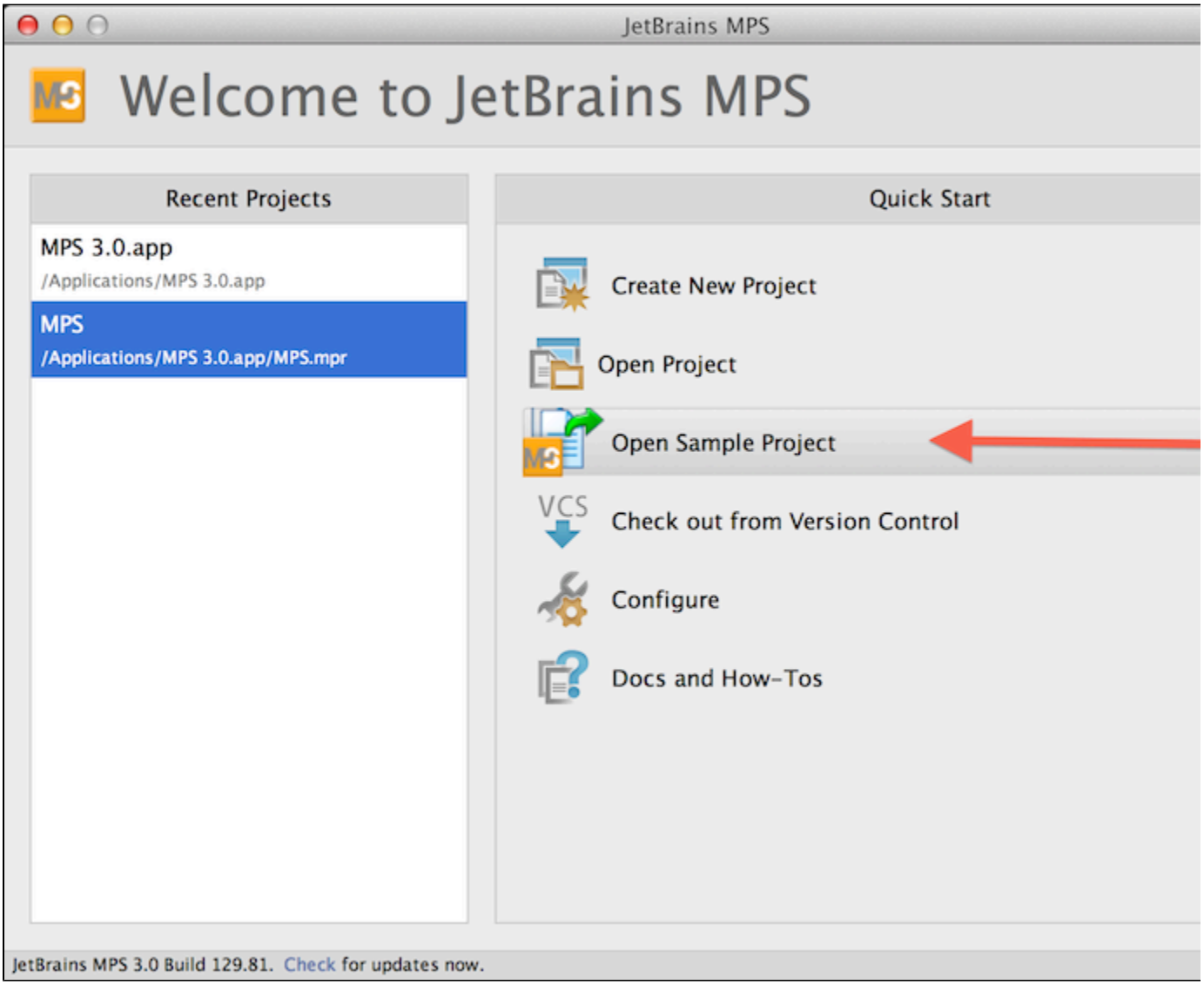
# Typesystem Debugging

For debugging typesystem MPS provides Typesystem Trace - an integrated visual tool that gives you insight into the evaluation process that happens inside the typesystem engine.

