not much more; the interpreter's semantics are the language's semantics, more or less.

Compared to Nock and Watt, self-sustaining interpreters in the Lisp and Smalltalk traditions seem relatively high-tech, and languages relatively low-tech. Nock's one-page spec has no types, variables, functions, etc. It has one control flow operation, recursion. It has one arithmetic operation, increment. Watt has a higher-order type inference engine, like (but very different from) Haskell's [10]. One two-layer design is Tarver's Qi [13], built on Common Lisp; Common Lisp is two orders of magnitude bigger than Nock.

There are many small Turing-complete automata (eg, SK combinators [12]) and self-interpreting interpreters (eg, Tromp's binary lambda calculus) [14]. Most such systems are "esoteric," ie, not designed for practical programming; in practice they exhibit unsolvable performance problems, unusable user interfaces, or both. Watt is an unusual language, but not an esoteric one.

Nock and Watt are defined by three files: `A`, `B`, and `C`. `A` is the Nock spec, in English and pseudocode. `B` is the Watt kernel, in Watt. `C` is the Watt kernel, as a Nock noun.

In pseudocode, Nock can be described as a partial function `N` of two arguments, a subject `S` (the input) and a formula `F` (the function), on failure not terminating, on success producing a product `P` (the output). Hence

```
N(S F) == P
```

To run a Watt function `W` on a data value `D`, we compute

```
N(D N(W C))
```

To verify the correctness of `C`, we check that

```
N(B C) == C
```

Note that Nock is the only execution function. There is such a thing as a Nock interpreter optimized for Watt; there is no such thing as a Watt interpreter.

The architecture is easy to understand if we think of Nock as a "functional assembly language." In this metaphor, Watt is C, `A` is the machine instruction set, `B` is the C source for `cc`, and `C` is the executable for `cc`.

These files (all available from `github:cgyarvin/urbit`) are quite small, both physically and semantically. `A` (`nock8.txt`) is 44 lines, 228 words, 1154 bytes; `B` (`watt297.watt`) is 3004 lines, 7921 words, 67K; `C` (`watt297.nock`) is 360K. Compressed size is a good proxy for Kolmogorov complexity, which is a proxy for semantic size; A gzips to 424 bytes, B 16K, C 42K.

(`C` is hefty because it includes the complete type of the kernel, which includes its complete abstract-syntax tree. If this is removed, yielding a kernel which can compile its successor but not user-level programs, `C` is 150K compressing to 19K.)

1.2 Practical

The Nock/Watt design may be elegant. It *is* elegant. At least, it fits on a floppy disk. But so what? Who needs that disk, and why? Watt is an open-source project, not a computer-science exercise. It needs a point.

Watt is a general-purpose programming language. As a pure functional language, it will never be fast enough to obviate Algol-type imperative programming, at least for inner loops. Most code is not bottleneck code, though, and the practical use of slow languages is considerable. For instance, Watt (in principle, not at present) can do anything Javascript can do, PHP can do, etc, etc.

But what can Watt do that Javascript can't? The conventional wisdom is that every new language needs a real use case that no existing language can satisfy. Conventional wisdoms are often right.

Watt is a functional specification language. Its niche is the precise definition of arbitrary functions, especially in network protocol and file-format standards. Today, these standards (such as RFCs) are normally written in English and pseudocode.

Why are Internet standards, for which precision is essential, expressed informally? Because the precision of a spec is is generally proportional to its size, and its size includes all its dependencies.

If the layout function in HTML 5 is defined in Java, its semantics depend on the entire Java environment (not just the JVM) against which the code runs. Java was originally designed as a write-once, run-anywhere language. Mostly, it is. But it retains many installation dependencies. Even the JVM, which is anything but trivial, is versioned and extensible. While quite stable, therefore, it is not as stable as possible.

Even extensibility is a vice in a specification language. Both Nock and Watt are versioned with an absolute convergence scheme: "Kelvin versioning." In Kelvin versioning, versions count not up by major and minor release, but down by integer degrees Kelvin. At absolute zero, all change ends.

Since the Nock spec is extremely small and cold (8 Kelvin), it should be extremely precise. A naive Nock interpreter is the proverbial page of code. An incorrect one demands unusual energy in incompetence. An efficient Nock interpreter can be quite internally complex, but its semantics are no different.

If two Nock interpreters disagree, one is buggy, and it is trivial to figure out which - no "standards lawyering" involved. And since since all Watt programs (Watt itself included) execute exclusively in Nock, a metacircular operating environment built on Nock should retain Nock's precision at all layers.

"Standards" is a broad category. The specific use case for Watt is the relatively new problem of dissemination or "content-centric" networking protocols [6]. A DN protocol is one whose semantics discard routing information, such as source IP. Hence, data must identify and authenticate itself.

