Description of the Expression Language

David Eigen

May 2003

1 Introduction

This document describes some of the more interesting issues in designing the language used to enter mathematical expressions in the Java applet program. My key aim in designing this language is to allow users to enter mathematical expressions as simply as possible, while still allowing a powerful programming language. The following examples illustrate the power and simplicity of the language by showing a mathematical formula, followed by how one would write it in the expression language:

<table>
<thead>
<tr>
<th>Mathematics</th>
<th>Expression Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c(t) = (t, t^2, t^3) )</td>
<td>( c(t) = (t, t^2, t^3) )</td>
</tr>
<tr>
<td>( T(X) = X' / |X'| )</td>
<td>( T(X) = X' / \text{length}(X') )</td>
</tr>
<tr>
<td>( f(u, v) = (u, v, u^2 - v^2) )</td>
<td>( f(u, v) = (u, v, u^2 - v^2) )</td>
</tr>
<tr>
<td>( N(X) = \text{unit}(X_1 \times X_2) )</td>
<td>( N(X) = \text{unit}(\text{cross}(X_{-1}, X_{-2})) )</td>
</tr>
<tr>
<td>( n(u, v) = N(f)(u, v) )</td>
<td>( n(u, v) = N(f)(u, v) )</td>
</tr>
<tr>
<td>( \text{Rot}(\theta) = \begin{bmatrix} \cos \theta &amp; -\sin \theta \ \sin \theta &amp; \cos \theta \end{bmatrix} )</td>
<td>( \text{Rot}(\theta) = \begin{bmatrix} \cos t &amp; -\sin t \ \sin t &amp; \cos t \end{bmatrix} )</td>
</tr>
</tbody>
</table>

The language features real, vector, matrix and function types; first-class functions; the ability to take derivatives analytically; and the ability to combine functions as one would write in mathematics, without the need for a wrapper lambda-form.
1.1 Notation

Throughout this discussion of the expression language, I will use the following notation. Text in\ntypewriter type\nshould be considered expressions, such as those the user might enter. $\mathbb{R}$ denotes the type for real numbers. A parenthesized list $(t_1, \ldots, t_n)$ denotes the vector type with component types $t_1, \ldots, t_n$. A list in square brackets, $[t_{11}, \ldots, t_{nm}]$ denotes the matrix type with $n$ rows and $m$ columns, with component types $t_{11}, \ldots, t_{nm}$. An arrow indicates a function type: $t_1, \ldots, t_n \rightarrow \tau$ denotes the function type whose argument types are $t_1, \ldots, t_n$ and whose return type is $\tau$. Finally, $e : t$ means that the expression $e$ has type $t$.

2 Building Expressions

Plaintext expressions are turned into executable objects as shown below. An executable object is a Java object that can be told to execute itself and return the resulting value.

\[
\text{tokenize} \quad \Rightarrow \quad \text{token tree} \quad \Rightarrow \quad \text{syntax tree} \quad \Rightarrow \quad \text{executable tree}
\]

The tokenize phase tokenizes the input, and reads it so that a tree is created according to how the token stream is parenthesized. Each node of the tree corresponds to a set of matching parentheses; the children of the node are created from the tokens within the parentheses. For example, the text $(x, \sin(2*x))$ would be tokenized into the tree

\[
( )
\]

\[
\begin{array}{c}
( ) \\
x \\
, \\
\sin \\
( ) \\
2 * x \\
\end{array}
\]

The parse phase parses a token tree into an abstract syntax tree. This is done in two steps: First, a built-in preprocessor transforms the token tree. Second is the actual parsing step, which creates an abstract syntax tree from the preprocessed token stream. The preprocessor is primarily used to group together some token strings with more parentheses (such as the first-class function construct,
func(<vars><expr>), which is transformed to (func(<vars><expr>)). This makes the parsing step easier, since token strings that should be treated as a group are in fact in their own node of the token tree.

The build phase compiles the syntax tree into an executable object tree. This allows type checking, derivative taking and function composition to be performed only once, before runtime. Right now, there is no static type-checking phase before the build phase. This actually proves to be a major problem for taking the derivatives of functions. In fact there is still an unresolved bug due to this, which I discuss in section 4.3.

