Brood

TOM SCHOUTEN

October 29, 2007

Abstract

Brood is a playing ground for exploring the space between the Scheme programming language, functional stack languages, and low level Forth. Brood is a programmable compiler. This paper serves the purpose of documenting the design of the system, and to make what I'm doing a bit more formal.

1 Introduction

1.1 Goals

The goal of the project is to create a programming tool chain for deeply embedded programming, structured as a lowlevel language based on Forth and a high–level meta programming system based on Scheme and functional stack languages. The following components are working:

- Low level. The base language is little more than an unsafe Forth stack machine model that is easily mapped to a concrete 8–bit machine.
- Simple macro system. The base language is designed with optimization and metaprogramming in mind, and has a first metaprogramming layer in the form of compositional macros.
- Interactivity. The system should be completely interactive and support incremental development from the low to the high level.

On top of this I plan to build a CONS based list processing machine, which can either host a lisp, or a dynamic concatenative language. I'm looking at both linear (tree) and nonlinear (graph) memory structures.

1.2 History

Originally, the Badnop Forth compiler was written in Forth. Because of the size of the target system, I was as good as forced to use cross compilation and tethered development. After having tasted Scheme and the Joy of functional programming for a while, I grew frustrated with the limitations of Forth employed as a meta–language, and wanted something more high–level.

The quest eventually lead to CAT, a set of Scheme macros that implements a compositional, strict, mostly functional language within Scheme. CAT is a layer that matches the impedance between Scheme and compositional / concatenative programming. CAT is used as the implementation language of the code generators (macros) that implement the Purrr18 Forth dialect.

Note that most of the ideas here are based largely on texts written by Manfred von Thun, about his programming language Joy[3], on the ideas behind the PLT Scheme macro/module system which is made by Matthew Flatt [2], and on a lot of inspiring texts about functional programming, Haskell, type theory, and category theory. I am seeing a lot of vague patterns that I cannot give a name yet, which indicates I need to read and think more about this matter. I have a bottom up mind, and can learn only by example. This system gives me the motivation to try to understand more theory. If you find blatant errors, please correct.

2 Forth Language Semantics

This section describes the system from two viewpoints. The first will be the semantics of the Purrr family of Forth languages, more specificly Purrr18 for the Microchip PIC18 architecture. The second will relate this semantics to syntactic operations on program text.

I will translate the problem of compiling Forth into the problem of re–arranging program text, which interpreted as a representation of a *function composition*, by applying *partial evaluation*. In order to get to that representation, Forth code needs to be processed because some Forth constructs do not fit in this view. In the following I will talk about two problem cases: *quoting* and *definitions*, after giving some definitions to pin down the meaning of a compositional language.

In the long run I am more interested in the (intermediate) compositional

low level language that is emerging from this approach. This language should eventually lead to a concatenative language tower, a series of bootstraps from low to high level. I see Forth as a very convenient syntax layer on top of this intermediate language, which helps guide its development.

2.1 Compositional language

Define a compositional language as a set of unary functions W. To correspond better to source code, we employ a reverse Polish notation for composition of the functions $f, g \in W$ as

$$[fg] = g \circ f,$$

where $\circ$ is the conventional notation for function composition. In the following we will use a verbatim font to indicate we're using program source text syntax, where function names are delimited by spaces. For example `[foo bar]` is a composition of the functions `foo` and `bar`.

Using our square bracket notation, the associativity of function composition can be expressed without parenthesis, as

$$[[ab]c] = [a[bc]] = [abc].$$

The application of a composition $[fg]$ to a value x is the same as applying g to the result of the application of f to x, or

$$[fg](x) = g(f(x)).$$

This rule is important for the program transformations we are about to discuss. The *execution* of a program is represented by function *application*. The expression on the right denotes the separate execution of the representations of f and g. The expression on the left denotes a program that is first transformed to a *composite* program $[fg]$ before it is applied to the data. The factors f and g are not represented separately.