Dissemination protocols are especially suited to functional specification. With irrelevant routing information discarded, the semantics of a dissemination client can be defined, uniformly across all nodes, as a single standard function on the list of packets received. Obviously, if this function is not precise, the protocol is ill-specified.

Many languages, even Javascript, may be well-specified enough to define useful and interesting dissemination protocols. For some simple functions, the old English-and-pseudocode RFC approach is serviceable as well. However, the challenge increases considerably if the protocol function is metacircular - that is, it extends its semantics with code delivered over the network.

Javascript has "eval," but (as in most languages), it is best regarded as a toy. You could also try this trick in Haskell. But how well would it work? Metacircularity in most languages is a curiosity and corner case. Programmers are routinely warned not to use it, and for good reason. Practical metacircularity is a particular strength of Nock, or any functional assembly language. Dynamic loading and/or virtualization works, at least logically, as in an OS.

We hypothesize that a metacircular functional dissemination protocol can serve as a complete, general-purpose system software environment - there is no problem it cannot solve, in theory or in practice. Such a protocol can be described as an "operating function" or *OF* - the functional analog of an imperative operating system.

How does an OF compute? Consider a dissemination protocol which implements a monotonic global namespace, in which new packets bind new names but do not rebind old ones. Such a namespace is familiar to any user of a modern revision-control system. With Nock and Watt, a value in this namespace can be defined as a pure, stateless, standard function of a name and a packet log.

In such an OF, all code that executes is either in the Watt kernel, in which case it is frozen, or installed as source in the namespace, in which case it has a global unambiguous name. Hence, no incompatibility appears at any layer.

The next layer in the Nock/Watt stack is Urbit, an OF. Nock is liquid helium; Watt is dishwater; Urbit is vaporware.

1.3 Mythical

A new language, especially one as weird as Watt, demands more than a prosaic problem statement. It needs an *origin myth.* And an origin myth it has.

The introductions above are, quite frankly, cover stories. Watt is not the author's unsupervised CS thesis, nor an open-source project. It is not even his own work - nor that of any man. Rather, it is alien technology.

Nock and Watt are the software infrastructure of the planet *Urth* (pronounced "Ürth"), one of whose spacecraft abducted the author on a heavy acid trip in the Mojave. The relationship between Urth and Earth is unclear. Urthlings are clearly familiar with ASCII, and some strings suggest non-trivial contact with English. Urth may even be Earth in the future, or the past, or an alternate timeline.

Beyond these cosmetics, however, Urth code is nothing like Earth code. Watt is *not in any known language family*. Its syntax is just as alien as its semantics. Even the indentation is insane. 0 is true, and 1 is false. Anything you recognize is likely to be a fundamental feature of the universe.

Does Earth need alien programming? Does it need to throw away all its code, and replace it with Urth's? Possibly. But not right away. Certainly, there is no "flag day" - Earth programmers are used to gluing together multiple paradigms. Consider the number of visions of programming in the components that comprise an ordinary LAMP Web server.

One benefit of the Maxwellian paradigm is that it enforces a strict hierarchical relationship between Earth code and Urth code: Earth always calls Urth, Urth never calls Earth. Because Urth semantics are rigorously defined, there is no place for ad-hoc extensions. Watt cannot call out to native libraries. Rather, native applications which want to execute rigorously defined functions must call Watt, which therefore functions as a scripting, extension or query language - a common niche.

The concept of a functional application is not meaningful. Unlike Haskell, with its logically external but closely-coupled I/O interpreter, Watt does not fight this point. We could imagine an Earth application that specified most of its functionality in Watt - for instance, a browser might so specify its HTML renderer. It would still be a C++ program, however. So Urth infiltrates, but never attacks.

2. Nock

The Nock spec, 8 Kelvin, is Figure 1.

Is this spec even precise? It *is* precise. But also terse. Nock does not exactly document itself - in standards-lawyer parlance, it is the normative text without the informative.

2.1 Nouns

Nouns are Nock's data model. Again, a noun is an atom or a cell; an atom is an unsigned integer of any size; a cell is an ordered pair of any two nouns.

The noun is the Platonic essence of acyclic data. XML, S-expressions, JSON, and the like all reduce naturally to nouns, which can be described as "S-expressions without the S."

Nouns are simpler than these formats, because they can be. They can be, because they are a data structure rather than a document format - nouns exist exclusively as the bottom layer in a two-layer execution model. As with actual assembly languages, nothing at the Nock layer is a UI. There is no generic noun syntax.

For instance, an atom can represent a number - or a byte string - or a binary file. (The convention is LSB first.) It has no hidden type information. How you print it is your own business. If you want to remember how to print it, you can explicitly keep its Watt type at runtime, and use a pretty-print function that looks at the type. Or just guess, as a generic pretty-printer must. The same goes for scanning nouns.

