Algorithmic Differentiation in Axiom

Jacob Smith, Gabriel Dos Reis, and Jaakko Järvi
Dept. of Computer Science, Texas A&M University
College Station, Texas, U.S.A
{thechao,gdr,jarvi}@cs.tamu.edu

ABSTRACT
This paper describes the design and implementation of an algorithmic differentiation framework in the Axiom computer algebra system. Our implementation works by transformations on Spad programs at the level of the typed abstract syntax tree — Spad is the language for extending Axiom with libraries. The framework illustrates an algebraic theory of algorithmic differentiation, here only for Spad programs, but we suggest that the theory is general. In particular, if it is possible to define a compositional semantics for programs, we define the exact requirements for when a program can be algorithmically differentiated. This leads to a general algorithmic differentiation system, and is not confined to functions which compute with basic data types, such as floating point numbers.

Categories and Subject Descriptors
I.1.2 [Algorithms]: Algebraic Algorithms; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms
Algorithms, Design, Languages, Theory

Keywords
Axiom, algorithmic differentiation, program transformation, symbolic-numeric computation

1. INTRODUCTION
Algorithmic differentiation (AD) is a technique for computing derivatives of a computer program avoiding both symbolic differentiation and divided difference approximations [10]. Tools for algorithmic differentiation transform an input program into another program, augmenting the original program with instructions to compute derivatives. The concept of algorithmic differentiation is very general, and does not require that symbolic expressions for the functions subject to differentiation are obtainable.

We demonstrate AD with the following example. Consider the Tchebychev polynomial of the first kind:

\[ T_n(x) = \cosh(n \, \text{arccosh}(x)) \]

This polynomial can be defined in the Axiom computer algebra system, discussed in Section 2, in (at least) two ways: directly enter it into Axiom from the command line as

\[ \text{Tchebychev}(x,n) \equiv \cosh(n \times \text{acosh}(x)) \]

or defining it as a recursive program, using a well known recurrence relation:

\[ \text{Tchebychev}(x: \text{Integer}, n: \text{Integer}): \text{Integer} \equiv \begin{cases} 1 & \text{if } n=0 \\ x & \text{if } n=1 \\ 2 \times x \times \text{Tchebychev}(x, n-1) - \text{Tchebychev}(x, n-2) & \text{otherwise} \end{cases} \]

Most certainly, this function definition is out of the range of conventional symbolic differentiation tools. An AD tool, however, can handle this function without trouble.

Several software packages exist for algorithmic differentiation [14, 16, 11, 3, 2], some extending a compiler [14], some implemented as a pre-processor [2], and some purely as a software library [11]. Typically these packages operate on computations on concrete data types, such as the built-in floating point types. In particular, such software are difficult to use with generic programming techniques — techniques that typically do not know actual data representations in advance — but instead state abstract requirements on the types they work with.

For its contributions, this paper brings the problematic of AD to the Computer Algebra community for effective and mathematical formalization. Indeed, we believe that this community could make a stronger effort improving its mathematical foundations. The paper discusses an algebraic differentiation system which is not limited to concrete data types, but rather can compute the algorithmic derivative of a program as long as the types it computes with satisfy suitable requirements. To specify these requirements, we have developed the beginnings of a theory for algorithmic differentiation. We present the theory in the context of the Axiom’s library extension language, but suggest that the ideas are more general. Our algorithmic differentiation framework, based on the presented theory, is a semantics-based transformation tool of Spad programs.

We have used tools from the theory of programming languages’ semantics to exhibit foundational algebraic structures underlying the practice of algorithmic differentiation. Finally we observe that our framework is designed as a library, without modification of the Axiom system.
2. THE AXIOM SYSTEM

Axiom is a strongly-typed general-purpose computational platform, supporting both numeric and symbolic computations. It uses type information, at compile time, to guide selection and application of operations. For example, given the recursive definition of the function \( \text{Tchebychev} \) from the previous section, the Axiom compiler generates direct calls to the multiplication and subtraction operations over Integer values for the fragment

\[
2 \times x \times \text{Tchebychev}(x, n-1) - \text{Tchebychev}(x, n-2)
\]
as opposed to inspecting the values held by the variables \( x \), \( n \), and the values returned by the function \( \text{Tchebychev} \) at run time and “discovering” the function to call. If the type specified for the variable \( x \) were unknown, or if \( x \) had a type for which no suitable operation named \( * \) could be found, then the expression would be rejected by the type checker and no code would be generated. This design aspect of Axiom contrasts remarkably with most general purpose computer algebra systems: they rely on run time type checking.