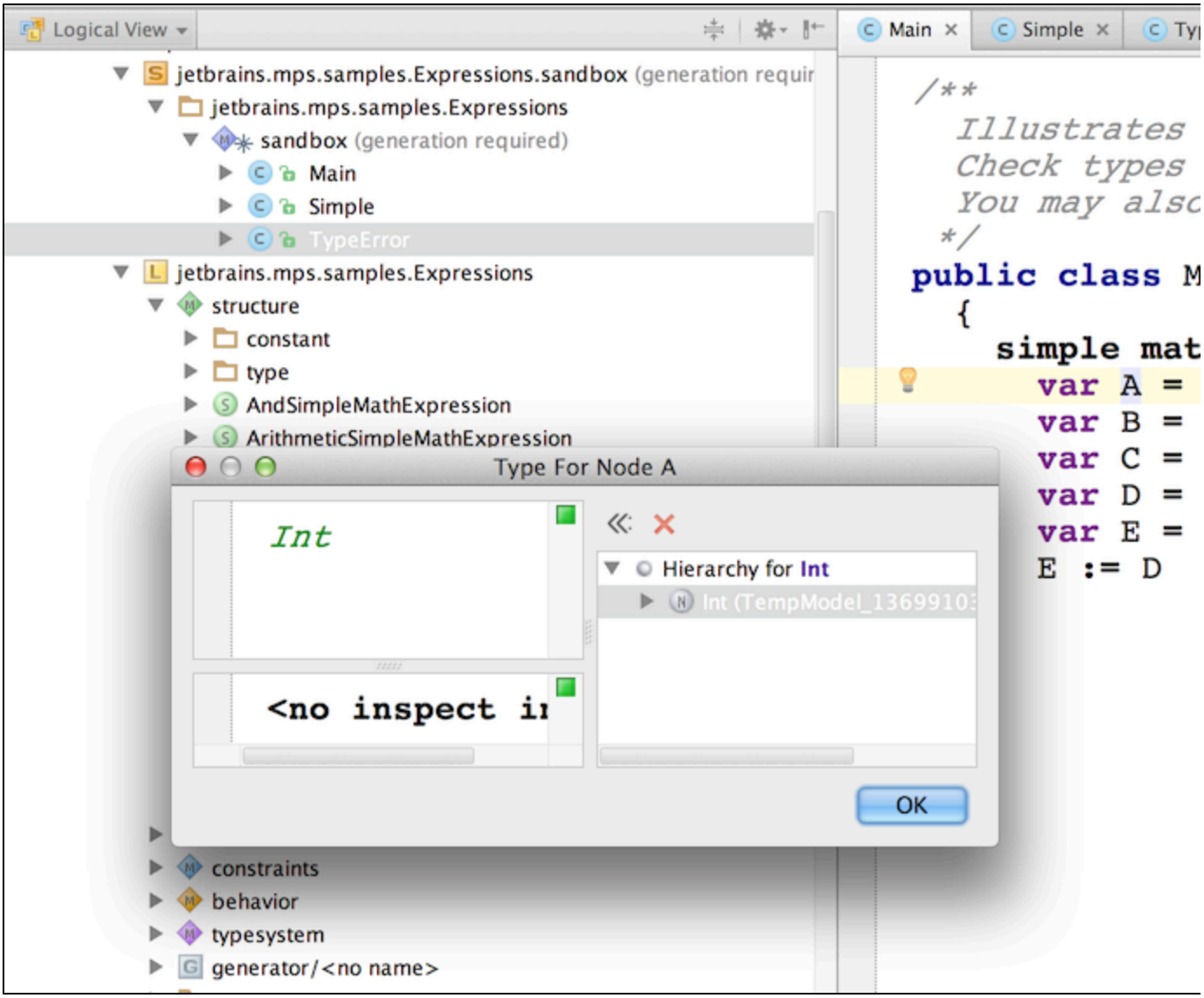
## Try it out for yourself

We prepared a dedicated sample language for you to easily experiment with the typesystem. Open the Expressions sample project that comes bundled with MPS and should be available among the sample projects in the user home folder.




### The sample language

The language to experiment with is a simplified expression language with several types, four arithmetic operations (+, -, *, /), assignment (:=), two types of variable declarations and a variable reference. The editor is very basic with almost no customization, so editing the expressions will perhaps be quite rough. Nevertheless, we expect you to inspect the existing samples and debug their types more than writing new code, so the lack of smooth editing should not be an issue.



