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 almost every DSL we have been building 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 the 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 and use a set of four 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 – any checkers and analyzers easier to build. However, at the core of all 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".

Reinventing this core functional language for each DSL is a huge waste of effort. An obvious solution to this problem is to develop a small functional language that can be used in 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.

1.1 Design Goals

Simplicity KernelF should be used use 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 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. It must be possible

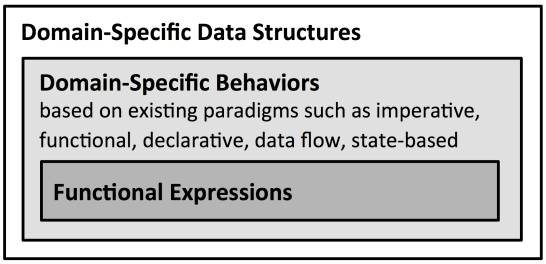

Figure 1. The three typical layers of a DSL: domain-specific data structures, behavior based on an existing paradigm, and at the core, functional expressions.

to add new types, new operators or completely new expressions, such as decision tables, to the language without invasively changing the implementation of the core language.

Embeddability Embedding refers to the ability to use the language at the core of arbitrary other languages. Several ingredients are needed: the exiting 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 (i.e., reference to) elements from this context.

Robustness The users of the DSLs that embed the 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 (pointer arithmetics in C is an obvious example). The language should also be structured in a way that makes advanced analyses, for example, through solvers, possible. It also ships with language abstractions for writing and running unit tests.

IDE Support In our experience, DSLs must ship 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[1]. 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.

2. Description

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. The example code is available here:

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

2.1 Types and Literals

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

```
val aBool   : boolean = true
val anInt   : int     = 42
val aReal   : real    = 33.33
val aString : string  = "Hello"
```

2.2 Basic Operators

KernelF provides the usual unary and binary operators, using infix notation. Precedence is similar to Java, parentheses are available.

```
42 + 33                      ==> 75 [int]
42 + 2 * 3                   ==> 48 [int]
aReal + anInt                ==> 75.33 [real]
aBool && true                ==> true [boolean]
"Hello, " + "World"          ==> "Hello, World" [string]
if [ aBool then 42 else 33 ] ==> 42 [int]
```

2.3 Null Values and Option Types

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

```
val maybe : option<int> = if [aBool then some(42) else none]
```

If the developer were to try to use `option<int>` as an `int` (e.g., by adding another value), a type error would be reported. Instead, the `int` value has to be extracted explicitly from the `option<int>`. The `with some` expression performs this task. Like an `if` expression it has two sub-expressions. The first one is evaluated if the option is `some(value)`, and provides access to the `value`. The second subexpression is evaluated otherwise, i.e., if the option was `none`. In this subexpression, it is syntactically impossible to access the (non-existent) value in the option.

```
with some maybe     => maybe.value else 0 ==> 42 [int]
```

[1] `SELECT <fields> FROM <table>` vs. `from <table> select <fields>`

There are two forms of the `with some` expression. The one above uses the `.value` notation to access the payload of the `some` value. The second one, shown below, associates a temporary variable, `v` in the example, with the payload. Both the `.value` and the temporary variable can only be used in some case, and not in the `else` branch.

```
with some v = maybe => v else 0           ==> 42 [int]
```

2.4 Error Handling using Attempt Types

Like null values via option types, KernelF also provides type system support for handling errors. The key are so-called `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.

As with option types, 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 `success(<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`. Note that you cannot mix non-attempt types and attempt types. So just returning `theHTML` instead of `success(theHTML)` would not be legal.

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

The client has to "unpack" the payload from the attempt type using the `try` expression. It works similarly to the `with some` expression in that, in the successful case, the `val` expression provides access to the payload of the attempt type. Errors can either be handled on by one (as shown in `??`), or with a generic `error` clause. The type checker makes sure that every error is handled, and intentions are available to add missing clauses. Also similar to `with some`, it is possible to assign a name to the called function, so that name can be used instead of `val` in the success case.

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

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; `??` 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).

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

KernelF also supports extension functions. They must have at least one argument that acts as the `this` variable. They can then be called in 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.

```
extension fun somethingInIt(this: list<int>) = this.size != 0
list(1, 2, 3).somethingInIt() ==> 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 takes two `int`s and returns and `int`. The second one represents functions that map one `int` 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.

```
typedef INT_BINOP : (int, int => int)
typedef INT_UNOP  : (int => int)
```

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 also define a closure, i.e., an anonymous function, and assign it to a similarly typed variable `mulCls`.