Axiom users can extend the system with libraries implemented in Spad, a general purpose programming language with affinity towards mathematical software construction and scientific computations. The Axiom system comes with more than a thousand implementations of computational mathematical entities. More information about the Axiom system and its use can be found in “the Axiom Book” [13, 5], the seminal papers of J. Davenport and collaborators [7, 6], and on Axiom web site [1].

3. SPAD PROGRAMMING LANGUAGE

This section describes the (abstract) syntax and the semantics of the Spad programming language. The intuitive behavior of a Spad program is described by its operational semantics. It uses type information, at compile time, to guide selection and application of operations. For example, given the recursive definition of the function \( \text{Tchebychev} \) from the previous section, the Axiom compiler generates direct calls to the multiplication and subtraction operations over Integer values for the fragment

\[
2 \times x \times \text{Tchebychev}(x, n-1) - \text{Tchebychev}(x, n-2)
\]
as opposed to inspecting the values held by the variables \( x \), \( n \), and the values returned by the function \( \text{Tchebychev} \) at run time and “discovering” the function to call. If the type specified for the variable \( x \) were unknown, or if \( x \) had a type for which no suitable operation named \( * \) could be found, then the expression would be rejected by the type checker and no code would be generated. This design aspect of Axiom contrasts remarkably with most general purpose computer algebra systems: they rely on run time type checking.

Axiom users can extend the system with libraries implemented in Spad, a general purpose programming language with affinity towards mathematical software construction and scientific computations. The Axiom system comes with more than a thousand implementations of computational mathematical entities. More information about the Axiom system and its use can be found in “the Axiom Book” [13, 5], the seminal papers of J. Davenport and collaborators [7, 6], and on Axiom web site [1].

![Abstract syntax of the Spad language.](image)

**Figure 1:** Abstract syntax of the Spad language. The notation \( Z \) represents an optional \( Z \), \( Z^+ \) a non-empty finite sequence of \( Z \); the square brackets are used for grouping.

1. the symbol \( * \) is a binary operation on objects of domain asserted belonging to the category being defined;
2. the identifier \( 1 \) denotes a constant object of domain belonging to the category being defined.

The following category definition

**Group()**: Category == Monoid with

\[
\text{inverse: } % \to %
\]

extends Monoid with the inverse operation, to capture the mathematical notion of group structure. One can think of Spad categories as specifying views on objects.

**Exports.** Definitions for Spad categories, Spad domains, and packages specify exported operations, i.e. the “public interface” in programming languages jargon, through either a WithExpr, or an Extension, or combination of both.

**WithExpr.** A WithExpr is essentially an unnamed Spad category consisting of a list of operation signatures (Signature).

**Extension.** Definitions for Spad categories and Spad domains may extend existing Spad categories or domains. An extension may specify either a type, or multiple categories through the Join operator. The latter form corresponds to multiple inheritance in object-oriented programming languages.
**Signature.** The specification of a type for an identifier can appear in a `WithExpr`, as a parameter declaration in `CallForm`, as the field of a record or union, or in a local variable definition.

**Type.** A type is a built-in type (`Boolean, Integer, Float`), a record or union, a function type, the name of a Spad category or Spad domain, or an instantiation of a Spad category or Spad domain. All field names specified by signatures in a record must be distinct. Similarly, all field names specified in a union must be distinct and unique in the enclosing scope; this applies recursively to any other union types directly referenced in the signature list of the union.