When it is possible to represent composite programs in a more efficient way than representing them separately, this composition is called *partial evaluation*, and acts as an *optimization*. This is always possible, but not always desirable. It might very well be that the cost of storing or executing the composition is *more* than the sum of the costs of the factors.

What we will elaborate on later are *idioms*. These are *necessary* partial evaluations of programs that cannot be represented as individual functions,

because they employ notions that are meaningless in the context of the program, but meaningful during compilation. The factors comprising idioms act as meta–programming tools. Most of the time they operate on types that cannot be instantiated at run–time, such as symbols.

Please note that partial evaluation is an operation on *syntax*, or on the *representation* of a program. This operation preserves semantics. There is another interesting syntactic property we might use for optimization and metaprogramming, which is *commutativity* of functions. I'll come back to this later.

Note that a continuation in a compositional language is extremely simple. It is the function composition following a certain function. For example, the continuation of b in $[abcd]$ is $[cd]$. Note that in an applicative interpreter that allows nested functions, a continuation is usually represented as a stack of partial continuations.

2.2 A real machine

Modeling an imperative machine language as a compositional language works well, as long as we take the entire machine state as the domain of the functions. The resulting language might not be so useful as a programming language, for example it might lack any kind of late value binding, but the functional representation is very useful for metaprogramming by partial evaluation.

There is one difficulty though, which is the representation of *branching*. We'll just assume that the compositional program $[abc]$, with b a branch instruction, is just a normal program. However, when evaluated, the subprogram b doesn't terminate. Since this depends only on the semantics, it can be completely ignored by the compiler, which operates on syntax only.

2.3 Forth is not compositional

Forth in its original form does not fit the compositional framework mentioned above. Take the Purrr18 program fragment

```
fsymbol hello
```

which contains two words. This fragment generates code that returns a pointer to the byte string `hello`, which is stored as a flash memory byte string constant.

The second word can clearly not be a function. The reason is that `fsymbol` changes the meaning of `hello`. What we need is a way to express the same effect, only by using individual functions. To this end we introduce a *quoting syntax*. Replace the previous program by

```
'hello *fsymbol
```

where both words are functions. The first one denotes the function that quotes a data object; a function that produces the symbol `hello` on the top of the stack. The second is a function that consumes this symbol, converting it to a pointer to a byte string stored in flash memory. The composition of these 2 functions has the same semantics as the original Forth fragment.

Does this mean the target run time type system contains symbols? We can do without a representation of symbols in the target system if all occurances of such types can be eliminated at compile time. The following sections illustrate how we can do this by employing partial evaluation of the program text, and an embedding space with a larger type system than that of the run time.

2.4 Reflection and Definitions

Following up on the previous note introducing quoting to get rid of a non–composable Forth constructs, there is a special class of words that take symbols which cause a different problem: the *definition* of new words. In Forth, these are words like `variable`, `constant`, and `:` (colon). In Purrr18, this is currently handled by implementing words like `*variable`, `*constant`, and `*:` which create new names in the dictionary at compile time.

However, this is a form of *reflection* which creates difficulties for source code processing and modularization. The language before and after the definition of a new symbol is different because of the presence of a new (or redefined) word. This can change the interpretation of the following code substantially.

Forth in its original form is highly reflective, which is of course a source of great power, and the primary reason why it can be so small and still do so much. Part of my effort is to *unroll* this reflectiveness, in order to simplify metaprogramming. To make a language easier to process as data, it's best to remove it's own ability to reflect, to process itself as data. As shown in [2] one does not have to loose too much reflectivity to make modularization easier, one merely needs to unroll the reflective loop into a directed acyclic

language graph. This exposes a *static* code structure, instead of a *dynamic* reflective loop.

Currently, an entire source file is a program, delimited by exit points ; and named entry points, which are : followed by a function name. This might change to a more standard source semantics where the source is a collection of macro and function definitions. A macro definition in Forth code is just an inlined function. Macros can contain non–compilable words, but function definitions can contain them only grouped into idioms.