There is overloading for built-in operators. For example, + can be used for scalar, vector or matrix addition. Since the types that an operator is applied to are known at build-time, the build phase type-checks the expression, and outputs the executable appropriate for the types. Thus, there are different executor objects for + in, for example, \( \mathbb{R} + \mathbb{R} \) and \((\mathbb{R},\mathbb{R}) + (\mathbb{R},\mathbb{R})\). Because of this, minimal (if any) type-checking needs to happen during runtime.

3 Function Composition

One of the key features unique to this language is that functions may be composed together without creating a wrapper lambda-form. For example, if \( f \) and \( g \) are both functions of type \( \mathbb{R},\mathbb{R} \rightarrow \mathbb{R} \), then the user may write \( f + g \) to mean the function of type \( \mathbb{R},\mathbb{R} \rightarrow \mathbb{R} \), where \((f+g)(u,v) = f(u,v) + g(u,v)\). There are two methods that I thought of for combining functions; I will call them function composition (a term taken from categorial grammar, a linguistic theory), and function building.

The method currently implemented is function building. However, it is quite possible that some variation of function composition is actually better, for reasons I will discuss later.

Function composition relies on having a static unification type-checker before the build phase. It uses the type judgment

\[
f : t_1, \ldots, t_n \rightarrow \tau \quad g_i : \alpha_1, \ldots, \alpha_k \rightarrow t_i \quad \text{or} \quad g_i : t_i \quad \forall i = 1, \ldots, n
\]

\[
f(g_1, \ldots, g_n) : \alpha_1, \ldots, \alpha_k \rightarrow \tau
\]

Here, \( f \) and the \( g_i \) are expressions of function type; of course, the application of \( f \) may have
a different syntax than the one shown (such as the infix syntax of $+)$. During the build phase, when a function application or operator syntax node is encountered that matches this type pattern, the resulting output is the function $\lambda(x_1, \ldots, x_k). f(h_1(x_1, \ldots, x_k), \ldots, h_n(x_1, \ldots, x_k))$, where

$$h_i(x_1, \ldots, x_k) = \begin{cases} g_i(x_1, \ldots, x_k) & \text{if } g_i \text{ is a function} \\ g_i & \text{if } g_i \text{ is not a function} \end{cases}.$$  

Although apparently simple, this approach has a major drawback. The type judgement, together with the standard function application type judgement, produces an ambiguity. Consider $h(f, g) = f(g)$. Then from the type judgments for application, $h$ can have type $(\alpha \to \beta), \alpha \to \beta$, or $(\alpha \to \beta), (\gamma \to \alpha) \to (\gamma \to \beta)$.

Given only this example, it might be the case these ambiguities are only temporary. That is, it is possible that although the types of functions can be ambiguous, the ambiguity will go away once they are applied enough. The following example, though, shows that the problem is much worse. Let $f(x) = \lambda x. \lambda y. x + 3$ and $g(x) = x$, and consider $f(g)(1)(2)$. This instance of $f$ can be typed in two ways: either $f$ takes $g$ directly, or $f$ takes $\mathbb{R}$ and is composed with $g$. Then $f : (\mathbb{R} \to \mathbb{R}) \to \alpha \to (\mathbb{R} \to \mathbb{R})$ or $f : \mathbb{R} \to \alpha \to \mathbb{R}$, with evaluations corresponding to $f(g) \Rightarrow \lambda y. x + 3$ in the first case, but $f(g) \Rightarrow \lambda x. \lambda y. x + 3$. Thus, there are two possible evaluations for $f(g)(1)(2)$: In the first case, $1$ is substituted for $y$ and $2$ is plugged into $g$. In the second case, the exact opposite happens. Thus, this expression can evaluate to either $2 + 3 = 5$ or $1 + 3 = 4$.

Function building is a more bottom-up approach. While building, functions are composed only when we reach a built-in operator, such as $+$. For example, let $f$, $g$ and $h$ be arbitrary user-defined functions, and consider building the expression $f(g, h)$. First, $(g, h)$ would be applied to $f$, and so we build the body of $f$. If we reach a built-in operator $\otimes$ that is applied to $g$ and $h$ and does not take function types, then we combine the functions: If $g$ and $h$ have the same number of arguments, then the result of $g \otimes h$ is the function $\lambda(x_1, \ldots, x_n). g(x_1, \ldots, x_n) \otimes h(x_1, \ldots, x_n)$. The argument and return types of this application of $g$ and $h$ are type-checked when the resulting function is applied. This process of combining functions propagates upward through the body of $f$, with a function combination potentially happening at each operator in $f$’s body. Note that the only type information we used was the number of arguments that $g$ and $h$ take. Thus, function building does not require a static type-checking phase before the build phase.

