

From Untyped to Polymorphically Typed Mathematical Web Services

William Naylor and Julian Padget

Department of Computer Science
University of Bath, UK

2006 08 11 / MKM2006

Outline

- 1 Preliminaries
- 2 Technical content
- 3 Outcomes
- 4 Conclusion
- 5 Algorithms

Aldor basics

Aldor: *Computer Algebra*

A Language for Describing Objects and Relations

Basic Aldor concepts:

- | Aldor | | Java |
|------------|---|----------------------------------|
| • Category | – | Interface |
| • Domain | – | Class |
| • Package | – | Class no representation (fields) |

Aldor basics

Aldor: *Computer Algebra*

A Language for Describing Objects and Relations

Basic Aldor concepts:

Aldor

Java

- Category – Interface
- Domain – Class
- Package – Class no representation (fields)

Aldor basics

Aldor: *Computer Algebra*

A Language for Describing Objects and Relations

Basic Aldor concepts:

Aldor

Java

- Category – Interface
- Domain – Class
- Package – Class no representation (fields)

Aldor basics

Aldor: *Computer Algebra*

A Language for Describing Objects and Relations

Basic Aldor concepts:

Aldor

Java

- Category – Interface
- Domain – Class
- Package – Class no representation (fields)

The Aldor Advantage

Building web services using Aldor has several advantages:

- **Interoperability:** *Aldor is designed to be used in interaction with other languages, e.g. Java*
- **Strong Typing:** *Sophisticated type structure designed to capture the structure of mathematics*
- **Efficiency:**
 - *Aldor sources may be compiled to native code*
 - *Executable speed comparable to C++*
 - *Small footprint since only the relevant libraries are linked*

The Aldor Advantage

Building web services using Aldor has several advantages:

- Interoperability: *Aldor is designed to be used in interaction with other languages, e.g. Java*
- Strong Typing: *Sophisticated type structure designed to capture the structure of mathematics*
- Efficiency:
 - *Aldor sources may be compiled to native code*
 - *Executable speed comparable to C++*
 - *Small footprint since only the relevant libraries are linked*

The Aldor Advantage

Building web services using Aldor has several advantages:

- Interoperability: *Aldor is designed to be used in interaction with other languages, e.g. Java*
- Strong Typing: *Sophisticated type structure designed to capture the structure of mathematics*
- Efficiency:
 - *Aldor sources may be compiled to native code*
 - *Executable speed comparable to C++*
 - *Small footprint since only the relevant libraries are linked*

Service Communication

- A web service must communicate with the world using some protocol
- It is preferable that the protocol is semantically unambiguous
- OpenMath provides such an unambiguous protocol for mathematical objects, *via* Content Dictionaries – definition repositories

Service Communication

- A web service must communicate with the world using some protocol
- It is preferable that the protocol is semantically unambiguous
- OpenMath provides such an unambiguous protocol for mathematical objects, *via* Content Dictionaries – definition repositories

Service Communication

- A web service must communicate with the world using some protocol
- It is preferable that the protocol is semantically unambiguous
- OpenMath provides such an unambiguous protocol for mathematical objects, *via* Content Dictionaries – definition repositories

OpenMath Example

The 2x2 matrix

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

becomes:

```
<OMA>
  <OMS cd = "linalg2" name = "matrix"/>
  <OMA>
    <OMS cd = "linalg2" name = "matrixrow"/>
    <OMI>1</OMI> <OMI>3</OMI>
  </OMA>
  <OMA>
    <OMS cd = "linalg2" name = "matrixrow"/>
    <OMI>2</OMI> <OMI>4</OMI>
  </OMA>
</OMA>
```

Aldor Code Wrapper

Considerations:

- Standardised representation on client side: *OpenMath*
- Standardised communication protocols: *Axis, SOAP (Well tested serialisation mechanisms exist for OpenMath)*
- The service implementer should not be required to be concerned with the communication protocols
- Problems arise due to the strongly typed nature of Aldor

Aldor Code Wrapper

Considerations:

- Standardised representation on client side: *OpenMath*
- Standardised communication protocols: *Axis, SOAP (Well tested serialisation mechanisms exist for OpenMath)*
- The service implementer should not be required to be concerned with the communication protocols
- Problems arise due to the strongly typed nature of Aldor