As a data model rather than a document format, nouns are expected to develop dag structure, though this is transparent as Nock cannot branch on it. Performance often depends on dagness, though; nouns must be serialized with care. However, Nock cannot generate cyclic or infinite structures, so memory management by pure reference counting is always straightforward and effective.

A common programming problem on all networks, Urth's or Earth's, is validation of untrusted foreign data. A type system layered above the data model, as implied in the Nock-Watt relationship, is excellent at this problem, which becomes almost trivial. In fact, the Watt programmer normally specifies a type with a normalization function or "mold," which types a generic noun. Thus, the type and the validator are one piece of code. Haskell, whose type system is much more powerful than Watt's, contends furiously with the problem of generic data.

2.2 Operators

Nock is specified as a set of noun reduction rules. Matches try to fit a dynamic noun to a static template. Non-matches fall through to the bottom and resolve to themselves - meaning that the interpreter does not terminate. Nock is "crash-only" [1] - its only failure mode is termination.

In this pseudocode (which must not be mistaken for Watt), we see five built-in operations, with prefix syntax, in the Nock spec. These are `?`, `^`, `=`, `/`, and `*`. They correspond to partial pure functions on nouns. The Urthlings have defined these operators by reduction, but neglected to name or describe them otherwise.

However, borrowing some names from Watt, we can say either "ask," "hat", "ben", "sol" and "ras" (to name by Watt syntax), or "trol," "vint," "sing," "frag" and "sail" (to name by Watt semantics). Watt is at all times pronounceable aloud, if you're willing to sound like an alien.

`*[a]` ("ras" or "sail") is Nock itself. While Nock is defined for every noun, the student soon notices that it exits (evaluates to itself) on any atom. Only for cells `[s f]` can it succeed. The student also notices that `s` seems to be the data, and `f` the function. Indeed, we say "subject" and "formula," or "sub" and "fol" for short.

`/[a b]` ("sol" or "frag") is a crude tree-addressing scheme. Consider it a sort of ghetto DeBruijn index. `a` is the address; `b` is the tree. The root is at 1; the head of the noun at `n` is `n + n`; the tail is `n + n + 1`.

Thus `/[7 478 152 93]` is `93`, as is `*[[478 152 93] 0 7]`. `/[5 [478 152] 19]` is `152`.

`?[a]` ("ask" or "dust") is `0`, "sic", also written `&` and pronounced "amp," and meaning true, if `a` is a cell; `1`, "non", also written `|` and pronounced "bar," if `a` is an atom.

`^[a]` ("hat" or "vint") is the increment of a if a is an atom. Otherwise it exits.

`=[a b]` is `&` if a equals `b` (ie, the same noun, not pointer equality, which cannot be tested); `|`, "non" otherwise.

2.3 Instructions

The first block of sail reductions, `*[a [b c] d]` through `5`, is the *essential* set - Nock is formally incomplete without them. The second block, `6` through `12`, is the *expanded* set, which exist only for prosaic practical reasons.

The only interesting essential instruction is the first, which provides a sort of "auto-cons" feature. For any two

```
1 Structures

    A noun is an atom or a cell.  An atom is any unsigned integer.
    A cell is an ordered pair of nouns.

2 Pseudocode

    [a b c] is [a [b c]]; *a is nock(a).  Reductions match top-down.

3 Reductions

    ?[a b]              0
    ?a                  1
    ^a                  (a + 1)
    =[a a]              0
    =[a b]              1

    /[1 a]              a
    /[2 a b]            a
    /[3 a b]            b
    /[(a + a) b]        /[2 /[a b]]
    /[(a + a + 1) b]    /[3 /[a b]]

    *[a [b c] d]        [*[a b c] *[a d]]
    *[a 0 b]            /[b a]
    *[a 1 b]            b
    *[a 2 b c]          *[*[a b] *[a c]]
    *[a 3 b]            ?*[a b]
    *[a 4 b]            ^*[a b]
    *[a 5 b]            =*[a b]

    *[a 6 b c d]        *[a 2 [0 1] 2 [1 c d] [1 0] 2 [1 2 3] [1 0] 4 4 b]
    *[a 7 b c]          *[a 2 b 1 c]
    *[a 8 b c]          *[a 7 [7 b [0 1]] c]
    *[a 9 b c]          *[a 8 b 2 [[7 [0 3] d] [0 5]] 0 5]
    *[a 10 b c]         *[a 8 b 8 [7 [0 3] c] 0 2]
    *[a 11 b c]         *[a 8 b 7 [0 3] c]
    *[a 12 b c]         *[a [1 0] 1 c]

    ^[a b]              ^[a b]
    =a                  =a
    /a                  /a
    *a                  *a
```

Figure 1. Nock 8K

Nock formulas `a` and `b`, the formula `[a b]` produces what in a Lisp dialect would be `(cons a b)`. This does not conflict with the atomic prefixes of the other instructions, because every Nock formula is a cell. It compacts formulas considerably.