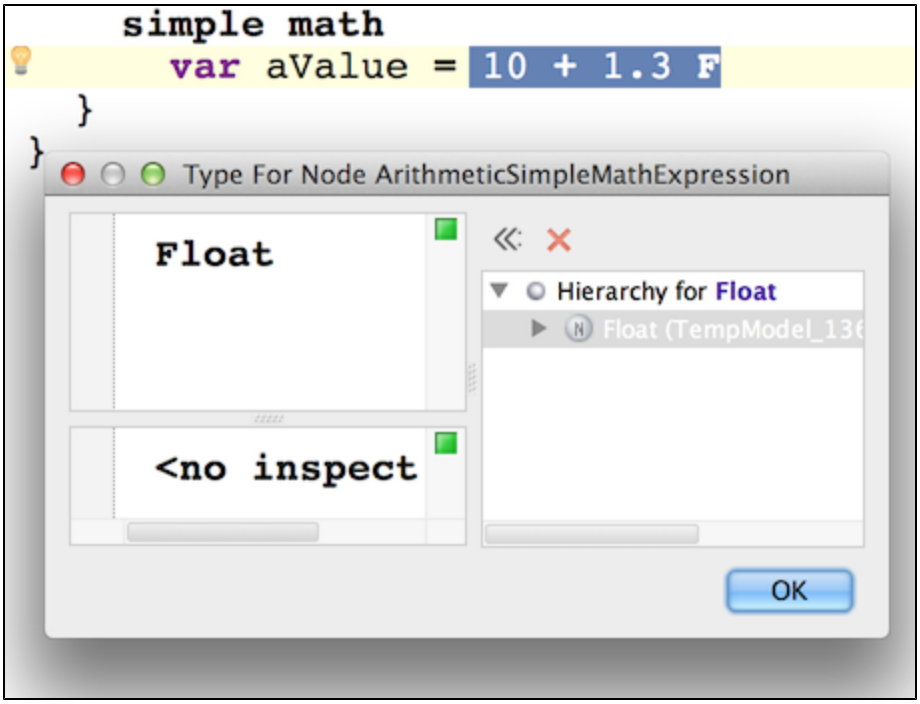

The language can be embedded into Java thanks to the SimpleMathWrapper concept, but no interaction between the language and BaseLanguage is possible.

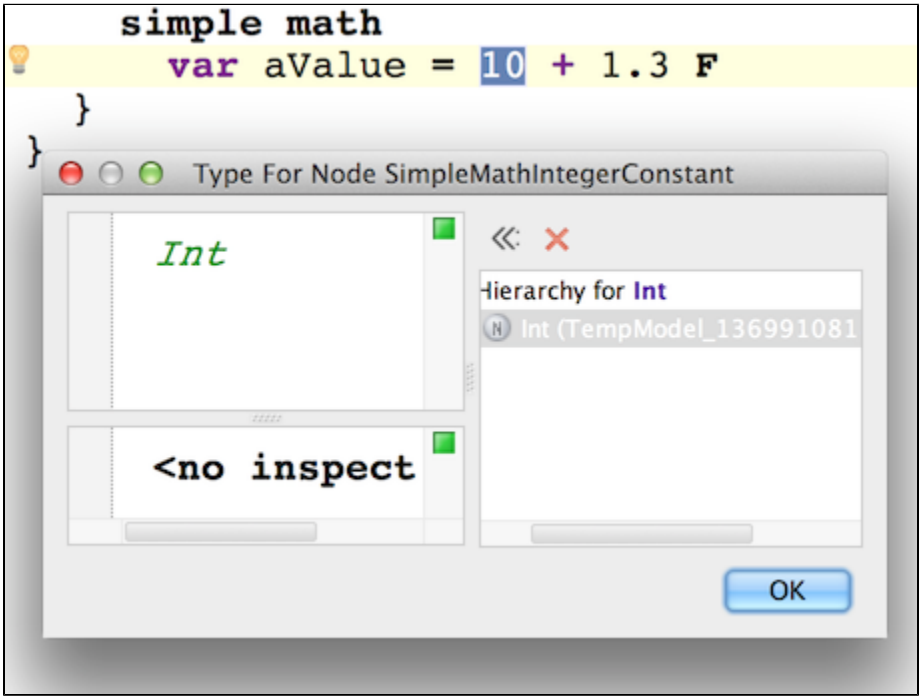
The expression language supports six types, organized by subtyping rules into two branches:

