KernelF - an Embeddable and Extensible Functional Language

Markus Voelter
independent/itemis AG
voelter@acm.org

Abstract

Expressions and simple functional abstractions are at the core of most DSLs we have built over the last years, in domains ranging from embedded software to medical systems to insurance contracts. To avoid reimplementing this functional core over and over again, we have built KernelF, an extensible and embeddable functional language. It is implemented based on JetBrains MPS, which facilitates extension and embedding. Because of this focus on embedding and the reliance on a language workbench, the design decisions driving KernelF are quite different from other functional languages. In this paper we give an overview over the language, describe the design goals and the resulting design decisions. We use a set of case studies to evaluate the degree to which KernelF achieves the design goals.

1. Introduction

After designing and implementing dozens of domain-specific languages (DSLs) over the last years, we have found a recurring pattern in the high-level structure of DSLs (see Fig. 1). All DSLs rely on domain-specific data structures, be they the structure of refrigerators, data schemas for legal contracts or insurance products or sensor and actor definitions in industrial automation. No two DSLs are similar in these structures. The behavioral aspects of DSLs is often based on versions of established behavioral paradigms, such as functional or object-oriented programming, rules executed by solvers or other rule engines, data flow models or state machines. Using an established behavioral paradigm makes the semantics of DSLs easier to tackle – and checkers and analyzers easier to build. However, at the core of the vast majority of these behavioral paradigms one can find expressions, and by extension, a small functional language: all of the mentioned paradigms require arithmetics, conditions or other simple "calculations". Many of them can benefit from having functions, records or enums.

Assuming the language implementation technology supports modular, reusable (embeddable and extensible) languages, reinventing this core functional language for each DSL is a huge waste of effort. Instead, a better solution to this problem is to develop a small functional language that can be reused in (and adapted for) all these DSLs. More specifically, the language should be extensible (so new expressions can be added) and embeddable (so it can be used in all the contexts mentioned above).

> In this paper we describe the design and implementation of KernelF, a modern functional core language built on top of MPS, a language workbench that facilitates modular and reusable language.

Access to the Code The core of KernelF (i.e., everything that is described outside the case studies in Sec. 7) is open source software. It lives in the IETS3 repository at

`https://github.com/IETS3/iets3.opensource`

The examples discussed in Sec. 2 can be found in the following root, in the above repo:

```
1 project: org.iets3.core (repo/code/languages/org.iets3.core)
```

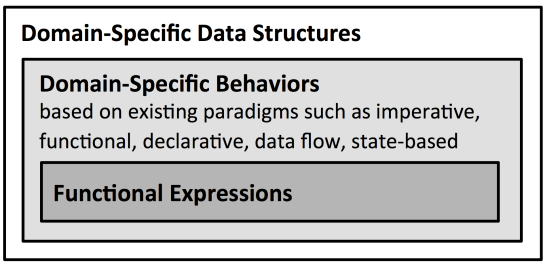

Figure 1. The three typical layers of a DSL: domain-specific data structures, behavior based on an existing paradigm, and at the core, functional expressions.

```
module:  sandbox.core.expr.os
model:   sandbox.core.expr.os.expressions
node:    Paper [TestSuite] (root)
url:     http://localhost:8080/select/org.iets3.core/
         r:3dff0a9d-8b1d-4556-8482-b8653b921cfb/
         7740953487934666415/
```

1.1 Design Goals

Simplicity KernelF should be used as the kernel of DSLs. The users of these DSLs may or may not be programmers – the overwhelming majority will not be experts in functional programming. These users should not be "surprised" or "overwhelmed". Thus, the language should use familiar or easy to learn abstractions and notations wherever possible.

Extensibility Extensibility refers to the ability to add new language constructs to the language to customize it for their domain-specific purpose. Specifically, it must be possible to add new types, new operators or completely new expressions, such as decision tables. These must be added to the language without invasively changing the implementation of KernelF itself.

Embeddability Embedding refers to the ability to use the language as the core of arbitrary other languages. To enable this, several ingredients are needed: the existing set of primitive types must be replaceable, because alternative types may be provided by the host language. More generally, the parts of the language that may not be needed must be removable. And finally, extension also plays into embedding, because embedding into a new context always requires extension of the language with expressions that connect to (i.e., reference) elements from this context (e.g., expressions that refer event arguments in a state machine language).

Robustness The users of the DSLs that embeds KernelF may not be experienced programmers – in fact, they may not see themselves as programmers at all. This means that the language should not have features that make it easy to make dangerous mistakes (such as pointer arithmetics). To the contraty, the language should be structured in a way that makes it straightforward to "do the right thing". For example, handling errors should be integrated into the type system as opposed to C's approach of making checking of `errno` completely optional. It should also enable advanced analyses, for example, through solvers, possible. Importantly, it should ship with language abstractions for writing and running unit tests to facilitate test-driven development.

IDE Support In our experience, DSLs must come with an IDE, otherwise they are not accepted by users. This means that an IDE must be available for the language, but also that the language should be designed so that it can be supported well by IDEs. Such support includes code completion, type checking, refactoring and debugging. In addition, programs should be executable (by an interpreter) directly in the IDE to support quick turnaround and the ability of end users to "play" with the programs.

Portability The various languages into which KernelF will be embedded will probably use different ways of execution. Likely examples include code generation to Java and C, direct execution by interpreting the AST and as well as transformation into intermediate languages for execution in cloud or mobile applications. KernelF should not contain features that prevent execution on any of these platforms. Also, while not a core feature of the language, a sufficient set of language tests should be provided to align the semantics of the various execution platforms.

2. KernelF Described

In this section we describe the KernelF language. The description is complete in the sense that it describes every important feature. However, it is incomplete in that it does not mention every detail; for example, several of the obvious binary operators or collection functions are not mentioned. They can be found out easily through code completion in the editor.

2.1 Types and Literals

Three basic types are part of KernelF: `boolean`, `number`, and `string`. This is a very limited set, but it can be extended through language engineering. They can also be restricted or entirely replaced if a particular host language wants to use other types.

```
val aBool:   boolean   = true
val anInt:   number    = 42
val aReal:   number{2} = 33.33
val aString: string    = "Hello"
```

Boolean types are obvious; for strings, it is worth mentioning that KernelF also support string interpolation, because this is usually more understandable to non-programmers than concatenating strings with `+`:

```
val concatString   = "Hello " + anInt + " and " + (3 + anInt)
val interpolString = '''Hello $(anInt) and $(3 + anInt)'''
```

The `number` type needs a little bit more explanation. A number has a range and a precision. The following patterns exist to specify number types:

```
// integer type, unlimited range
number             => number[-inf|inf]{0}
// positive integer
number[0|inf]      => number[0|inf]{0}
// integer type, range as specified
number[10|20]      => number[10|20]{0}
// decimal type with 2 decimal places, unlimited range
number{2}          => number[-inf|inf]{2}
// range as specified, precision derived from range decimals
number[3.3|4.5]    => number[3.3|4.5]{1}
```

The precision of numbers can be modified with the `precision` operator:

```
type preciseT: number[0|10]{5}
type roundedT: number[0|10]{2}
type wholeT:   number[0|10]{0}
val precisePI: preciseT = 3.14156
val roundedPI: roundedT = precision<round up to 2>(precisePI)
val wholePI    wholeT   = precision<cut to 0>(0)
test case Precision {
  assert precisePI equals 3.14156 <number[0|10]{5}>
  assert roundedPI equals 3.15    <number[0|10]{2}>
  assert wholePI   equals 3       <number[0|10]>
}
```

There are also operators to ensure a value stays in its bounds but cutting too big or too small values.

```
val high = limit<wholeT>(20)
val mid  = limit<wholeT>(5)
val low  = limit<wholeT>(-1)
test case TestLimit {
  assert high equals 10 <number[0|10]>
  assert mid  equals 5  <number[0|10]>
  assert low  equals 0  <number[0|10]>
}
```

We tried to use the various brackets consistently. We use regular round parentheses for value constructors, functions calls, built-in functions (like `limit` above) and for precedence. We use angle brackets for everything that relates to types, specifically type arguments (as in `list<int>`). Finally, we use square brackets for tuples, indexed collection access, number ranges (as shown above). Curly braces are used for blocks and in the special case of number precision.

2.2 Basic Operators

KernelF provides the usual unary and binary operators, using infix notation. Precedence is similar to Java, parentheses are available. Note that the type system performs type inference (discussed in more detail in Section 5.2). As part of that, it performs basic arithmetic computations on the ranges of numeric types.

```
42 + 33                  ==> 75     <number[75|75]{0}>
42 + 2 * 3               ==> 48     <number[48|48]{0}>
aReal + anInt            ==> 75.33  <number[75.33|75.33]>
if aBool then 42 else 33 ==> 42     <number[33|42]{0}>

type tt: number[-10|10]
val n3, n4: tt             = 0
val n34: number[-100|100]  = n3 * n4
```

A few less trivial operators are also available, expressed as member functions. For example, you can test for membership in a list of values or a range:

```
val fourtyTwo = 42
fourtyTwo.oneOf[33, 42, 666] ==> true  <boolean>
fourtyTwo.inRange[0..42]     ==> true  <boolean>
// notice open upper bracket: excluded upper limit
fourtyTwo.inRange[0..42[     ==> false >boolean>
```

2.3 Null Values and Option Types

Option types are used to handle `null` values in a typesafe way. The constant `maybe` in the code below can either be an actual number value, or nothing (i.e., `none`), depending on the `if` condition and the value of `aBool`. This is why the constant is typed as an `option<number>` instead of just `number`. The `if` expression then produces either `none` or 42.

```
val maybe : option<number> = if aBool then 42 else none
```

Most operators, as well as many dot operations, are overloaded to also work with `option<T>` if they are defined for `T`. If one of the arguments is `none`, then the whole expression evaluates to `none`. In this sense, a `none` value "bubbles" up. Note that the type system represents this; the `+` operator and the `length` call in the example below are also option types!

```
val nothing : opt<number> = none
val something : opt<number> = 10
val noText : opt<string> = none

nothing + 10 ==> none <option[number[-inf|inf]{0}]>
something + 10 ==> 20 <option[number[-inf|inf]{0}]>
noText.length ==> none <option[number[0|inf]{0}]>
```

To test whether an option actually contains a value, you can use the `isSome` expression as shown below:

```
val maybeHasAValue : boolean = isSome(maybe)
```

To explicitly extract the value from an option type (i.e., to essentially transform an `option<T>` to a `T`), a special form of the `if` expression can be used for this purpose, as shown in the example below. As mentioned above, the `isSome` expression is as a query that tests if the option contains a value; inside the `then` part, the `val` expression refers to the value extracted from the option; `val` cannot be used in the `else` branch, so it is syntactically impossible to access the (then non-existent) value in the option.

```
if isSome(maybe) then maybe else 0 ==> 42 <number[42]>
```

If the name `val` is ambiguous, then the name can be changed using an `as` clause; the example also illustrates that several expressions can be tested at the same time.

```
if isSome(f(a)) as t1 && isSome(f(c)) as t2
   then t1 + t2 else 0
```

A shorthand operator `opt ?: alt` is also available; it returns the value inside the `option` if the `option` is a `some`, and the `alt` value otherwise:

```
val anInt = maybe(a, b) ?: 0
```

2.4 Error Handling using Attempt Types

In the same way that KernelF encodes null checks into the type system using option types, KernelF also provides type system support for handling errors using `attempt` types. An attempt type has a base type that represent the payload (e.g., return value in a function) if the attempt succeeds. It also has a number of error

literals that have to be handled by the client code. An attempt type is written down as `attempt<baseType|err1, err2,..,errN>`. As a consequence of type inference, such a type is hardly ever written down in a program.

Error handling has two ingredients. The first step is reporting the error. In the example below, this is performed in the `getHTML` function. Depending on what happens when it attempts to retrieve the HTML, it either returns the payload or reports an error using `error(<error>)`. The type inference mechanism inferes the type `attempt<string|timeout, err404>` for the `alt` expression and, transitively, the function `getHTML`.

```
fun getHTML(url: string) : attempt<string|timeout, error404>
   = alt |..successful..  => theHTML |
         |..timeout..     => error(timeout)   |
         |..unreachable.. => error(error404)  |
```

The client has to "unpack" the payload from the attempt type using the `try` expression. In the successful case, the `val` expression provides access to the payload of the attempt type. Errors can either be handled one by one (as shown in Figure **??**), or with a generic `error` clause.

```
val toDisplay : string =
        try getHTML("http://mbeddr.com") => val
          error<timeout>  => "Timeout"
          error<error404> => "Not Found"
```

As with the unpacking of options using `isSome`, it is possible to assign a name to the result of the called function, so that name can be used instead of `val` in the success case:

```
try getHTML("http://mbeddr.com") as data => data
  ...
```

If not all errors are handled, the type of the `try` expression remains an attempt type. In the above example, we may not handle the `error404` case:

```
val toDisplay =
        try getHTML("http://mbeddr.com") => val
          error<timeout>  => "Timeout"
```

In this case, the type of `try`, and hence of `toDisplay`, would be `attempt<string|error404>`. This way, error handling can be delegated to an upstream caller. To force complete handling of all errors, two strategies can be applied. The first one involves a type constraint to express that the success type is expected:

```
val toDisplay: string =
        try getHTML("http://mbeddr.com") => val
          error<timeout>  => "Timeout"
```

In an incomplete case, where not all errors are handled (either individually or with a generic `error` clause), the type of `try` will remain an `attempt` type with the non-handled errors. If an explicit return type expects a non-attempt type, this type incompatibility will return in an error. A way of forcing the try expression to handle all errors is to use the `complete` flag, as shown below. It reports an error on the `try` expression directly if not all errors are handled:

```
val toDisplay =
        // try will have error b/c error404 is not handled
        try complete getHTML("http://mbeddr.com") => val
          error<timeout>  => "Timeout"
```

Similar to option types, the attempt types are also overridden wrt. to their success type for the same operators and dot expressions. The error literals are propagated accordingly.

```
getHTML("http://mbeddr.com").length ==> 4
           <attempt[number[0|inf]{0}|[error404, timeout]]>
getHTML("http://doesntExist.com").length ==> error(error404)
           <attempt[number[0|inf]{0}|[error404, timeout]]>
```

2.5 Functions and Extension Functions

Even though function syntax may be domain-specific, KernelF includes a default abstraction for functions. Functions have a name, a list of arguments, an optional return type and an expression as the body; the code below shows a few examples. The body can use the block expression, which supports values as temporary variables (similar to a `let` expression, but with a more friendly syntax). As with variables, the return type is optional.

```
fun add(a: number, b: number) = a + b
fun addWithType(a: number, b: number) : number = a + b
fun biggerFun(a: number) = {
  val t1 = 2 * a
  val t2 = t1 + a
  t2
}
```

KernelF also supports extension functions. They must have at least one argument, the one that acts as the `this` variable. They can then be called using dot notation on an expression of the type of the first argument. In contrast to regular functions, the advantage is in IDE support: code completion will only show those functions that are valid for the first argument. Note that, at least for now, no polymorphism is supported.

```
ext fun isSomethingInIt(this: list<number>) = this.size != 0
list(1, 2, 3).isSomethingInIt() ==> true <boolean>
```

2.6 Function Types, Closures, Function References and Higher-Order Functions

KernelF has full support for function types, closures and function references as well as higher-order functions.