Also notable is 2, which can be seen as a sort of ”eval” at the Nock level. However, because it does not in any sense compile, it should be seen as more of a jump operation. 2 is the only mechanism for indirect control flow. In Lisp terms, Nock has ”eval” but not ”apply.”

The expanded set, `6` to `12`, is more recondite but less interesting. Observe that expanded instructions are basically built-in macros. At the Nock level, though, macros are useful only for mechanical performance, not human usability. They could be elided, but Nock is simple enough to afford some slack.

Briefly, `6` is if-then-else; `7` is function composition; `8` pushes a variable; `9` is a calling convention. These make Nock formulas somewhat smaller and faster. `10`, `11`, and `12` are semantically useless, but send hints to the execution engine; `10` consolidates for reference equality, `11` drops an arbitrary, unspecified hint, `12` labels for acceleration.

2.4 Programming

Is Nock even Turing-complete? It sure is. The basic trick of Nock programming is to create subject nouns which contain both data and code, often known as "cores," then sail to a formula in the core, using the whole core as the subject.

For instance, lambda - a primitive to most - is no more than a design pattern to Nock. As a noun, a lambda (or "gate") is a special case of a more general noun pattern called a "core," which is conventionally a triple of the form

```
[[sample context] battery]
```

sometimes shortened to

```
[[sam con] bat]
```

The battery is a bush of formulas, addressed by fragment. The sample and context are the argument and the environment. If this core is at `[0 1]`, the sample is at `[0 4]`, the context (ie, environment) at `[0 5]`, the formula at `[0 3]`.

To call the function, replace the sample with your argument, and invoke the formula with the whole core as the subject.

In the gate (ie, lambda) convention, the battery is a single formula, and the caller replaces the sample without changing the context. The result is a function call. For instance, if you have a gate at [0 1] and you want to call it with the argument 47, you can use the formula

```
[2 [[[1 47] [0 5]] [0 3]] [0 3]]
```

A lambda call if you've ever seen one. However, like any design pattern, but unlike the "ultimate" lambda, it can be generalized. For instance, the caller can revise the context, as well as the sample. Or the battery may be not just one formula, but a tree thereof - a sort of method table, all sharing the same data. Thus we see something like the object-oriented pattern as well.

2.5 Performance

An efficient Nock, as with any interpreter, is a hard problem. Following the famous advice of Knuth, it is a problem on which we have chosen to procrastinate. However, it does eventually have to be solved.

Like `cc`, a Nock compiler produces code that is ugly but not entirely illegible, and heavy but not absurdly bloated. Unlike regular assembly language, Nock is inherently slow. But not unamenable to optimization.

Nock's only arithmetic operation is increment. It is a fun if painful exercise to write a Nock decrement formula. (It is more badass if you use no instruction above `5`.) However, the successful student will note that her decrement is `O(n)`, in the value of the sample.

Which does not bode well for efficient execution. However, we note a simple solution: we can write a souped-up Nock interpreter, which recognizes the Nock formula of the decrement function in the standard library, and does it fast in C. Unless this "jet" is buggy, it does not extend or violate Nock; the programmer cannot tell it is there.

From the C programmer's perspective, Watt is really no more than a library for specifying your C functions. The Nock interpreter invokes your C and can test it easily against its specification. A smart interpreter could even assign itself a test overhead budget, such as 10% of runtime, and check jets randomly in production execution.

In future, acceleration could and probably should be pushed up to the user level by something like a functional C interpreter. If such an interpreter is itself jet-propelled, the implementation can employ all kinds of dynamic compilation techniques, while retaining its precision. Formal equivalence testing of functional and imperative programs remains an extremely hard theoretical problem. But empirical testing is sufficient in most practical cases.

Also, nouns (being acyclic) are very easy to manage in memory. Reference counting is so obvious that it is not yet implemented. Instead, the current optimized interpreter uses two opposing and symmetrical stacks, each computation placing its compacted result on one stack, and temporary data on the opposite stack. To compute the temporary data, the stacks reverse. Saved temporary pointers are copied on assignment.

This design is not appropriate for all algorithms and must be heavily supplemented by judicious hints, including a refcount hint. However, it has the advantage of (a) zero fragmentation and (b) straightforward, on-the-fly tail-call elimination. With little or no tuning, the current interpreter runs at about 5 million steps per second on a 2Ghz Core Duo, directly out of the Nock tree with no bytecode translation, not using the C stack and executing tail-call loops in fully compacted heap space.

Another implementation trick is to attach a 31-bit short hash, or "mug," to every cell and large atom. This is computed lazily, as needed; all nouns have the slot. A mug should not be mistaken for a secure hash. But it is convenient for many associative data structures.