1. Element -> Number -> Float -> Long -> Int
2. Element -> Bool

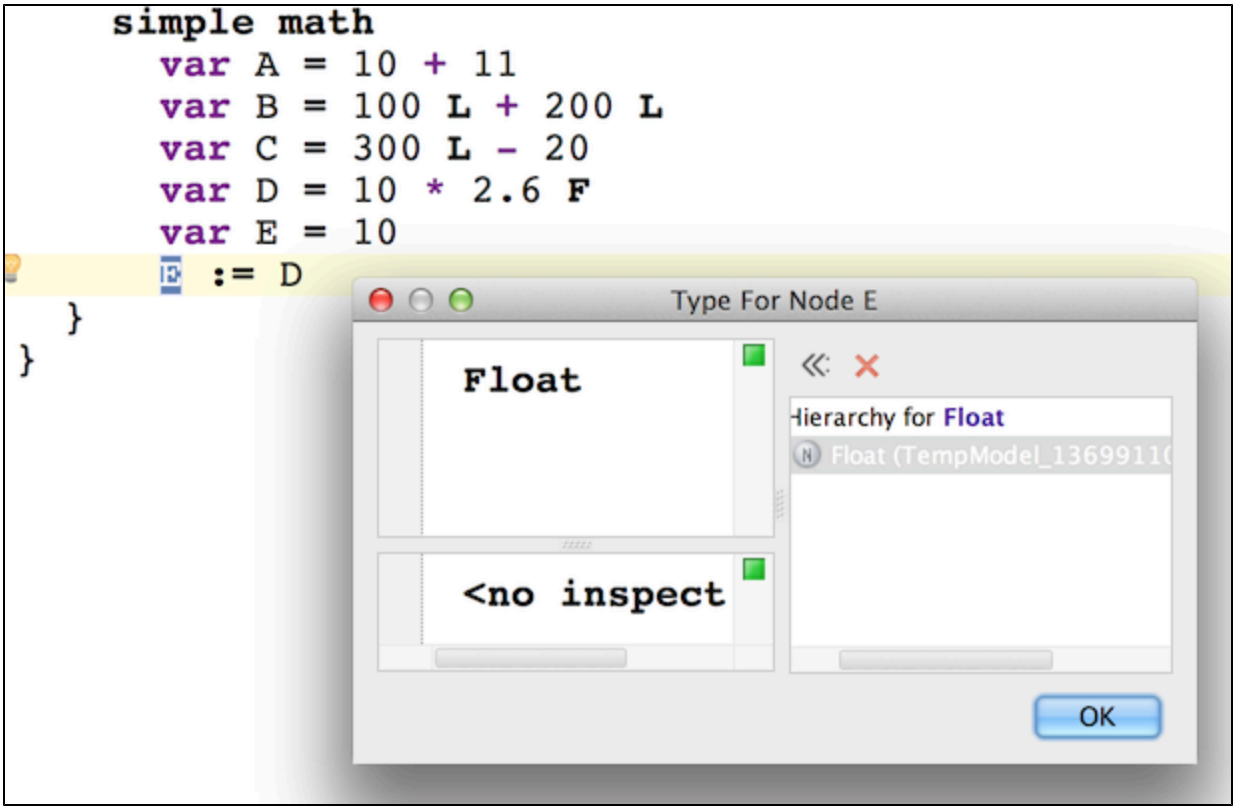
## Inspecting the types

If you open the Simple example class, you can position the cursor to any part of the expression or select a valid expression block. As soon as you hit Control/Cmd +Shift + T, you'll see the type of the selected node in a pop-up dialog.



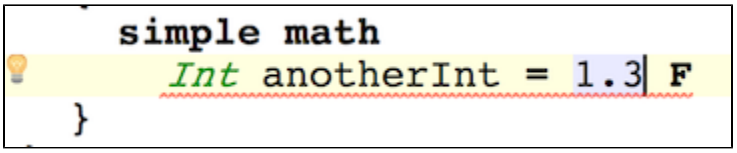

The Main sample class will give you a more involved example showing how Type-inference correctly propagates the suitable type to variables:



Just check the calculated types for yourself.