**DomainDef.** A Spad domain definition provides implementations for views specified by categories. A domain definition has an interface specification part (`Exports`) stating the categories and possible additional signatures it implements, and an implementation part called `capsule`. The implementation part may define the representation of the object belonging to the domain, and provide definitions for operations declared in its `Exports`. For example, the program fragment

```plaintext
IntMonoid(): Monoid ==
  Rep := Integer
  (a:%) * (b:%) == (rep a + rep b)$Integer
  1:% == 0::Integer
```

provides an implementation `IntMonoid` for the `Monoid` specification as follows:

- the object representation domain is `Integer`;
- “multiplication” of two objects in `IntMonoid` is the value obtained by adding their respective underlying values (returned by the `*` operator);
- the `Integer` constant `0` is the underlying value of the unit of `IntMonoid`.

Note that a Spad domain almost always references the “current domain” using the symbol `%`.

**PackageDef.** A package definition provides implementations for functions that operate on a Spad domain. Unlike a Spad domain, a package does not define a `Representation` and does not reference the symbol `%`. Like a Spad domain, it has an `Exports` part and an implementation part.

**Capsule.** The implementation part of a Spad domain or package is its capsule. A capsule may specify the representation of a domain (if it is the implementation of a domain), and specifies a sequence of toplevel definitions for operations on the Spad domain objects, or the operators in a package.

**Representation.** A Spad domain specifies the underlying representation of its objects by assigning a type expression to the identifier `Rep`. A `Representation` can occur only in a Spad domain definition.

**Definition.** A (delayed) definition is the binding of an identifier or a function call expression to a Spad category, Spad domain, or an ordinary function. The body of the definition is evaluated when needed. That evaluation may happen only once for a given argument list. Even though the evaluation is delayed, the body is still fully type checked at the definition point. The Spad language, as understood by the Spad compiler, does not allow ordinary function definitions at toplevel. However, they are the core of the language understood by the interpreter. For uniformity, we include toplevel function definitions in the Spad subset we describe.

**CallForm.** A call form consists of an identifier and a parenthesized sequence of signatures declaring formal parameters. A call form is needed in the definitions of a Spad category, Spad domain, and function.

**Statement.** Statements appear in the body of function definitions. A statement is either an expression, a one or two-arm if-statement, an iteration where the body of the iteration (a statement) is controlled by a list of iterators, a local variable definition, or a an assignment.

**Iterator.** An iterator is either a sequence of items `x` drawn from a sequence `e`, possibly filtered by a predicate `p`, or a repeated evaluation of a predicate.

**Expression.** An expression is either a constant, variable, function call, member selection, type-case expression, an assignment, or a qualified expression. A qualified expression is an expression that contains free variables and is immediately followed by their definitions in a `where-clause`. We assimilate expressions built with built-in operations — such as addition on integers, etc. — as function calls.

**Variable.** A variable is the use of a name declared with a given type.

**Constant.** A constant is a built-in value, such as `342 Integer`, `true Boolean`, `1 Integer → Integer → Integer`, etc.

**Identifier.** An identifier is a finite sequence of characters. The set of identifiers in Spad is countably infinite.

### 3.2 Language features

The Spad programming language supports elements of dependent types. That is, the Spad type system allows types to depend on types, but also on values. For example, the following definition, part of the Axiom’s standard library,

```plaintext
Variable(sym:Symbol): _
  Join(SetCategory, CoercibleTo Symbol) with
  coerce : % -> Symbol
    ++ coerce(x) returns the symbol
  variable(): () -> Symbol
    ++ variable() returns the symbol
    == add
      coerce(x:%):Symbol == sym
      coerce(x:%):OutputForm == sym::OutputForm
    x = y == true
    latex(x:%):String == latex sym
```

defines a parameterized domain `Variable` whose argument is a value of type `Symbol`, an Axiom standard domain. So the types `Variable x` and `Variable y` designate different domains, because the domain arguments `x` and `y` are not equal when considered as values of type `Symbol`. Dependent types enables an unusual direct style of implementation of mathematical structures.
The Spad programming language also supports general overloading; in particular, a function can be overloaded on its return type. The overload resolution algorithm exploits all context information, including arguments and target types, to select the best matching function. Implicit conversion is supported through the coerce operator. The above example says that a value of type symbol can be converted to a Symbol, or to an OutputForm, the Axiom preferred domain for output operations.

