# Status of the New Backend

Karim Abdelhak, Bernhard Bachmann
University of Applied Sciences Bielefeld
Bielefeld, Germany

February 5, 2024

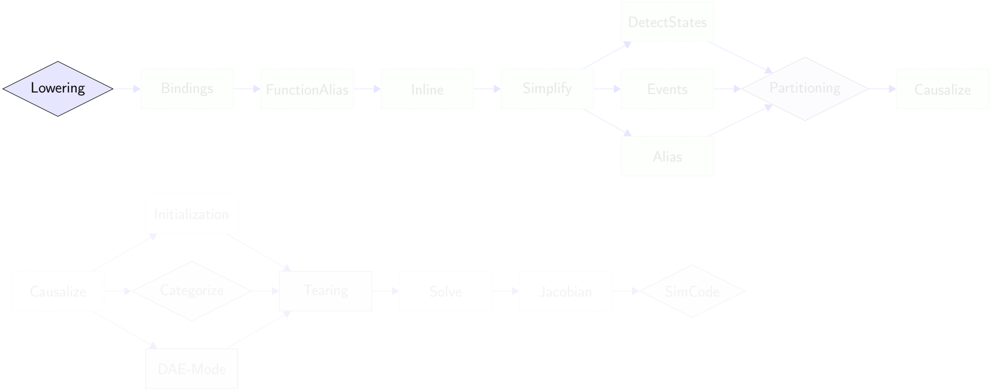
Hochschule Bielefeld
University of Applied Sciences and Arts