## Type errors

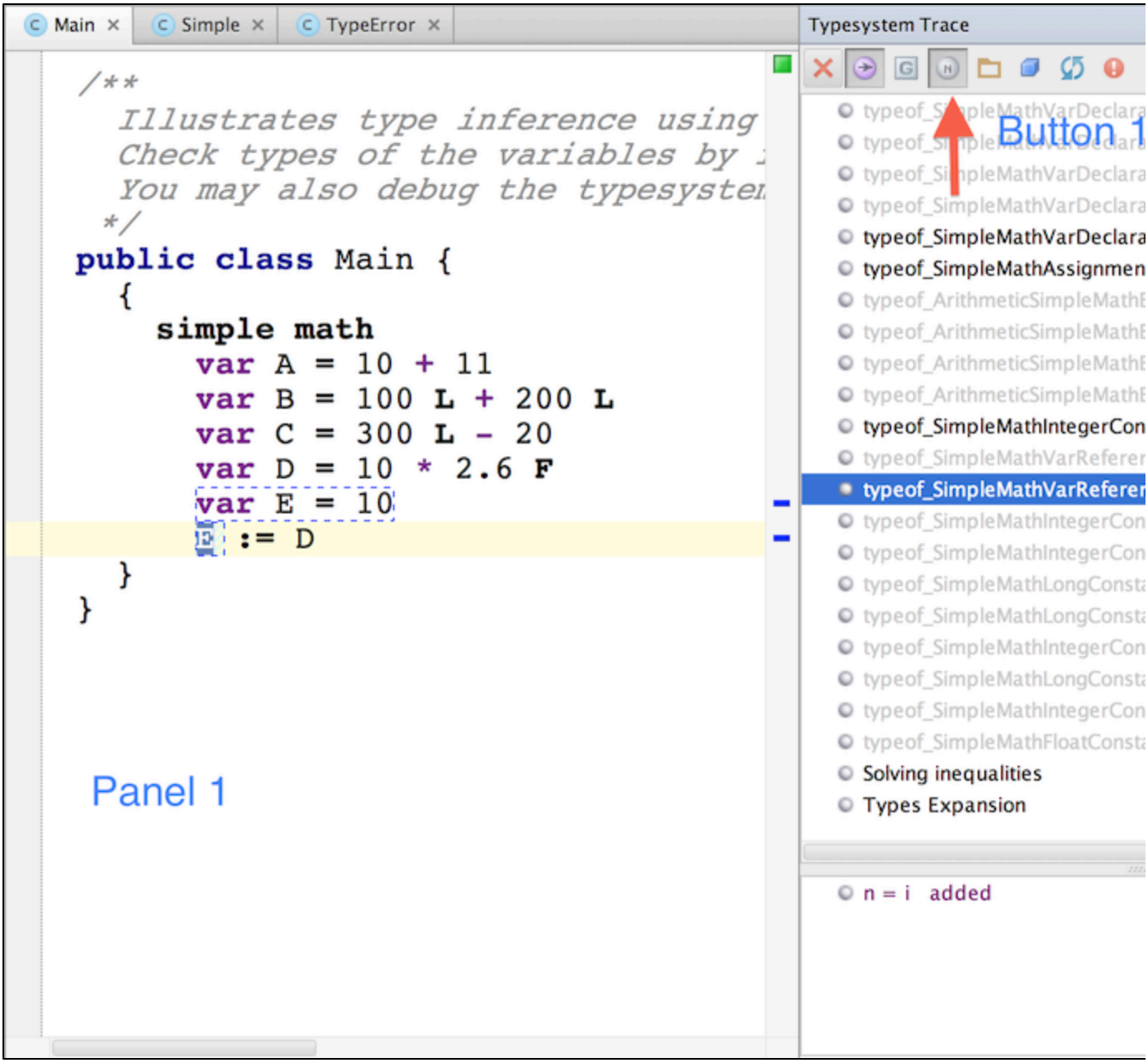
The TypeError sample class shows a simple example of a type error. Just uncomment the code (Control/Cmd + /) and check the reported error:




Since this variable declaration declares its type explicitly to be an Int, while the initializer is of type Float, the type-system reports an error. You may check the status bar at the bottom or hover your mouse over the incorrect piece of code.