Aldor Code Wrapper

Considerations:

- Standardised representation on client side: *OpenMath*
- Standardised communication protocols: *Axis, SOAP (Well tested serialisation mechanisms exist for OpenMath)*
- The service implementer should not be required to be concerned with the communication protocols
- Problems arise due to the strongly typed nature of Aldor

Aldor Code Wrapper

Considerations:

- Standardised representation on client side: *OpenMath*
- Standardised communication protocols: *Axis, SOAP (Well tested serialisation mechanisms exist for OpenMath)*
- The service implementer should not be required to be concerned with the communication protocols
- Problems arise due to the strongly typed nature of Aldor

The ExpressionTree Domain

OpenMath – type information is not mandated
v.s. Aldor – strongly typed

- Need an interface domain: `ExpressionTree` *with functions to perform the translations*,
 $\lambda : \text{ExpressionTree} \rightarrow X : \text{Parsable}$
 $\lambda : R : \text{ExpressionType} \rightarrow \text{ExpressionTree}$
- `ExpressionTree` exports functions to convert to: TeX, Fortran, C, Lisp, Maple or Axiom format
- Extend `ExpressionTree` with a function to export `OpenMath`

The ExpressionTree Domain

OpenMath – type information is not mandated
v.s. Aldor – strongly typed

- Need an interface domain: `ExpressionTree` *with functions to perform the translations*,
 $\lambda : \text{ExpressionTree} \rightarrow \text{X:Parsable}$
 $\lambda : \text{R:ExpressionType} \rightarrow \text{ExpressionTree}$
- `ExpressionTree` exports functions to convert to: TeX, Fortran, C, Lisp, Maple or Axiom format
- Extend `ExpressionTree` with a function to export `OpenMath`

The ExpressionTree Domain

OpenMath – type information is not mandated
v.s. Aldor – strongly typed

- Need an interface domain: `ExpressionTree` *with functions to perform the translations*,
 $\lambda : \text{ExpressionTree} \rightarrow \text{X:Parsable}$
 $\lambda : \text{R:ExpressionType} \rightarrow \text{ExpressionTree}$
- `ExpressionTree` exports functions to convert to: TeX, Fortran, C, Lisp, Maple or Axiom format
- Extend `ExpressionTree` with a function to export `OpenMath`

The ExpressionTree Domain

OpenMath – type information is not mandated
v.s. Aldor – strongly typed

- Need an interface domain: `ExpressionTree` *with functions to perform the translations*,
 $\lambda : \text{ExpressionTree} \rightarrow \text{X:Parsable}$
 $\lambda : \text{R:ExpressionType} \rightarrow \text{ExpressionTree}$
- `ExpressionTree` exports functions to convert to: TeX, Fortran, C, Lisp, Maple or Axiom format
- Extend `ExpressionTree` with a function to export `OpenMath`

Example

We transform a `DenseMatrix(Integer)` into various formats:

```
%1 >> mat:DenseMatrix(Integer) := [[1,2],[3,4]]
matrix [[1,3],[2,4]] @ DenseMatrix(AldorInteger)
```

```
%2 >> exmat := extree(mat)
(matrix 2 2 1 3 2 4) @ ExpressionTree
```

```
%3 >> axiom(stdout,exmat) -- Axiom format
matrix [[1,3],[2,4]] () @ TextWriter
```

```
%4 >> maple(stdout,exmat) -- maple format
linalg[matrix](2,2,[1,3,2,4]) () @ TextWriter
```

```
%5 >> tex(stdout,exmat) -- tex format
\pmatrix{
1 & 3 \cr
2 & 4 \cr} () @ TextWriter
```

Example

We transform a `DenseMatrix(Integer)` into various formats:

```
%1 >> mat:DenseMatrix(Integer) := [[1,2],[3,4]]
matrix [[1,3],[2,4]] @ DenseMatrix(AldorInteger)
```

```
%2 >> exmat := extree(mat)
(matrix 2 2 1 3 2 4) @ ExpressionTree
```

```
%3 >> axiom(stdout,exmat) -- Axiom format
matrix [[1,3],[2,4]] () @ TextWriter
```

```
%4 >> maple(stdout,exmat) -- maple format
linalg[matrix](2,2,[1,3,2,4]) () @ TextWriter
```

```
%5 >> tex(stdout,exmat) -- tex format
\pmatrix{
1 & 3 \cr
2 & 4 \cr} () @ TextWriter
```