We start by using a `typedef` to define abbreviations for two function types. The first one, `INT_BINOP` is the type of functions that take two `number`s and return a `number`. The second one represents functions that map one `number` to another. Using typedefs is not necessary for function types, they can also be used directly. But since these types become long'ish, using a `typedef` makes sense.

```
type INT_BINOP : (number, number => number)
type INT_UNOP  : (number => number)
```

Next, we define a function `mul` that is of type `INT_BINOP`. We can verify this by assigning a reference to that function (using the colon operator) to a variable `mulFun : INT_BINOP`. Alternatively we can define a closure, i.e., an anonymous function, and assign it to a similarly typed variable `mulCls`. Closures use the vertical bar for deliniation.

```
fun mul(a: number, b: number) = a * b
val mulFun: INT_BINOP         = :mul
val mulCls: INT_BINOP     = |a: number, b: number => a * b|
```

We can now define a higher-order function `doWithTwoInts` that takes two integers as arguments, as well as value of type `INT_BINOP`. The body of the function executes the function or lambda, forwarding the two arguments. The next two lines verify this behavior by calling `doWithTwoInts` with both `mulFun` and `mulCls`.

```
fun doWithTwoInts(x: number, y: number, op: INT_BINOP) =
                                              op.exec(x, y)
doWithTwoInts(2, 3, mulCls) ==> 6 <number>
doWithTwoInts(2, 3, mulFun) ==> 6 <number>
```

Finally, KernelF also supports currying, i.e., the binding of some of a function's arguments, returning new functions with correspondingly fewer arguments. The value `multiplyWithTwo` in the example below is a function that takes one argument, because the other one has already been bound to the value 2 using `bind`. We could add an optional type to the constant (`val multiplyWithTwo: INT_UNOP = ...`) to verify that the type is indeed `INT_UNOP`. For demonstration purposes we define another higher-order function and call it.

```
val multiplyWithTwo = mulCls.bind(2)
fun doWithOneInt(x: int, op: INT_UNOP) = op.exec(x)
doWithOneInt(5, multiplyWithTwo) ==> 10 <number>
```

2.7 Collections

KernelF has `lists`, `sets` and `maps`. All are subtypes of `collections`. While KernelF does not have generics in general, the collections are parametrized with their element types. They are also covariant.

```
val reals     = list(1.41, 2.71, 3.14)
val names     = set("Markus", "Markus", "Tamas")
val hometowns = map("Markus"->"Heidenheim",
                    "Tamas" ->"Puspokladany")
val col : collection<real> = reals
```

The collections support the usual simple operations, a few are shown in the following example code. Of course, like all other values in KernelF, collections are immutable; the operations to not modify the value on which they are called, the return a modified copy. This is illustrated by the second line, where the original `reals` list is still `list(1.41, 2.71, 3.14)`.

```
reals              ==> list(1.41, 2.71, 3.14)
                           <list<number[0.00|100.00]{2}>>
reals.add(1.00)    ==> list(1.41, 2.71, 3.14, 1.00)
                           <list<number[0.00|100.00]{2}>>
reals.at(1)        ==> 2.71  <number[0.00|100.00]{2}>
reals[2]           ==> 3.14  <number[0.00|100.00]{2}>
names.isEmpty      ==> false      <boolean>
names.size         ==> 2          <number>
hometowns["Tamas"] ==> "Budapest" <string>
```

Notice that the `reals.add(1.00)` will lead to an error because it tries to add a `1.00` to a list of `number[1.41|3.14]{2}`, i.e. `1.00` is out of range! To fix this, the `reals` collection must be given an explicit type, for example `number[0.00|100.00]`[1].

The usual higher order functions on collections are also available. They can be used in three forms: you can pass in a function reference, a closure (both introduced before), and also a shorthand version of the closure, where the `it` argument is implicit. The latter is the default.

```
val ints = list(1, 2, 3, 4)
fun isGreaterTwo(it: number) = it > 2
ints.where(:isGreaterTwo)       ==> list(3, 4)
                                    <list<number[1|4]>>
ints.where(|number r => r > 2|) ==> list(3, 4)
                                    <list<number[1|4]>>
ints.where(|it > 2|)            ==> list(3, 4)
                                    <list<number[1|4]>>
```

More examples are shown below; the list of operations is expected to grow over time.

```
ints.map(|it + 1|) ==> list(2, 3, 4, 5) <list<number>>
ints.any(|it < 0|) ==> false            <boolean>
ints.all(|it > 3|) ==> false            <boolean>
```

There is also a `foreach` which requires the lambda expression inside to have a sideeffect; it "performs" the sideeffect and then returns the original list.

Inside `where`, `foreach` and `map`, the variable `counter` is available; it has a zero-based index value of the current iteration (i.e., 0 in the first iteration, 1 in the second, etc.).

2.8 Tuples

Tuples are non-declared multi-element values. The type is written as `[T1, T2, .., Tn]`, and the literals look essentially the same way: `[expr1, expr2, .., exprN]`. Tuple elements be accessed using an array-access-like bracket notation.

```
ext fun minMax(this: list<number>) = [this.min, this.max]
ints.minMax()    ==> [1, 4] <[number, number]>
ints.minMax()[0] ==> 1      <number>
ints.minMax()[1] ==> 4      <number>
```

2.9 Records and Path Expressions

Like Tuples, records are structured data, but they are explicitly declared. KernelF has them primarily for

[1] In later version of the type system, a suitable type might be derived automatically. Currently, the element added to a list must be a subtype of the element in the list.

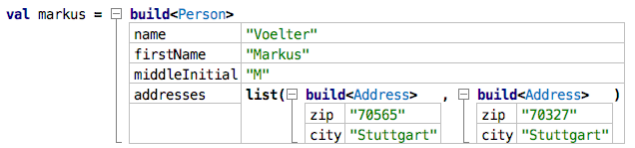

Figure 2. The builder expression uses collapsible trees to build hierarchical structures.

completeness; we expect most data structures to be domain-specific and hence contributed by a language that embeds KernelF.

```
record Company {
  offices: list<Office>
  emps   : list<Person>
}
record Person {
  lastName     : string
  middleInitial: option<string>
  firstName    : string
}
record Office {
  branchName: string
}
```

A literal syntax is also supported:

```
val officeLuenen = #Office{"Luenen"}
val comp = #Company{
  list(#Office{"Stuttgart"},
       officeLuenen),
  list(#Person{"Markus", none , "Voelter"},
       #Person{"Tamas", "M", "Szabo"})
}
```

Path expressions can be used to navigate along nested records structures, as shown in the examples below.

```
comp.emps.firstName
   ==> list("Voelter", "Szabo") <collection<string>>
comp.emps.firstName.last
   ==> "Szabo" <string>
comp.emps.map(|Person p => "Hello " + p.firstName|).first
   ==> "Hello Markus" <string>
```

In addition, a semi-graphical builder expression is available for constructing complex structures. An example is shown in Figure 2. It can be used for any hierarchical structure, not just records, if a suitable adapter is provided.

Like all other values in KernelF, record instances are immutable. However, there is a convenient syntax to "modify" record instances, i.e., create copies with some member values changed:

```
val me         = #Person{"Markus", none, "Voelter"}
val meWithX    = me.with(firstName = old + "X",
                         lastName  = lastName + "X")
val meSwitched = me.with(firstName = lastName,
                         lastName  = firstName)
val brother    = me.with(firstName = "Mathias")

brother    ==> #Person{"Mathias", none, "Voelter"}  <Person>
meWithX    ==> #Person{"MarkusX", none, "VoelterX"} <Person>
meSwitched ==> #Person{"Voelter", none, "Markus"}   <Person>
```

Grouping The `groupBy` operation supports grouping the entries in an existing collection by a key. The result is a new collection of type `group<KT, MT>` where `KT` is the type of the key expression and `MT` is the type of the members of each group (the type of the original collection). In the example in Figure 3 the `KT` is `string` and the `MT` is `Item`. On a variable of type `group<KT, MT>` one can use the `key` operation to retrieve the current group's key, and `members` to acccess all the members of that group.

```
record Item {
  name: string
  price: int
  cat: string
}

val m1 = #Item{"Jim", 100, "Book"}
val m2 = #Item{"Jim", 200, "CD"}
val p1 = #Item{"Peter", 100, "CD"}
val data = list(m1, m2, p1)
val authorCats = data.groupBy(it.name)
                     .project(author = it.key, cats = it.members.map(it.cat).distinct)
val texts = authorCats.map('''The author $(it.author) published $(it.cats.join(", ")) ''')
                      .terminate(". ")
```

Figure 3. Example code showing grouping, projection of anonymous records and string joining/termination.

Anonymous Records The `project` operation supports the on-the-fly creation of anonymous records. In the example in Figure 3, we create one that has two fields, `author` and `cats`.[2] `project` is typed to a collection of this anonymous record. As a consequence of type inference, the anonymous record can be used with full IDE support; however, since the type has no name it cannot be mentioned in the program. So, for example, the `authorCats` value could not be annotated with an explicit type, and it cannot be used as a function argument (because this would require an explicit type).

String Lists Lists of strings can be transformed into a single string using the `join(s)` and `terminate(s)` expressions. `join` separates two subsequent strings by `s`, whereas `terminate` terminates each one with `s`. Figure 3 shows an example.

2.10 Enums

Enums are also supported in KernelF, with regular and valued flavors. Regular enums just define a list of literals; their type is the enum itself (see the use of `Color` in the code snippet below). Literals can be marked as `qualified`, which means that their literals have to be referenced using enum name before the colon to deal with potentially overlapping literal names.

```
enum Color               { red, green, blue }
enum Starbucks qualified { large, venti, monster }
val ocean: Color = blue
val coffee       = Starbucks:large
```

Valued enums associate an arbitrary value with each literal; all values of a particular enum must be of the same type. That type is declared after the name of the enum, adding that type makes an enum a valued

[2] This is short for `categories` and does not related to the animal :-).

enum. From an enum literal reference, you can get the associated value using the `value` operation.

```
enum StarbuckSizes<number> {
  big   -> 100
  venti -> 200
  mega  -> 300
}

enum Family<Person> {
  me        -> #Person{"Markus", none, "Voelter"}
  myBrother -> #Person{"Mathias", none, "Voelter"}
}

me.value.firstName ==> "Markus" [string]
big.value          ==> 100      [number]
```

2.11 Unit Tests and Constraints

Tests Built-in support for unit tests is important, because, as we describe in Sec. 5.3, the semantics of KernelF is defined via a test suite; so we needed the ability to conveniently write collections of unit tests even during the implementation of KernelF. Test support is also essential to help users write good code.

At the core of the unit test support is the test case: a test case has a name and a number of test case items. The default item is an `assert`ion that compares an expected and an actual value. The comparison operator itself is equation by default, but can be extended through language extension. The second test case item is `confail`, which expects a constraint failure to occur as part of the evaluation of the actual result (see below for constraints). The constructs that can go into a test case, the test case items, can be extended as well. For example, users can add set up or tear down code if they want to test expressions with side effects. A test suite finally groups tests, plus other top level contents (records, functions, constants). It is also possible to reference entities outside the test suite. Fig. 10 shows an example.

Constraints KernelF also supports checking of runtime constraints. Several forms exist, all illustrated in Fig. 4:

- Attached to a value: it is checked after the value has been computed.
- Attached to a typedef or record, it is checked whenever a value is checked against an explicitly declared type: when assigning to a value, when returning from a function, and when passing an argument into a function. For chained typedefs, the constraints are joined in a conjunction ("anded" together). Constraints for records are also checked when the record is instantiated using a record literal `#R{..}` or when it is "changed" using the `with` operation.
- Type `check` on an expression: it checks the type and also the constraints associated with the type.

```
fun plus3times2(i: number) {
  val v where [it > i] = i + 3
  v * 2
}

fun oddOrEven(i: number) where [pre  i.range[1..4]
                                post res.in[0, 1]] = alt [i == 1 => 0
                                                          i == 2 => 1
                                                          i == 3 => 0
                                                          i == 4 => 1]
type posint: number where [it >= 0]
type age: posint where [it < 120]

record Bounds {
  min: number
  max: number
} where [min <= max]

fun heuristic(age: age) = √<2 * age : posint>
```

Figure 4. Constraints can be attached to values, to functions (in the form of pre- and postconditions), to records, and to types. In the latter case, they are checked whenever a type is explicitly specified in values, function arguments, return types and type constraint expressions.