# Section 1

## Overview

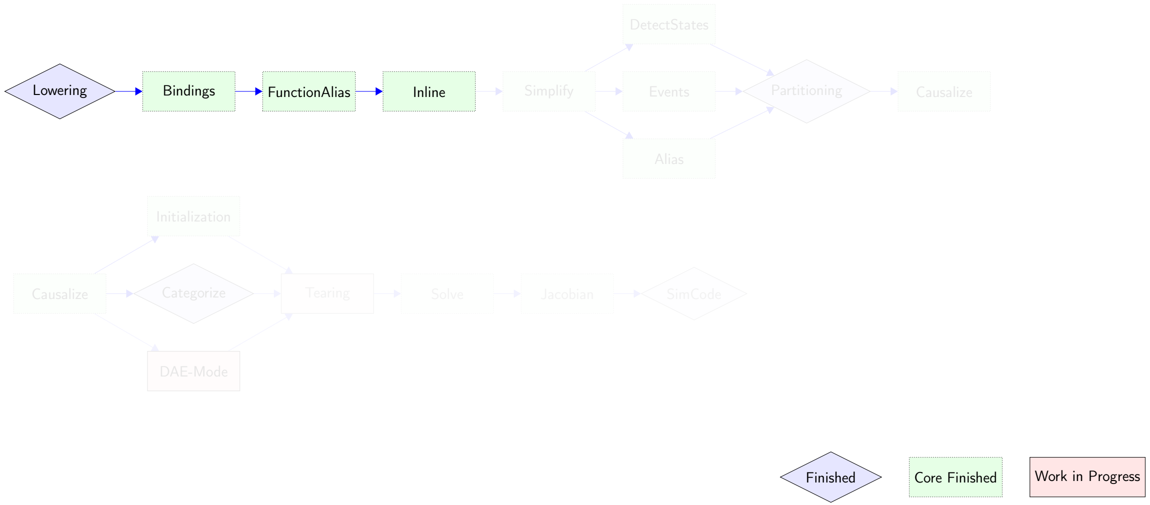
# Backend Modules

## Status on Array-Handling



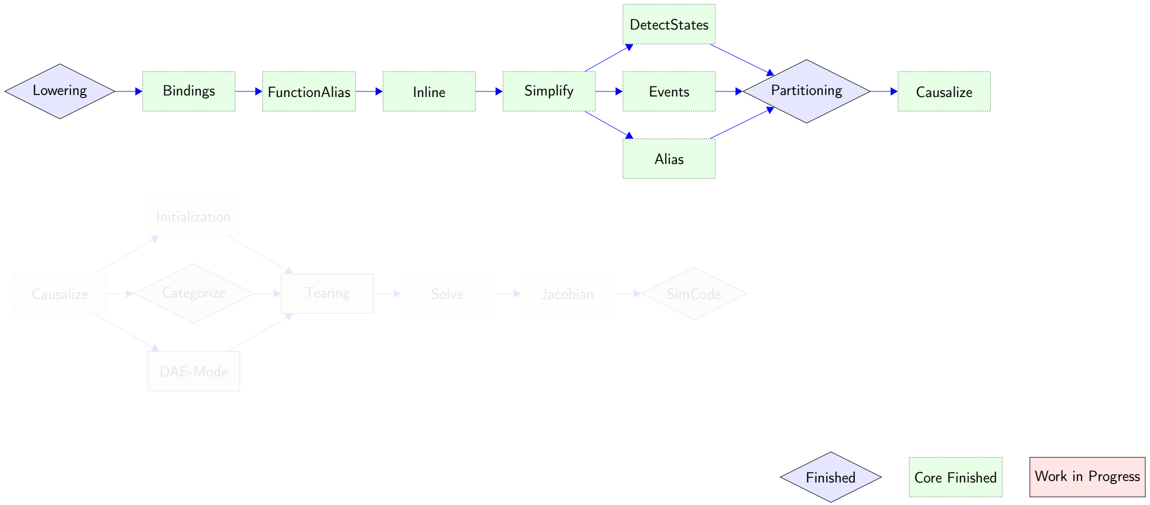

# Backend Modules

## Status on Array-Handling



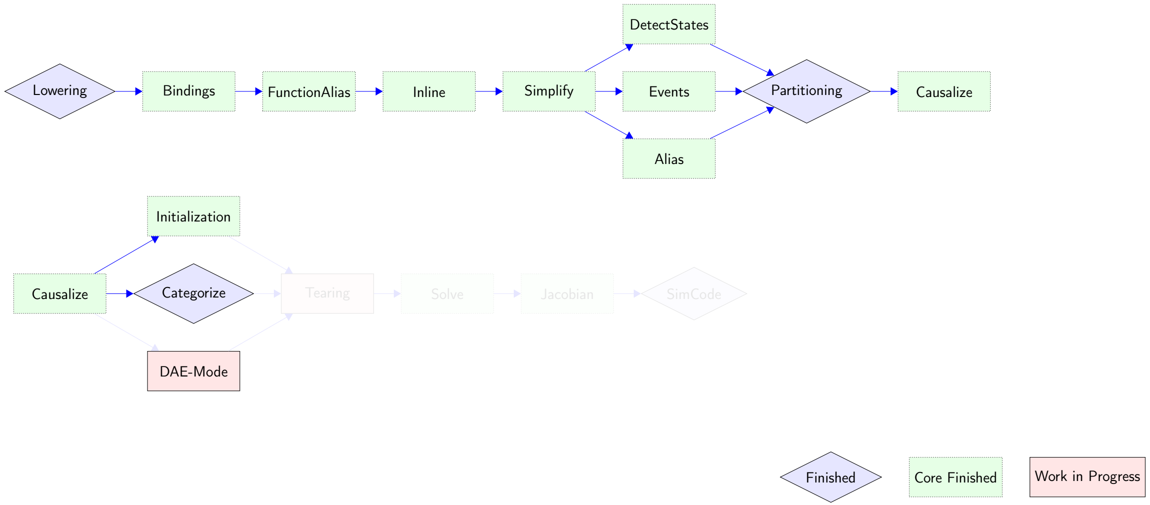
# Backend Modules

## Status on Array-Handling



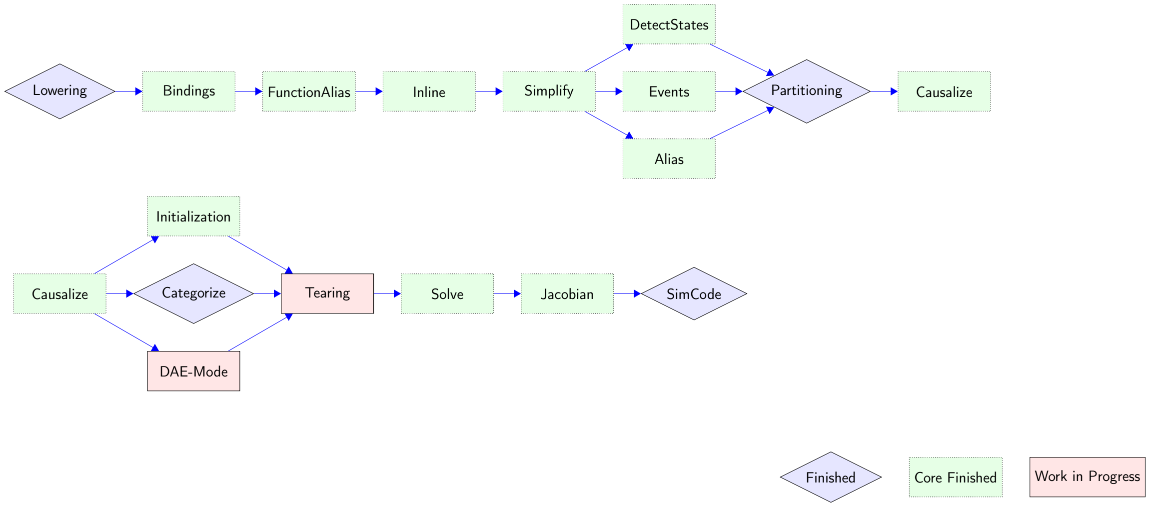
# Backend Modules

## Status on Array-Handling



# Backend Modules

## Status on Array-Handling



Section 2

# Bindings

# Creating Binding Equations

## Debugging

Flag: `-d=dumpBindings`

## Outline

1. Create binding equations for simulation (Bindings Module).
2. Create binding equations for initialization (Initialization Module).

## Challenges

- Correctly parse bindings of multi dimensional variables.
- Correctly parse record bindings. Some records are bound themselves, for some one needs to create binding equations for the elements.
- Correctly parse external object bindings (e.g. alias).

## Motivation

Mandatory to have a balanced model in the first place.

# Creating Binding Equations

## Debugging

Flag: `-d=dumpBindings`

## Outline

1. Create binding equations for simulation (Bindings Module).
2. Create binding equations for initialization (Initialization Module).

## Challenges

- Correctly parse bindings of multi dimensional variables.
- Correctly parse record bindings. Some records are bound themselves, for some one needs to create binding equations for the elements.
- Correctly parse external object bindings (e.g. alias).

## Motivation

Mandatory to have a balanced model in the first place.

# Creating Binding Equations

## Debugging

Flag: `-d=dumpBindings`

## Outline

1. Create binding equations for simulation (Bindings Module).
2. Create binding equations for initialization (Initialization Module).

## Challenges

- Correctly parse bindings of multi dimensional variables.