Example

We transform a `DenseMatrix(Integer)` into various formats:

```
%1 >> mat:DenseMatrix(Integer) := [[1,2],[3,4]]
matrix [[1,3],[2,4]] @ DenseMatrix(AldorInteger)
```

```
%2 >> exmat := extree(mat)
(matrix 2 2 1 3 2 4) @ ExpressionTree
```

```
%3 >> axiom(stdout,exmat) -- Axiom format
matrix [[1,3],[2,4]] () @ TextWriter
```

```
%4 >> maple(stdout,exmat) -- maple format
linalg[matrix](2,2,[1,3,2,4]) () @ TextWriter
```

```
%5 >> tex(stdout,exmat) -- tex format
\pmatrix{
1 & 3 \cr
2 & 4 \cr} () @ TextWriter
```

Example

We transform a `DenseMatrix(Integer)` into various formats:

```
%1 >> mat:DenseMatrix(Integer) := [[1,2],[3,4]]
matrix [[1,3],[2,4]] @ DenseMatrix(AldorInteger)
```

```
%2 >> exmat := extree(mat)
(matrix 2 2 1 3 2 4) @ ExpressionTree
```

```
%3 >> axiom(stdout,exmat) -- Axiom format
matrix [[1,3],[2,4]] () @ TextWriter
```

```
%4 >> maple(stdout,exmat) -- maple format
linalg[matrix](2,2,[1,3,2,4]) () @ TextWriter
```

```
%5 >> tex(stdout,exmat) -- tex format
\pmatrix{
1 & 3 \cr
2 & 4 \cr} () @ TextWriter
```

Example

We transform a `DenseMatrix(Integer)` into various formats:

```
%1 >> mat:DenseMatrix(Integer) := [[1,2],[3,4]]
matrix [[1,3],[2,4]] @ DenseMatrix(AldorInteger)
```

```
%2 >> exmat := extree(mat)
(matrix 2 2 1 3 2 4) @ ExpressionTree
```

```
%3 >> axiom(stdout,exmat) -- Axiom format
matrix [[1,3],[2,4]] () @ TextWriter
```

```
%4 >> maple(stdout,exmat) -- maple format
linalg[matrix](2,2,[1,3,2,4]) () @ TextWriter
```

```
%5 >> tex(stdout,exmat) -- tex format
\pmatrix{
1 & 3 \cr
2 & 4 \cr} () @ TextWriter
```

Parsing OpenMath (to Aldor) 1

- `OpenMath` \rightarrow `ExpressionTree` \Rightarrow Parsing OpenMath
- Must extend `OpenMath` with the function:
`parse! : % -> ExpressionTree`
- Naïve method : *traverse table of conversion methods*
- Complexity = $O(mn)$ where
 m = # categories of OpenMath objects
 n = size of object to be translated

Parsing OpenMath (to Aldor) 1

- `OpenMath` \rightarrow `ExpressionTree` \Rightarrow Parsing OpenMath
- Must extend `OpenMath` with the function:
`parse! : % -> ExpressionTree`
- Naïve method : *traverse table of conversion methods*
- Complexity = $O(mn)$ where
 m = # categories of OpenMath objects
 n = size of object to be translated

Parsing OpenMath (to Aldor) 1

- `OpenMath` \rightarrow `ExpressionTree` \Rightarrow Parsing OpenMath
- Must extend `OpenMath` with the function:
`parse! : % -> ExpressionTree`
- Naïve method : *traverse table of conversion methods*
- Complexity = $O(mn)$ where
 m = # categories of OpenMath objects
 n = size of object to be translated

Parsing OpenMath (to Aldor) 1

- `OpenMath` \rightarrow `ExpressionTree` \Rightarrow Parsing OpenMath
- Must extend `OpenMath` with the function:
`parse! : % -> ExpressionTree`
- Naïve method : *traverse table of conversion methods*
- Complexity = $O(mn)$ where
 m = # categories of OpenMath objects
 n = size of object to be translated