- Attached to functions in the form of pre- and postconditions. They are checked before and after the execution of the function, respectively.

Constraint failures lead to a target platform-specific form of diagnostic output. The default implementation in the interpreter throws a `ConstraintFailureException` (whose occurence can be tested using the `confail` test item). The output of the exception logs a stacktrace of the failed constraint; see below. The long URL in line two is the URL of the node in the MPS source code that failed; you can paste it into your browser, and MPS will select the particular node.

```
1 ERROR: Postcondition failed for res.inRange[0..1]
2        http://localhost:8080/select/DEFAULT/r:3dff0a9...
3   at [Function] PaperDescription.oddOrEven(10)
4      [Function] PaperDescription.function1(10)
5      [Function] PaperDescription.function2(10)
```

In case of a failed constraint, execution terminates. If, in the example above, the error should be communicated back to the caller, regular error handling should be used:

```
fun oddOrEven(i: number) = alt | i == 1    => success(0)   |
                               | i == 2    => success(1)   |
                               | i == 3    => success(0)   |
                               | i == 4    => success(1)   |
                               | otherwise => error(range) |
```

For constraints on types, it is also possible to query the conformance of a value against this type explicitly from the program (i.e., without throwing a runtime exception). Types can contribute constraints as well as custom error messages that can be reported to the user.

```
type Speed:    number[-50|250]
type FwdSpeed: Speed where it >= 0

val validSpeed1  = check<Speed>(-10)
val validSpeed2  = check<Speed>(50)
val invSpeed     = check<Speed>(300)
val invFwdSpeed1 = check<FwdSpeed>(-10)
val invFwdSpeed2 = check<FwdSpeed>(300)
```

```
test case TestConstraintsCheck {
  assert validSpeed1.ok equals true
  assert validSpeed2.ok equals true
  assert if validSpeed1 then 1 else 0 equals 1
  assert invSpeed.ok equals false
  assert invSpeed.err equals "value is over minimum (250)"
  assert invFwdSpeed1.ok equals false
  assert invFwdSpeed2.ok equals false
}
```

If you want to test for constraints explicitly, you cannot assign the type to the variable, because this would lead to a constraint *before* the explicit test gets invoked. Thus, the following code would be illegal, because the assertion in the test case would never be executed; the runtime constraint check in `val aSpeed: Speed = 300` would occur first.

```
val aSpeed: Speed = 300
val validSpeed = check<Speed>(aSpeed)
test case TestConstraintsCheck {
  assert validSpeed.ok equals true
}
```

Using an unconstrained integer (or not specifying a type at all) solves this problem:

```
val aSpeed: number = 300
val validSpeed = check<Speed>(aSpeed)
```

Forcing Types Assigning a "bigger" type to a "smaller" type is prevented by the type system; thus the following error:

```
val bigRge   : number[0|100] = 50
val smallRge : number[10|20] = bigRge // error
```

However, in the following piece of code, *we know* that the value will fit into `number[10..20]`, even though the type system cannot figure it out[3] and will report an error.

```
val smallRge : number[10|20] =
  if bigRge > 20 then 20 else bigRge
```

To solve this issue, you need an explicit type cast:

```
val smallRge : number[10|20] =
  cast<number[10|20]>(if bigRge > 20 then 20 else bigRge)
```

A cast essentially prevents type checks and delegates checking to runtime; in other words, the runtime constraint checks of the target type are applied to the value returned by the `cast` expression (range between 10 and 20 in this case). Note that, because of type inference, the type of the `val` can be omitted, resulting in the following code:

```
val smallRge =
  cast<number[10|20]>(if bigRge > 20 then 20 else bigRge)
```

To recap: a type specified on an argument or value is checked by the type system. A cast type is not checked by the type system, but the value has to conform to the target type at runtime. Note that there is no way to avoid *all* static and runtime checks; KernelF always at least provides runtime safety.

[3] Future version of the type system may be able to figure it out by improving the number range caclulcaton.

2.12 Type Tags

A type tag [1] is additional information attached to the type, that is tracked and checked by the type system. Consider a web application that processes data entered by the user. A function `process(txt: string)` may be defined to handle the data entered by the user. To ensure that `txt` does not contain executable code (cf. code-injection attacks), the string has to be sanitized. Until this happens, the data must be considered tainted [2]. Type tags can be used to ensure that a function can only work with sanitized strings:

```
// returns an arbitrary string
fun getData(url: string) : string { "data" }
// accepts a string that must be marked as sanitized
fun storeInDB(data: string<sanitized>) : boolean = ...
...
// v is a regular string
val v = getData("http://voelter.de")
// trying to pass it storeInDB fails because it
// does not have the sanitized tag
val invalid = storeInDB(v) // error
// sanitize is a special operator that cleans up the string,
// and them marks it as sanitized; passing to storeInDB works
val valid = storeInDB(sanitize[v])
```

The `sanitized` tag is an example of a unary tag. A type can be marked to have the tag (`<tag>`), to *not* have the tag (`<!tag>`), or to be unspecified. The tag definition determines the type compatibility rules between those three options. For `sanitized`, a type with no specification corresponds to `<!sanitized>`; in other words, if we don't know, we cannot assume the string has been sanitized.

In addition, the system supports n-ary tags as well. They define a set of tag values (e.g., `confidential, secret, topsecret`) with an ordering between them (e.g., `confidential < secret < topsecret`). The type checking for tags takes this ordering into account, as is illustrated by the code below:

```
val somethingUnclassified : string                = "hello"
val somethingConfidential : string<confidential> = "hello"
val somethingSecret       : string<secret>       = "hello"
val somethingTopSecret    : string<topsecret>    = "hello"

fun publish(data: string) = ...
val p1 = publish(somethingUnclassified)
val p2 = publish(somethingConfidential) // ERROR
val p3 = publish(somethingSecret)       // ERROR
val p4 = publish(somethingTopSecret)    // ERROR

fun putIntoCIAArchive(data: string<confidential+>) = ...
val a1 = putIntoCIAArchive(somethingUnclassified) // ERROR
val a2 = putIntoCIAArchive(somethingConfidential)
val a3 = putIntoCIAArchive(somethingTopSecret)
val a4 = putIntoCIAArchive(somethingSecret)

fun tellANavyGeneral(data: string<secret->) = ...
val g1 = tellANavyGeneral(somethingConfidential)
val g2 = tellANavyGeneral(somethingSecret)
val g3 = tellANavyGeneral(somethingTopSecret) // ERROR
```

```
val g4 = tellANavyGeneral(somethingUnclassified)
```

3. Stateful KernelF

3.1 Effects Tracking

KernelF at its core is a functional language and none of the expressions in KernelF have a side effect. This means, for example, that an execution engine can cache the results of functions that are called repeatedly with the same arguments; the default KernelF interpreter does this. However, KernelF may be extended to support expressions with side effects or be embedded in a language that has effects. Then, it must be possible to analyze which functions (or other parts of programs) can be cached, and which cannot because they have effects. Similarly, it must be allowed to call a function with an effect without capturing its return value (which is an error otherwise).

To enable this, KernelF supports effects tracking. It distinguishes between read and write effects, and for write effects it also tracks idempotence.

Consider the following example:

```
fun standardize/RM(data: number) {
  val filtered = filter(data)
  effect[data]
  if filtered > data then filtered else data
}
```

Here, `effect[..]` is a demo expression provided by a language extension that has a side effect. This is signalled to the checker by implementing `IMayHaveEffect` in the language concept and returning an `EffectDescriptor` from its `effectDescriptor` method; the descriptor has Boolean flags for the various supported kinds of effects.

Because it is called inside the `standardize` function, that function must also be marked to have an effect. This is done by entering `/R` (reads), `/M` (modifies) or `/RM` (reads + modifies) behind the function name; an error will be reported otherwise. The mechanism also works for function types: you can mark a function type as allowing effects, by entering the flag after the arrow in the function type; this is shown in the argument of the function below. If declared this way, it is legal to pass in functions that has an effect (or not).

```
fun doSomethingWithAnEffect/RM(f: ( =>/RM string)) =
   f.exec/RM()
```

Note that the function *call* (to `exec` in this case) is automatically marked to have an effect if the called function has an effect.

3.2 Boxes

Immutable data means that you cannot change a value once it has been created. For primitive types, this is intuitive:

```
val a = 1 + 2
val b = 3 + a
val x = a + b
```

`1 + 2` creates a new value `3`, and adding `a` and `b` creates a new value `c`. Values can also not be reassigned because *anybody* who has a reference to `x` now sees the value of `x` change.

```
val x = a + b
x = x + 1 // invalid
```

Instead you have to invent a new name for the new value, however, this leads to many new (temporary) names. Let us look at collections. Assume you have a list of three elements and you add a fourth one:

```
val l1 = list(1, 2, 3)
val l2 = l1.plus(4)
assert l1.size == 3
assert l2.size == 4
```

Here, too, the original list remains unchanged and you get a new list, one that now has a fourth element, as the result of `l1.plus(4)`.

So, how do you store changing global state, for example, a database of measurements? Using a new variable for each updated "state of the database" is not a solution because it is *the* database that is supposed to change. One solution would be to introduce variables (as opposed to the *values* used so far):

```
var db = list(1, 2, 3) // note the r instead of the l
fun store/M(x: int) {
  db.add(x)
}
```

For this to work, you will have to mark the `add` operation to have an effect, which will, transitively, also give `store` an effect. However, `add` does not exist on immutable lists, so you need a whole second set of APIs for mutable collections. The `list` in this example cannot be the same `list` as the one used earlier; it's a *mutable* list, maybe called `mlist`. In clonclusion, you need mutable versions of all collections. This approach is a valid solution, and some languages, for example, Scala, use it. However, it is a lot of work and should be avoided.

Boxes Boxes are an alternative approach that do not require mutable version of all immutable data structures. Boxes explicitly values inside. The box itself is immutable (i.e., its own reference stays stable), but its contents can change:

```
val globalcounter: box<int> = box(0)
fun incrCounter() {
  globalcounter.update(globalcounter.val + 1)
}
```

Apart from creation, boxes have two operations: a `val` that returns the contents of the box, as well as a `update(newval)` that sets a new value. The former has a `read` effect, the latter a `modify` effect. When you update the box's content, you pass a *new* value; you do

```
state machine CounterToMax {
  event init(start: posInt)
  event inc(by: posInt)
  var counter: posInt = 0
  var invalids: posInt = 0
  state init {
    on init -> count: counter := start
  }
  state count {
    on inc [by < 10] -> count: counter := counter + by
    on inc [by >= 10] -> count: invalids := invalids + 1
  }
}
```

Figure 5. A state machine is an example of natively mutable data structure.

not need additional APIs for changing value. The boxes themselves are generic, as shown with the next example of boxed collections:

```
val db = box(list(1, 2, 3)) // we're back to a value here!
fun store(x: int) {
  db.update(db.val.plus(4))
}
```

The big advantage of this approach is that no mutable data structures are required, the original immutable APIs (plus the generic box functionality) are enough. However, the syntax is a little bit chatty. To make it more consise, the `it` expression provides access to the current content of the box:

```
val globaxlcounter = box(0)
val db             = box(list(1, 2, 3))
fun incrCounter() { globalcounter.update(it + 1) }
fun store(x: int) { db.update(it.plus(4)) }
```

Interpreter In terms of implementation, for example, in an interpreter, boxes are really just wrapper objects with a method to get and set a generic `java.lang.Object` box content. The `val` and `update` operations call those methods on the runtime Java object.

```
1 public class BoxValue {
2   private Object value;
3   public BoxValue(Object initial) { this.value = initial; }
4   public void set(Object newValue) { this.value = newValue; }
5   public Object get() { this.value; }
6 }
```

3.3 Native Mutable Data

The reason for boxes is that existing immutable data types can be used in a mutable way. However, this is only useful if you have immutable data structures to reuse this way in the first place; some data structures are inherently mutable, and they can use a box-free syntax.

State Machine Example The embodiment of changing state are state machines, and Figure 5 shows minimal one that represents a (slightly contrived) counter:

It defines two events, one to initialize the machine's `counter`, and the other one to increment it. It has two states, an initial state and operational state. The `init` event goes from the initial `init` state to the `count` state, where it then accepts `inc` events. If the `by` value is less than 10, the `counter` gets incremented, otherwise a counter of `invalids` incrementation attempts is increased.

Since the state machine's purpose is to *represent changing state over time*, we don't have to pretend anything is immutable. This is why we allow an assignment operator `:=` inside a state machine. Inside a state machine you can also read one of its variables by just mentioning its name (as in `invalids + 1`); you don't need the `val`.

The following code shows how to use a state machine from client code:

```
val ctr = start(CounterToMax).init(0) // start creates
     instance
fun doStuffWithCounter() {
  ctr.inc(5)      // now 5
  ctr.inc(3)      // now 8
  ctr.inc(20)     // invalid; still 8
  assert ctr.counter  == 8
  assert ctr.invalids == 1
}
```

Note that even though there is mutable state (and the various operations on state machines have effects), there are no boxes; no `update` or `val` is required. However, internally the state machines still have box semantics (in the implementation, several interfaces for `IBoxLike` things are used to generalise box-like behavior). But state machines have been purpose-built to have state, there is no need to reuse existing immutable APIs, as was the case for primitive operators and collections.