- Correctly parse record bindings. Some records are bound themselves, for some one needs to create binding equations for the elements.
- Correctly parse external object bindings (e.g. alias).

## Motivation

Mandatory to have a balanced model in the first place.

# Creating Binding Equations

### Debugging

Flag: `-d=dumpBindings`

### Outline

1. Create binding equations for simulation (Bindings Module).
2. Create binding equations for initialization (Initialization Module).

### Challenges

- Correctly parse bindings of multi dimensional variables.
- Correctly parse record bindings. Some records are bound themselves, for some one needs to create binding equations for the elements.
- Correctly parse external object bindings (e.g. alias).

### Motivation

Mandatory to have a balanced model in the first place.

# Creating Binding Equations

## Debugging

Flag: `-d=dumpBindings`

## Outline

1. Create binding equations for simulation (Bindings Module).
2. Create binding equations for initialization (Initialization Module).

## Challenges

- Correctly parse bindings of multi dimensional variables.
- Correctly parse record bindings. Some records are bound themselves, for some one needs to create binding equations for the elements.
- Correctly parse external object bindings (e.g. alias).

## Motivation

Mandatory to have a balanced model in the first place.

# Creating Binding Equations

## Debugging

Flag: `-d=dumpBindings`

## Outline

1. Create binding equations for simulation (Bindings Module).
2. Create binding equations for initialization (Initialization Module).

## Challenges

- Correctly parse bindings of multi dimensional variables.
- Correctly parse record bindings. Some records are bound themselves, for some one needs to create binding equations for the elements.
- Correctly parse external object bindings (e.g. alias).

