# Demo Outline

## Use of Shadow Models for KernelF2

The demo will focus on one of the use cases, KernelF2 (discussed in Section 2.1); showing demos of all three use cases would result in too little detail for each one.

The next code snippet shows a selection of the KernelF2 base language concepts. We define a couple of values and use various operators and types. Functions, generics and algebraic data types are demonstrated as well.

```
module BaseLanguage {

  val x1 = 10
  val x2 : int = 20
  val x3 = 10 * 20
  val x4 = x2 > x1
  val x5 = "Hello, " + "World"
  val x6 = true == x4
  val x7 = true && true
  val x8 : int = if x2 > x1
                 then 0
                 else 1

  fun add(i: int, j: int) = i + j

  val x9 = add(30, 40)

  fun<T> id(v: T) : T = v
  val x10 : int = if id(true)
                  then id(2)
                  else id(1)

  algebraic Maybe = | Some(int)
                    | None()

  fun extract(m: Maybe, v: int) : int = match(m)
                                          [Some(_@n)  => n]
                                          [           => v]

}
```

Some of the expressions, such as `true && true` or `10 * 20` could be evaluated statically; and indeed, there is a Shadow Models transformation for the KernelF2 base language that performs such simplifications:

```
module BaseLanguage {

  val x1 = 10
  val x2 : int = 20
  val x3 = 200
  val x4 = x2 > x1
  val x5 = "Hello, World"
  val x6 = true == x4
  val x7 = true
  val x8 : int = if x2 > x1
                 then 0
                 else 1

  fun add(i: int, j: int) = i + j

  val x9 = add(30, 40)

  fun<T> id(v: T) : T = v
  val x10 : int = if id(true)
                  then id(2)
                  else id(1)

  algebraic Maybe = | Some(int)
                    | None()

  fun extract(m: Maybe, v: int) : int = match(m)
                                          [Some(_@n)  => n]
                                          [           => v]

}
```

Now we demonstrate language extension and desugaring. This shows three of the extension constructs defined in the `kernelf2.sugar` language: `enums`, `alt` expressions and decision tables. These are not part of the core, but defined in a modular extension; their semantics is defined through a reduction to the base language. `?maybe?` is a Boolean expression that we have added for demo purposes to avoid making up arbitrary Boolean expressions all the time.

```
module Extensions {

  enum Color { red, green, yellow, yellow }

  val aColor : Color = red

  fun decide(a: int, b: int) = alt ⎡ a > b      => 1 ⎤
                                   ⎢ a == b     => 2 ⎥
                                   ⎣ otherwise  => 3 ⎦

  val res =           | ?maybe? | ?maybe? | ?maybe?
              ?maybe? | 1       | 3       | 5
              ?maybe? | 2       | 4       | 6

}
```

Below is the reduced version of the previous code: `enum`s become constants that follow a particular naming convention, the `alt` expression becomes nested `if`s, and the decision table is translated to nested `alt` expressions which are then in turn reduced to `if`s. Notice how the reduced version contains errors that are produced by type checks of the base language: declarations must have unique names and the `<!>` expression must never show up. In this case, `<!>` is produced by the transformation because the decision table has these `?maybe?` expressions, so the type system cannot statically figure out whether the table is complete and free from overlaps. **In the demo, I will of course show how changes are propagated incrementally.**

```
module Extensions {

  val Color_red = -309736043
  val Color_green = -1313361145
  val Color_yellow = -1555955472
  val Color_yellow = -1555955472

  val aColor : int = Color_red

  fun decide(a: int, b: int) = if a > b
                               then 1
                               else if a == b then 2 else 3

  val res = if ?maybe?
            then if ?maybe?
                 then 1
                 else if ?maybe? then 2 else <!>
            else if ?maybe?
                 then if ?maybe?
                      then 3
                      else if ?maybe? then 4 else <!>
                 else if ?maybe?
                      then if ?maybe?
                           then 5
                           else if ?maybe? then 6 else <!>
                      else <!>
}
```

Below we can see an error message in the source model that is lifted from the shadow. No analysis happens on the level of the `enum` declaration itself. Lifting errors is a core use case for Shadow Models.

```
module Extensions {
                                         [LIFTED] this is the duplicate
  enum Color { red, green, yellow, yellow }

  fun decide(a: int, b: int) = alt ⎡ a > b      => 1 ⎤
                                   ⎢ a == b     => 2 ⎥
                                   ⎣ otherwise  => 3 ⎦

  val res =           | ?maybe? | ?maybe? | ?maybe?
              ?maybe? | 1       | 3       | 5
              ?maybe? | 2       | 4       | 6

}
```

