Chemical Compiler

Phil Senum

August 25, 2011

1 Introduction

In our previous work, we developed chemical modules to perform specific algebraic and computational tasks. Specifically, we have shown ways to operate on variables represented in a chemical system in unary, including operations such as copying and decrementing. We gave a brief overview of a method by which these basic modules could be combined to perform more complicated operations. In this paper, we describe a process by which these modules may be systematically connected into sequential chains of operations to perform sophisticated arithmetic in a chemical target.

2 System Organization

Traditional compliers operate in the following way:

1. Input, in the form of a human-readable behavioral description, is fed to the compiler.
2. The compiler transforms the input into computer-readable tokens.
3. Each token is combined with surrounding tokens to form complete statements.
4. Each statement is broken down to machine code, and optimizations are added.
5. The resulting machine code is optimized further, if desired.

Our compiler uses the same first three steps in its compilation. The last two are revised. Specifically, instead of breaking down statements into machine code, statements are broken down into sets of chemical reactions. These chemical reaction sets, which we call chemical "modules", are designed to perform specific algebraic tasks such as addition and subtraction of integer quantities.

Our compiler operates in the following way:

1. Input language, a subset of the C language, is read by the compiler.

2. The input file is split into tokens; groups of tokens are arranged into a list of statements.
3. Each statement in the list is split into one or more fundamental statements.
4. The list of statements is made into a directed graph.
5. Each node in the graph is translated into a set of chemical reactions by picking the appropriate template.

3 Chemical Modules

The design of the chemical modules as described in previous works performed adequately for the simple operations which we were modeling. However, the design did not scale well to allow for sequential execution of multiple modules. Although certain sophisticated operations such as raise-to-power were implementable, there was a large amount of complexity added to the system that was not fully understood. Here, we describe a slightly modified set of modules that perform the same operations, with the advantage of a scalable design.

3.1 Previous Model

In the previous model, several assumptions are made about the input species and start signal.

1. Nothing else needs to operate on the input species, x, while the module is operating.
2. The start signal, g, is supplied from a small, fast injection by an external source.

3.2 Available Operations

3.2.1 Single-Molecule Source Modules

The first set of reactions perform operations based on single molecules of a source.

3.2.1.1 Copy

$$a + g + d_{\text{ab}} \xrightarrow{\text{slow}} a' + g \tag{1}$$
$$g + a_{\text{ab}} \xrightarrow{\text{slow}} d \tag{2}$$
$$d + d \xrightarrow{\text{fast}} d \tag{3}$$
$$d + a' \xrightarrow{\text{slow}} d + a + b \tag{4}$$
$$d + g \xrightarrow{\text{fast}} d \tag{5}$$

The new model works as follows.

1. Assuming the operation has not already been completed, indicated by the absence of the done signal, d_{ab}, the input species, x, is transferred to a temporary type, x', in the presence of the start signal, g. This is handled by reaction 1.

2. Once all of the input species, x, has been transferred to its temporary type, the done signal, d, is created to indicate the operation has completed. This is handled by reaction 2.

3. The done signal, d, is maintained at a small quantity. This is handled by reaction 3.

4. In the presence of the done signal, d, the temporary type, x', is transferred back to the input species, x, with a copy being put in the output species, y. This is handled by reaction 4.

5. The start signal, g, is kept out of the system by the presence of the done signal, g. This is handled by reaction 5. Under perfect circumstances, this is not a necessary step. However, due to the nature of the creation of absence indicators (always created; destroyed in the presence of the corresponding type), it was observed that reaction 1 would occasionally fire in the presence of the done signal, d. By keeping the start signal, g, removed from the system, this is less likely to happen.

Usage of the module is almost the same as the previous model.

1. If present, flush the done signal, d, from the system.

2. Place the species to copy in the input species, x.

3. Make an injection of the start signal, g.

Several assumptions are made about the input species and start signal.

1. Nothing else needs to operate on the input species, x, while the module is operating.

2. The start signal, g, is continuously supplied by some external source. The module will operate when the start signal, g, transitions from being absent from the system to being present in the system.

3.2.1.2 Clear Same as Copy except for reaction 4:

$$d + a' \xrightarrow{\text{slow}} d \tag{6}$$

3.2.1.3 Double Same as Copy except for reaction 4:

$$d + a' \xrightarrow{\text{slow}} d + a + a \tag{7}$$

3.2.2 Double-Molecule Source Modules

These modules perform operations based on pairs of molecules of a source.