## Motivation

Mandatory to have a balanced model in the first place.

Section 3

# Function Alias

# Introducing Function Alias

## Debugging

Flag: `-d=dumpCSE`

## Outline

1. Gathering and replacing all function calls in the model.
2. Creating the auxiliary equations for the replaced function calls.

## Challenges

- Only create a single alias for identical function calls.
- Wrap the auxiliary equations in the iterators (+when/if conditions) of the function call.
- Do not replace impure functions, inlineable functions and functions in algorithms that do not strictly depend on the inputs.
- Create multiple function alias variables and wrap them in a tuple if the function has multiple outputs.

# Introducing Function Alias

## Debugging

Flag: `-d=dumpCSE`

## Outline

1. Gathering and replacing all function calls in the model.
2. Creating the auxiliary equations for the replaced function calls.

## Challenges

- Only create a single alias for identical function calls.
- Wrap the auxiliary equations in the iterators (+when/if conditions) of the function call.
- Do not replace impure functions, inlineable functions and functions in algorithms that do not strictly depend on the inputs.
- Create multiple function alias variables and wrap them in a tuple if the function has multiple outputs.

# Introducing Function Alias

## Debugging

Flag: `-d=dumpCSE`

## Outline

1. Gathering and replacing all function calls in the model.
2. Creating the auxiliary equations for the replaced function calls.

## Challenges

- Only create a single alias for identical function calls.
- Wrap the auxiliary equations in the iterators (+when/if conditions) of the function call.
- Do not replace impure functions, inlineable functions and functions in algorithms that do not strictly depend on the inputs.
- Create multiple function alias variables and wrap them in a tuple if the function has multiple outputs.

# Introducing Function Alias

## Debugging

Flag: `-d=dumpCSE`

## Outline

1. Gathering and replacing all function calls in the model.
2. Creating the auxiliary equations for the replaced function calls.

## Challenges

- Only create a single alias for identical function calls.
- Wrap the auxiliary equations in the iterators (+when/if conditions) of the function call.
- Do not replace impure functions, inlineable functions and functions in algorithms that do not strictly depend on the inputs.
- Create multiple function alias variables and wrap them in a tuple if the function has multiple outputs.

# Introducing Function Alias

## Debugging

Flag: `-d=dumpCSE`

## Outline

1. Gathering and replacing all function calls in the model.
2. Creating the auxiliary equations for the replaced function calls.

## Challenges

- Only create a single alias for identical function calls.
- Wrap the auxiliary equations in the iterators (+when/if conditions) of the function call.
- Do not replace impure functions, inlineable functions and functions in algorithms that do not strictly depend on the inputs.
- Create multiple function alias variables and wrap them in a tuple if the function has multiple outputs.

# Introducing Function Alias

## Debugging

Flag: `-d=dumpCSE`

## Outline

1. Gathering and replacing all function calls in the model.
2. Creating the auxiliary equations for the replaced function calls.

## Challenges

- Only create a single alias for identical function calls.
- Wrap the auxiliary equations in the iterators (+when/if conditions) of the function call.
- Do not replace impure functions, inlineable functions and functions in algorithms that do not strictly depend on the inputs.
- Create multiple function alias variables and wrap them in a tuple if the function has multiple outputs.

# Introducing Function Alias

## Motivation

- Call identical function calls only once.
- Function calls in algebraic loops that don't depend on the iteration variables will be extracted entirely from the strong component to not be evaluated multiple times during the process of solving the algebraic loop.
- Function calls in algebraic loops that depend on iteration variables can be extracted to be only evaluated as torn inner equations when using proper tearing methods. This results in function calls never being part of a residual equation.
- If applied before the Inlining module it ensures that it can properly resolve all inlinable operator record functions.

# Introducing Function Alias

## Motivation

- Call identical function calls only once.
- Function calls in algebraic loops that don't depend on the iteration variables will be extracted entirely from the strong component to not be evaluated multiple times during the process of solving the algebraic loop.
- Function calls in algebraic loops that depend on iteration variables can be extracted to be only evaluated as torn inner equations when using proper tearing methods. This results in function calls never being part of a residual equation.
- If applied before the Inlining module it ensures that it can properly resolve all inlinable operator record functions.