## Type-system Trace

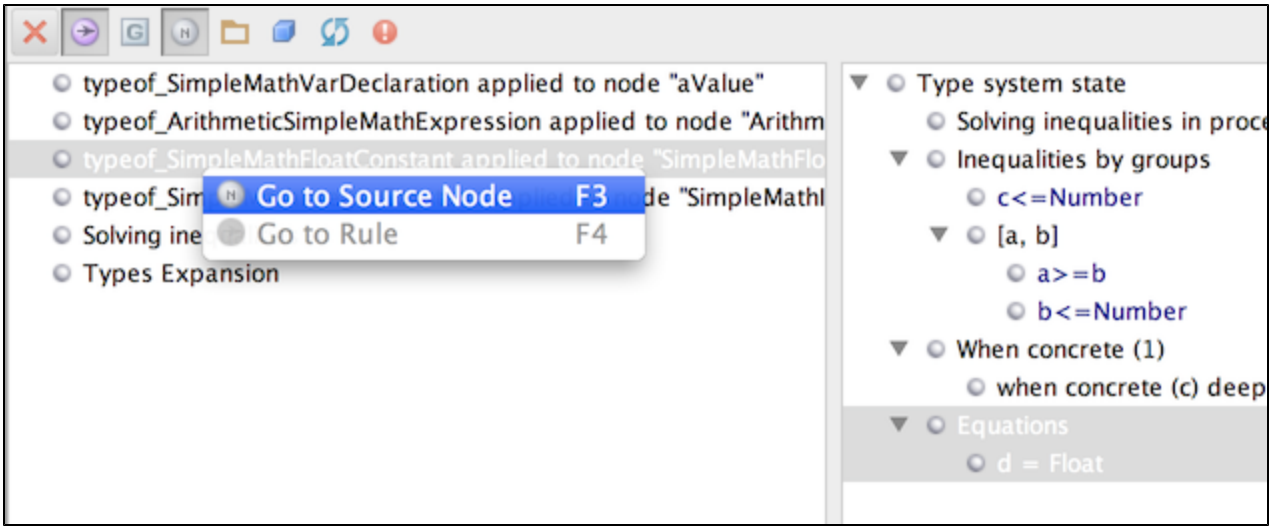
When you hit Control/Cmd + Shift + X or navigate through the pop-up menu, you get the Typesystem Trace panel displayed on the right hand-side.