3.2.2.1 Decrement

$$a + g + d_{\text{ab}} \xrightarrow{\text{slow}} a' + g \tag{8}$$
$$g + a_{\text{ab}} \xrightarrow{\text{slow}} d \tag{9}$$
$$d + a' + a' \xrightarrow{\text{fast}} d + a + a' + a^{\text{rx}} \tag{10}$$
$$a^{\text{rx}} \xrightarrow{\text{slow}} \varnothing \tag{11}$$
$$d + a^{\text{rx}}_{\text{ab}} + a' \xrightarrow{\text{slow}} d \tag{12}$$
$$d + d \xrightarrow{\text{fast}} d \tag{13}$$
$$d + g \xrightarrow{\text{fast}} d \tag{14}$$

3.2.2.2 Increment Same as Decrement except for reaction 12:

$$d + a^{\text{rx}}_{\text{ab}} + a' \xrightarrow{\text{slow}} d + a + a \tag{15}$$

3.2.2.3 Halve Same as Decrement except for reaction 10:

$$d + a' + a' \xrightarrow{\text{fast}} d + a + a^{\text{rx}} \tag{16}$$

3.2.3 Comparison Modules

Note that each comparison module does not specifically implement a "while" or "if" loop, it simply performs the comparison and branches either into the loop or past the loop. The difference between the two loops is handled by program flow control; in the case of a "while" loop, the last statement in the loop branches back to the head of the loop, whereas with an "if" loop, the last statement proceeds past the end of the loop.

3.2.3.1 Greater than Zero

$$g + a \xrightarrow{\text{slow}} g_{\text{next}} + a \tag{17}$$
$$g + a_{\text{ab}} \xrightarrow{\text{slow}} g_{\text{skip}} \tag{18}$$
$$g_{\text{next}} + g \xrightarrow{\text{fast}} g_{\text{next}} \tag{19}$$
$$g_{\text{next}} + d_{\text{last}} \xrightarrow{\text{fast}} g_{\text{next}} \tag{20}$$

3.2.3.2 Equal to Zero

$$g + a_{\text{ab}} \xrightarrow{\text{slow}} g_{\text{next}} \tag{21}$$

$$g + a \xrightarrow{\text{slow}} g_{\text{skip}} + a \tag{22}$$

$$g_{\text{next}} + g \xrightarrow{\text{fast}} g_{\text{next}} \tag{23}$$

$$g_{\text{next}} + d_{\text{last}} \xrightarrow{\text{fast}} g_{\text{next}} \tag{24}$$

3.2.3.3 Arbitrary Comparison Arbitrary comparison is achieved via a similar process to the comparison operations described in our previous work. Each comparand (each side of the comparison) is first copied to a temporary type, as arbitrary comparison is a destructive operation.

3.3 Sequential Firing of Modules

Each module has a number of signals that can be used to track the progress of the operation. These include:

- the start signal, g, which enables the operation;
- the done signal, d, which prevents multiple executions of the same operation; and
- a temporary type, usually x', which is present whenever the operation is currently executing.

We can design chemical reactions that use the presence and absence of these signals to determine the current state of each operation. With this in mind, we design a new module which enables the next statement in the program after the current statement has completed. Reactions 25–28 implement this module.

$$a'_{\text{ab}} + d \xrightarrow{\text{slow}} g'_{\text{next}} + d \tag{25}$$

$$g'_{\text{next}} + a' \xrightarrow{\text{fast}} a' \tag{26}$$

$$g'_{\text{next}} \xrightarrow{\text{slow}} g_{\text{next}} \tag{27}$$

$$g_{\text{next}} + d \xrightarrow{\text{fast}} g_{\text{next}} \tag{28}$$

1. We know that the current operation has completed when the done signal, d_n, is present and the temporary type, x', is absent. We always preserve the population of non-absence-indicator types by allowing them to appear as both a reactant and a product in any reaction in which they are used. Based on this condition, we generate the pre-reactant for the next start signal, g'_{n+1}. This is all accomplished by reaction 25.

2. In the case that we errorenously generated the next start signal, we can catch it before it becomes the next start signal. (Simulation results show that absence indicators will occasionally take part in a reaction when the type they are indicating is actually present.) This is accomplished by reaction 26.

3. We allow the pre-reactant for the next start signal to become the next start signal. This is accomplished by reaction 27.

4. We need to reset the module back to its initial state. This is achieved by removing the done signal, d_n, from the system. We know it is safe to do so when the next start signal has been created. This is accomplished by reaction 28.