# Introducing Function Alias

## Motivation

- Call identical function calls only once.
- Function calls in algebraic loops that don't depend on the iteration variables will be extracted entirely from the strong component to not be evaluated multiple times during the process of solving the algebraic loop.
- Function calls in algebraic loops that depend on iteration variables can be extracted to be only evaluated as torn inner equations when using proper tearing methods. This results in function calls never being part of a residual equation.
- If applied before the Inlining module it ensures that it can properly resolve all inlinable operator record functions.

# Introducing Function Alias

## Motivation

- Call identical function calls only once.
- Function calls in algebraic loops that don't depend on the iteration variables will be extracted entirely from the strong component to not be evaluated multiple times during the process of solving the algebraic loop.
- Function calls in algebraic loops that depend on iteration variables can be extracted to be only evaluated as torn inner equations when using proper tearing methods. This results in function calls never being part of a residual equation.
- If applied before the Inlining module it ensures that it can properly resolve all inlinable operator record functions.

## Structures for Function Alias

```
record CALL_ID
  Expression call;
  Iterator iter;
  // Option<Expression> when_condition
  // Option<Expression> if_condition
end CALL_ID;

record CALL_AUX
  Expression replacer;
  EquationKind kind;
  Boolean parsed;
end CALL_AUX;

UnorderedMap<Call_Id, Call_Aux> map;
```

# Structures for Function Alias

```
record CALL_ID
  Expression call;
  Iterator iter;
  // Option<Expression> when_condition
  // Option<Expression> if_condition
end CALL_ID;

record CALL_AUX
  Expression replacer;
  EquationKind kind;
  Boolean parsed;
end CALL_AUX;

UnorderedMap<Call_Id, Call_Aux> map;
```

## Structures for Function Alias

```
record CALL_ID
  Expression call;
  Iterator iter;
  // Option<Expression> when_condition
  // Option<Expression> if_condition
end CALL_ID;

record CALL_AUX
  Expression replacer;
  EquationKind kind;
  Boolean parsed;
end CALL_AUX;

UnorderedMap<Call_Id, Call_Aux> map;
```

Section 4

# Inline

# Inlining Function Calls

## Debugging

Flag: `-d=dumpBackendInline`

## Main Outline

1. Collecting all inlineable functions from the function tree (+native functions).
2. Inline inlineable functions in all equations.
3. Inline all record constructors and tuple equations.
4. Additional functionality: Inline for-equations with iterators of size 1.

## Function Inline Outline

1. The input variables of the call have to be mapped to the input variables of the interface.
2. If any input variables are records, the mapping has to be extended to their record elements.
3. The bindings of local variables have to be evaluated using the existing input mapping. Furthermore, the local variables and their evaluated bindings have to be added to the mapping.

# Inlining Function Calls

## Debugging

Flag: `-d=dumpBackendInline`

## Main Outline

1. Collecting all inlineable functions from the function tree (+native functions).
2. Inline inlineable functions in all equations.
3. Inline all record constructors and tuple equations.
4. Additional functionality: Inline for-equations with iterators of size 1.

## Function Inline Outline

1. The input variables of the call have to be mapped to the input variables of the interface.
2. If any input variables are records, the mapping has to be extended to their record elements.
3. The bindings of local variables have to be evaluated using the existing input mapping. Furthermore, the local variables and their evaluated bindings have to be added to the mapping.

# Inlining Function Calls

## Debugging

Flag: `-d=dumpBackendInline`

## Main Outline

1. Collecting all inlineable functions from the function tree (+native functions).
2. Inline inlineable functions in all equations.
3. Inline all record constructors and tuple equations.
4. Additional functionality: Inline for-equations with iterators of size 1.

## Function Inline Outline

1. The input variables of the call have to be mapped to the input variables of the interface.
2. If any input variables are records, the mapping has to be extended to their record elements.
3. The bindings of local variables have to be evaluated using the existing input mapping. Furthermore, the local variables and their evaluated bindings have to be added to the mapping.

# Inlining Function Calls

## Debugging

Flag: `-d=dumpBackendInline`

## Main Outline

1. Collecting all inlineable functions from the function tree (+native functions).
2. Inline inlineable functions in all equations.
3. Inline all record constructors and tuple equations.
4. Additional functionality: Inline for-equations with iterators of size 1.