More generally, hints can add all kinds of transparent baggage to data; for instance, they can request that a list carry a linear index for O(1) access, or even that an RDBMS cache database indexes. Since hints are in all cases semantically transparent, they can be specified by informal convention. No work at all has been done in this area.

Finally, since Nock is referentially transparent, memoization (result caching) hints are also straightforward. Indeed, Watt's type-inference algorithm would be quite intractable without memoization.

3. Watt

Unlike most self-sustaining designs, Watt is a typed language. Like Haskell, Watt has polymorphic, higher-order type inference, pattern matching, and monad syntax. Fortunately, the resemblance is entirely superficial.

(If you've had a bad time with metamathematical FP - Haskell, ML, F#, etc - Watt is functional programming for stupid people like you. If you love Haskell, you are very smart and can learn Watt with no effort at all. If you prefer a Lisp or Smalltalk dialect, after some brainwashing you may come to understand that your old way was just an archaic, broken form of Watt. And if you adore Perl, you'll worship Watt's syntax, which is even more perverse and arcane.)

Watt is more than just a language. It is a kernel which compiles itself to Nock - and compiles user-level programs that actually do stuff. For both, it needs a spartan but sensible standard library. For instance, Nock has increment but no decrement, so the Watt kernel must contain a decrement formula.

(When Watt "bootstraps," compiling the next version of itself, it must not install the current kernel's formulas in the next kernel. Otherwise, it would not be a new kernel. But when it compiles an ordinary end-user program, this program must be able to reuse the kernel's standard library, type and all. If this sounds tricky, it is.)

Watt 297K, 3004 lines of code, is divided into tiers, 1 through 6. Tier 1, basic arithmetic, about 100 loc. Tier 2, basic containers, 200 loc. Tier 3, miscellaneous noun surgery, 300 loc. Tier 4, associative containers, 150 loc. Tier 5, monadic parsing, 300 loc. Tier 6, Watt in Watt, 2000 loc.

(Bear in mind: the whole kernel gzips to under 16K. This suggests that Watt source is a heavy newline consumer, which it is. It also suggests that the kernel is entirely uncommented. Which it is. (Reference code should never contain comments.))

Because Tier 6 contains a code generator, a type inference engine, and a top-down combinator grammar, it is and must be fairly meaty. The best code samples, indeed, are in its depths. We do not have space to more than look at samples, but let's look at some anyway.

Figure 2 is `play`, a gate over the `rose` core, which is over `plow`, which is all of Tier 6. (Good style in Watt encourages relatively deep core nesting.) `play` is the main loop of Watt type inference; it computes output type from input type and gene, without verifying or generating code.

If you understand this, you may be an alien yourself. Which is weirder? The syntax, or the semantics? Some may find it hard to say. Let's take a brief tour of the language.

3.1 Semantics

Semantically, Watt's most unusual feature is the lack of one. Watt has no concept, as found in most languages functional or imperative, of "scope" or "environment." Watt is neither lexically nor dynamically scoped. It's not scoped at all.

All Watt computations have access to one and only one noun - the Nock subject. If you want to compute a function, the Nock formula for that function must be somewhere in the subject. And you want to read a variable, ditto. There is nowhere else to address - least of all, heaven forfend, I/O.

`plow` exports a variety of compilation gates, but all share a common ingredient: a subject type, generally named `sut` (and shared across the `rose` core), and a Watt gene, generally `gen`. The gene is an abstract syntax tree, computed directly from the atomic source file.

Compilation gates combine three primitives, `play` (seen above), `make` (which makes the formula), and `show` (which checks correctness). Combinations are `shop` for `show` and `play`, `pass` for `show` and `make`, `wish` for `make` and `play`.

Watt is a static type system. There is no type data at runtime. There is no place to hide it. And all equivalence is structural, since there is no scope to define identity.

A Watt type defines a set of nouns, and ascribes semantics to that set. Core types, ie types of nouns containing formulas, are not exceptions to this rule. The formula is precisely defined as the product of type and gene; the core type contains the gene.

Type inference in Watt is exclusively *forward*. Type flow never flows backwards, as in a Hindley-Milner [2] or other unification algorithm. Watt is a pure inference language - there are no type definitions. To compile is to convert input type and gene, to output type and formula.

The disadvantage of forward type inference (FTI), as against unification type inference (UTI), is that FTI is much less powerful, and hence requires more human assistance. Watt can infer the type of tail-recursive computations; for head recursion, it requires a declaration (which it checks).

The advantage of FTI is that FTI is much less powerful, and hence easier for humans to learn and follow. As a user interface, Watt's type inference is like a manual transmission; to use it is to know the algorithm, and apply it unconsciously as you read or write code. There is no suggestion that Watt type inference will "do the right thing" - the programmer's job is to manage it.