# Implementation - Simplifications

Here is the entry point to the transformation; it contributes to a predefined transformation that attaches the results of transformations to the shadow repository, a Shadow-Models internal data structure that is also visualized in the IDE. This contribution iterates over all the models in MPS that contain a `Module` (the root concept of KernelF2, see examples above) and transforms each of them through the fork `moduleFork`. It copies its input while applying two transformations (declared elsewhere) to a fixpoint.

```
namespace Repo {

  transformation t0 contributes to ShadowRepository.Repository [ i0: Repository ]
    [<< ... >>]
  -> [ o0: Repository {
           modules: Module {
             name: "kf2"
             models: map _.modules.models.where({~it => it.rootNodes.ofConcept<Module>.isNotEmpty; }) -> Model {
               name: _.name + ".reduced"
               rootNodes: map _.rootNodes.ofConcept<Module> -> fork moduleFork [_]
             }
           }
         }                                                                                                   ]

  fork moduleFork [i0: Module] {
    root:        copy i0
    auto apply:  Desugar.desugar
                 Simplfy.simplify
    fixpoint:    true
  }

}
```

The next screenshot shows the `simplify` transformation. The first entry declares an abstract transformation from `Expr` to `Expr`; the other two polymorphically override the declaration for the `LogicalNotExpr` and the `PlusExpr`. The former transforms a `!true` to `false` and the second one performs the static addition of number literals.

```
namespace Simplfy {

  abstract transformation simplify overrides ... [ alt: Expr ]
    [<< ... >>]
  -> [ o0: Expr {   } ]

  transformation? t2 overrides simplify [ i0: LogicalNotExpr ]
    [ i0.expr.isInstanceOf(TrueLit) ]
    [<< ... >>]
  -> [ o0: FalseLit {   } ]

  transformation? t4 overrides simplify [ i0: PlusExpr ]
    [ i0.left.isInstanceOf(NumLit) && i0.right.isInstanceOf(NumLit) ]
    [<< ... >>]
  -> [ o0: NumLit {
           value: i0.left:NumLit.value + i0.right:NumLit.value
       }                                                       ]

}
```

The concrete `simplify` transformations are defined in the same language as its abstract declaration; this is not the case for the `desugar` transformation. Only the abstract declaration is specified in the `kernelf.core` language, with language extensions expected to provide the overrides:

```
namespace Desugar {

  abstract transformation desugar overrides ... [ alt: Expr ]
    [<< ... >>]
  -> [ o0: Expr {   } ]

}
```

# Implementation - Enums

Let's take a look at the reduction of `enums` to constants. The respective transformation overrides the `desugar` transformation for `EnumDecls`. The transformation iterates over the literals in the `enum`, and trsansforms each of them into a `Constant` (note how the rule is allowed to create multiple outputs as opposed to just one, as defined in the abstract declaration). The `name` and the `value` properties of the created `Constant` are computed with Java expressions that access the properties of the input node. The mapping between the each `EnumLiteral` and the resulting `Constant` is stored in the `enumLit-ToConst` mapping label.

```
label enumLitToConst : EnumLit -> Constant

 transformation t17 overrides Desugar.desugar [ i0: EnumDecl ]
   [<< ... >>]
-> [ o0+: map _.literals -> enumLitToConst(_) <- Constant {
            name: i0.name + "_" + _.name
            value: NumLit {
              value: (i0.name + "_" + _.name).hashCode()
            }
         }                                                       ]
```

We need two more transformations. First, we have to transform every `EnumType` (as in `val x: Color = …`) into an `IntType` (`val x: int = …`), because all `enums` are transformed to `int` constants. And whenever we reference an enum literal using a `EnumLitRef` expression, we have to transform it to a reference to the particular `Constant` that has been created from the referenced literal; we can find it by querying the previously populated label.

```
 transformation t19 overrides Desugar.desugar [ alt: EnumType ]
   [<< ... >>]
-> [ o0: IntType {  } ]

 transformation t21 overrides Desugar.desugar [ l: EnumLitRef ]
   [<< ... >>]
-> [ o0: ConstantRef {
          const -> enumLitToConst(l.lit)
       }                                  ]
```

# Implementation – Error Lifting for Enums

Error Lifing relies on detecting particular error messages on output nodes of the transformation and, if one exists, back-propagating a new error to one or more of the input nodes. To this end, transformation authors override a predefined operation `liftMessage` "inside" the creator of the target node. Here is the example for the enums:

```
 transformation t17 overrides Desugar.desugar [ i0: EnumDecl ]
   [<< ... >>]
-> [ o0+: map _.literals -> enumLitToConst(_) <- Constant {
            name: i0.name + "_" + _.name
            value: NumLit {..}
            op ShadowRepository.liftMessage(text: string, lifter: IMessageLifter): void {
              if (text.contains("duplicate")) {
                lifter.liftMessage("this is the duplicate", _);
                lifter.liftMessage("duplicate literal names", i0);
              }
            }
         }                                                                              ]
```

The operation implementation, written inside the `Constant`, checks if the error message(s) reported by MPS on this `Constant` contain the word *"duplicate"*. If so, we propagate a message *"this is the duplicate"* to the currently transformed literal, and another error *"duplicate literal names"* to the input `EnumDecl`. The framework takes care automatically of removing the lifted errors if their causes go away.