In this way, each module is enabled by its g signal, signals its completion with the d signal, and automatically enables the next module in the sequence by asserting the next g signal. In this way, sequential execution of commands is achieved.

4 Code Parsing

4.1 Language Structure and Tokens

We adopt a subset of the C language for our own use. The following operations are supported:

- Assignment to variables
  - Can source from other variables or constants
  - Supports addition and subtraction of multiple variables and constants
- Comparison between multiple variables or variables and constants
  - Includes support for "if" and "while" loop types
  - Supports all six major comparisons: $>$, $\geq$, $<$, $\leq$, $\neq$, and $=$.

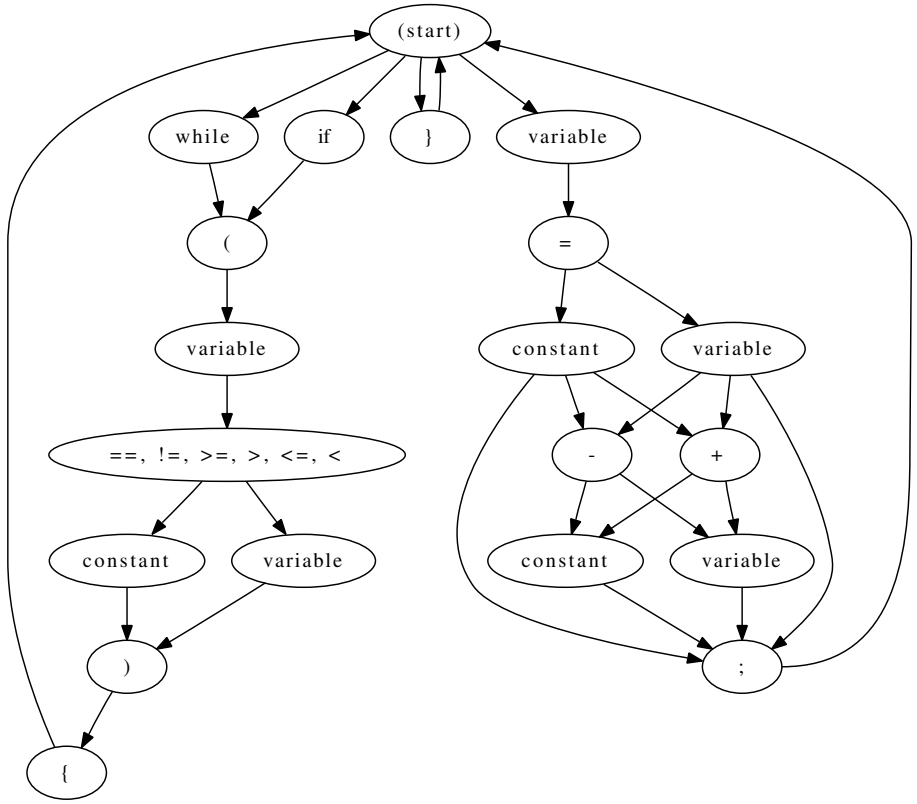

Figure 1: The tokens and flow of our language.

4.2 Fundamental Statements

Although our language allows for a wide array of statements, only a small number of these are directly implementable with chemical modules. All other statements must be modified or broken down into one or more of these fundamental statements.

- `x = 0;` (Clear)
- `x = x + y;` (Copy)
- `x = x - y;` (Copy)
- `x = x + 1;` (Increment)
- `x = x - 1;` (Decrement)
- `x = x + x;` (Double)

4.3 Mapping of All Statements to Fundamental Statements

A total of 59 unique statements are recognized by the compiler. These are shown in Table 2. Each one of these statements can be parsed in one of 13 ways into one or more fundamental statements, as shown in table 1.

4.4 If and While loop handling

"if" and "while" loops are implemented via our greater than/equal to zero modules (where possible) or comparison module. Program flow control determine which type of loop is used.

5 Mapping Chemical Modules to Parsed Code

After the file has been parsed and all statements have been correctly re-written, the compiler then links the statements. Each statement is marked with up to three numbers: its own index, the index of the proceeding statement (and the index of the statement to skip to in the case of comparison statements, when a condition is not satisfied).

After this is complete, the compiler simply steps through the statements line-by-line, creating the set of chemical reactions for each statement based on the templates described earlier. Each statement gets a computational block of reactions and a block of control reactions. The computational block implement the computation itself; the control reactions ensure that the statements are executed in the correct order.