Parsing OpenMath (to Aldor) 2

- The parsing method we use:
Store functions on a hash table, where
 - *Keys – Characterisations of OpenMath objects*
 - *Values – functions with signature:*
 $\lambda : \text{OpenMath} \rightarrow \text{ExpressionTree}$
that implement the conversion method
- Parsing of an OpenMath object:
recursively apply functions from #table
- Complexity = $O(n)$, where
n = size of object to be translated

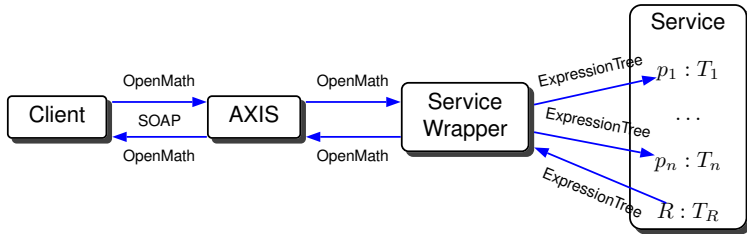
Parsing OpenMath (to Aldor) 2

- The parsing method we use:
Store functions on a hash table, where
 - *Keys – Characterisations of OpenMath objects*
 - *Values – functions with signature:*
 $\lambda : \text{OpenMath} \rightarrow \text{ExpressionTree}$
that implement the conversion method
- Parsing of an OpenMath object:
recursively apply functions from #table
- Complexity = $O(n)$, where
 n = size of object to be translated

Parsing OpenMath (to Aldor) 2

- The parsing method we use:
Store functions on a hash table, where
 - *Keys – Characterisations of OpenMath objects*
 - *Values – functions with signature:*
 $\lambda : \text{OpenMath} \rightarrow \text{ExpressionTree}$
that implement the conversion method
- Parsing of an OpenMath object:
recursively apply functions from #table
- Complexity = $O(n)$, where
 n = size of object to be translated

The Client Server Architecture



Theoretical and Practical Limitations

- Category considerations:
 - *It is important that types have the required Categories*
 - Parsable, ExpressionType
- Parsing OpenMath:
 - *The OpenMath domain must have translation to ExpressionTree capability*
 - *Translation function must exist on hash table*
- Function Objects:
 - *Function objects: first class objects*
 - *Basis of many Aldor Types:*
e.g. DenseUnivariateTaylorSeries

Theoretical and Practical Limitations

- Category considerations:
 - *It is important that types have the required Categories*
 - Parsable, ExpressionType
- Parsing OpenMath:
 - *The OpenMath domain must have translation to ExpressionTree capability*
 - *Translation function must exist on hash table*
- Function Objects:
 - *Function objects: first class objects*
 - *Basis of many Aldor Types:*
e.g. DenseUnivariateTaylorSeries

Theoretical and Practical Limitations

- Category considerations:
 - *It is important that types have the required Categories*
 - `Parsable`, `ExpressionType`
- Parsing `OpenMath`:
 - *The `OpenMath` domain must have translation to `ExpressionTree` capability*
 - *Translation function must exist on hash table*
- Function Objects:
 - *Function objects: first class objects*
 - *Basis of many Aldor Types:*
e.g. `DenseUnivariateTaylorSeries`

Function Objects

- Function objects are not members of domains:
Specialist code for function objects
- `ExpressionTree` framework not rich enough:
 - *Specialist code to translate directly between OpenMath and function objects*
 - *Extend descriptive mechanism (possibly take parameters)*
- Translation from Aldor to OpenMath:
Can't decompose functions in Aldor
 - *Apply to `ExpressionTree` variables: parameter types must be `ExpressionTree`*
 - *Interpolation of function:*
 - inefficient*
 - function must be interpolable*

Function Objects

- Function objects are not members of domains:
Specialist code for function objects
- `ExpressionTree` framework not rich enough:
 - *Specialist code to translate directly between OpenMath and function objects*
 - *Extend descriptive mechanism (possibly take parameters)*
- Translation from Aldor to OpenMath:
Can't decompose functions in Aldor
 - *Apply to `ExpressionTree` variables: parameter types must be `ExpressionTree`*
 - *Interpolation of function:*
 - inefficient*
 - function must be interpolable*