2.5 Types and Meta Types

In our application, the target run time type system is very simple. If we need to make it explicit, for the PIC18 it is an 8 x N array of bits. Call $\mathcal{T}$ the *target type*, the collection of possible machine states. Note that we avoid first class functions.

All *primitive* machine instructions $w \in W_{\mathcal{T}}^{p}$ are endomaps of this set

$$W_{\mathcal{T}}^{p} : \mathcal{T} \to \mathcal{T}.$$

Composing these primitive functions gives a larger set of *words*

$$W_{\mathcal{T}} = \{[ab\ldots] \mid a, b, \ldots \in W_{\mathcal{T}}^{p}\}.$$

In order to represent the meta language we hinted at in the previous section, we need a larger type system. Let's extend $\mathcal{T}$ with $\mathcal{M}$, the *meta type*. $\mathcal{M}$ could contain values like symbols, numbers, and even programs from $W_{\mathcal{T}}$. Endomaps of this extended space are collected in

$$W_{\mathcal{M}} : \mathcal{T} \times \mathcal{M} \to \mathcal{T} \times \mathcal{M}.$$

Target words can be embedded in this space of functions, by means of an *embedding* map

$$e : W_{\mathcal{T}} \to W_{\mathcal{M}},$$

which maps real programs to meta–programs. Let's say a program $p_m \in W_{\mathcal{M}}$ is *compilable* if there exists an element $p_t \in W_{\mathcal{T}}$ such that $p_m = e(p_t)$. Note that compilability is a semantic property.

2.6 Syntax

In order to perform compilation, it is necessary to work on a syntactic representation of programs. Define $X_{\mathcal{T}}$ and $X_{\mathcal{M}}$ as the set of *program strings*, respectively associated to the sets of functions $W_{\mathcal{T}}$ and $W_{\mathcal{M}}$.

Each $s_t \in X_{\mathcal{T}}$ can be associated to a $w_t \in W_{\mathcal{T}}$, and each $s_m \in X_{\mathcal{M}}$ can be associated to a $w_m \in W_{\mathcal{M}}$. These maps

$$i_{\mathcal{T}} : X_{\mathcal{T}} \to W_{\mathcal{T}} \text{ and } i_{\mathcal{M}} : X_{\mathcal{M}} \to W_{\mathcal{M}}$$

are called *interpretation* maps and relate a program text to its semantics in the appropriate space of functions. These maps are *onto* and thus define equivalence relations, creating equivalence classes of program strings that map to the same semantics.

The existence of the map i allows us to define the embedding maps for the syntactic domain as any

$$e_s : X_{\mathcal{T}} \to X_{\mathcal{M}},$$

for which $i_{\mathcal{M}} \circ e_s = e \circ i_{\mathcal{T}}$, illustrated by the diagram

$$\begin{array}{ccc} X_{\mathcal{T}} & \xrightarrow{i_{\mathcal{T}}} & W_{\mathcal{T}} \\ \downarrow{\scriptstyle e_s} & & \downarrow{\scriptstyle e} \\ X_{\mathcal{M}} & \xrightarrow{i_{\mathcal{M}}} & W_{\mathcal{M}} \end{array}$$

Note that e_s is not unique, and one typically chooses this to be the *rpn assembler*.

Let's illustrate this embedding with an example for Purrr18. Program texts in $X_{\mathcal{T}}$ are strings of machine instructions, where each instruction is represented by an expression in parenthesis. Each such expression has a meaning in $W_{\mathcal{T}}$ The meaning is attributed by the map $i_{\mathcal{T}}$. The equality

$$e_s(\texttt{[(movlw 123)]}) = \texttt{[123 movlw]}$$

associates the syntactic representation of the machine instruction that loads the number 123 in the working register with the rpn assembler code for Purrr18.

We say a program text is compilable if its semantics are compilable. This property is transferred from the semantic to the syntactic domain by the interpretation map $i_{\mathcal{M}}$. This gives an equivalence class $\{X^{\mathrm{nc}}_{\mathcal{M}}, X^{\mathrm{c}}_{\mathcal{M}}\}$ of uncompilable and compilable program texts.