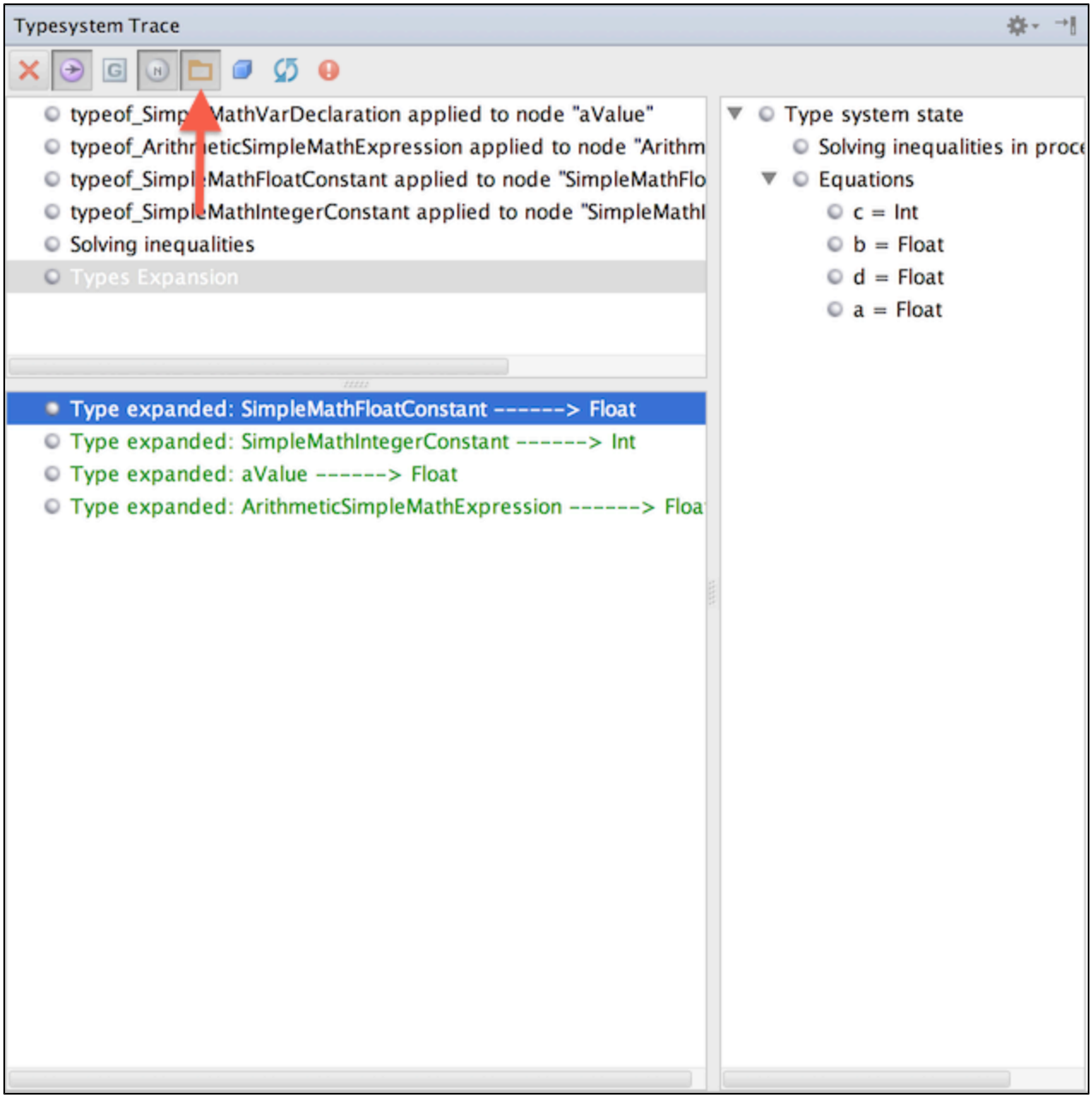

The Trace shows in Panel 2 all steps (i.e. type system rules) that the type-system engine executed. The steps are ordered top-to-bottom in the order in which they were performed. When you have Button 1 selected, the Panel 2 highlights the steps that directly or indirectly influence the type of the node selected in the editor (Panel 1). Panel 3 details the step selected in Pan el 2 - it describes what changes have been made to the type-system engine's state in the step. The actual state of the engine's working memory is displayed in Panel 4.

## Step-by-step debugging

The Simple sample class is probably the easiest one to start experimenting with. The types get resolved in six steps, following the typesystem rules specified in the language. You may want to refer to these rules quickly by pressing F4 or using the Contro l/Cmd + N "Go to Root Node" command. F3 will navigate you to the node, which is being affected by the current rule.




1. The type of a variable declaration has to be a super-type of the type of the initializer. The aValue variable is assigned the a type-system variable, the initializer expression is assigned the b type-system variable and a>=b (b sub-type or equal type to a) is added into the working memory.
2. Following the type-system rule for Arithmetic Expressions, b has to be a sub-type of Number, the value 10 is assigned the c variable, 1.3F is assigned the d variable and a when-concrete handler is added to wait for c to be calculated.
3. Following the rules for float constants d is solved as Float.
4. Following the rules for integer constants c is solved as Int. This triggers the when-concrete handler registered in step 2 and registered another when-concrete handler to wait for d. Since d has already been resolved to Float, the handler triggered and resolves b (the whole arithmetic expression) as Float. This also solves the earlier equation (step 2) that b <=Number.
5. Now a can be resolved as Float, which also solves the step 1 equation that a>=b.
6. If you enable type expansions by pressing the button in the tool-bar, you'll get the final expansions of all nodes to concrete types as the last step.


Typesystem Trace
typeof_Simp MathVarDeclaration applied to node "aValue"
typeof_Arithmetic SimpleMathExpression applied to node "Arithm
typeof_Simple MathFloatConstant applied to node "SimpleMathFlo
typeof_SimpleMathIntegerConstant applied to node "SimpleMathI
Solving inequalities
Types Expansion
Type expanded: SimpleMathFloatConstant ------> Float
Type expanded: SimpleMathIntegerConstant ------> Int
Type expanded: aValue ------> Float
Type expanded: ArithmeticSimpleMathExpression ------> Floa
Type system state
Solving inequalities in proce
Equations
c = Int
b = Float
d = Float
a = Float