### 3.3 Semantics

The computational rules used to evaluate Spad programs are those of *eager* semantics, and functions arguments are passed by value. We sketch the semantics of Spad programs in two ways: operational semantics and denotational semantics. The operational semantics gives an intuitive idea of the behavior of Spad programs, whereas the denotational semantics lets us associate mathematical functions to Spad programs. The latter allows us to formally talk about the notion of a derivative of a Spad program.

#### 3.3.1 Operational semantics

The Spad language is imperative in the sense that its programs operate on stores by explicit modification. Values of types are those of eager semantics, and functions arguments are passed by reference. We sketch the semantics of Spad programs as follows: 

- **Location values:** object locations are in Value
- **Boolean values:** true ∈ Value and false ∈ Value
- **Integer values:** integer constants \( n^{int_{ern}} \) are in Value
- **Float values:** float constants \( f^{float} \) are in Value
- **Functions:** if \( f \) is a defined function of type \( \sigma_1 \rightarrow \sigma_2 \), then the constant \( f^{int_{ern}} \) is in Value
- **Aggregates:** if \( c_i^{\tau_i} \) are values of type \( \tau_i \) in Value, then the tuple \( \langle c_1^{\tau_1}, \ldots, c_n^{\tau_n} \rangle^{\tau_1 \times \ldots \times \tau_n} \) is in Value. Tuples represent record values. Similarly, if \( c^{\tau_1} \) in Value, so is \( c^{\tau_2 \rightarrow \tau_1} \). It represents a value of a field of type \( \tau_1 \) in a union \( \tau_2 \).

The behavior of a Spad program is a sequence of configurations \( \langle p, \sigma, \Gamma \rangle \) where \( p \) denotes fragments of Spad constructs, \( \sigma \) the store of values, and \( \Gamma \) the current environment of bindings of variables to types and expressions. The notation \( \Gamma, x^r == e \) denotes an environment obtained by extending \( \Gamma \) with the binding \( x^r == e. \) The == e part may be missing. A store \( \sigma \) is a mapping from memory locations to Spad values. We use the notation \( \sigma[v/l] \) to designate an updated function defined by

\[
\sigma[v/l](x) = \begin{cases} 
  v & \text{if } x = l \\
  \sigma(x) & \text{otherwise}
\end{cases}
\]

Each configuration is defined by structural induction on the syntax of Spad, as specified in Figure 2.

#### 3.3.2 Denotational semantics

The basic idea of algorithmic differentiation rests on the notion that a computer program computes a function whose range has a ring structure; and the collection of such functions can be endowed with a differential algebra structure. The theory of denotational semantics [15] is a useful tool in laying down the necessary theoretical framework for meaningful discussion of computing the derivative of computer programs. We seek for a standard denotational semantics \( [\bullet] \) of the Spad programming language, that respects the operational semantics outlined in §3.3.1, i.e.,

\[
t \rightarrow v' \Rightarrow [t] = [v'].
\]

### 4. ELEMENTS OF ALGEBRAIC THEORY OF ALGORITHMIC DIFFERENTIATION

The very idea of transforming a program \( P \) to another program \( P' \) such that the function computed by the program \( P' \) is the derivative of the function computed by the program \( P \) assumes that we are given a semantics function that computes the meaning of a program.

In our framework, we define a program written in the Spad language as a SPAD-algebra, where SPAD is the functor structuring the Spad language abstract syntax (Figure 1.) Let’s call \( S \) the collection of all SPAD-algebras We are interested in meaning functions \( [\bullet] : S \rightarrow D \) where the semantics domain \( D \) is suitable for talking about derivatives.

#### 4.1 Algorithmic differential rings

An algorithmic differentiation of Spad program is a transformation \( \Phi : S \rightarrow D \) such that

- the function \([\Phi (-)] : S \rightarrow D \) respects composition, i.e.,

\[
[\Phi (P_1; P_2)] = [\Phi P_2] \circ [\Phi P_1]
\]

where we have used ; to indicate sequencing, i.e. the action of executing program \( P_1 \) first, followed by the execution of the program \( P_2 \). This key functorial property embodies the usual chain rule from calculus.