The act of *compilation* is defined for compilable programs as a map

$$c : X^{\mathrm{c}}_{\mathcal{M}} \to X_{\mathcal{T}}.$$

The act of *optimization* is defined as the identification of a particular $s_t \in X_{\mathcal{T}}$ from the equivalent representations of w_t, which is optimal according to some measure on $X_{\mathcal{T}}$.

2.7 Idioms

Metaprogramming becomes interesting when we can use non–compilable functions $f, g, \ldots \in W_{\mathcal{M}}$ that lead to a compilable composition $[fg\ldots]$.

This way some *specific* target behaviour can be *factored* into a composition of words that are not representable on the target side, but nevertheless serve as an abstraction mechanism. We can define $f, g, \ldots$ independently in source code, but use them together so they can be represented on the target after composing the functions. We call these compositions *idioms*.

An example to make this a bit more concrete. Suppose $\mathcal{M}$ contains a subset of symbols. That way some word f could produce a symbol, and some word g could consume it consume it, transforming it into a target program. Due to the production/consumption of this symbol, the words f, g can not be compilable, but the composition $[fg]$ can be. So during the optimization phase we could replace the words f, g with a compilable word $h = [fg]$, partially evaluating the composition to a single function. One could call these words "quarks".

For example. thake the code `['foo @]`. Neither of both words are compilable, but they compose into a word that is, namely one that generates the instruction to fetch a value from some numerically addressed data location, say 123, and put it on the stack. For the Purrr18 language this is the target machine program `[(save) (movf 123)]`. Note that the machine instruction `(movf 123)` counts as a single word: machine instructions are denoted by using original syntax, delimited by parentheses, while composition is still denoted by square brackets.

Classic immediate Forth words fall into this class also. An example are the words `for` and `next`. Neither of them is compilable, but a composition that

starts with `for` and ends with `next`, and constists of a compilable composition in the middle, is itself compilable.

2.8 Optimizations

If there is a subcomposition that can be represented more efficiently as a *specialized* function, it is always possible to take out the original composition and replace it with this function. Note that it is alwasy *possible* to do this, but might only be *desirable* for a small subset of compositions in a program.

Usually this works well when the composition involves *static data*, which is data known at compile time. For example in the program `[foo 1 + bar]`, the composition `[1 +]` could be implemented as a single operation `increment`, leading to the resulting code `[foo increment bar]`.

3 Implementation

The implementation of the partial evaluation scheme mentioned in the previous section is done using *macros*, which operate on syntactic representation of the semantics explained above.

Define $\mathcal{C}$ as the set of compilation states. In practice, $\mathcal{C}$ contains stacks of (scheme) values. A macro is then defined as $c \in W_{\mathcal{C}}$, which contains functions on $\mathcal{C} \times X_{\mathcal{I}}^*$.

The set $X_{\mathcal{I}}^*$ can be seen as an intermediate representation which is slightly bigger than $X_{\mathcal{I}}$, in that it can represent some elements from $X_{\mathcal{M}}$, including symbols and pseudo operations, but is not large enough to represent $X_{\mathcal{M}}$ entirely.

A macro creates program strings. The *compiler* of a program string $s \in X_{\mathcal{M}}$ is a composition of macros directly derived from the program source text by associating each word in the program source to a corresponding macro that will generate a target program text fragment and/or change the compilation state.

The target program text is obtained by applying the compiler to an empty compilation state and an empty program string.

- REPRESENT. Forth is translated to a compositional representation in $X_{\mathcal{M}}$. This representation is given a *postponed* semantics, $p : X_{\mathcal{M}} \rightarrow$

$W_{\mathcal{C}}$, which associates a compiler to the program text. This representation is obtained by relating each syntactic element s in the program text to a macro $m \in W_{\mathcal{C}}$ which will compile its semantics.