This is the advantage of our approach: each chemical module is self-contained. To execute a "line of code," the only thing that needs to be done is to assert

Table 1: Statement parsing

1	2	3	4	5
`x = 0;`	`x = x + y;`	`x = y;`	`x = y + z;`	`x = x + x;`

`x = 0;`	`x = x + y;`	`x = 0;`
		`x = x + y;`
			`x = 0;`
			`x = x + y;`
			`x = x + z;`
				`x = x + x;`

6	7	8	9	10
`x = y + y;`	`x = x + 1;`	`x = y + 1;`	`x = x - 1;`	`x = y - 1;`

`x = 0;`
`x = x + y;`
`x = x + x;`
	`x = x + 1;`	`x = 0;`
		`x = x + y;`
		`x = x + 1;`
			`x = x - 1;`	`x = 0;`
				`x = x + y;`
				`x = x - 1;`

11	12	13		
`x = x - y;`	`x = y - x;`	`x = y - z;`		

`x = x - y;`	`_temp1 = 0;`
	`_temp1 = _temp1 + x;`
	`x = 0;`
	`x = x + y;`
	`x = x - _temp1;`
		`x = 0;`
		`x = x + y;`
		`x = x - z;`		

Table 2: All statements recognized by the compiler.

`x = 0;`	1	`x = c;`	3	`x = 1 - 0;`	3	`x = c - x;`	12
`x = x - x;`	1	`x = 1;`	3	`x = y + z;`	4	`x = 1 - x;`	12
`x = y - y;`	1	`x = c + 0;`	3	`x = y + c;`	4	`x = y - z;`	13
`x = c - c;`	1	`x = 0 + c;`	3	`x = c + y;`	4	`x = c - y;`	13
`x = 0 - 0;`	1	`x = c + c;`	3	`x = x + x;`	5	`x = 1 - y;`	13
`x = 1 - 1;`	1	`x = c + k;`	3	`x = y + y;`	6	`x = y - c;`	13
`x = 0 + 0;`	1	`x = c + 1;`	3	`x = x + 1;`	7	`x = x;`	Warn
`x = x + y;`	2	`x = 1 + c;`	3	`x = 1 + x;`	7	`x = x + 0;`	Warn
`x = y + x;`	2	`x = c - k;`	3	`x = y + 1;`	8	`x = x - 0;`	Warn
`x = x + c;`	2	`x = c - 0;`	3	`x = 1 + y;`	8	`x = 0 + x;`	Warn
`x = c + x;`	2	`x = c - 1;`	3	`x = x - 1;`	9	`x = 0 - x;`	Error
`x = y;`	3	`x = 1 - c;`	3	`x = y - 1;`	10	`x = 0 - y;`	Error
`x = y + 0;`	3	`x = 1 + 0;`	3	`x = x - y;`	11	`x = 0 - c;`	Error
`x = 0 + y;`	3	`x = 0 + 1;`	3	`x = x - c;`	11	`x = 0 - 1;`	Error
`x = y - 0;`	3	`x = 1 + 1;`	3	`x = y - x;`	12		

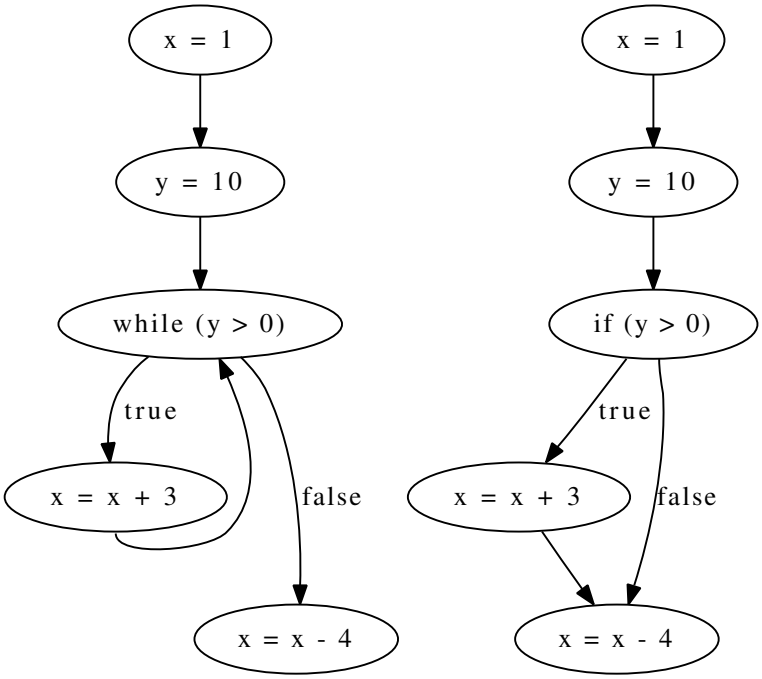

Figure 2: The difference between if and while loops.

the proper start signal, g_n, and wait for the operation to complete. With the added control blocks, the signals are asserted in the proper order automatically.