Interpreter Let's look at the interpreter. To implement the variable references inside state machines, we use an interface `ICanBeUsedAsLValue` to mark that they can be used on the left side of an assignment (an "lvalue"). The interface has a method `isUsedAsLValue` that detects structurally, from the AST, if a particular variable reference is on the left side of an assignment. The interpreter uses this method to determine what it needs to evaluate to: the box if it is used as a lvalue, and the box contents otherwise. Here is the generic interpreter for the assignment; note how it relies on the runtime representation of things that can be lvalues to implement the `ILValue` interface to generically implement this functionality:

```
1  Object rvalue = #right; // recursively call interpreter
2  Object lvalue = #left;  // on the two arguments
3  if (lvalue instanceof ILValue) { // must be an ILValue
4    // which has update method
5    ((ILValue) lvalue).updateValue(rvalue);
6  } else {
7    throw new InvalidValueException
8              (node.left, "not_an_ILValue");
9  }
10 return rvalue;
```

In the case of state machines, the interpreter plays together with, the `VarRef` concept that represents references to state machine variables:

```
SmValue currentMachine = (SmValue) env[SmValue.THIS];
SMVarValue value = currentMachine.getVar(node.var);
if (node.isUsedAsLValue()) {
  return value;          // returns the box
} else {
  return value.value();  // returns box contents
}
```

It first retrieves the currently executing instance of the state machine from the environment (the triggers put that there), and then asks the current state machine for the variable that it references. Note that this returns the `ILValue`-implementing class that represents the variable. Then comes the crucial distinction: if the current variable reference is used in lvalue position, we return the `ILValue` (so that the assignment interpreter can call `update`). Otherwise we directly return the contents of the box (e.g., an `int`)

3.4 Transactions

Take a look at the following code:

```
type intLE5: int where it <= 5
val c1: box<intLE5> = box(0)
val c2: box<intLE5> = box(0)
fun incrementCounters(x1: int, x2: int) {
  c1.update(it + x1)
  c2.update(it + x2)
}
fun main() {
  incrementCounters(1, 1)
  incrementCounters(3, 5)
}
```

Boxes respect the constraints on their content type: if you set a value that violates a constraint, than the update fails. What actually happens then is configurable, at least in KernelF's default interpreter: output a log message and continue, or throw an exception that terminates the interpreter. While, in the second case, the program stops anyway, and so it does not matter which value is set, in the first case we run into the problem that, for the second invocation of `incrementCounters`, `c1` is updated correctly, but the update of `c2` is faulty. Transactions can help with this:

```
fun incrementCounters(x1: int, x2: int) newtx{
  c1.update(it + x1)
  c2.update(it + x2)
}
```

A transaction block is like a regular block, but if something fails inside it (interpreter: an exception is thrown), it rolls back all the changes to mutable data inside that transaction. Because the box contents themselvers are immutable, the interpreter simply stores the value of each box (or more generally, `ITransactionalValue`) before it performs the update and remembers them in the transaction. On rollback, it just re-sets the value. This also works with state machines, and with combinations

```
val aCounter = accu0()
val greenEntered = accu0()
val redLeft = accu0()

val ff = start(FlipFlop)

fun txOnFlipFlop/M(m: FlipFlop) = newtx{
  m.trigger
  m.trigger
  assert(false)
}

state machine FlipFlop {
  event trigger
  state red {
    on trigger [aCounter.val < 5] -> green: aCounter.inc/M()
    exit: redLeft.inc/M()
  }
  state green {
    entry: greenEntered.inc/M()
    on trigger [aCounter.val < 5] -> red: aCounter.inc/M()
  }
}
```

Figure 6. An example of using transactions with different mutable data structures.

of state machines and other boxes, as shown in the example below where the state machine modifies other global data.

The language also supports nested transactions (which can be rolled back individually) as well as the distinction between starting a new transaction (with `newtx`) and a block requiring to be executed in an *existing* transaction (using `intx`).

Interpreter The reason why transctions work also with state machines is that the current total state of a state machine is also an immutable object; in other words, it also implements `ITransactionalValue`. The implementation of the transaction in the interpreter looks like this:

```
Transaction tx = new Transaction(node);
env[Transaction.KEY] = tx; // store in env for nested calls
try {
  Object res = #body;
  tx.commit();
  return res;
} catch (SomethingWentWrong ex) {
  tx.rollback();
} finally {
  env[Transaction.KEY] = null;  // no tx active anymore
}
```

This form of transactional memory is also used in Clojure, as far as I understand.

3.5 State Machines

We have introduced basic state machines above. In this section we'll introduce the remaining features of state machines.

Nested States States can be nested. A state S that itself contains states considers the first F one as the initial state. Any entry into S automatically enters F, recursively.

Actions State machines support entry and exit actions on states as well as transition actions. Ordering of their execution is always exit-transition-entry. For nested states, the exit actions are executed inside-out, the entry actions are executed outside in.

Automatic Transitions In addition to transitions that are triggered by events (expressed using the `on` keyword), automatic transitions are also supported. They are introduced by the keyword `if` and do not include a triggering event, only a guard condition. They are executed upon state entry (after the entry actions) or if no triggered transition fires.

Timeouts A particular use case for automatic transitions is to use the `timeInState` variable in the guard condition to implement time-dependent behaviours. It contains the time since the last (re-)entry of the state. Notice that if a transition `on E -> S` fires, this counts as a reentry. If you want to "stay" in the state, then avoid the `-> S`. Note that if you do not specify a target state, then the transition must have an action. A transition with no action and no target state is illegal (because it does not do anything).

3.6 Clocks

KernelF supports clocks. There is a built-in type `clock` whose values have a `time` operation that returns the current time millis of the underlying clock. New values of type `clock` can be created by using two expressions: `systemclock` returns a clock that represents the clock of the underlying system. `artificialclock(init)` returns a clock initialized to the `init` value. Note that `artificialclock` is also of type `artificialclock`, which, in addition to `time`, also has an `advanceBy(delta)` operation that moves the clock forward by `delta` units. The `tick` operation corresponds to `advanceBy(1)`.

Artificial clocks are useful for testing. However, built-in expressions such as the `timeInState` mentioned above default to the *global clock*. By default, the global clock is the `systemclock`. If you want to use an artificial clock for testing. you must register it as the global clock using the `§global-clock` pragma.

4. Tooling

4.1 MPS-based IDE

The KernelF *language* is of course not dependent on any particular IDE. However, what makes KernelF relevant (and not just another functional language) is its extensibility and embeddability. For this, it relies on MPS' meta programming facilities. In other words, KernelF can only be sensibly used within MPS. This also means thart the IDE support MPS provides is *the* IDE support for KernelF. Like for any other langauge, MPS provides syntax highlighting, code completion, goto definition, find usages, and type checking. Because MPS is a projectional editor, it also implicitly provides formatting. Since all of this is pretty standard, we will not discuss this further.

What is worth mentioning is that this IDE support also automatically works for all extensions of KernelF, and it keeps working if KernelF is embedded into another language. No ambiguities arise from combining grammars, and no disambiguation code has to be written.

4.2 Interpreter

KernelF comes with an in-IDE interpreter that directly interprets MPS' AST. The semantic implementation of the language concepts is implemented in Java. Note that it is *not* optimized for performance (in which case a completely different architecture would be required), but for quick feedback for DSL code, in particular for test cases. The interpreter can be executed on `assert` entries in test cases; it can be started either from the context menu or with `Ctrl/Cmd-Alt-Enter`. Complete test cases and test suites can also be executed using the same menu/keys.

Notice that the interpreter performs extensive caching for expressions that have no effects. In particular, function calls with the same arguments are executed only once (per interpreter session) if the function has no effect. It is thus important that effect tracking is implemented correctly in language concepts.

4.3 Read-Eval-Print-Loop

KernelF ships with a read-eval-print-loop (REPL; Figure 7 shows an example). It is represented as its own root and is persisted; but its interaction is more like a console in the sense that whenever you evaluate an entry (using `Ctrl/Cmd-Alt-Enter`) the next one is created and focused. Each entry is numbered, and you can refer to each one using the `$N` expression.

By default, each entry in a REPL is evaluated once, and you "grow" the REPL by adding new expressions. However, by checking the `downstream updates` option, you can change any REPL expression, and all the transitively dependent ones are then reevaluated as well.

The easiest way to start a REPL is to select any expression in a KernelF program and use the `Open REPL` intention. It then creates a new REPL, adds the expression in the first entry and evaluates it. By using the `Close and Return` button in the REPL, the REPL is deleted and the node from which it was opened is put back in focus.

4.4 Debugger

One of the benefits of a functional language is that there is no program state to evolve; all computations can be

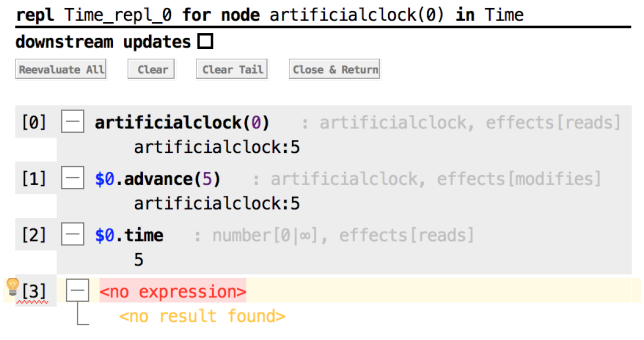

Figure 7. An example of a REPL session on a clock expression.

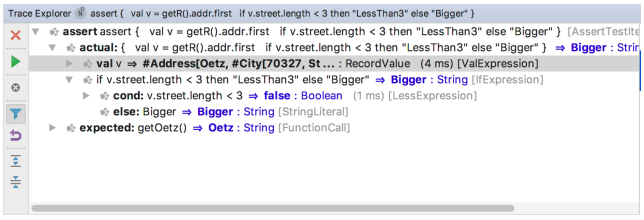

Figure 8. The frame tree as shown in the debugger.

seen as a tree of computed values. This means that debugging does not require the step-and-inspect style we know from imperative languages. Instead, debugging can just illustrate the computation tree in a convenient way.

KernelF ships with a debugger that is based on this approach. Fundamentally, a computation in the KernelF interpreter collects a trace, and this trace can be inspected.[4] The debugger, also known as the tracer, can be invoked for anything that has no upstream dependencies, i.e., test case `assertions`, gloval `values` and `functions` that have no arguments. Other domain-specific "main program like"-constructs may be available in a DSL. Whereas the interpreter is invoked via `Ctrl-Alt-Enter`, the debugger is invoked with `Ctrl-Alt-Shift-Enter` (or the `Show Trace` menu item in the context menu of the respective program node).

Debugger Components The debugger comes with three components: the frame tree, the value inspector and the code decorator; we will discuss each in turn. The frame tree shows a hierarchy of frames. Frames are "coarse-grained" entities in the computation tree such as functions and function calls, local values or `if` expressions. Importantly, the tree does not show the program nodes, it shows the computation steps involving these program nodes. This is important, because any node may be executed several times during as comoputation, but with differnet values, producing a differnet result (consider `val f = x() + x()`, recursion, or the lambdas in higher-order functions). The frame tree shows the hierarchical nesting of those computation steps. Each node in the tree has an optional label (for example, `cond` or `then`), the (abbreviated) syntax, the (abbreviated) value and the time it took to compute it[5]. The tree node shows a yellow `[E]` if that node has (had) an effect. If the node throws a constraint failure, this is highlighted in red, in place of the blue value.

Next to the frame tree we see the value inspector. When clicking on a node in the tree, the inspector shows the structure (if any) of the value of the tree node. For example, an instance of a `record` as a tree, and if an expression returns an MPS node, that node is clickable, selecting that node in the MPS editor.

When double-clicking a node in the frame tree, the respective node is decorated in the source. As shown in Figure 9, it associates a value with each AST node. Depending on the node's complexity, it shows no value at all (for literals, because the value would be the same as the node syntax), or shows it next to or below the node. The color is goverened by the nesting depth. The decorated code always represents one particular value assignment. Thus, to debug the values for `lambda` in the iterations of a `coll.where(lambda)` higher order functions, you would click on the respective nodes in the frame tree, highlighting each instance in the code.

Debugger UI The debugger opens a new frame tree for each root for which the user opens the debugger. The red X closes the current tab. The green arrow reexecutes the same root, if it is reexecutable (as determines by the debugged program node). This is useful after updating the code. Node that the expansion state of the tree is retained across reexecutions. The little grey round X removes all code decorations created by the current tab. The blue filter icon toggles between the regulae tree where only coarse-grained frames are shown and a view where *all* interpreter steps are included. While this is usually overwhelming, it can sometimes be useful. The reset arrow reverts the tree to its original expansion state (see below). The collapse all and expand all buttons should be obvious.

Breakpoints and Run To Breakpoints and Run To are two features known from classical debuggers. A breakpoint stops execution on a specific statement, and Run To runs the program until it reaches a particular statement. We have adapted these ideas in the tracer to the world of debugging functional code. A program node can be marked as `REVEAL` using an intention. Marked this way, when the debugger is invoked, the tree is expanded to show all instances of that node, marked with a red

[4] The trace can also be collected from other sources, for example, a KernelF program that has been generated to Java code, as long as the runtime also collects trace data.