Note that there is no ambiguity in this method. In fact, in the ambiguous example from before, we...
necessarily get that $f(x) = \lambda y. g + 3$, and so $f(x)(1)(2) = 2 + 3 = 5$.

An unsatisfying property of function building is that it does not readily allow functions to be opaque, since the result of function building depends on the internal definition of $f$. A nicer solution might try to compose functions as early in the application process as possible. It’s not clear how this would work, though, and it is even possible it may confuse the user (since composition could happen at different times for different cases).

An advantage of function building is that it is not necessary to have a static type checker. Since the language has parametric polymorphism, a unification type checker would be necessary, and operator overloading would make it hard to implement. Furthermore, from the discussion above, it is not at all clear how to put function composition into such a type checker. This is the largest reason I decided to go with function building. However, as I discuss in section 4.3, there is a big problem with taking derivatives that is related to this decision, and there is a (very rarely encountered) bug still in the program due to this.

4 Derivatives

4.1 Introduction

Derivatives of a function are taken during the build phase. This allows complete executable objects to be built, eliminating any cost of derivatives at runtime. Derivatives are performed as a macro-transformation on the syntax of the body of the function. For example, say $f(x, y) = x^2 + y$, and consider the expression $f_x(3, 5)$. When $f_x$ is built, the body of $f$ is transformed by derivative and simplifying transformations:
The body of $f_{x:x}$ is set to the final tree above.

### 4.2 The Chain Rule

It is possible that functions have argument types that are not just real numbers (e.g., the vector type $(\mathbb{R}, \mathbb{R})$). Furthermore, even if we do not allow the user to take derivatives with respect to arguments of non-scalar type, this case still comes up. For example, consider the two function expressions $g(x, y) = f(x, y)$ and $f(A) = (-A_2, A_1)$. Here, $g$ is a function of two real numbers, while $f$ is a function of a 2D vector. If we want to take the derivative of $g$ with respect to $x$, we would use the chain rule and take the derivative of $f$. Thus, the derivative program must be able to handle derivatives of functions with respect to non-real variables, using a definition for this that is consistent with the chain rule.

The chain rule used by the derivative program makes use of the ability to have vectors as the component types of vectors. The chain rule is formulated as follows:

$$d_x(f(a_1, \ldots, a_n)) = (\nabla f(a_1, \ldots, a_n), (d_x f)(a_1, \ldots, a_n)) \cdot ((d_x a_1, \ldots, d_x a_n), 1)$$  \hspace{1cm} (1)

Here, the dot product is recursively defined:

$$u \cdot v = \begin{cases} uv & u : \mathbb{R} \text{ or } v : \mathbb{R} \\ \sum_{i=1}^n u_i \cdot v_i & u : (t_1, \ldots, t_n) \text{ and } v : (\tau_1, \ldots, \tau_n) \\ \sum_{i=1}^n \sum_{j=1}^m u_{ij} \cdot v_{ij} & u : [t_{11} \ldots t_{nm}] \text{ and } v : [\tau_{11} \ldots \tau_{nm}] \end{cases}$$  \hspace{1cm} (2)
The gradient $\nabla f$ is defined as usual: if $f$ is a function of $n$ variables, $\nabla f = (d_1 f, \ldots, d_n f)$. However, any of the argument types of $f$ might not be $\mathbb{R}$. The following procedure extends the notion of derivative to non-real types in order to make the chain rule work.

Let $t_1, \ldots, t_n$ be the argument types of $f$. For each argument $i$, $t_i$ is a tree whose internal nodes are vector and matrix types, and whose leaf nodes are $\mathbb{R}$ (for now, ignore function types as arguments; I will discuss them later). Call these leaf nodes $l_1, \ldots, l_k$, and denote by $\langle e_1, \ldots, e_k \rangle$ the construction (of either expressions or types) obtained from plugging each $e_j$ into the place $l_j$. For example, if $t_i = ((\mathbb{R}, \mathbb{R}), \mathbb{R})$, then $\langle 1, 2, 3 \rangle_i = ((1, 2), 3)$. Define $d_i f$ as

$$d_i f = \langle d_{i_1} f, \ldots, d_{i_k} f \rangle_i$$

where $d_{i_j} f$ is the expression obtained by differentiating $f$ with respect to the argument $i$ using the macro transformations, and using $d_{i_j} = (0, \ldots, 0, 1, 0, \ldots, 0)_i$, where the 1 is in the $j^{th}$ place.