## Function Inline Outline

1. The input variables of the call have to be mapped to the input variables of the interface.
2. If any input variables are records, the mapping has to be extended to their record elements.
3. The bindings of local variables have to be evaluated using the existing input mapping. Furthermore, the local variables and their evaluated bindings have to be added to the mapping.

# Inlining Function Calls

## Debugging

Flag: `-d=dumpBackendInline`

## Main Outline

1. Collecting all inlineable functions from the function tree (+native functions).
2. Inline inlineable functions in all equations.
3. Inline all record constructors and tuple equations.
4. Additional functionality: Inline for-equations with iterators of size 1.

## Function Inline Outline

1. The input variables of the call have to be mapped to the input variables of the interface.
2. If any input variables are records, the mapping has to be extended to their record elements.
3. The bindings of local variables have to be evaluated using the existing input mapping. Furthermore, the local variables and their evaluated bindings have to be added to the mapping.

# Inlining Function Calls

## Debugging

Flag: `-d=dumpBackendInline`

## Main Outline

1. Collecting all inlineable functions from the function tree (+native functions).
2. Inline inlineable functions in all equations.
3. Inline all record constructors and tuple equations.
4. Additional functionality: Inline for-equations with iterators of size 1.

## Function Inline Outline

1. The input variables of the call have to be mapped to the input variables of the interface.
2. If any input variables are records, the mapping has to be extended to their record elements.
3. The bindings of local variables have to be evaluated using the existing input mapping. Furthermore, the local variables and their evaluated bindings have to be added to the mapping.

# Inlining Function Calls

## Debugging

Flag: `-d=dumpBackendInline`

## Main Outline

1. Collecting all inlineable functions from the function tree (+native functions).
2. Inline inlineable functions in all equations.
3. Inline all record constructors and tuple equations.
4. Additional functionality: Inline for-equations with iterators of size 1.

## Function Inline Outline

1. The input variables of the call have to be mapped to the input variables of the interface.
2. If any input variables are records, the mapping has to be extended to their record elements.
3. The bindings of local variables have to be evaluated using the existing input mapping. Furthermore, the local variables and their evaluated bindings have to be added to the mapping.

# Inlining Function Calls

## Challenges

- Define what *inlineable* means.

## Motivation

- Make the inlined function body susceptible for symbolic manipulation.
- Remove most record equations and remove all tuple equations.
- Correctly handle ignored outputs.

# Inlining Function Calls

## Challenges

- Define what *inlineable* means.

## Motivation

- Make the inlined function body susceptible for symbolic manipulation.
- Remove most record equations and remove all tuple equations.
- Correctly handle ignored outputs.

# Inlining Function Calls

## Challenges

- Define what *inlineable* means.

## Motivation

- Make the inlined function body susceptible for symbolic manipulation.
- Remove most record equations and remove all tuple equations.
- Correctly handle ignored outputs.

# Inlining Function Calls

## Challenges

- Define what *inlineable* means.

## Motivation

- Make the inlined function body susceptible for symbolic manipulation.
- Remove most record equations and remove all tuple equations.
- Correctly handle ignored outputs.

# Example: Inlining operator record functions

Considering following *Modelica* model record_inlining three record functions will be inlined:

1. The record constructor Complex.'constructor'.fromReal
2. The overloaded operator Complex.'*'.multiply
3. The overloaded operator Complex.'^'

```
model record_inlining
  Complex a,b,c,d;
equation
  a = Complex(sin(time), cos(time));
  b = Complex(time, tan(time));
  c = a * b;
  d = a ^ b;
end record_inlining;
```

# Example: Inlining operator record functions

Considering following *Modelica* model `record_inlining` three record functions will be inlined:

1. The record constructor `Complex.'constructor'.fromReal`
2. The overloaded operator `Complex.'*'.multiply`
3. The overloaded operator `Complex.'^'`

```
model record_inlining
  Complex a,b,c,d;
equation
  a = Complex(sin(time), cos(time));
  b = Complex(time, tan(time));
  c = a * b;
  d = a ^ b;
end record_inlining;
```