3.2 Type

Figure 3 is the Watt type system - sans algorithms. More precisely, this code is a "mold," a gate that ascribes type to an untyped noun (or exits). Thus, if `x` is any noun (type `%blur`), `(foobar x)` is `x` as a `foobar`, with `(foobar)` or `*foobar` (the latter should be folded as a constant, but is not yet) as a sample.

We observe ten forms of `*type`. First, some basics. `%blur` (pronounced "mit blur," the ASCII string "blur" as

```
::::
::
  play
=+  [gen=*gene]
|=  #(..rose %play 6)
^-  *type
?-  -.gen
    %bone   [%cube p.gen]
    %bran   [%face p.gen $(gen q.gen)]
    %cast   $(gen p.gen)
    %dust   [%fork [%cube 0] [%cube 1]]
    %flac   $(sut $(gen p.gen), gen q.gen)
    %germ   $(gen q.gen)
    %grit   grit(sut $(gen p.gen))
    %hint   $(gen q.gen)
::
    %mack
  =+  lar=(seek p.gen)
  =+  mut=(turn q.gen =+([p=*gene q=*gene] |=([p $.-.$(gen q)])))
  ?~  q.lar
    (edit(sut r.lar) mut)
  [%hold (edit(sut q.u.q.lar) mut) r.u.q.lar]
::
    %pank   [%core sut [%hard (fill:gull q.gen)]]
    %plin   [%fork [%cube 0] [%cube 1]]
    %sail   %blur
    %sing   [%fork [%cube 0] [%cube 1]]
    %stil   sut
    %tash   [%core sut [%soft sut (fill:gull q.gen)]]
    %trol   (eith(sut $(gen q.gen, sut (gain p.gen))) $(gen r.gen))
    %twix   [%cell $(gen p.gen) $(gen q.gen)]
    %vint   %atom
    %wost   $(gen p.gen)
    %zalt   [%cube $(gen p.gen)]
    %zemp   $(gen q.gen)
    %zike   %blot
    %zoot   seed
    %zush   $(bug .^(bug), gen p.gen)
    *       $(gen (open gen))
==
```

Figure 2. play.rose.plow

an atom LSB first), means "any noun." %blot means "no nouns." %atom means "any atom." [%cube p=*] means the constant p. %cell is a recursively typed cell. %face names a subtree, or "part."

More interesting are the algebraic combinations, %fork meaning set union or "either type," and %fuse meaning set intersection or "both types." In particular, %fork is symmetrical (semantics do not depend on order), whereas %fuse is asymmetrical (order matters).

The semantics of a %fuse are that p filters q, pruning all fork branches in q whose intersection with p is empty. This allows pattern matching to bind variables.

%hold implements manual laziness for recursion control. Since Watt can infer the type of recursive programs, it must be lazy in following the recursion, or it will look forever. Since Watt is a strict language, it must defer computation explicitly. A %hold simply means the result type of q as played with the subject p.

```
::::
::
    type
  |?
    %blur
    %blot
    %atom
    [%cell p=*type q=*type]
    [%core p=*type q=*<[%hard p=*spec] [%soft p=*type q=*spec]>]
    [%cube p=*]
    [%face p=*term q=*type]
    [%fork p=*type q=*type]
    [%fuse p=*type q=*type]
    [%hold p=*type q=*gene]
  ==
::::
::
      spec
    (book term gene)
```

Figure 3. type.plow

Manual laziness is useful in controlling infinities. By searching in an explicit control pool, a conservative algorithm can check that it is trying to solve a problem it is already solving, and prune. It is not quite clear how a lazy language would express this - perhaps, by simply not using the feature. However, as discussed below, abuse of soft cores *can* send Watt's inference algorithm into an infinite loop. This is Watt's way of teaching you not to abuse soft cores.

Our last and most interesting type form is the `%core`, which introduces code. Its `%spec` is an associative array from term (ie, name) to gene. The spec of one common core pattern, the "gate" or lambda, will contain one gene, whose term is zero - written $, pronounced "buc" or "blip."

Watt does not compute type signatures. Instead, it just leaves the source genes in the core type. A noun in the type set defined by a core is always a cell, whose head is the type p, and whose tail is the bush of formulas generated by the genes in the core spec.

A core overloads the "part" namespace served by `%face`. If the spec exports a name, the reference sails to the formula, using the whole core as the subject, and the Nock generated by the gene as the formula. Here is something like a method call.

But different, as this "hook" has no place for an argument - it is a pure function of the core, which serves here as an object. For a true method, the hook must produce a gate. The calling convention thus has two hook steps: one to produce the gate, a second to activate the revised gate. In short, a hook is a dynamic attribute, not a method; a method is a hook that produces a gate. In turn, calling the function of that gate means invoking the hook $, which is 0, the null term.