- if two programs \( P_1 \in S \) and \( P_2 \in S \) have meanings \([P_1] = f : T \rightarrow U_1\) \([P_2] = g : T \rightarrow U_2\), where \( U_1 \) is a strict extension of a differential ring \( (U, \delta) \), then the following identities hold

\[
\delta ([\Phi P_1] + [\Phi P_2]) = \delta ([\Phi P_1]) + \delta ([\Phi P_2])
\]

\[
\delta ([\Phi P_1] \cdot [\Phi P_2]) = \delta ([\Phi P_1]) \cdot [P_2] + [P_1] \cdot \delta ([\Phi P_2]).
\]

These two identities relate the meaning of the transformed programs to the usual mathematical notion of derivation, namely additivity and Leibniz rule.

Note that if the domains of computation are that of polynomials, or power series, with usual derivation operation then the chain rule holds. In more general domains of computations, however, we add the chain rule as a requirement. Consequently, we require that all our domains of computations where we carry differentiation are actually algorithmic differential rings, and not just differential rings.

#### 4.2 Strategies of derivative evaluation

Based on the chain rule and associativity of function composition, one can develop various strategies for evaluating the derivative. For example, given the program

\[
P_1 ; \ldots ; P_n
\]

the meaning of its algorithmic differentiation transform is

\[
[\Phi (P_1 ; \ldots ; P_n)] = [\Phi P_n] \circ \cdots \circ [\Phi P_1]
\]
which can be evaluated using various computation strategies. One approach is a “naive” reading of the composition from right to left, leading to so-called forward mode where the first instruction is transformed, then the second, etc., and the derivatives are propagated forward. This approach is simple to comprehend and implement. However, it has the inconvenience that it generates computations with complexity expressed in terms of (indepedent) input variables. Therefore it might not be very efficient for computing gradients of scalar functions or many variables.

Another approach is a reading of the compositions from left to right. That strategy requires the composition of derivatives of functions not yet executed. Consequently, its actual implementation requires running the original programs first, then ‘reverting’ the sequencing of computations to propagate the derivatives generated, thus leading to so-called reverse mode. This composition strategy has the property that its complexity is in terms of the number of (dependent) output variables. It is therefore a good candidate for computing the gradient of a scalar function of many variables. These two strategies of computation are “extreme” in some sense, and an AD tool may actually use a mix of association of compositions.

4.3 Control flow and differentiability

The practical realization of algorithmic differentiation of a program \( \Phi \) builds on two concepts. First, the simulation of the operation semantics (§3.3.1) of \( \Phi \) yields a program \( \Phi \) that is in simple form. Note that this stage is parameterized by the operation semantics of the programming language being used (Spad, in our case). From this simple program \( \Phi \) one extracts the control flow and the data flow graphs. This data flow graph is usually called a computational graph [10] of the program \( \Phi \). The control flow graph extracted from \( \Phi \) is such that each node of the graph is a basic block, that is a maximal sequence of instructions without “jumps”. Each basic block is therefore a straight line program and, through the denotational semantics §3.3.2, defines a differentiable function. However at the joint points of the control flow graph, there is no guarantee that we obtain a differentiable function, e.g. the case of the absolute value function. Even when the mathematical function being computed is differentiable, it may be that its expression as an algorithm contains transfers of control that introduce artificial anomalies. Consider

\[
\text{funnyId(x: Integer): Integer ==>}
\text{if x=2 then 2 else x}
\]

which computes the identity function in a curious way. Its transform is

\[
\text{funnyId(x: Jet Integer): Jet Integer ==>}
\text{if x.value = 2 then jet(2,0) else x}
\]

with derivative 0 at \( x = 2 \), which obviously is wrong. We cannot statically expect to detect all such contrived constructs and produce the right results. However, the theory of data flow analysis as abstract interpretation [4] provides a framework for studying those issues, which we do not investigate further here.

5. THE SPAD COMPILER

The Axiom system can operate in interactive and “batch” modes. The interactive mode uses an interpreter, the batch mode a compiler. The compiler can be invoked from within the interpreter by issuing the system command \( \text{compiler} \).