### 4.2.1 Chain Rule: An Example

This definition coincides with the normal definition of the chain rule. As an example, take $f : \mathbb{R}^2 \to \mathbb{R}^3$ and $g : \mathbb{R} \to \mathbb{R}^2$, and consider differentiation with respect to the (real-valued) argument of $g$. Say $f$ is a function of $u = (u_1, u_2)$, $f(u)$, and $g$ is a function of $x$, $g(x)$. Using the ordinary definition of the chain rule, we know that

$$d_x(f(g)) = \begin{bmatrix} \partial f_1/\partial u_1 & \partial f_1/\partial u_2 \\ \partial f_2/\partial u_1 & \partial f_2/\partial u_2 \\ \partial f_3/\partial u_1 & \partial f_3/\partial u_2 \end{bmatrix} \begin{bmatrix} \partial g_1/\partial x \\ \partial g_2/\partial x \end{bmatrix}$$

We can view these matrices as vectors of vectors; this gives us

$$d_x(f(g)) = \left( \begin{bmatrix} \partial f_1/\partial u_1 \\ \partial f_2/\partial u_1 \\ \partial f_3/\partial u_1 \end{bmatrix} \right) \cdot \left( \begin{bmatrix} \partial f_1/\partial u_2 \\ \partial f_2/\partial u_2 \\ \partial f_3/\partial u_2 \end{bmatrix} \right) \cdot \begin{bmatrix} \partial g_1/\partial x \\ \partial g_2/\partial x \end{bmatrix}$$

where the “dot product” used is the one defined in (2) (note that the distinction between a “row vector” and a “column vector” is not used due to this definition).
The result of the dot product in (4) is the same as the matrix multiplication in (3):

\[
\begin{pmatrix}
\frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} \\
\frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} \\
\frac{\partial f_3}{\partial u_1} & \frac{\partial f_3}{\partial u_2}
\end{pmatrix}
\begin{pmatrix}
\frac{\partial g_1}{\partial x} \\
\frac{\partial g_2}{\partial x}
\end{pmatrix}
= 
\begin{pmatrix}
\frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} \\
\frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} \\
\frac{\partial f_3}{\partial u_1} & \frac{\partial f_3}{\partial u_2}
\end{pmatrix}
\begin{pmatrix}
\frac{\partial g_1}{\partial x} + \frac{\partial f_1}{\partial u_1} \\
\frac{\partial f_2}{\partial u_1} + \frac{\partial f_2}{\partial u_2} \\
\frac{\partial f_3}{\partial u_1} + \frac{\partial f_3}{\partial u_2}
\end{pmatrix}
\]

Furthermore, equation (4) is precisely what the chain rule (1) gives us: Since \( f \)'s argument \( u \) is of type \((\mathbb{R}, \mathbb{R})\), the procedure in section 4.2 gives us

\[
d_u f = \langle d_1 f, d_2 f \rangle_u \\
= \left( \frac{\partial f}{\partial u} \bigg|_{d_u u = (1, 0)} \right) \left( \frac{\partial f}{\partial u} \bigg|_{d_u u = (0, 1)} \right)
\]

\[
= \left( \frac{\partial f}{\partial u_1}, \frac{\partial f}{\partial u_2} \right)
\]

\[
= \left( \frac{\partial}{\partial u_1} (f_1, f_2, f_3), \frac{\partial}{\partial u_2} (f_1, f_2, f_3) \right)
\]

Similarly, since \( g \)'s argument is of type \( \mathbb{R} \), using the same procedure we get

\[
d_x g = \frac{\partial g}{\partial x} \bigg|_{d_x x = 1} \\
= \frac{\partial}{\partial x} (g_1, g_2)
\]