Many kinds of invocation resolve to a single Watt instruction, `%mack`, which constructs a core, revises it, and applies it. For instance, the common calling convention, `%mung`, expands to `%mack`; it generates the gate, replaces the sample at `[0 4]`, and invokes $ at `[0 3]`.

The crucial question in "macking" is: does the new data work with the old formulas? If revising a cell (or more precisely, creating a revised copy of the cell), we can replace any subtree with a value of a completely different type. However, in revising a core, the formula is read-only. The new data has to work with the old code.

The two kinds of core (hard and soft) represent a manual two-speed approach to this question. When a hard core is macked, the type is preserved; the new data must be physically compatible with the old type. When a soft core is macked, the type is changed; the old formulas must correctly execute the old genes with the new data. (This can be checked by making the old genes with the new data, and comparing the formulas to the old formulas - a crude hack dignified as "empirical polymorphism.")

Hardcore macking is monomorphic; softcore macking is polymorphic. Think of Watt type inference as an ancient Russian motorcycle, little more than an engine, two wheels and a towel to sit on, with a two-position throttle: "Sedate" and "Isle of Man." Without soft cores, Watt could not infer naturally polymorphic types, such as typed lists. Without hard cores, it would constantly crash and burn due to infinite garbage type.

Watt 297 does not have a gene for inheritance, but soft polymorphism should make this future upgrade straightforward. In inheritance with method replacement, subclasses are not subtypes, because changing the formula changes the set of nouns. Thus, all object-oriented interfaces must be softcore.

3.3 Syntax

Learning Watt is like learning Chinese. Can you learn a hundred characters of Chinese? Then Watt's syntax should be no problem. "But you must abandon your linear, Western way of thinking."

As we see, Urth knows ASCII. (Could ASCII be a fundamental feature of the universe?) However, Urth has its own three-letter names for all the ASCII graphics glyphs (Figure 4).

Furthermore, ASCII not being capacious enough for Watt's needs, we create synthetic glyphs with Watt digraphs (Figure 5). Each digraph has its own four-character, one-syllable Urth name, matching a Watt gene stem.

Digraphs can be pronounced phonetically as their glyph sequence, or semantically as their gene; eg, "askamp" or "chan" for `?&`. The meaning of digraph glyphs is formally significant, but often (as in Chinese) reveals some meaningful pattern. For instance, digraphs beginning with `?` tend to branch or test in some way.

Why not keywords? And why the cryptic names? At first, we thought strings like "lonk," "tash" and "mung" might be clues to the strange language of the Urthlings. But increasingly, we suspect that even on Urth these are just random strings.

Every language is a UI. And keywords, or even symbolic instruction names, act as what UI gurus call a "false affordance." They are very good at convincing you that you know a language, when actually you don't. If `%lonk` was called "bond," `%tash` was called "load," and `%mung` was called "call" - as they were in one experimental version - you might think you know what they meant. Whereas you don't.

The user experience of cryptic names, most famously Unix `grep`, is that when you learn a system, you know that you know it. As scientists have shown [8], this makes novice users more comfortable and confident. Initial immersion requires a small feat of associative memory, but humans are quite good at associative memory.

Another approach accounting for Watt's alien appearance is its indentation. A common if banal problem in functional languages is their tendency, as call nesting depth increases, to slide off the right side of the screen. Also irritating are large piles of terminators. These can be conquered by significant whitespace, which brings various troubles of its own.

Watt indentation (in which whitespace is not significant) does not indicate nesting depth. Also, to fight the terminators, Watt has two separate syntactic modes, "tall" and "wide."

In wide mode, all expressions are terminated, and whitespace is one space. In tall mode, whitespace is a newline and an indent; tuple bulbs are unterminated, and distinguished by two-space backsteps; list bulbs earn a two-space tabstop and the terminator ==. Also, the line comment digraph `::`, pronounced "digven" or "bosh," is restful to the eyes and good for building large decoration marks.

4. Status

An alpha release of Watt, continually updated, is available at `github:cgyarvin/urbit`. It has has all the flaws of an alpha-release language: slowness, poor debugging support, no test coverage, slowness, unstable syntax, unstable semantics, poor UI, missing features, and slowness. (Alpha performance, optimized strictly by need, is not even a relative guide to product performance.)

In particular, the core of the type system is largely unverified - the C jets compile the Watt code, but does that code compile itself? Pending more thorough testing, the answer must be: "probably not." Thus kernel 297 is unlikely to be correct - the Watt definition is no more than a spec.

Watt, which was developed without official or commercial support, is unpatented and in the public domain. In fact, a good case can be made that, since it is simply a mathematical model, Nock and Watt are not legally patentable in most jurisdictions. Of course, this is true for all software, but it seems especially true for Maxwellian software.