[5] We might evolve the tracer to also support a simple form of profiling in the future.

Figure 9. Decorated code that associates values with syntax nodes.

```
test suite PaperDescription      execute automatically  : false
                                 Only local declarations: true

val aBool: boolean = true
val anInt: number = 42
val aReal: number{2} = 33.33
val aString: string = "H"
fun f(n: number) where [post res < 10] = 2 * n

test case BasicOperators [fail] {
  assert  42 + 33                 equals 75      [1 ms]     <number[75|75]{0}>
  assert  42 + 2 * 3              equals 48      [1 ms]     <number[48|48]{0}>
  assert  aReal + anInt           equals 75.33   [1 ms]     <number[-∞|∞]{2}>
  assert  aBool && true           equals true    [0 ms]     <boolean>
  assert  aString + ", W"         equals "H, W"  [1 ms]     <string>
  confail f(20)
  assert  if aBool then 42 else 33 equals 44     actual: 42 <number[33|42]{0}>
}
```

Figure 10. Test suites in KernelF. They can either be executed automatically (as part of MPS' type system) or on demand (by pressing `Ctrl-Enter` at any level in the suite). Color coding highlights success and failure.

`[R]`. This way it is easy to identify a particular node in an execution trace. Run To means that you execute the program to a particular point. In the tracer, the `Select Next Trace` selects the next trace for the node on which it is called in the tree. `Select All Traces` highlights all of them.

4.5 Test Execution

The default execution mechanism for test suites is the built-in interpreter. Depending on the `execute automatically` flag, tests are run automatically (technically, in the MPS type system) or manually. In the latter case, `Ctrl-Alt-Enter` triggers a test item, a test case or a test suite, depending on where `Ctrl-Alt-Enter` is pressed.

4.6 Coverage Measurement

KernelF is being used for a wide range of applications, some of them in safety-critical areas. It is thus important to ensure the quality of the language itself, plus its extensions. This is why KernelF ships with a coverage analyzer for its test cases. The coverage analyzer provides structural and interpreter coverage checking. An assessment reports various staticstics on the coverage, as shown in Figure 11.

Structural coverage means that the analyzer checks that all properties, children, and references are used in test cases. Heuristics assess the average complexity and size of the expressions in the test case. Minimum and maximum complexity thresholds can be defined to force developers to write "unit tests" (low complexity/size) and "integration tests" (higher complexity/size).

Interpreter coverage refers to the coverage of the interpreter that runs the language by default. It verifies that the evaluator for all language concepts is executed at least once. By marking branches in the interpreter, one can also ensure that all relevant branches in the interpreter code are executed at last once. Furthermore, if the interpreter works with collections (such as an argument list of a function), one can check that the interpreter runs at least once with an empty list, with a list of one element, and with a list of more than one element. Finally, the interpreter coverage analyzer can also track the ranges and distributions of numeric (and potentially other) values to make transparent the range of numbers used to exercise the interpreter.

The main limitation of the analyzer is that it does not analyze combinatorial coverage, i.e., the possible combinations of language concepts and/or value ranges.

4.7 Test Case Generation

KernelF supports test case generation; an example is shown in Figure 12. While this requires a more detailed explanation, here are the core characteristics. The generator works on any language construct that accepts a vector-style input, such as functions. There are different producers[6], currently we support `random` (which creates the specified number of random values that are each compatible with the n-th vector element's type) and `eqclass` (which seelcts "interesting" values for each type and then generates vectors with all permutations). If a vector is executed, several things can happen:

- A precondition (if one is given) can fail, reported as `[PRE] error message`. Using an intention, such vectors can be marked as `Invalid Input`, which, when running the vector again, makes the vector green. A second intention can physically remove all `Invalid Input` vectors.
- A postcondition (if one is given) can fail. This is a genuine test failure and must be adressed.

[6] Currently they work only for primitive types, not for collections or records. This will be improved in the future.

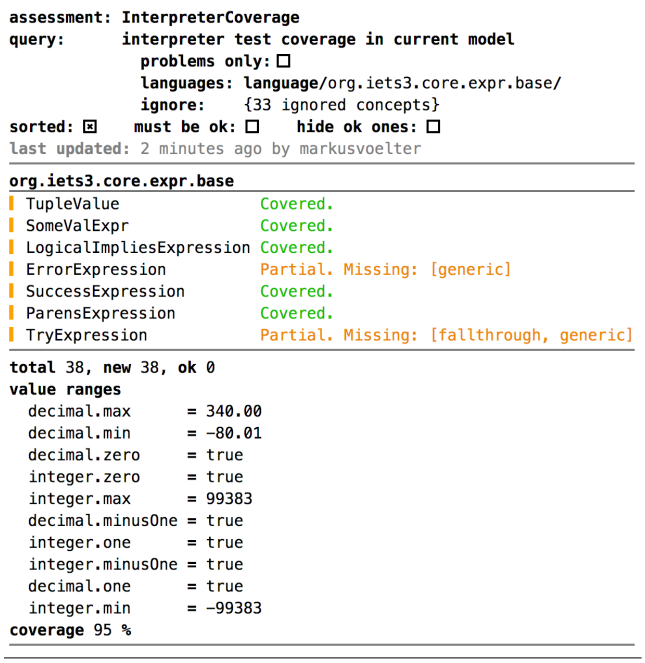

Figure 11. Example of interpreter coverage measurement. Users specify the language, concepts that should be ignored (because they are not interpreted and should hence not be part of the coverage analyis). The analyzer reports missing branches, calculates a coverage ratio, and tracks number ranges.

- If an expected result is specified, and the vector evaluates to something else, this is also a failure that must be adressed.
- If not result value is given, and no constraints fail, all vectors will succeed. The actual values can then be copied into the `result` column using an intention. While this looks initially pointless, such vectors are useful as a safety net for downstream refactorings of the test subject.

4.8 Mutation Testing

The testing infrastructure also supports mutation testing. Mutation testing is about judging the quality of a test suite by making "random" changes to the test subject and then detecting if one or more tests in the suite fail. If no test fails, this means that the tests are not specific enough. A high quality test suite is one where for each introduced mutation at least one test fails. The changes performed by the mutator are extensible; currently we support

- Replacement of boolean subexpressions with true and false
- Negation of boolean expressions
- Replacement of some arithmetic operations with others, e.g. + with *, - with /
- Replacement of some boolean relations with others, e.g. > with >=, == and <=
- Exchange of the `then` and `else` part of conditions.

Currently we support mutation testing only vector test itemis, i.e., those that define a set of test vectors for a single test subject. They are also used for the test case generation discussed above.

Figure 13 shows an example. A vector test item is used for the function `add`, and, using an intention, we have attached a `mutator` to the item. Using another intention, the specified number mutation attempts can be executed. Technically, we create a clone of the current model for each mutation; those mutations where the set of tests does *not* fail are kept around; the other are deleted (unless `keep all` is set to true). Another intention can be used to delete all the mutant models.

The original model, the one where we started the mutation process, contains pointers to all the mutated nodes to provide an overview of the problematic code locations; they are attached to the mutator with the `->` notation. Following the references leads to the mutated code which shows the new and the original node side-by-side. A couple of examples are shown in Figure 14. Note that the mutator can also touch indirectly used functions; the particular scope of the mutations is defined by the test subject adapter.

5. Design Decisions

Based on the goals for KernelF outlined in Sec. 1.1, we have made a number of design decisions which we outline in this section.

5.1 Exploit Language Workbench Technology

The core functional abstractions, and the design for robustness are independent of the technologies used for implementing the language. However, the support for embedding, extension and the IDE support relies on the fact that KernelF is designed to be used with language workbenches that support modular language extension and embedding. Specifically, we have built it on top of Jetbrains MPS.

By deciding to rely on the capabilities of MPS, IDE support comes essentially for free (a few refactorings, such as extracting an expression into a value, have been implemented manually). Similarly, the language does not require an elaborate type system or meta programming support to enable extension and embedding. Instead, we rely on the language workbench to achieve extension and embedding.

5.2 The Type System

Static Types KernelF is statically typed. This means that every type is known by the IDE (as well as the interpreter, or a future generator). If a user is interested

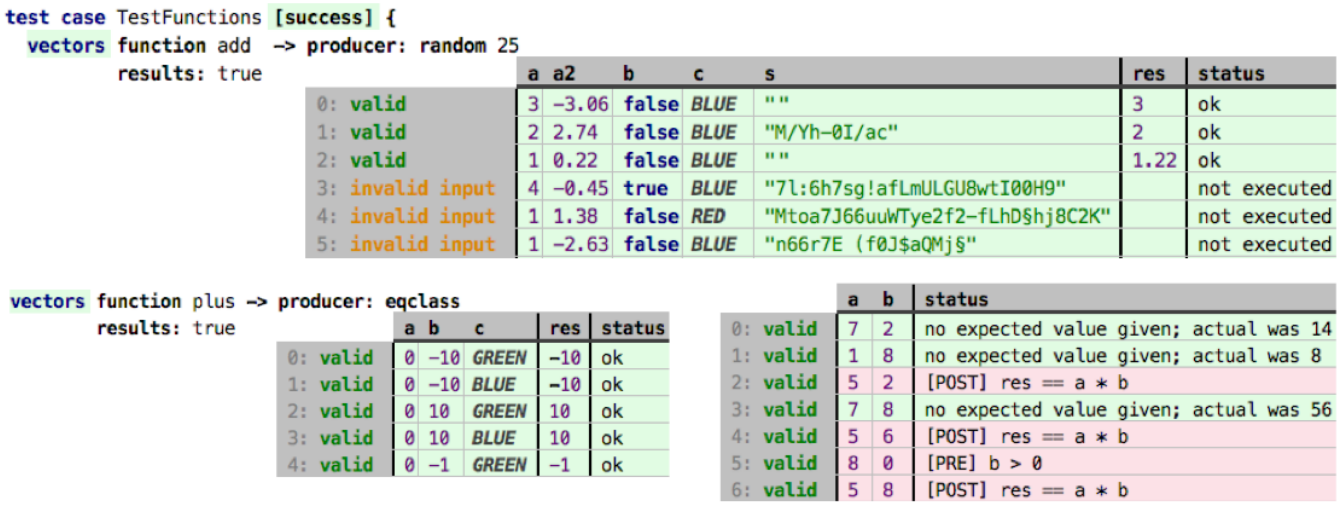

Figure 12. A couple of examples for test case generation; refer to the text for details.

```
fun doodle(a: number[1|5]) = if true then a else a * 1

fun add(a: number[1|5]) = alt [a > 3     => a + 1
                               otherwise => doodle(a)]

test case TestFunctions [incomplete] {
  vectors function add                          -> producer: random 30
          results: true                            { 30 entries }
          mutator: # of mutations 20
                   keep all:      false
                     -> ParensExpression
                     -> NumberLiteral
                     -> LogicalNotExpression
                     -> ParensExpression
                     -> ParensExpression
}
```

Figure 13. An example of mutation testing.

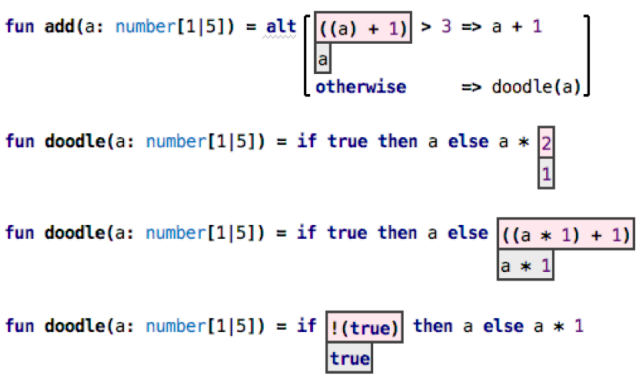

Figure 14. Highlighting of code mutations. The mutated code is red, the original one is grey.

in the type of an expression, they can always press `Ctrl-Shift-T` to see that type. This helps with the design goals of Simplicity and IDE Support, but also with Robustness, because more aspects of the semantics can be checked statically in the IDE. Examples include suntyping errors as well as violations of number ranges.

Numeric Types An early version of KernelF had `int` and `real` types, implemented as Java `long` and `double` in the interpreter. We received feedback immediately that doubles and longs are not suitable, and that the implementation should be changed to use `BigDecimal` and `BigInteger` – to get rid of the range limitations. Further feedback from business domains led to the need for explicitly specified ranges (most quantities in business domains have a range) and an explicitly specified precision (number of decimal digits). Instead of making this an optional (project-specific) extension of KernelF, we decided to replace the `int` and `real` types with `number[min|max]{prec}`, as explained in the introduction. The feedback from our users is very positive.

The type system performs simple range computations, such as those listed below.

- Number literals have a type that has a singleton range based on their value and number of decimal digits (e.g., `42.2` has the type `number[42.2|42.2]{1}`.
- Supertypes of numeric types merge the ranges (for example, the supertype of `number[5|5]`, `number[10|20]` and `number[30|50]` is `number[5|50]`. This is an overapproximation (i.e., simplification in the type system implementation), because the type system could know that, for example, the value `25` is not allowed. However, to implement this, a number type would have to have *several* ranges; we decided that this would be too complicated (both for users and the language implementor) and induce performance penalties in type checking; so we decided to live with the overapproximation.
- For arithmetic operations (currently `+`, `-`, `*` and `/`), the type system computes the correct result

ranges; for example, if variables of type `number[0|5]` and `number[3|8]` are added, the resulting type is `number[3|13]`.

- A division *always* results in an infinite precision value; if a different precision is required, the `prevision<>()` operator has to be used. Since we cannot technically represent infinite precision currently, we approximate it with a precision of 10.