- COMPILE. The resulting compiler is evaluated to produce a program text. Each macro macro is applied to the propagated stated from left to right. Each macro has some intelligence about how to combine the accumulated target program text $\in X_{\mathcal{T}}^*$ with compilation state data $\in \mathcal{C}$ generated during its compilation, into new code and change of compilation state. Each macro *eagerly* employs partial evaluation of the generated code, using mostly local information.

- ASSEMBLE. When the program text has a correct semantics of a compilable program, invocation of the compiler results in a program text in $X_{\mathcal{T}}^*$ that can be represented in $X_{\mathcal{T}}$ without loss of information, by replacing remaining symbolic names by numbers.

The REPRESENT stage actually contains a number two separate stream processors: the first one is bound to the filesystem, and implements the `load` word, which inlines forth files, and the `load-ss` which inlines Scheme files. The second stage (independent of the filesystem) converts all non–compositional Forth constructs to compositional ones, and separates macro definitions from macro instantiations. This is called the *parser* stage.

4 Thoughts on Stack Languages

In the previous section we talked about how compilation, optimization and metaprogramming can be implemented as syntactic operations based on the associativity property of function composition.

In and of itself a compositional language can hardly be called a progamming language. It is more of a *machine model* applicable to many languages that can be modeled as a function composition, i.e. most real machine languages without *branching*.

In this section we discuss some of the properties of two classes of specialized compositional languages: the compositional stack languages (CSLs), and the compositional tuple languages (CTLs).

4.1 CTL → pure CSL

A compositional *tuple* language (CTL) is a category F where the objects are sets of tuples T_i, and the arrows are functions $f_{ij} : T_i \to T_j$ bringing an i–tuple to a j–tuple. A compositional stack language (CSL) is a category W with a single object S, a set of stacks, where the arrows are functions (words) $w : S \to S$. This makes a CSL a monoid. We denote a stack with angle brackets $\langle \ldots, a_2, a_1 \rangle$, where a_1 is the top element.

We call a CSL *pure* if it has a *parent* CTL which is taken to the CSL by means of the functor *snarf*

$$\sigma : F \to W.$$

This functor maps each T_i to S by associating to each i–tuple $t_i = (a_i, \ldots, a_1)$ a stack $\langle a_i, \ldots, a_1 \rangle = \sigma(t_i)$, and to each morphism f_{ij} a morphism $w : S \to S$ which can be composed of the operations

- Split $S \to S \times T_i$, mapping a stack $s \in S$ into a pair of a stack and a tuple (s_b, t_i), where $t_i \in T_i$ is the i–tuple containing the top i elements of s, and s_b the bottom, which contains the rest of the stack.
- Evaluate $S \times T_i \to S \times T_j$, which maps $(s_b, t_j) = (s_b, f_{ij}(t_i))$,
- Join $S \times T_j \to S$, which maps the tuple (s_b, t_j) back into a stack by pushing all elements of the j–tuple t_j onto s_b.

This is a functor because it preserves the identity morphisms, and composition of morphisms

- $\sigma(1_{T_i}) = 1_S$
- $\sigma(f_{jk} \circ f_{ij}) = \sigma(f_{jk}) \circ \sigma(f_{ij})$

If a CSL is pure all operations in W are *local* to the top of the stack. For each $w : S \to S$ there are numbers i, j such that

$$w(\langle \ldots, a_{i+1}, a_i, \ldots, a_1 \rangle) = \langle \ldots, a_{i+1}, b_j, \ldots, b_1 \rangle,$$

which allows us to associate it to an $f_{ij} : T_i \to T_j$. Note that this cannot be expressed as a functor $W \to F$ because we cannot map the set of stacks S to the sets of i–tuples T_i. The conversion from CTL to CSL looses information about the arity of the functions.

In Brood, this functor is seen in action at the point where Kat is derived from Scheme, and where (a subset of) Purrr18 is derived from a functional language implementing 8 bit arithmetic.

4.2 Why stacks?

In essence, the data stack is part of an explicit representation of a *continuation* of the application of a function $t_j = f_{ij}(t_i)$ to a tuple $t_i \in T_i$, as part of a larger computation $c \in T_p \to T_q$.