The interpreter and the compiler understand slightly different dialects of the Spad language. This is due partly by design, and partly by a turbulent history. The compiler is intended for library development (large scale programming), whereas the interpreter is intended for convenient interactive conversation (small scale programming). Also, the interpreter supports type inference (“guessing”), as opposed to the compiler, which for the most part restricts itself to type checking. Since our framework is primarily intended for library development, we focus on the compiler component of the Axiom system. We emphasize that our framework works with both the compiled and the interpreted dialects of Spad.

An input Spad source file is decomposed into a stream of tokens by the Spad lexer. The token stream is transformed into a parse tree by the Spad parser. That parse tree undergoes further transformation by a post-parser transformer, resulting in a parse form. The parse form is still close to Spad source. For example, here are the parse forms of the definitions of Monoid and IntMonoid examples from §3.1:
The parse form is then transformed into another internal abstract syntax tree, which is used as input to the semantic analyzer. The job of the semantic analyzer is to type check the abstract syntax tree, to resolve dependencies on previously compiled programs and load them if necessary. The output of the semantic analyzer is a fully typed abstract syntax tree, which is translated by the code generator into Lisp code. A copy of the resulting Lisp code is saved on disk for future use, and another copy is given to the run time system (a Lisp system) for evaluation. The current Axiom system uses the GNU Common Lisp (GCL) implementation. GCL is capable of compiling Lisp code to native object code, which is subsequently loaded into the running Lisp image. As a result, the Axiom compiler compiles input Spad programs to native object code for execution.

6. IMPLEMENTATION

Our AD framework is entirely implemented as an Axiom library, meaning that no source code modification to the Axiom compiler is required. The implementation consists of:

- a small library interface to the Spad compiler, to retrieve the parse forms and the typed abstract syntax tree over objects of the SExpression domain;
- Axiom domains and packages working on the output of the compiler interface.

Additionally, we have an unparsy that pretty prints the internal representation as Spad code. At the time of the writing, we support only the forward mode, and we are working on the reverse mode. We do not support direct computation of gradient yet. The gradient could be obtained from repeated calls to compute partial derivative in several directions; however, that is inefficient and potentially incorrect, because if the function depends on global variables, there is no guarantee that successive calls yield the desired values.

6.1 Transformation to simple form

The main thrust of our transformer is that it requires the input program be in simple form (§4.3), which simulates the operational semantics outlined in §3.3.1. Since it is unrealistic to require users to write codes in that specific form, we have implemented a package called ADCatExpand that transforms an arbitrary Spad program into simple form. The expander “walks” the AST, identifying and expanding any expressions that are not in simple form. For example, all the arguments of function calls that are not constants or variables are replaced with temporary variables, such that every operation consists of the name of the operation and references to variables. This transformation does not change the meaning of the input program.

6.2 First order prolongation

Our AD framework uses the theory of jets [8]. If the input program computes a mathematical function \( f: M \rightarrow N \) as determined by the standard semantics, then our tool generates a program that computes the first prolongation

\[ \text{jet}^1(f): \text{jet}^1 M \rightarrow \text{jet}^1 N. \]

More precisely, the framework defines a domain Jet parameterized by a domain from the Ring category. For example, Jet(Integer) and Jet(Float) designate the first order jet of Integer and Float, respectively.

Jet(T: Ring): Public == Private where
| Public ==> with |
| jet(T, T) > | |
| ++ construct a jet value from a pair |
| elt: (T, T) -> T |
| ++ retrieve the 'value' field of a jet |
| elt: (T, T) -> T |
| ++ retrieve the 'delta' field of a jet |
| setelt: (T, T) -> T |
| ++ set the 'value' field of a jet |
| setelt: (T, T) -> T |

Private == add

Rep := Record(val: T, der: T)
jet(v, d) == [v, d]
elt(jet: %, x: T) ||- jet.val
setelt(jet: %, x: T) ||- jet.val

Note that here Jet takes types T of the Ring category; in reality, Jet should take types T from the category of algorithmic differential ring as explained in §4.1. That is a current limitation of our framework, partly due to lack of enough expressivity of the Spad language.