Plugging these into (1), we get

\[
d_x (f(g)) = (d_u f(g), (0, 0, 0)) \cdot (d_x g, 1) \\
= d_u f(g) \cdot d_x g + (0, 0, 0) \cdot 1 \\
= \left( \frac{\partial}{\partial u_1} (f_1, f_2, f_3), \frac{\partial}{\partial u_2} (f_1, f_2, f_3) \right) \cdot \frac{\partial}{\partial x} (g_1, g_2) + (0, 0, 0) \\
= \begin{pmatrix}
\frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} \\
\frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} \\
\frac{\partial f_3}{\partial u_1} & \frac{\partial f_3}{\partial u_2}
\end{pmatrix}
\begin{pmatrix}
\frac{\partial g_1}{\partial x} \\
\frac{\partial g_2}{\partial x}
\end{pmatrix}
\]

Thus, (4) is exactly what we get from applying the chain rule (1).
4.3 Function Argument Types

Functions can be arguments to functions. Taking the derivative with respect to a function does not make sense, and thus the user should not be allowed to do this. However, this case may still come up in the chain rule. Suppose \( F : \mathbb{R}, (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R}, \) \( f : \mathbb{R} \rightarrow \mathbb{R}, \) and \( g : \mathbb{R} \rightarrow \mathbb{R}, \ g(x) = F(x, f). \) Then the user may take the derivative \( g'. \)

This can be handled simply leaving out the derivatives for function arguments in the chain rule. We can define the chain rule as

\[
d_x(f(a_1, \ldots, a_n)) = (\nabla^* f(a_1, \ldots, a_n), (d_x f)(a_1, \ldots, a_n)) \cdot (d_x a_i)_{i \in I}, 1)
\]  

(5)

where \( \nabla^* f = (d_i f)_{i \in I} \), and \( I = \{i : t_i \text{ does not contain a function type}\} \). Here, \( t_i \) is the \( i^{th} \) argument type of \( f \).

While I believe this works if we know the argument types of \( f \), we do not always know \( f \)'s argument types if we do not have a static type-checker before the build phase. Without a static type-checker, I find \( f \)'s argument types by seeing what types it (or its derivative) is applied to. Since the derivative function might not be immediately applied, I lazily put off taking the derivative of a function until it is applied; thus, whenever a derivative is actually taken, the types of the arguments it is applied to are known.

However, with this strategy it is not always possible to ignore function argument types. For example, if \( X : \mathbb{R}^2 \rightarrow \mathbb{R}^3 \) and \( c : \mathbb{R} \rightarrow \mathbb{R}^2 \), then the user may write \( X_u(c)(3) \). Then \( X_u \) is directly applied to \( c \), but \( c \) has type \( \mathbb{R} \rightarrow (\mathbb{R}, \mathbb{R}) \). To solve this, I take the derivative using the macro transformations, plugging in a placeholder node for \( d_i i \). When the derivative function is applied and building the placeholder node results in a non-function type, the argument type of the function is known. At this point, an escaper is invoked, and the derivative syntax tree is rebuilt with the appropriate value(s) for \( d_i i \).

This approach (although is very complicated) works for functions that are applied to functions, but whose argument types are not actually function types. However, it breaks down for functions whose argument types actually are function types. This is a bug that currently remains in the program. For example, suppose \( F : \mathbb{R}, (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R}, \ F(x, f) = f(x) ; \ f : \mathbb{R} \rightarrow \mathbb{R}; \) and \( g : \mathbb{R} \rightarrow \mathbb{R}, \ g(x) = F(x, f). \) Then the program will not correctly be able to handle \( g' \). In our normal use of the
program, cases like this have not come up much. This is due to the fact that the chain rule only
applies when the syntax in the body of a function contains a function application. It does not apply
when the function itself is found through function applications. For example, in the expression
$N(X)(c)'$, the program first creates the function $N(X)(c)$, and then takes its derivative; the chain
rule is never used. In the expression $\text{func}(t)(N(X)(c)(t))'$, the chain rule applies, but $N(X)(c)$ is
the function being applied; thus, the chain rule is not applied to $N(X)(c)$—this function’s derivative
is found after the applications are performed.

This bug is a major bug, though, and I hope to fix it in the future. However, it’s not clear exactly
how this can be fixed. One approach would be to have some sort of a type checking phase before
taking the derivatives. In section 3, though, I show that there are problems with using unification
type checkers. An observation that might help here, however, is that we don’t need to know full
types of arguments, only whether they must be functions (due to being applied, etc.). Another
approach could be to substitute function arguments before taking the derivative.