The continuation is a function $k \in T_j \to T_q$ which is represented in $S \to S$ by a pair of functions d, r. The function d is a multivalued constant, mapping the empty stack to the part of the stack that's left invariant by the function f_{ij} under consideration. The function r is usually represented as a list (stack) of function compositions.

The function k takes a tuple from T_j, which is the result of applying f_{ij} to an element of T_i. The codomain of k is the same as the codomain of the computation c of which k is part.

In the representation set $S \to S$, it is easy to see d as a representation of the *past* of a computation and r as the representation of the future, by expression a computation as

$$c = [dt_i f_{ij} r].$$

Here t_i produces the i constant data items used by f_{ij} which is here directly represented as a stack word. We can call $t_j = [t_i f_{ij}]$, the evaluation of f_{ij} independent of its continuation represented by the past d and the future r.

It is clear that a CSL language can be used to *implement* a CTL language. Now, why would we be using this low–level representation to write programs? The main reason is to be found in *steering words*. In a CSL we can factor a program into more general terms than a CTL. For example, the operation `dup` needs to be defined only once, and can be applied regardless of stack size. In a CTL we would need a `dup` for each T_i with $i > 0$.

So in short, it's more convenient to write programs in a CSL, but for thinking about the language, it can be more convenient to switch to the parent CTL. Both give us *point free* notation.

What I am still looking for, is a way to associate *arrows* as they are used in Haskell, to stack languages.

4.3 Closures

Stack languages make it easier to distinguish between functions that are *closures* created at run–time, and those that depend only on immediately supplied arguments.

Because the absence of variable names, dynamic closures need to be created using an explicit dynamic *curry* operation c. The advantage of this is that a CSL language which supports first class functions but no first class closures, for example because it lacks dynamic memory management, can be easily extended by adding the combinator c.

To see this, let's compare the lambda terms $M = \lambda x.x + 1$ and $N = \lambda a \lambda x.x + a$. The former can be derived from the latter as $M = (N1)$.

To represent M as a stack word $\in S \rightarrow S$, we could write it as the composition [1+], where 1 and + are the snarfed constant function from $T_0 \rightarrow T_1$ and the binary operator from $T_2 \rightarrow T_1$ respectively. This composition is *static*, meaning it can be completed at compile time.

To represent N we need some kind of combinator trick. For example $[(+)c]$, where $(+)$ denotes the function snarfed from $T_0 \rightarrow T_1$ that pushes the function + to the top of the stack, and c denotes the curry operation, which combines a value with a composition, by turning the value into a function that generates the value, and composing that function with +. This operation is *dynamic* in that it creates a function value at run–time.

4.4 Linear memory management

Stack languages make it feasible to eliminate the need for dynamic garbage collection, by using linear memory management. This is a consequence of the presence of explicit `dup` and `drop` operators for accessing values. Compare this to β–reduction for lambda expressions, which doesn't account for the number of times a bound variable is referenced.

With linear memory management we mean that the entire data collection is a (binary) tree. Each `cons` cell in this tree is referenced only once, and all operations on the tree preserve its topology. This can be implemented efficiently using *hash consing*. See [1].

The key observation in a meta–programming setting is that linear data structures are allowed to refer to non–linear ones, as long as the non–linear collector can traverse the linear data tree to identify reachable data. Non–linear structures are *not* allowed to refer to linear structures directly.

Compiled code could be part of the non–linear part of the system, managed by *meta–collection*. Closures and other lists can be dynamic, but linear.

An advantage of this is that a linear core system can run without the need for memory management by garbage collection, making it suitable for hard real–time tasks without requiring a real–time garbage collector. This pattern

of splitting a program into a static kernel and a dynamic system interacting with it in a controlled way, i.e. using message passing, can be found in quite a lot of systems. However, compositional languages make this idea practical without giving up too much expressive power.