The generated program is defined as an overloaded function on the jets. In particular, we don’t use a fancy symbol mirroring the mathematical “functorial notation” \( \text{jet}^1(f) \). A benefit from that approach is that the generated program is structurally similar to the input program. A fundamental drawback is that it prevents us from defining functions on jet spaces which are not prolongations. For example, we would like Jet(T) to be a member of the Ring category but doing so requires defining operators \( + \) and \( \times \), which conflict with operators we may be prolonging when working with Jet(Integer) or Jet(Float) for example. This is an aspect that we plan to improve on in future work, as it makes perfect sense to define functions on jets (to represent differential equations) that are different from prolongations.

6.3 Initial environment

When our library is loaded into Axiom, it starts with an environment that contains the prolongations of certain operators on built-in types:

```lisp
\( \text{ADCatExpand} \)

\( \text{ADInitialEnvironment} \)

\( \text{ADInitialEnvironment}(\text{T: Ring}) : \text{Public} == \text{Private} \)

where

Public ==> with

"*: (Jet T, Jet T) -> Jet T
"*: (Jet T, Jet T) -> Jet T
"*: (Jet T, Jet T) -> Jet T
"*: Jet T -> Jet T

if T has Field then
```
6.4 Forward Mode

The core of our framework’s user interface consists of functions defined in the `ADCatForward` package. There are two versions: the first takes an s-expression (the internal data-structure used to hold the AST) and returns an s-expression that contains the differentiated code. The second takes a String (a file-name), and returns a list of s-expressions corresponding to the first prolongation of all domains, categories, packages, etc. found within that file.

The AD function is a simple layer on top of a routine that implements a Visitor Pattern over the AST. The AD function walks the AST until all derivatives have been generated. We use the category membership assertion to reject invalid input functions, e.g. those with arguments and return types that are not from `Ring`. For example, since the type `String` is not asserted to belong to `Ring`, we reject a request to compute the derivative of a function that returns a `String`. This is another place where we rely on type annotation for meaningful transformation.

6.5 Examples

6.5.1 The GRADIENT paper.

Our first example is the function “f” from Monagan and Neuenschwander [14] that we have translated to Spad as:

```spad
f(x: Float, n: Integer): Float ==
if n = 0 then return 0
else
  a : Float := 0
  b : Float := x
  for k in 1..(n-1) repeat
    h : Float := b
    b := log(a*b)
    a := h
  end for
  return(G2668)
```

Monagan and Neuenschwander’s example did not include explicit typing for the first argument. Here, we have chosen `Float` to emphasize the point that our framework (and Algorithmic Differentiation in general) does not require that function parameters be “symbols”.

After preliminary transformation to “simple form” (automatically done by the framework), followed by the actual AD computation, the output program is:

```spad
f(x: Jet(Float)), n: Jet(Integer)): Jet(Float) ==
if n.value = Zero() then
  G2668 := Zero()
  return(G2668)
else
  a : Jet(Float) := x
  b : Jet(Float) := x
  for k in (One())..(n.value - One()) repeat
    h : Jet(Float) := b
    G2669 := a + b
    b := log(G2669)
    a := h
  end for
  return(G2668)
```

Note that all operator names have retained their original spelling. This is because the AD transformer has inserted the first prolongation of the operators as an overloaded function defined on the first prolongations of its source and target. This means that the correct operation is selected based on its argument types, not solely on its name.