```
fun mul(a: int, b: int) = a * b
val mulFun: INT_BINOP   = :mul
val mulCls: INT_BINOP   =|a: int, b: int => 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: int, y: int, op: INT_BINOP) =
                                        op.exec(x, y)
```

```
doWithTwoInts(2, 3, mulCls) ==> 6 [int]
doWithTwoInts(2, 3, mulFun) ==> 6 [int]
```

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` is a function that takes one argument, because the other one has already been bound to the value 2. We could add an optional type to the constant 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 [int]
```

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 arguments.

```
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 (use code completion in the IDE to find the remaining operations). Of course, all operations do not modify the value on which they are called, as illustrated by the second line, where the original `reals` list is still `list(1.41, 2.71, 3.14)`.

```
reals.add(1)       ==> list(1.41, 2.71, 3.14, 1) [list<real>]
reals              ==> list(1.41, 2.71, 3.14)    [list<real>]
reals.at(1)        ==> 2.71                      [real]
reals[2]           ==> 3.14                      [real]
names.isEmpty      ==> false                     [boolean]
names.size         ==> 2                         [int]
hometowns["Tamas"] ==> "Puspokladany"            [string]
```

The expected 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: int) = it > 2
ints.where(:isGreaterTwo)    ==> list(3, 4) [list<int>]
ints.where(|int r => r > 2|) ==> list(3, 4) [list<int>]
ints.where(|it > 2|)         ==> list(3, 4) [list<int>]
```

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

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

2.8 Tuples, Records, and Path Expressions

Tuples are non-declared multi-element values. The type declaration is written as `[T1, T2, .., Tn]`, and the literals look essentially the same way: `[expr1, expr2, .., exprN]`

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

Records are structured data, they are explicitly declared. KernelF has them primarily for 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", some("M"), "Szabo"})
}
```

Finally, path expressions can be used to navigate tree 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]
```

2.9 Unit Tests and Constraints

Built-in support for unit tests is important, because, as we describe in Sec. 3.3, 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.

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. By default, the only 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. Similarly, 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. A test suite finally groups tests, plus other top level contents (records, functions, constants). It is also

```
test suite PaperDescription2    execute automatically  : true
                                Only local declarations: true

val aBool : boolean = true
val anInt : int = 42
val aReal : real = 33.33
val aString : string = "Hello"

test case BasicOperators {
  assert 42 + 33                    equals 75             [327 ms]
  assert 42 + 2 * 3                 equals 48             [2 ms]
  assert aReal + anInt              equals 75.33          [3 ms]
  assert aBool && true              equals true           [2 ms]
  assert "Hello, " + "World"        equals "Hello, World" [3 ms]
  assert if [aBool then 42 else 33] equals 44             actual: 42
}
```

Figure 2. 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.

possible to reference entities outside the test suite. Fig. 2 shows an example.

KernelF also supports checking of runtime constraints. Four forms exist, all illustrated in Fig. 3. The first one is a constraint attached to a value. It is checked after the value has been computed. The second one are constraints on types defined via `typedefs`. They are checked whenever a value is checked against a 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 anded. The third form is a type annotation on an arbitrary (nested) expression; it checks the type and also the constraints associated with the type. The fourth option are pre- and postconditions on functions. 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 interpreter produces the following result for the failure of the postcondition shown in Fig. 3. Note that the long URL in line to 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.in[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)
```

Note that in case of a failed constraint, the program continues normally, but an error is logged. If, in the example, above, the error should be communicated back to the called, the regular error handling should be used:

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

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

fun oddOrEven(i: int) where [pre  i.range[1..4]
                             post res.in[0, 1] ] = alt [i == 1 => 0
                                                        i == 2 => 1
                                                        i == 3 => 0
                                                        i == 4 => 1]

typedef posint: int where [it >= 0]
typedef age: posint where [it < 120]

fun heuristic(age: age) = √[2 * age]^posint
```

Figure 3. Constraints can be attached to values, to functions (in the form of pre- and postconditions), 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.

3. Design Decisions

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

3.1 Exploit Language Workbench Technology

The fact that KernelF is a functional language, the core functional abstractions and the design for robustness is 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 a elaborate type system or meta programming support to enable extension and embedding. Instead, a relatively number of specific design and implementation decisions enable extension and embedding; we detail those below.

3.2 The Type System

Static Types KernelF is statically-typed. This means that every type is known by the IDE. If a user is interested in the type of an expression, they can always press `Ctrl-Shift-T` to see the type of any expression. 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.

Type Inference To avoid the need to explicity specify types (especially the `attempt` types can get long), KernelF supports type inferenc; this also helps with

SIMPLICITY. The types of all constructs are inferred, with the following exceptions:

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

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` or `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 define their own "generic" language extensions.

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).