We are making the simplifying tradeoffs consciously, because, in the extreme, we would have to implement a type system that supports dependent types (or abstract interpretation of code); this is clearly out of scope.

Type Inference To avoid the need to explicitly specify types (especially the `attempt` types, collections and number types can get long), KernelF supports type inference; this also helps with SIMPLICITY. The types of all constructs are inferred, with the following exceptions:

- Arguments and record members always require explicit types because they are declarations without associated expressions from which the type could be inferred.
- Recursive functions require a type because our type system cannot figure out the type of the body if this body contains a call to the same function.

If a required type is missing, an error message is annotated. Users can also use an intention on nodes that have optional type declarations (functions, constants) and have the IDE annotate the inferred type.

No Generics KernelF does not support generics in user-defined functions, another consequence of our goal of SIMPLICITY. However, the built-in collections are generic (users explicitly specify the element type) and operations like `map`, `select`, or `tail` retain the type information thanks to the type system implementation in MPS. As a consequence of the extensibility of KernelF, users can also define their own "generic" language extensions, similar to collections.

Option and Attempt Types To support our goal of ROBUSTNESS, the type system supports option types and attempt types. Option types are useful to explicitly deal with null values and force client code to deal with the situation where null (or `none`) is returned. Similarly, attempt types deal systematically with errors and force the client code to handle them (or return the `attempt` type its own caller).

We decided not to implement full support for monads; for our current use cases, this is acceptable and keeps the implementation of the type system simpler, which supports our goal of extensibility. Note that, because many operations and operators for `T` also work for `opt<T>`, users can defer dealing with options and errors until it makes sense to them; no nested `if isSome(...) ...` are required.

Effect Tracking and Types Effect tracking, as discussed in Sec. 3.1, is not implemented with the type system: an effect is not declared as part of the type signature of a function (or other construct). There are two reasons for this decision. First, for various technical reasons of the way the MPS type system engine works, this would be inefficient. Second, language extenders and embedders would have to deal with the resulting complexity when integrating with KernelF's type system. Instead, the analysis is based on the AST structure and relies on implementing the `IMayHaveEffect` interface and overriding its `hasEffect` correctly. While this is simpler for the language implementor or extender, a drawback of this approach is an overapproximation in one particular case: if you declare a function to take a function type that has an effect, then, even if a call passes a function without an effect, the call will still be marked as having an effect:

```
fun f*(g: ( =>* string)) = g.exec()* // declaration
f*(:noEffect)                        // call
```

We are working on an interprocedural data flow analysis, which will solve this problem.

5.3 Definition of the Semantics

The semantics of KernelF are given by the interpreter that ships with the language, together with a sufficiently large amount of test cases. No other formal definition of the language semantics is provided. KernelF does not ship with a generator, because, in the interest of PORTABILITY, a generator would always be target platform-specific. To align the semantics of generators with the reference semantics given by the interpreter, one can simply generate the test cases to the target platform and then run them there – if all pass, the (functional) semantics are identical.

5.4 Extension

We provide more details on extension and embedding in Sec. 7, but here is a quick overview of the typical approaches used for extension of KernelF.

Abstract Concepts A few concepts act as implicit extension points. They are defined as abstract concepts or interfaces in KernelF, so that extending languages can extend these concepts. They include `Expression` itself, `IDotTarget` (for things on the right side of a dot expression), `IFunctionLike` (for function-like callable entities with arguments), `IContracted` (for things with constraints or pre-/postconditions) and `Type` (as the super concept of all types used in KernelF). `IToplevelExprContent` is the interface implemented by all declarations (records, functions, typedefs).

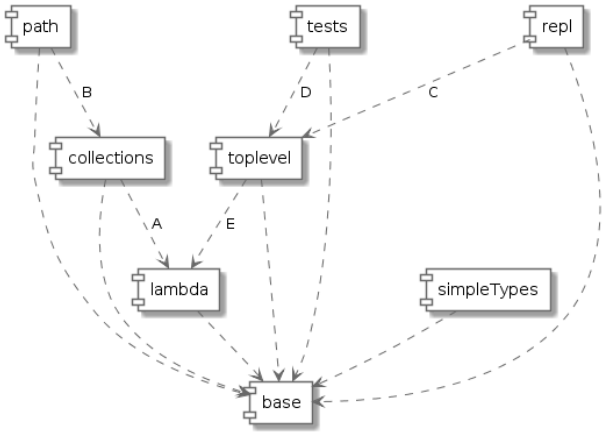

Figure 15. Dependencies between the language modules in KernelF.

Syntactic Freedom A core ingredient to extension is MPS' flexibility regarding the concrete syntax itself. As we show in Sec. 7.1, tables, trees, math or diagrams are an important enabler for making KernelF rich in terms of the user experience.

5.5 Embedding

Making a language embeddable is more challenging – at least with MPS – than making it extensible. We outline the core approaches here:

KernelF is Modular The language itself is modular; it consists of several MPS languages that can be (re-)used separately, as long as the dependencies shown in Figure 15 are respected. Importantly, it is possible to use only the basic expressions (`base`), or expressions with functional abstractions `lambda`. Nothing depends on the `simpleTypes`, so these can be exchanged as well (discussed below). We briefly discuss the dependencies (other than those to `base`) between the languages and explain why they are acceptable:

- **A:** required because of the higher-order functions (`where, map`) on the collections
- **B:** `path` navigation usually also has 1:n paths, which requires collections
- **C:** `repl` is a utility typically used when developing larger systems, which usually also use `toplevel` expressions; so the dependency does not hurt.
- **D:** `tests` are themselves top level elements; also, a dependency on `toplevel` does not hurt for a *test* model.
- **E:** the functions in `toplevel` require generic function-like support from `lambda`

Removing Concepts In many cases, embedding a language into a host language requires the removal of some of the concepts from the language. One way of achieving this is to use only those language modules that are needed; see previous paragraph. If a finer granularity is needed the host language can use constraints to prevent the use of particular concepts in specific contexts. A concept whose use is constraint this way *cannot be entered* by the user – it behaves exactly as if it were removed.

Exchangeable Primitive Types One particular part of a language that may have to be removed (or more specifically, exchanged) is the set of primitive types. As per what we have said in the previous paragraph, users can decide to not user `kernelF.primitiveTypes` or constrain away some of the primitive types. However, the type system rules in the `kernelF.base` language relies on primitive types (some built-in expressions must be typed to Boolean or integer). This means that the types constructed in those rules types must also be exchangeable. To make this possible, KernelF internally uses a factory to construct primitive types. Using an extension point, the host language can contribute a different primitive type factory, thereby completely replacing the primitive types in KernelF.

Structure vs. Types The types and the underlying typing rules can be reused independent from the language concepts. For example, if a language extension defines a its own data structures (e.g., a relational data model), the collection types from KernelF can be used to represent the type of a `1:n` relation. Examples are given in the case studies.

Scoping Scopes are used to resolve references. Every DSL (potentially) has its own way of looking up constants, functions, records, typedefs or its own domain-specific declarations. To make the lookup strategy configurable, KernelF provides an interface `IVisibleElementProvider`. Host language root concepts can implement this interface and hence control the visibility of declarations.

Overriding Syntax Imagine on embeds KernelF into a language that uses German keywords. In this case the concrete syntax (in particular, the keywords) of KernelF must be adapted. MPS' support for multiple editors for the same concepts makes this possible.

Extension Finally, embedding KernelF into a host language usually also requires extending KernelF. For example, if KernelF expression were to be used as guards in a state machine, then a new expression `EventArgRef` would be required to refer to the arguments of the event that triggered the current transition; an example is the reference to `data` after the `if` in the following snippet:

```
state machine Protocol {
  state Waiting {
    on PacketReceived(data: list<number>)
      if data.size > 0 -> Active
  }
```

```
  state Active { ... }
  ...
}
```

To this end, everything discussed in Sec. 5.4 is relevant to embedding as well.

5.6 Miscellaneous

Algebraic Data Types not Essential Option types can be seen as a special case of algebraic data types, with the following definition:

```
type option<T> is some<T> | none;
```

Similarly, attempt types could also be built with a generic algebraic data type language. However, we decided against having algebraic data types in the core of the language (they might become available as an extension) for two reasons. First, as we have outlined at the beginning of Sec. 1, we expect domain-specific data structures to be contributed by the host language, so sophisticated means of modeling data, of which algebraic data types are an example, are unnecessary. Second, by making attmpt and option types first class, we can provide support for them with special syntax and type checks (e.g., the `try` expression for attempt types) or by making an existing concept aware of them (the `if` statement wrt. option types).

No Monads We decided to not add a generic facility for (user-definable) monads, for two reasons. First, they are probably at odds with our design goal of SIMPLICITY: our users will probably not be able to understand them. More importantly, however, they make the type system much more complicated to implement in MPS. This, in turn, is a problem for extensibility, because extension developers would have to deal with this complexity.

No Exceptions KernelF does not support exceptions. The reason is that these are hard or expensive to implement on some of the expected target platforms (such as generation to C); PORTABILITY would be compromised. Instead, attempt types and the constraints can be used for error handling.

Not Designed for Building Abstractions KernelF is not optimized for building custom structural or behavioral abstractions. For example, it has no classes and no module system. The reason for this apparent deficiency lies in the layered approach to DSL design shown in Fig. 1: the DSLs in which we see KernelF used ship their own domain-specific structural and behavioral abstractions. More generally, if sophisticated abstractions are needed (for example, for concurrency), these can be added as first-class concepts through language engineering in MPS (cf. Sec. 5.1).

Keyword-rich In contrast to the tradition of functional languages, KernelF is relatively keyword-rich; which means, it has relatively many first-class language constructs. There are several reasons for this decisions, the main reason being simplified analyzability: if a language contains first-class abstractions for semantically relevant concepts, analyses are easier to build. These, in turn, enable better IDE support (helping with SIMPLICITY and making the language easier to explore for the DSL users) and also make it easier to build generators for different platforms (PORTABILITY) Finally, in contrast to languages that do not rely on a language workbench, the use of first-class concepts does not mean that the language is sealed: new first-class concepts can be added through language extension easily.

5.7 A specific example: "unpacking" options

In this section we provide a more detailed discussion of one particular language design decision to illustrate how user expectations and MPS tool capabilities lead to the final solution. We struggled with this one for a while, and this section illustrates the thought process. The example is about "unpacking" option values, i.e., checking if a valuf of type `option<T>` contains a `T` and not `none`.

The Starting Point We started with a first-class concept `with some`, plus an expression `val` that would provide access to the optioned value if it is `some` and not `none`. Having a first-class concept makes analyses simple to build, because it is simple to recognize a check for `some` because the language concept directly expresses it.

```
fun f(x: option<number>) = with some x => val + 10
```

We also experimented with using a dot expression to access the optioned value:

```
fun f(x: option<number>) = with some x => x.val none 10
```

This second version would not work for complex expression such as function calls, since repeating the complex expression before the dot is syntactically ugly and leads to errors if the called function has side effects. We decided on the first alternative.

Naming However, this alternative will result in a problem if several `with some` expressions are nested because `val` would be ambiguous. The name of the expression used to refer to the value must be changeable. One solution would be to define a value explicitly:

```
fun f(x: number, y: number) = {
  val xval = with some maybe(x) => val none 10
  with some maybe(y) => val + xval none 20
}
```

However, this is too verbose. We came up with two versions of an abbreviation to define names for the tested value:

```
fun f(x: number) = with some v = maybe(x)   => v none 10
fun f(x: number) = with some maybe(x) as v  => v none 10
```

We preferred `<expr> as <name>` over `<name> = <expr>` because it cannot be confused with an assignment (which we do not support in KernelF). It is also easier from the perspective of the user, because you can add the name (syntactically and in terms of typing sequence) after the expression the user wants to test. Finally, KernelF already has a facility for optionally naming things with an `as` suffix. The above can then be written as:

```
fun f(x: number, y: number) = {
  with some maybe(x) as xval
     => with some maybe(y) as yval => xval + yval
                                  none 0
     none 0
}
```

To avoid nesting, we allowed comma-separated tests:

```
fun f(x: number, y: number) =
  with some maybe(x) as xval, maybe(y) as yval
     => xval + yval none 0
```

Using If Expressions The first-class concept `with some` turned out to be ugly, and also introduced new keywords for something where users intuitively wanted to use an `if`; so we allowed the `if` statement to be used, again with the same options:

```
fun f(x: option<number>) = if isSome(x)     then val   else
     10
fun f(x: option<number>) = if isSome(x)     then x.val else
     10
fun f(x: number) = if isSome(maybe(x))      then val   else
     10
fun f(x: number) = if isSome(maybe(x) as v) then v     else
     10
```

A problem with using the existing `if` expression is that users can construct arbitrarily complex expressions, such as the following:

```
fun f(x: option<number>) =
  if isSome(x) || g(x) then val else 10
```

In this case it cannot (easily) be statically checked that inside the `then` branch, `x` always has a value. To enforce this, we ensure that the `isSome` expression is the topmost expression in the `if`; it cannot be combined with others. This is trivial to check structurally and avoids the need for advanced semantic analysis of complex expressions.