The above program, despite the syntactic similarity with the original program `f`, does compute numerical values of the derivative of the function computed by `f`. To see that this is the case, we use Axiom’s `Expressions` — even with its strong emphasis on algebraic computations, Axiom provides the parameterized domain `Expression` for symbolic manipulations. Replacing `Float` with `Expression Integer` in the original function `f` and transforming that function with our AD tool results in the function below. The result, again, is similar to the original function; only type annotations have changed and temporaries have been introduced to hold intermediate values:

```spad
f(x: Jet(Expression(Integer)), n: Jet(Integer)): Jet(Expression(Integer)) ==
if n.value = Zero() then
  G2668 := Zero()
  return(G2668)
else
  a : Jet(Expression(Integer)) := Zero()
  b : Jet(Expression(Integer)) := x
  for k in (One())..(n.value - One()) repeat
    h : Jet(Expression(Integer)) := b
    G2669 := a + b
    b := log(G2669)
    a := h
  end for
  return(G2668)
```

Evaluation of `f(jet(x,1),jet(4,0))`.delta yields:

```
log(x) + 2x + 1
```

which coincides with the derivative of `f(x,4)`:

```
log(log(log(x) + x) + x log(x)) + x log(x)
```
6.5.2 Tchebychev polynomials.

Next, we consider the Tchebychev polynomial defined by recurrence relation as shown in §1. Here, again we considered the original function taking a Float, and another taking Expression Integer to check the symbolic evaluation. The first prolongation of

\[ \text{Tchebychev}(x: \text{Expression Integer}, n: \text{Integer}): \text{Expression Integer} == \\
\text{if } n=0 \text{ then return } 1 \\
\text{if } n=1 \text{ then return } x \\
2x*\text{Tchebychev}(x,n-1) - \text{Tchebychev}(x,n-2) \]

is computed as:

\[ \text{Tchebychev}(x: \text{Jet Expression Integer}, n: \text{Jet Integer}): \text{Jet Expression Integer} == \\
\text{if } n.\text{value } = 0 \text{ then } \\
G2716 := 1 \\
\text{return } G2716 \\
\text{if } n.\text{value } = 1 \text{ then return } x \\
G2718 := 2 * x \\
G2721 := 1 \\
G2720 := n - G2721 \\
G2719 := \text{Tchebychev}(x,G2720) \\
G2717 := G2718 + G2719 \\
G2723 := n - 2 \\
G2722 := \text{Tchebychev}(x,G2723) \\
G2717 - G2722 \]

Evaluating \( \text{Tchebychev}(\text{jet}(x,1), \text{jet}(5,0)).\text{delta} \) yields

\[ 80x - 60x + 5 \]

which agrees with the derivative of \( T_5 = 16x^5 - 20x^3 + 5x \).

7. Conclusion

Automatic differentiation is a well-known technique for computing derivatives, and matured software packages exist for applying it in practice. However, we found the theory of algorithmic differentiation incomplete. This paper explores such a theory. We base our work on top of the Axiom computer algebra system, which allows highly generic programs to be written in terms of “categories,” or classes of algebras, instead of concrete data-types. We outline the formal semantics of Spad, the library extension language of Axiom, and define the exact requirements for when Spad programs can be subjected to algorithmic differentiation.

We apply our theory in an implementation of an algorithmic differentiation tool for Spad. The implementation transforms Spad programs, at the level of typed abstract syntax trees, into programs that compute derivatives as well as their original values. Our prototype implementation illustrates the benefits of the algebraic approach to algorithmic differentiation.

7.1 Related Work

M. Monagan and W. Coene [14] implemented the forward mode of AD in Axiom, called GRADIENT, in the Maple computer algebra system. The GRADIENT package was later extended by D. Villard and M. Monagan [16] to cover the reverse mode. To the best of our knowledge, our work is the first documented attempt at implementing AD for the Axiom computer algebra system.

7.2 Future Work

There are several directions we would like to extend our work. First, we are working on finishing the reverse mode implementation, as well on better support for higher order derivatives exploiting sparsity and symmetries. The “Axiom way” of implementing computational mathematics is to rely on strong typing to structure programs. Our initial work on interfacing with the Spad compiler suggests that we need Spad domains and packages for strongly typed representations of Spad programs themselves. Finally, and probably most importantly from the “Axiom way” of doing computational mathematics, we will continue the development of formal semantics of the Spad programming languages as well as the algebraic theory of algorithmic differentiation, to gain better understanding of algorithmic differentiation, and to stimulate the construction of generic AD libraries.

Acknowledgments

The authors thank the anonymous referees for their helpful suggestions for improvement, and Yinan Fan and James Huang for early discussions about our framework.

8. References