Function Objects

- Function objects are not members of domains:
Specialist code for function objects
- `ExpressionTree` framework not rich enough:
 - *Specialist code to translate directly between OpenMath and function objects*
 - *Extend descriptive mechanism (possibly take parameters)*
- Translation from Aldor to OpenMath:
Can't decompose functions in Aldor
 - *Apply to `ExpressionTree` variables: parameter types must be `ExpressionTree`*
 - *Interpolation of function:*
 - inefficient*
 - function must be interpolable*

The Service Manager

A service manager has been created, which allows:

- Creation of Aldor services
- Invocation of Aldor services
- Listing deployed services
- Removing deployed services

This paper is concerned with the first two points

Automatic MSDL Generation

- Can only do signatures (inputs, outputs).
pre-conditions, post-conditions require AI
processing of service
- Injective homomorphism from Aldor Types to OpenMath
Types
- Types in program: correct \Rightarrow MSDL: correct
- Use hashtable based $O(n)$ algorithm, where n is the size
of the (parameterised) type

Automatic MSDL Generation

- Can only do signatures (inputs, outputs).
pre-conditions, post-conditions require AI
processing of service
- Injective homomorphism from Aldor Types to OpenMath
Types
- Types in program: correct \Rightarrow MSDL: correct
- Use hashtable based $O(n)$ algorithm, where n is the size
of the (parameterised) type

Automatic MSDL Generation

- Can only do signatures (inputs, outputs).
pre-conditions, post-conditions require AI
processing of service
- Injective homomorphism from Aldor Types to OpenMath
Types
- Types in program: correct \Rightarrow MSDL: correct
- Use hashtable based $O(n)$ algorithm, where n is the size
of the (parameterised) type

Automatic MSDL Generation

- Can only do signatures (inputs, outputs).
pre-conditions, post-conditions require AI
processing of service
- Injective homomorphism from Aldor Types to OpenMath
Types
- Types in program: correct \Rightarrow MSDL: correct
- Use hashtable based $O(n)$ algorithm, where n is the size
of the (parameterised) type

Conclusion

- Presented a solution to delivering Aldor applications as web services
- Allows construction and invocation of bespoke services, using OpenMath for communication
- Particular problems resolved include:
 - Bridging the Untyped \leftrightarrow Strongly typed gap
 - Efficient parsing of OpenMath objects
- Automatic generation of MSDL

Creation of an Aldor Service 1/2

```

serviceWrap():() == {
    { $x_1, \dots, x_n$ } ← read OpenMath arguments from the default Input Stream
    { $E_1, \dots, E_n$ } ← convert { $x_1, \dots, x_n$ } to ExpressionTree
    ret:R ← service_code( $E_1, \dots, E_n$ ) — R is the return type of the service
code
    return openmath(stdout,extree(ret))
}
  
```

```

service_code( $E_1$  : ExpressionTree,  $\dots$ ,  $E_n$  : ExpressionTree):R == {
  import from Partial( $D_1$ )  $\dots$  Partial( $D_n$ ) and  $D_1 \dots D_n$ ,
    where  $D_1 \dots D_n$  are the arguments to the service code.
  
```

```

  { $e_1, \dots, e_n$ } ← {retract(eval( $E_1$ )) $\dots$ retract(eval( $E_n$ ))}
  
```

convert the ExpressionTree objects into objects of the specific types.

```

  the rest of the service code }
  
```

Creation of Aldor Service 2/2

input: `prog` – The service code

Extract the arguments and their types from the *interface function* of `prog`.

if The argument types are not of category `Parsable` **then**
 throw a `TypeNonParsable` exception, whose detail records the types which are not
 of category `Parsable`

end if

Extract the return type of the *interface function* of `prog`.

if The return type is not of category `ExpressionType` **then**
 throw a `TypeNonParsable` exception

end if

Build the program detailed on the previous slide where the arguments are those given as the parameters of `prog`

Compile the constructed program, and store the executable in the service database.