We had the idea of interpreting an option type as Boolean to allow this syntax:

```
fun f(x: option<number>) = if x then val else 10
```

However, we discarded this option because, for our target audience, we think that too much type magic is too complicated. Another idea was to use the name of the tested variable (if it is a simple expression) in the `then` part, and type it to the content of the option. This would allow the following syntax:

```
fun f(x: option<number>) = if isSome(x) then x else 10
```

This is harder to implement because the type of `x` is now different depending on the location in the source. This is not easily possible with MPS' type system. Alternatively, the second `x` could be made to be a different language concept (which comes with a different type), but then one has to prevent the use of the original `x` in the `then` part. This would require all reference concepts to be aware of the mechanism; every scoping function would have to call a filter method. While this makes language extension a little bit harder (users have to call the filtering function), we decided that this is worth it: since one cannot do anything else inside the `then` part, providing the "unpacked" value there makes sense.

Final Design We settled on the following syntax. The `if` conforms to users' expectations, the `as` avoids confusion with assignments, and we provided the magic of "automatic unpacking" inside the `then` part.

```
fun f(x: option<number>) = if isSome(x) then x else 10
fun f(x: number) = if isSome(maybe(x) as v) then v else 10
```

For multiple tested values we now use `&&` instead of the comma, because the `&&` is used in logical expressions already as a conjunction; note that other logical operators are not supported on `isSme` tests.

```
fun f(x: number, y: option<number>) =
    if isSome(maybe(x)) as xval && isSome(y)
       then xval + y else 0
```

6. Evolution over Time

6.1 Number Types

Initially, KernelF had been designed with the usual types for numbers: `int` and `float`. However, even in our very first customer projects it turned out that those numberic types are really too much focussed on the need of programmers (or even processors), and that almost no business domain finds those types useful. Thus we quickly implemented the number types as described earlier. Since this happened during the first real-world use, so this evolution did not involve any migration of existing, real-world models of customers, making the evolution process very simple.

6.2 Transparent Options and Attempts

Initially, option types and attempt types were more restricted than what has been described in this paper. For example, if a value of `option<T>` is expected, users had to explicitly construct a `some(t)` instead of just returning `t`. Similarly for attempt types: users had to return a `success(t)`. Options and attempts also were not transparent for operators. For example, the following code was illegal, users first had to unpack the options to get at the actual values, which lead to hard to read nested `if` expressions.

```
val something : opt<number> = 10
val noText    : opt<string> = none
something + 10 ==> 20 <option[number[-inf|inf]{0}]>
noText.length  ==> none <option[number[0|inf]{0}]>
```

The reasons for the initial decision to do it in the more strict way were twofold. One, we thought that the more explicit syntax would make it clearer for users what was going on (less magic). Instead it turned out it was perceived as unintuitive and annoying. The second reason was that the original explicit version was easier to implement in terms of the type system and the interpreter, so we decided to go with the simpler option.

The migration to the current version happened after significant end-user code had been written, and so we implemented an automatic migration where possible: all `some(t)` and `success(t)`were replaced by just `t` by migration script that was automatically executed once users opened the an existing model once the new language version was installed. The unnecessary unpackings were flagged with a warning that explained the now possible simpler version. We expected users to make the change manually because we were not able to reliably detect and transform all cases, and because automated non-trivial changes to users' code is often not desired by users.

6.3 Enums with data

Originally, enums, as described in Section 2.10, were available only in the traditional form, i.e., without associated values. However, it turned out that one major use case for enums was to use them almost like a database table, where the structured value of one enum literal would refer to another enum literal (through using tuples or records in as their value type):

```
enum T<TData> {
  t1 -> #TData(100, true,  u1)
  t2 -> #TData(200, false, u2)
  t3 -> #TData(300, true,  u2)
}

enum U<number> {
  u1 -> 42
  u2 -> 33
}
```

6.4 Records

According to our own design goal to keep KernelF small and simple, and in particular, the assumption that the host language would supply all (non-primitive) data structures, we originally did not have records. However, it turned out that this was a bidge too far: records are useful as temporary data structures, even if the hosting DSL defines the notion of a component, class or insurance contract. Records are also useful for testing many other language constructs.

However we did not add advanced features to records, such as inheritance; we reserve such features for host language domain-specific data types. However, the internal implementation infrastructure for records is based on interfaces. This way, it is very easy for extension developers to create their own, record-like structures that, for example, use custom syntax or support things like inheritance. This extension hook has been used in several KernelF-based DSLs by now.

6.5 Range Qualifiers

A very common situation is to work with ranges of numbers. With the original scope of KernelF, for example, one could use an `alt` expression to compute a value `r` based on slices of another value `t`:

```
val r = alt | t < 10           => A |
            | t < 10 && t < 20 => B | // or t.range[10..20]
            | t > 20           => C |
```

However, as our users told us, this is perceived as unintuitive. The situation gets worse once uses range checks as part of decision tables, where many more such conditions have to be used. Our solution to this approach was to create explicit range qualifiers, so one could write the following code:

```
val r = split t | < 10   => A |
                | 10..20 => B |
                |  > 20   => C |
```

Note that these are not really expressions, because, for example in `< 10`, there is no argument given on which the check has to be performed. That argument is implicit from the context. This is why these range qualifiers can only be used in surrounding expressions that have built specifically for use with range qualifiers. The `split` expression is an example. We decided to make this part of the core KernelF language instead of an extension because these constructs are used regularly.

6.6 Enhanced Effects Tracking

Originally, there was only one effect flag: an expression either had an effect or it did not. However, when extending KernelF with mutable data, it quickly became clear that we have to distinguish between read and mofify effects because, for example, a precondition or a condition in an `if` is allowed to contain expression that have read effects, but it is an error for them to have write effects. Interpreting "has effect" as "has modify effect" also does not work, because, even for expressions with read effects, caching is invalid.

So far we have decided not to distinguish further between different kinds of effects (IO, for example), because this distinction is irrelevant for our main use of effect tracking, namely caching in the interpreter.

7. Case Studies

7.1 The Utilities Extension

Context Our first case study is an extension of the core KernelF languages with more end-user friendly ways of writing complex expressions: decision tables, decision trees and mathematical notations. Figures 16, 17 and 18 show examples.

```
fun pricePerMin(time: int, region: int) =
```

	region == EUROPE	region.in[USCAN, ASIA]
time.range[0..6]	12	10
time.range[7..17]	20	22
time.range[18..24]	17	20

Figure 16. A decision tables makes a decision over two dimensions, plus an optional default value.

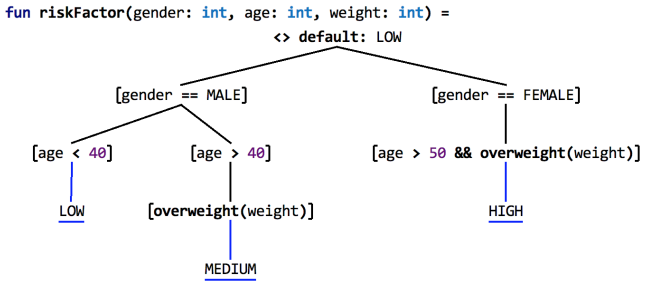

Figure 17. A decision tree directly captures a step-wise decision-making procedure found in many technical and scientific domains.

```
fun weighted(quarters: collection<FinData>) =
  quarters.size
    ∑   let [ √(q.pe / |q.rev|) ]
  i = 0     with q = quarters.at(i)
```

Figure 18. The mathematical notation helps capture mathematical calculations in a way a domain expert might write them down on paper.

Notations and Abstractions The abstractions used should be fairly obvious. Their natural notations are extremely helpful when building languages for non-programmers, since the same notations would be used in the proverbial Word document that is often the basis for capturing knowledge (informally) in non-programmer organizations. The fact that first-class logical and mathematical abstractions are used has, however, additional benefits: for example, for decision tables their completeness[7] and overlap-freedom can be checked. In our particular implementation, we do this by translating the table to the corresponding logical formulae in the Z3 solver. Errors are highlighted directly in the table.

For a table with n rows (r_i) and m columns (c_j), we detect incompleteness if the following formula is satisfiable:

$$\neg \bigvee_{i,j=1}^{n,m} (r_i \wedge c_j)$$

Similarly, an overlap between conditions ca be found by checking the following conjunctions:

$$\forall i,k = 1..n, j,l = 1..m :$$
$$i \neq k \wedge j \neq l \Rightarrow r_i \wedge c_j \wedge r_k \wedge c_l$$

[7] Assuming the range of the type is defined.

```
<default> editor for concept SumExpression
  node cell layout:
    LOOP
    lower:        [> { name } : % varType % = % lower % <]
    upper:        % upper %
    body:         % body %
    symbol:       SumSymbolSerif
    parentheses:  (node)->boolean {
                    Utils.hasFollowingExpression(node.body);
                  }
```

Figure 19. The definition of the editor for the $\sum$ Expression essentially maps the structural members (`body`, `lower`, `upper`) to predefined slots in the notational primitive for math loops.

If nested `if` expressions would be used instead of the table, no assumption about completeness can be made, and the checks could not be performed (unless the user annotates the set of nested `if` expressions with some `must be complete` annotation).

Implementation Structurally, all the new language concepts – decision tree, decision table, fraction bar, square root symbol and sum symbol – all extend `kernelf.base.Expression` so they can be used wherever an expression is expected, particular, as the implementation of functions. Some concepts are wrappers around functions; for example, the content cells of decision tables are instances of `DectionTableContent`, which in turn contain the `value` expression, but also point to their respective row and column headers to define their position in the table.

In terms of notation, we reuse existing notational primitives we have developed over the years for tables, trees and mathematical symbols. Once these are available, the definition of the concrete syntax is straightforward. Fig. 19 shows the editor definition for the sum symbol. The editors for the table and the tree are a little bit more complicated, since they dynamically construct the tree and table structures. The integration with the solver is the subject of the next subsection.

7.2 Solver Integration

Context In this case study we take a closer look at the integration of the solver: this explains more details about the architecture of the solver integration hinted at above, and it is also a case study in the use of KernelF itself.

Working with the solver, we have found a set of recurring "questions" that one asks from the solver: are the following set of expressions complete, are they overlap free, do they contract themselves, is one a subset of another, or are two expressions identical (while have different structure, think deMorgan laws). Answering many of these questions requires an often initially

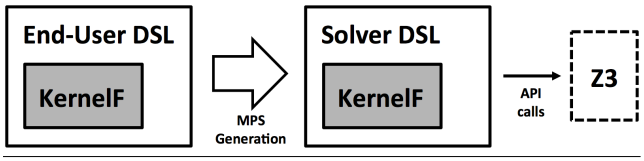

Figure 20. The integration of the solver, Z3 in our case, into end user-facing DSLs.

unintuitive encoding of the expressions in the solver (e.g., using negations).

Notations and Abstractions To avoid users' having to implement such encodings over and over again, we have developed a set of *solver tasks* that represent these questions. As shown in Fig. 20, a problem that should be addressed with the solver must be translated to one or more suitable solver tasks; these are then mapped to the solver, taking into account the unintuitive encodings. This simplifies the use of the solver (for typical problems) to the developer of a DSL. In addition, by isolating the DSL developer from the actual solver API, it also makes the solver exchangeable without any effect on the end user-DSLs: only the solver DSL with its tasks has to get a new mapping to a new solver.

Consider the following simple `alt` expression:

```
fun decide(a: int) = alt | a < 0  => 1 |
                         | a == 0 => 2 |
                         | a > 0  => 3 |
```

For this to be correct, the three conditions should be complete (there should not be a value for `a` that is not covered by any option) and it should be overlap free (for any value of `a`, only one option should apply). Below we show the encoding of these two problems in the solver DSL (layout changed to save space). These formulations are considerably simpler than the two mathematical formulae given earlier; the mapping to the solver API takes care of the mathematical encoding.

```
variables:
  a: int
relationships:
  <none>
checks:
  completeness { a < 0, a == 0, a > 0 }
  non-overlapping { a < 0, a == 0, a > 0 }
```

Implementation The Solver DSL *embeds* the KernelF expressions. To do this, the checks (completeness, non-overlapping, etc.) have children of type `kernelf.base.Expression`, as well as a type check that ensures them to be of Boolean type.

Note that the solver tasks must be self-contained, i.e., no external references are allowed. So the uses of the `a` variable in the expressions in the solver task are references to the `a` declared in the `variables` part, not to the argument of the `decide` function from which the checks are derived. This is an example of an extension required because of the embedding: a new expression, the `SolverTaskVarRef` has been introduced as part of the Solver DSL. During the transformation from the end user-visible DSL (in this case, the `alt` Expression of KernelF itself), the references to function arguments are replaced with `SolverTaskVarRef`. The mechanics of how this is done is outside the scope of this paper.

```
record Car { speed : int }
val car = #Car{0}
test case carStuff {
  assert car.accelerate to 10 with 2 equals #Car{10} [34 ms]
}
```

Figure 21. An example of natural language syntax for extension function calls: in the test case, the function is called with a multi-word text string instead of a positional argument list.

```
@syntax{accelerate to  @[to]  with  @[at] }
ext fun accelerate(this: Car, to: int, at: int)
        = #Car{to}
```

Figure 22. Associating a natural language function call syntax with an existing extension function.

7.3 Natural Language Function Calls

Context The KernelF should be usable for business DSLs where its users are non-programmers. While such users can easily deal with operators such as `+` or `&&`, the notion of function calls with its parentheses and positional arguments can be hard to communicate. These users often want a more "natural language-like" syntax.

Notations and Abstractions We continue to use regular functions and extension functions, but provide an additional call syntax, as shown in Fig. 21. The syntax is associated with existing extension functions via an annotation, as shown in Fig. 22. In this annotation, a text template can be specified that embeds the arguments at the respective places. The notation also extends into the code completion menu: when you press control space after the dot on a `Car` instance, you will get a proposal `accelerate to @[to] with @[at]`. Goto definition on the function call also still works as expected.