Constants are handled by creating another chemical type (basically, creating another variable) that holds the value of the constant.

Subtraction is handled through a second chemical type, x_s to each x:

$$x_s + x \xrightarrow{\text{fast}} \varnothing \tag{29}$$

For instance, if we want to decrease the value of x by 5, we would copy a constant of value 5 to x_s.

6 Example: Multiplcation

The following loop implements the operation `z = x * y;`.

```
x = 10;
y = 20;
z = 0;
while (x > 0)
{
        z = z + y;
        x = x - 1;
}
```

The code is compiled in the following way:

1. The code is read by the compiler, tokenized, and turned into a list of computer-readable statements. A directed graph, similar to the one shown in Figure 3.

2. Each statement in the list is broken down into fundamental statements. In this case, the first three statements, which simply initialize variables, can be removed; instead, the initial quantity of the corresponding species is set to these quantities. The remainder of the statements are already fundamental statements.

3. Each fundamental statement is assigned a chemical block type. A new directed graph, similar to the one shown in Figure 4 is formed.

4. Each chemical block is assigned an index. This index is used to create species g_n and d_n for each chemical block. Chemical reactions are written according to the templates listed above; the copy and decrement operations each get an execution block and a control block, whereas the compare block simply stands alone. In this case, fourty-four chemical reactions are written.

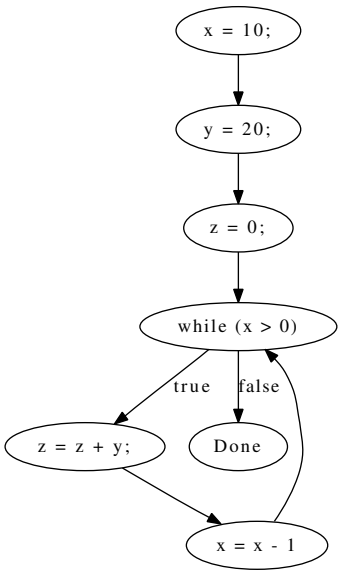

Figure 3: Multiply operation, pre-compilation.

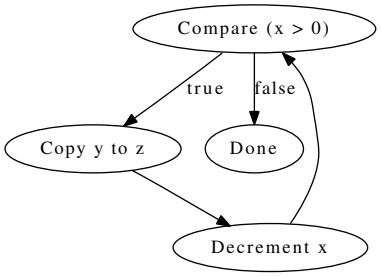

Figure 4: Multiply operation, post-compilation.

7 Simulation Results

7.1 Raise to Power

```
x = 5;
p = 3;
y = x;
d = 0;
p = p - 1;
while (p > 0) {
        w = x;
        while (w > 0) {
                d = d + y;
                w = w - 1;
        }
        y = d;
        d = 0;
        p = p - 1;
}
```

Input statements: 15
Output statements: 15
Reactions: 180

7.2 Square Root

The following code finds the floor of the square root of an input, `x`, by iteratively incrementing and squaring `y` until it finds a square of `y` that is larger than `x`.

```
x = 9;
y = 1;
ysq = 1;
while (ysq <= x) {
        y = y + 1;
        ysq = 0;
        n = y;
        while (n > 0) {
                ysq = ysq + y;
                n = n - 1;
        }
}
y = y - 1;
```

Input statements: 13
Output statements: 15
Reactions: 192

7.3 Sort

The following code implements a bubble-sort routine, sorting `a`, `b`, `c`, and `d`.

```
a = 10;
b = 12;
c = 9;
d = 25;
inOrder = 0;
while (inOrder == 0) {
        inOrder = 1;
        if (a > b) {
                t = a;
                a = b;
                b = t;
                inOrder = 0;
        }
        if (b > c) {
                t = b;
                b = c;
                c = t;
                inOrder = 0;
        }
        if (c > d) {
                t = c;
                c = d;
                d = t;
                inOrder = 0;
        }
}
```

Input statements: 26
Output statements: 43
Reactions: 504

8 Discussion

This research is intended as a proof-of-concept for the field of chemical compilation. We implement computation based on an imperative programming paradigm; other paradigms, such as functional programming, may serve as better frontends for chemical computation.