# Example: Inlining operator record functions

## Inlining the multiplication operator for complex numbers

```
encapsulated operator '*' "Multiplication"
  function multiply "Multiply two complex numbers"
    import Complex;
    input Complex c1 "Complex number 1";
    input Complex c2 "Complex number 2";
    output Complex c3 "= c1*c2";
  algorithm
    c3 := Complex(c1.re*c2.re - c1.im*c2.im, c1.re*c2.im +
        c1.im*c2.re);
    annotation(Inline=true);
  end multiply;
end '*';
```

# Example: Inlining operator record functions

## Inlining the multiplication operator for complex numbers

```
Inlining: Complex.'*'.multiply(a, b)
  -- Result: Complex.'constructor'.fromReal(a.re * b.re - a.im
      * b.im, a.re * b.im + a.im * b.re)

Inlining: [RECD] (2) c = Complex.'constructor'.fromReal(a.re *
    b.re - a.im * b.im, a.re * b.im + a.im * b.re)
  -- Result: [SCAL] (1) c.re = a.re * b.re - a.im * b.im
  -- Result: [SCAL] (1) c.im = a.re * b.im + a.im * b.re
```

# Example: Inlining operator record functions

## Inlining the multiplication operator for complex numbers

```
Inlining: Complex.'*'.multiply(a, b)
  -- Result: Complex.'constructor'.fromReal(a.re * b.re - a.im
       * b.im, a.re * b.im + a.im * b.re)

Inlining: [RECD] (2) c = Complex.'constructor'.fromReal(a.re *
     b.re - a.im * b.im, a.re * b.im + a.im * b.re)
  -- Result: [SCAL] (1) c.re = a.re * b.re - a.im * b.im
  -- Result: [SCAL] (1) c.im = a.re * b.im + a.im * b.re
```

# Example: Inlining operator record functions

## Inlining the power operator for complex numbers

```
encapsulated operator function '^'
  "Complex power of complex number"
  import Complex;
  input Complex c1 "Complex number";
  input Complex c2 "Complex exponent";
  output Complex c3 "= c1^c2";
protected
  Real lnz=0.5*log(c1.re*c1.re + c1.im*c1.im);
  Real phi=atan2(c1.im, c1.re);
  Real re=lnz*c2.re - phi*c2.im;
  Real im=lnz*c2.im + phi*c2.re;
algorithm
  c3 := Complex(exp(re)*cos(im), exp(re)*sin(im));
  annotation(Inline=true);
end '^';
```

# Example: Inlining operator record functions

## Inlining the power operator for complex numbers

```
Inlining: Complex.'^'(a, b)
 -- Result: Complex.'constructor'.fromReal(exp(0.5 * log(a.re
    * a.re + a.im * a.im) * b.re - atan2(a.im, a.re) * b.im) *
     cos(0.5 * log(a.re * a.re + a.im * a.im) * b.im + atan2(
    a.im, a.re) * b.re), exp(0.5 * log(a.re * a.re + a.im *
    a.im) * b.re - atan2(a.im, a.re) * b.im) * sin(0.5 * log(
    a.re * a.re + a.im * a.im) * b.im + atan2(a.im, a.re) *
    b.re))
```

Section 5

# Summary

# Results

- Overview
- Large TestSuite NB OB
- Recent Coverage Scalable TestSuite PowerGrids

# Summary

## Recent Development

- Bindings (+Initialization) Module
- FunctionAlias Module
- Inline Module

## Current Development

- Adjacency Matrix Improvements (+Tearing)
- Enable Sparse Solvers

## Upcoming Plans

- Pseudo-Array Index Reduction
- Resizable Arrays after Compilation

# Summary

## Recent Development

- Bindings (+Initialization) Module
- FunctionAlias Module
- Inline Module

## Current Development

- Adjacency Matrix Improvements (+Tearing)
- Enable Sparse Solvers

## Upcoming Plans

- Pseudo-Array Index Reduction
- Resizable Arrays after Compilation

# Summary

## Recent Development

- Bindings (+Initialization) Module
- FunctionAlias Module
- Inline Module

## Current Development

- Adjacency Matrix Improvements (+Tearing)
- Enable Sparse Solvers

## Upcoming Plans

- Pseudo-Array Index Reduction
- Resizable Arrays after Compilation