# Implementation – Alt Expressions

We conclude the demo with the transformation of the `AltExpression`. What makes this interesting is that a flat list of alternatives must be transformed to a tree of nested `IfExpressions`. Transforming lists to trees is achieved conveniently with the fold function in functional programming, which is why we have added a corresponding transformation primitive to the Shadow Model transformation language.

A `foldR` function joins output for the *n*-th element of the list to what has been constructed for the *n-1*-th list element. In this case, we create a new `if` that reuses the condition of the current option and puts its value into the `then` part. The `else` part of the currently constructed `if` is whatever the previous iteration has created, represented by the `acc` (short for accumulator) expression inside the `foldR`. For the first entry in the `alt`'s list of options, we pass a seed value to `foldR`; in our case it is the `NeverLit`, rendered as `<!>`.

```
transformation t0 overrides Desugar.desugar [alt: AltExpr]
   [<< ... >>]
-> [o0: foldR alt.cases, NeverLit {
        op ShadowRepository.liftMessage(text: string, lifter: IMessageLifter): void {
          lifter.liftMessage("This alt is not guaranteed to succeed; unhandled combination.");
        }
      }, IfExpr {
        cond: copy it.cond
        thenPart: copy it.val
        elsePart: ElsePart {
          expr: acc
        }
      }                                                                                      ]
```

This transformation will create a set of nested `if`s, where the last one always has an `<!>` in its `else` part. However, if we look above, the transformation of the `alt`-expression in the `Extensions` module did not include an `else <!>` at the tail. Why is this? The reason is the `otherwise` in the last option, it's basically a catch-all clause. That last option will be transformed to by the transformation above:

```
if true then 3 else <!>
```

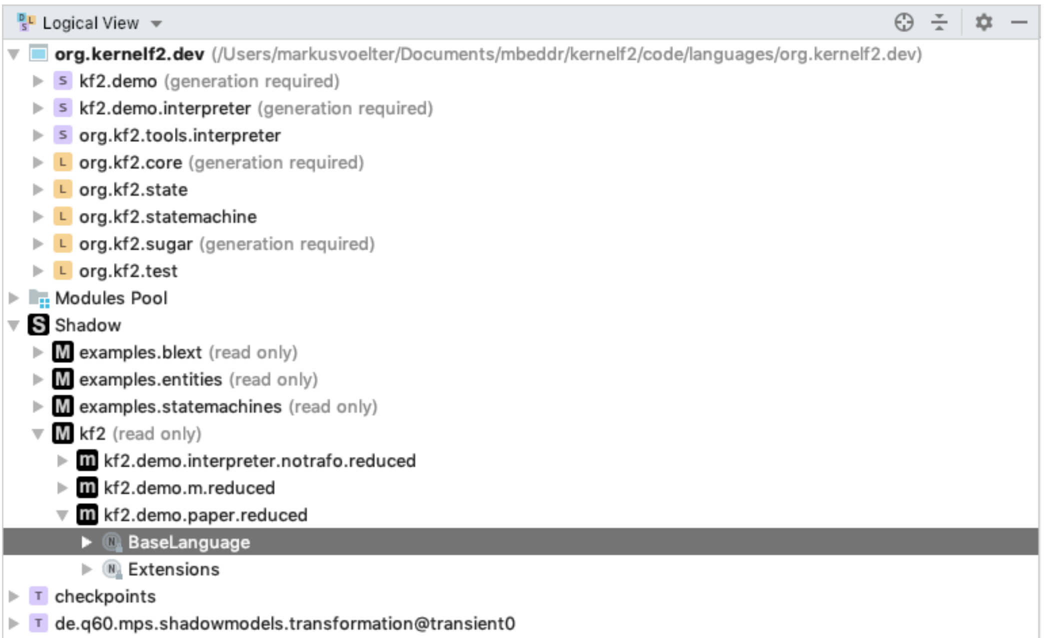
However, the set of base language simplifications defines one that transforms an `if true` to the `then` part, because we know statically the `else` option will never occur:

```
transformation? t39 overrides simplify [i0: IfExpr]
   [ i0.cond.isInstanceOf(TrueLit) ]
   [<< ... >>]
-> [o0: copy i0.thenPart ]
```

The case above thus simply becomes `3` and the `else <!>` goes away. As a corollary, this means that whenever a `<!>` is created as the result of the transformation of an `alt` expression (and survives simplification), then this is an indication of an error in the `alt` expression; which is why we add a corresponding lifter to the transformation, as can be seen above.

# IDE Integrations

The Shadow Repository in the MPS project view that shows the results of all transformations; double clicking any root opens it in an editor, and the incremental transformations can be observed in realtime.




The Fork Explorer shows the execution of the transformations, including the inputs it processes and the output nodes it creates.