This is one of the core ideas behind brood as a metaprogramming system, which is related to *Packet Forth*. At the time of this writing, PF is still a separate project. Brood does contain *poke*, an attempt to build a linear machine based on restructuring of binary trees.

4.5 Modules

Names are an important programming tool. Compositional stack languages in their basic form use a flat global name space. This can get messy. For practical programming, some form of name space control is necessary. I suggest something like MzScheme's module system.

5 CAT

CAT, the language used to implement the compositional macros discussed above is a compositional language implemented as MzScheme syntax extensions. To reduce confusion with the term *macro* used in this paper, I will refer to a scheme macro as an *extension*.

5.1 Base CAT

Base CAT is not much more than a mapping from scheme to a compositional syntax. In scheme, this is implemented by the `base:` extension. This creates functions $s \to s$, where s is the set of stacks of values. A stack is represented by a CDR–linked list, used as the argument list for scheme procedures. Here are some examples related to their defining scheme expression:

```
(base:)      ≡  (lambda s s)
(base: 123)  ≡  (lambda s (cons 123 s))
(base: +)    ≡  (lambda (b a . s) (cons (+ a b) s))
```

Symbols can be bound by the lexical environment. This means the extension can be used to create closures. The code

```
(let ((plus
       (lambda (b a . s)
         (cons (+ a b) s))))
  (base: 1 plus))
```

will bind the name `plus` and interprets it as a function. The result is a function which adds 1 to the top of the stack. This also works for quoted symbols, which are treated as constants. The code

```
(let ((abc 123))
  (base: 'abc))
```

will create a function that loads the number 123 on the stack. This might seem confusing at first, but it is very similar to the way `syntax-case` deals with pattern variables. Inside CAT, a quote means the quoted atom is treated as literal data. With Scheme metaprogramming CAT, quoted symbols can be interpolated.

5.2 Extensions

The CAT language has a fixed syntax with pluggable semantics. The following syntacting elements are fixed, but their interpretation can be varied per language. The defaults are

- A non–quoted *symbol* always refers to a function.
  - If a binding is present in the surrounding *lexical* environment, its value will be used as function.
  - If the module namespace in which the expression occurs contains a name prefixed with `rpn.`, its value will be used as function.
  - If the name occurs free, it will be taken from the global name space `(base)`.
- A *list* is treated as a function which loads an *abstraction* on the stack. The abstraction is a function built from the composition of the functions in the list.
- A *quoted* expression is interpreted as a constant function. If the expression contains a *symbol* which has a lexical binding, it is substituted by the associated value.

- A *quasiquoted* expression is interpreted as a constant function. All *unquoted* lists are replaced by their function values.
- Anything other than a symbol or a list denotes a *constant* function.

5.3 State

The Base CAT is not purely functional; some small set of operations including have side effects, including IO. Since CAT is strict, this doesn't give problems. However, there is no doubt that a purely functional *programming style* has certain benefits.

Therefore, CAT has an extension that allows hidden *threading* of state through a composition. Because of the use of a stack one does not need to resort to *monads* to perform this threading. The reason is that a stack language is a specific instance of a monad language.

Threading is performed by storing the state on top of the stack. The associated syntax transformation will automaticly *dip* pure functions so they ignore the state atom, except for passing it on the next function in a composition.

For example the `state:` macro will compile the state access words `state@` and `state!` to operate on the state, but will transform other words so they just pass it on. The code

```
(state: 1 state@ + state!)
```

implements a counter. This code is equivalent to

```
(base: (1) dip  ;; 1
       dup      ;; state@
       (+) dip  ;; +
       drop)    ;; state!
```

after syntax expansion. Note how state fetch and store map directly to the language's duplication and deletion operators!

It would be interesting to explore the relationship if this fairly simple but useful syntactic extension, to the more general picture of monads and arrows [Hughes].

6 Purrr State Management

There are two kinds of state to be managed in the Brood system: *target* state and *host* state. Target state consists of binary code stored on a microcontroller's flash ROM and symbolic metadata contained in a `.state` file. Host state is the *functional* code that implements the compiler.