Implementation The template is attached to the function definition using an annotation. Annotations are an MPS feature where it is possible to attach annotation nodes `A` to other nodes `N` without the definition of `N` having to be aware of it.

The annotation contains a `Text` node that supports entering arbitrary, multi-line, unstructured text. `Text` nodes consist of a sequence of `IWord` nodes. Languages can define their own concepts that implement `IWord`, which is how the `@[..]` placehoders are built: the `NatLangFunctionArgRef` concept implements `IWord` and has a reference to a `FunctionArgument`, scoped to

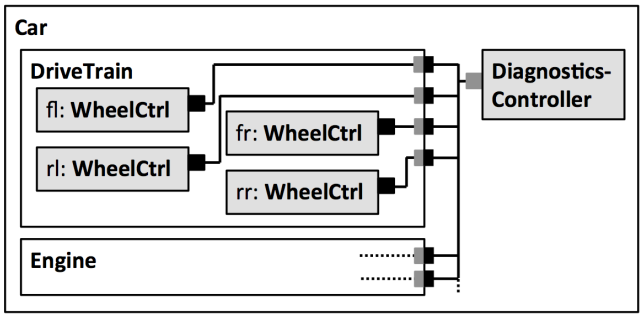

Figure 23. An example components-based system with delegating connectors.

the arguments of the function to which the annotation is attached.

The implementation of the caller syntax is a little bit more work: ca. 100 lines of customer MPS cell provider are required to render the custom cell. While, as a downside, this requires some detailed knowledge of how the MPS editor works, the plus side is that it is possible at all to add this kind of natural language-influenced function call syntax at all. Once this cell provide is available, it is embedded into the function call expression's editor like any other cell.

7.4 Components Language

Context Components-based software development relies on composing systems from reusable components with well-defined interfaces. Components expose interfaces through ports which are then connected hierarchically. One problem with this approach is that cross-cutting functionality, such as the diagnostics shown in Fig. 23, leads to a lot of connectors, some of them may even have to be delegated through many layers of component assembly. This is tedious and error prone.

Notations and Abstractions To solve this problem, some ports should be connected programmatically, i.e., by using expressions that enumerate instances and ports of specific types (e.g., the `client` port of the `WheelControl` instances) to connect those to other ports (e.g., the `server` port of the `DiagnosticsController` instance). In our system, one can write expressions, such as the following:

```
component Car {
  connect many this.allinstances<WheelController>
                   .map(|it.ports<IDiagnostics>|)
  to DiagnosticsController.server
  // more component contents
}
```

Notice the special-purpose expressions: `allinstances<T>` returns all recursively nested component instances of type `T`, and `port<P>` returns all ports with port type `P` of a given component. `this` represents the component in which we write the expression. `map` is reused from `kernelF.collections`. An alternative formulation could have been to directly recursively get all ports of type `IDiagnostics` (however, this would not illustrate the use of `map`):

```
this.allports<IDiagnostics>
```

Implementation The `Component` concept owns the `connect many` clauses, which, in turn, embed `kernelf.Expression`. Several new expressions have been implemented for this language:

The `this` expression is used to refer to the component in which the `connect many` clause lives. It is typed to be a `ComponentType` that in turn refers to the surrounding `Component`.

`allinstances` is not an expression, but an `IDotTarget`, the concept that can be used on the right side of a `DotExpression`. A can be child constraint ensures that it can only be used if the context (the expression on the left side of the dot) is of type `ComponentType`:

```
1 parentNode:DotExpression.expr.
2   type.isInstanceOf(ComponentType);
```

Since this expression returns the list of all instances of the `ComponentType` specified as the argument, it is typed to be a list of this component type: `<ListType(baseType: # allinstances.component.copy)>;` This is an example of where KernelF's existing collections are reused as the types of custom expressions; `ListType` is KernelF's regular type for lists. This way, all list operations from KernelF can be used on the collection of component instances returned by `allinstances`. The expression to return all ports `allports` works basically the same way.

A type system rule verifies that the two expressions of the `connect many` clause are typed to `PortType` or `collection<PortType>`.

7.5 Variability Models

7.6 A DSL for Medical Applications

Context A language has been developed for specifying medical algorithms. The main abstractions are components (with input/output parameters and user interactions) for realizing modularity, as well as state machines to implement asynchronous, verifiable behavior inside the components. Expressions play a role in many contexts such as invariants for data structures as well as transition guards.

As part of the behavior in the algorithms, interactions with databases are required. For example, averages of the last N measurements of certain medical parameters are required. In addition, such parameters must be stored in the database in the first place. The expressions to achieve this are expressions with sideeffects; this allows us to illustrate how effect tracking works in KernelF.

The system has several record definitions to represent system data:

```
record Patient { id: string }
record BSLMeasurement {
  measuredAt: [date,time]
  quantity  : sugarLevel
}
record BlooodPressureMeasurement {
  measuredAt: [date,time]
  systolic  : int
  diastolic : int
} where systolic  < 120 &&
       diastolic < 80  &&
       diastolic < systolic
```

The database is essentially a unstructured store that can store data for a patient. The data is tagged by the kind; we use the name of the record we want to store as that kind. The special-purpose expression `db-store` performs storage; it takes the ID of the patient as a key, and then associates the value `val` with the tag derived from the name of the record.

```
ext fun storeBloodSugarValue*(this: Patient,
                              val:  BSLMeasurement)
   = db-store*[this.id][BSLMeasurement => val]

ext fun storeBP*(this: Patient,
                 val:  BloodPressureMeasurement)
   = db-store*[this.id][BloodPressureMeasurement => val]
```

The `db-store` operation obviously has a sideeffect – storing the data in the database. To tell this to the type system, the `db-store` language concept implements `IMayHaveEffect`. Transitively, the two functions shown above also have an effect, as shown by the asterisk behind their names. If a function calls one of these, that function also gets an effect (the `cur-`expressions are keyword expressions).

```
fun userEnteredBSLMeasurement*(value: int)
   = cur-patient.storeBloodSugarValue*(
        #BSLMeasurement{[cur-date,cur-time], value})
```

Finally, if functions with sideeffects are called from a lambda, this lambda also is marked with a sideeffect:

```
fun valueSubmitted*(v: int, action: ( int =>* boolean)) {
  action.exec(v)*
  log("value submitted: " + v)
}

fun mainLoop*() {
  ..run UI code..
  valueSubmitted*(10,
     |v: int => userEnteredBSMeasurement*(v)|* )
}
```

8. Related Work

8.1 Dynamic Languages

A widespread approach for building embedded DSLs is the use of dynamic languages that support reflection and flexible syntax. Prime examples are Groovy and Ruby. However, the approach is not suitable for our purposes, for several reasons. First, the implementation based on reflection prevents static analysis and (automatic) IDE support. Second, the syntax of extensions is limited to the freedom given by the grammars of the respective language.[8] In addition, the languages are all not purely functional and provide no support for explicit effects tracking. We discarded this option early and clearly.

8.2 Other Base Languages

mbeddr C mbeddr[] is an implementation of C in MPS. As we have shown in many publications, it is very extensible; it uses the same extension mechanisms as KernelF because it is built on MPS as well. However, it is unsuitable as a general-purpose embeddable base language for DSLs because (a) it implicitly relies on many C abstractions such as the primitive data types and some operators and (b) it has all the non-functional abstractions available in C and (c) comes with its own C-specific module system.

mbeddr C is implemented in a modular way, i.e., even the core of C is split into several languages. One of them, `com.mbeddr.core.expressions`, contains only the C expressions and primitive types. In particular, it does not have user-defined data types, pointers, statements, or a module system. The idea was to make this a kind of core expression language to be hosted in other DSL. In practice, this works well *as long as that DSL generates to C*. However, even in this core language subset, there are many implicit assumptions about C, making it unsuitable as a generic, embeddable expression language; building an interpreter is also tough. It also misses many useful features, such as higher-order functions.

When we started seeing the need for a core expression language, we thought about generalizing the mbeddr expressions; however, we decided against it and started KernelF: the required changes would have been too great, making mbeddr C too complicated. The use cases are just too different.

MPS BaseLanguage MPS ships with a language called BaseLanguage – it wears its purpose clearly on its sleeve. It is fundamentally a slightly extended version of Java (for example, it had higher order functions and closures long before they were standardized as part of Java 8). It also ships with a set of (modular) extensions for meta programming, suppliying language constructs, to, for example, create, navigate and query ASTs.

BaseLanguage has been used successfully – by us and others – as the basis for DSLs. If those DSLs either extend Java or at least generate to Java, BaseLanguage is a great fit and the recommended way to go. Even though it is not built in a modular way, MPS' support for restricting languages (using constraints) is powerful enough to cut it down to a subset that is relevant in any particular DSL.

[8] Both of these points are clearly illustrated by a customer's (not very satisfying) attempt at building a whole range of business DSLs with Groovy.

However, similar to mbeddr C, it suffers from its tight connection to Java in terms of data types, operators and assumptions about the context in which expressions are used. The fact that it is not a purely functional language and does not support effects tracking also makes it much harder to analyze. It also has several features, such as generics, that make it harder to extend. Finally, its long evolution in MPS also means that it carries around a lot of baggage; we decided that it is worth the effort to build a new, clean base language.

Xbase/Xtend Xbase is a functional language that ships with Xtext. Similar to KernelF, its purpose is to be extended and embedded in the context of DSLs. Xtend is a full programming language (with classes, modules and effects) that embeds Xbase expressions. Similar to Kotlin and Ceylon, its goal is to be a better, cleaned up Java, while not being as sophisticated/complex as Scala. For the purposes of being an embeddable base language, Xtend's scope is too big (like Java or C), so we limit our discussion in this paragraph to Xbase.

In terms of its suitability as a base language, Xbase suffers from several problems. The most obvious one for our use case is that it is implemented in Xtext, and is thus useless for MPS-based languages. Of course, this does not say anything about its conceptual suitability as a core language. However, there are also two siginificant conceptual problems. First, because of the fact that it is implemented in Xtext, its support for modular extension or embedding are limited: one cannot use several independently developed extensions in the same program in a modular way. Consequently, no such extensions are known to us, or documented in the literature. Second, Xbase is very tightly coupled to Java: it uses Java classes, generates to Java and even its IDE support is realized by maintaining Java shadow models in the background. While this is a great benefit for Java-based languages (the goal of Xbase), it is a drawback if that dependency is not desired.

In terms of its core abstractions, many of the ideas of Xbase are similar: everything is an expression, functional abstractions, no modules or statements (those are supplied by Xtend).

8.3 Lisp-Style Languages

Lisp-style languages have a long tradition of being extensible with new constructs and being used at the core of other systems, such as Emacs. Racket takes this to an extreme and allows significant syntactical flexibility for Lisp itself. We decided against this style of languge for several reasons:

First, while, generally, it is a matter of taste (and of getting used to it) whether developes like or hate the syntax, it is very clear that (our) end users do not like it. Thus, adopting this syntactical style was out of the question.

Second, existing Lisp implementations are parser-based, and even the meta-programming facilities rely on integrated parsing through macros. This limits the syntactic freedom to textual notations in general, and to the capabilities of the macro system more specifically. We needed more flexibility, as shown throughout this paper.

Third, we wanted language extensions to be first-class: instead of defining them through meta programming, we wanted the power of a language workbench. Of course we could have implemented (a version of) Lisp im MPS and then used MPS' extension mechanisms to build first-class extensions. However, then we would not make use of Lisp's inherent extensibility, while still getting the end-user-unsuitable syntactic style – clearly not a good tradeoff.

Finally, Lisp language extensions only extend the *language*, not the IDE. However, for our use cases, the IDE is just as important as the language itself, so any language extension or embedding must also be known to the IDE. Lisp does not support this (at least not out of the box).

8.4 Embeddable Languages

Lua is a small, extensible and embeddable language. In contrast to KernelF, it is not functional – it has effects and statements. Also, the notion of extension relates to extending the C-based runtime system, not the frontened syntax. So, out of the box, Lua would not have been an alternative to the development of KernelF.

However, we could have reimplemented Lua in MPS and used MPS' language engineering facilities for syntactic extension. While possible, this would still mean that we would use a procedural language as opposed to a functional one, which was at odds with our design goals.

On the plus side is Lua's small and efficient runtime system. While we did not perform any comparisons, it is certainly faster than our MPS-integrated AST interpreter. However, performance considerations are not a core requirement for the IDE-integrated interpreter. If fast execution is required, we would generate to Java or C, or implement an optimized interpreter in C.

9. Conclusion

We have built KernelF as a base language. This means that it must be extensible (so new, domain-specific language constructs can be added), embeddable (so it can be used as part of a variety of host languages) and things users do not need must be removable or replaceable. Our case studies show that we have resoundingly achieved this goal. Since developing KernelF, we have used it

in all customer projects that required expressions or a full-blown programming language.

Why were we successful? Two factors contribute. One is that we have built KernelF after years and years of building DSLs. So we had a pretty good understanding of the features required for the language, and to make it extensible and embeddable. In particular, the design that enables extensiblity was based on our experience with mbeddr C, which has proven to be extensible as well. We also had a good understanding of what features *not* to include, because they are typically contributed by the hosting DSL. The second factor is MPS itself. As we have analyzed in [**?**], MPS supports this kind of modular language engineering extremely well; it has literally been designed for this purpose. In conjunction with the team's experience, the leads to an extremely powerful tool.

Future Work Can a general monad system be built so that extension developers don't have to care (much)? A generic logging system for expectations and constraints, PPC?

References

[1] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. *ACM SIGPLAN Notices*, 34(5):192–203, 1999.

[2] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, pages 201–220, 2001.