3.3 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 general, because, in the interest of PORTABILITY, a generator would always be target platform-specific. To align the semantics of generators with that 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.

3.4 Extension

We provide more details on extension and embedding in Sec. 4, but here is a quick overview of the typical approaches that are used for extension.

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).

Syntactic Freedom A core ingredient to extension is MPS' flexibility regarding the concrete syntax itself. As we show in Sec. 4.1, tables, trees, math or diagrams are an important enabler for making KernelF rich in terms of the user experience.

3.5 Embedding

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

Removing Concepts In many cases, embedding a language into a host language requires the removal of some of the concepts from the language. While one cannot technically *remove* concepts from a language definition (non-invasively), 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. Additionally, KernelF itself consists of six different MPS language definitions: `base, primitiveTypes, lambda, collections, toplevel, tests`. By not using some of them, whole groups of concepts are effectively 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). So those rule-supplied 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.

The Type System 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.

Interfaces for Scoping Scopes are used to resolve references. Every DSL (potentially) has its own way of looking up constants, functions, records or typedefs. To make this possible, KernelF provides an interfaces (`IConstantScopeProvider`, `IFunctionScopeProvider`, ...). Host language root concepts can implement these interface and hence control the visibility of these concepts.

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: int[]) if data.size > 0 -> Active
  }
  state Active { ... }
  ...
}
```

To this end, everything discussed in Sec. 3.4 is relevant to embedding as well.

3.6 Miscellaneous

No Algebraic Data Types Option types can be seen as a special case of algebraic data types, with the following definition:

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

We decided against supporting algebraic data types (and user-defined generics) for reasons of SIMPLICITY. Also, 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.

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

```
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 4. A decision tables makes a decision over two dimensions, plus an optional default value.

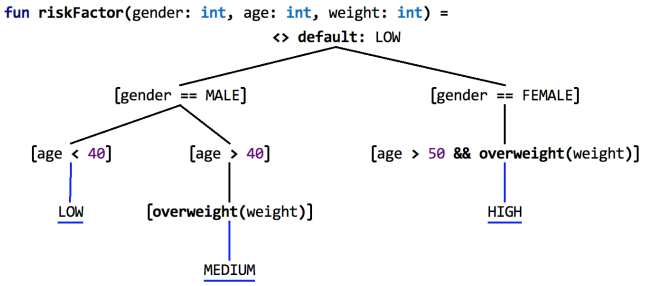

Figure 5. A decision tree directly captures a step-wise decision-making procedure found in many technical and scientific domains.

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. 3.1).

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.

4. Case Studies

4.1 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 4, 5 and 6 show examples.

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

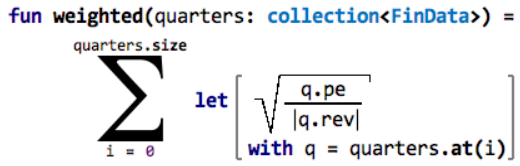

Figure 6. The mathematical notation helps capture mathematical calculations in a way a domain expert might write them down on paper.

organizations. The fact that first-class logical and mathematical abstractions are used has, however, additional benefits: for example, for decision tables their completeness[2] 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:

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

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. 7 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.

[2] 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 7. 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.

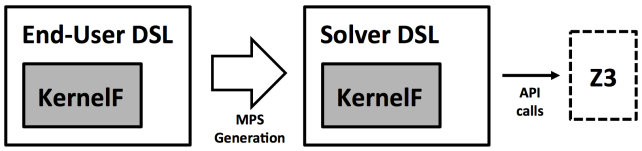

Figure 8. The integration of the solver, Z3 in our case, into end user-facing DSLs.

4.2 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 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. 8, 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.

4.3 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. 9, 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>
```

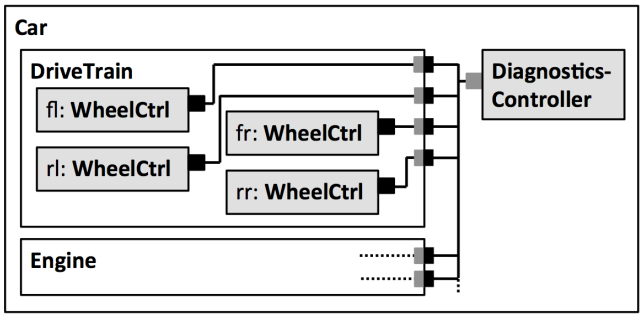

Figure 9. An example components-based system with delegating connectors.

```
                .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`.

new expressions (this, allports, allinstances) new types (port type, instance type) existing collections (so we can map) interpreter

4.4 A Schema Language

4.5 A DSL for Medical Applications

4.6 Cloud-based Application Development

5. Related Work

6. Conclusion

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