Acknowledgments

Thanks to Aaron Beckerman, Adam Bliss, Matt Brubeck, Stanislav Datskovskiy, Prakhar Goel, J. Storrs Hall, Alex Jacobson, Raph Levien, Ivan Matosevic, Daniel Nagy, Francois-Rene Rideau, and Colin Smith for much valuable feedback. Substantial funding was provided by the infinitely gracious contributors to the MMBF, who know who they are and what they're owed. Finally, the infinite patience of Jen Kollmer was at all times indispensable.

References

[1] G. Candea and A. Fox. Crash-only software. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[2] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM. ISBN 0-89791-065-6. doi: http://doi.acm.org/10.1145/582153.582176.

[3] S. Feldman. A conversation with alan kay. *Queue*, 2(9):20–30, 2005. ISSN 1542-7730. doi: http://doi.acm.org/10.1145/1039511.1039523.

[4] P. Graham. Arc at 3 weeks. `http://www.paulgraham.com/arcll1.html`, 2001.

[5] P. Graham. First priority: core language. `http://www.paulgraham.com/core.html`, 2008.

[6] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named

amp	&	dig	:	nub	-	tic	‘
ask	?	dot	.	pat	@	toq	"
bar	|	dus	}	pel	)		
ben	=	hat	^	pod	+		
bot	’	hop	!	ras	*		
buc	$	lep	(	red	>		
cab	_	lom	;	sac	\		
com	,	mit	%	sig	~		
dax	#	mon	]	sol	/		
der	<	nom	[	sud	{		

Figure 4. Urth glyphs

!!	"hopven"	%zike	:+	"digpod"	%trex	?|	"askbar"	%dorn
!#	"hopdax"	%zush	:-	"dignub"	%twix	?~	"asksig"	%fent
!%	"hopmit"	%zoot	:^	"dighat"	%quax	^$	"hatbuc"	%germ
!:	"hopdig"	%zalt	:~	"digsig"	%slax	^%	"hatmit"	%velt
!‘	"hoptic"	%zole	;+	"lompod"	%fist	^*	"hatras"	%mave
%*	"mitras"	%teck	;-	"lomnub"	%mast	^+	"hatdig"	%pock
%+	"mitpod"	%bung	=+	"benpod"	%gant	^-	"hatnub"	%cast
%-	"mitnub"	%fung	=-	"bennub"	%tang	^:	"hatpod"	%stil
%.	"mitdot"	%gnum	=<	"bender"	%claf	^=	"hatben"	%bran
%:	"mitdig"	%mung	=>	"benred"	%flac	^?	"hatask"	%hint
%=	"mitben"	%mack	?!	"askhop"	%vern	^@	"hatpat"	%grit
%^	"mithat"	%tung	?&	"askamp"	%chan	^~	"hatsig"	%wost
%_	"mitcab"	%frit	?*	"askras"	%moze	|%	"barmit"	%tash
%~	"mitsig"	%gath	?-	"asknub"	%grel	|*	"barras"	%pank
.*	"dotras"	%sail	?.	"askdot"	%lort	|-	"barnub"	%vamp
.=	"dotben"	%sing	?:	"askdig"	%trol	|:	"bardig"	%sunt
.?	"dotask"	%dust	?<	"askder"	%marg	|=	"barben"	%lome
.^	"dothat"	%vint	?=	"askben"	%plin	|?	"barask"	%rond
:*	"digras"	%prex	?>	"askred"	%gram	|~	"barsig"	%lonk

Figure 5. Urth digraphs

content. In *CoNEXT '09*, pages 1–12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-636-6. doi: http://doi.acm.org/10.1145/1658939.1658941.

[7] A. C. Kay. The early history of smalltalk. *SIGPLAN Not.*, 28(3):69–95, 1993. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/155360.155364.

[8] L. D. Leon, W. G. Harris, and M. Evens. Is there really trouble with unix? In *CHI '83: Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 125–129, New York, NY, USA, 1983. ACM. ISBN 0-89791-121-0. doi: http://doi.acm.org/10.1145/800045.801595.

[9] J. McCarthy. History of lisp. *SIGPLAN Not.*, 13(8):217–223, 1978. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/960118.808387.

[10] S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/definition/`.

[11] I. Piumarta. Making colas with pepsi and coke. `http://piumarta.com/papers/colas-whitepaper.pdf`, 2005.

[12] M. Schönfinkel. Über die bausteine der mathematischen logik. *Math Annalen*, 92:305–316, 1924.

[13] M. Tarver. Functional programming with qi. `http://www.lambdassociates.org/Book/page000.htm`, 2009.

[14] J. Tromp. Binary lambda calculus and combinatory logic. In *Proceedings of the 5th Conference on Real Numbers and Computers, RNC5*, page 214, 2003.