Host state is *transparent*, which means it can be put in one–to–one relation with source code. This code is either internal Brood source code, or language extension macros coming from a Purrr project. The host state is thefore just a *cache* and can be automaticly managed and ignored by the Purrr programmer.

Target state is not transparent. It is created as a consequence of compilation: Purrr code is translated into binary target code, some metadata to allow symbolic access of that code (dictionary), and the source code for the macros defined in the project code used to compile other parts in the project code. Once code is compiled, it looses all relationship with its original source code, and becomes a separate entity.

In other words, Purr is an incremental *image based* development system. It also used *early binding*. This is quite a low–level approach, and gives up *safety* for *concreteness*. We give up safe (transparent, automatic) synchronization between source code and target state for a more direct access all the way down to the real machine code.

The the core of Brood itself is not developped as a stateful, image–based system. Instead it is written incrementally in a transparent declarative bottom–up fashon, using PLT Scheme's module system. Such an approach makes more sense for purely functional programs that contain no cyclic dependencies. Brood is almost entirely functional: only I/O and compiler compilation cache are managed in an imperative way.

For Purrr, there are several practical reasons for using an image based approach. Implementing an incremental transparent system for target code is simply more difficult. It requires a module system to manage dependencies. In order to reach a small incremental development cycle, one also has to cut down on upload times. This requires some kind of filesystem on the target flash ROM, and a way to perform linking, which is complicated due to the write–once nature of the code memory, and makes some optimizations impossible. I believe the safety this gains is not worth the cost of complexity and loss of performance.

Instead, a purely bottom up incremental style is a very simple alternative

which requires no filesystem, and no re-linking, so we can resort to a simpler and more efficient early binding strategy. While it makes cyclic dependencies harder to solve, it has the very interesting property that adding new code cannot break stored old code. It can only break run–time data structures.

A disadvantage is that changes to the bottom of a code stack do not automaticly propagate to code written on top of this. Managing these kind of changes is the responsability of the programmer: from time to time a complete reload is necessary. This can be annoying sometimes, but can mostly be solved by adding some (ad–hoc) form of late binding. In Purrr the easiest way to do this is to use `route` or `execute`.

This paradigm is key to interactive Forth development. Variants can also be found in image–based Lisp and Smalltalk. However, the latter two use *late binding*, which makes image based development more reflective: barring arbitrary limitations due to optimization, essentially everything can be changed.

So, the moral of the story: as a programmer interface, we stick with the Forth paradigm. For the task at hand, it seems to be the better aproach.

7 Goals

The front–end for Brood is currently the Purrr interactive compiler used for CATkit and Sheep. The most immediate goal is to make that system usable for real–world development, meaning people other than me that can't cross the Purrr/Brood barrier.

- Move Purrr to purely functional macros.
- Add a module system for namespace management.

8 Remarks

I wrote a *pattern matching language* for writing a Forth peephole optimizing code generator. This is the main abstraction that is used to implement the partial evaluation step as a set of greedy recombination rules. This pattern matching language alludes to the fact that the intermediate target code representation is really *statically typed*, where each assembler instruction represents a different type.

One of the great benefits of working with Forth is *incremental development*, which means the ability to add new code to a running system. The standard Forth approach is to make a system self–hosting, by including a compiler in a running system. I've opted to not do that for a very compelling and obvious reason. Because of the small size of the systems we intend to write code for, making it self–hosting is a waste of available technology: it's much more comfortable to move the compiler and interaction system to a more complex development host. This enables the use of a more resource hungry high level language and better abstraction. It took me a while to realize this, coming from a fairly lowlevel C and Forth world. In short, all the machinery which would normally run on a target system is presented in such a way as if it does, but in fact, it runs on a different system. This gives some consistency to the user interface.

CATkit is a small board with knobs, to run the Sheep, a 1-bit noise maker implemented in Purrr18.

Poke is a compositional language VM based on linear memory management, combined with *external* non–linear memory management. What this means is that the reflection system has access to the code the core VM is running, but not to its data structures.