# DESIGN, EVOLUTION AND USE of *KernelF*

## An Extensible and Embeddable Functional Language

KernelF is a functional language built on top of MPS. It is designed to be highly extensible and embeddable in order to support its use at the core of domain-specific languages, realising an approach we sometimes call Funclerative Programming. "Funclerative" is of course a mash-up of "functional" and "declarative" and refers to the idea of using functional programming in the small, and declarative language constructs for the larger-scale, often domain-specific, structures in a program. We have used KernelF in a wide range of languages including health and medicine, insurance contract definition, security analysis, salary calculations, smart contracts and language-definition. In this paper, I illustrate the evolution of KernelF over the last two years. I discuss requirements on the language, and how those drove design decisions. I showcase a couple of the DSLs we built on top of KernelF to explain how MPS was used to enable the necessary language modularity. The paper also serves as a reference for KernelF.s

**Markus Voelter**
voelter@acm.org

# The Design, Evolution, and Use of KernelF
## An Extensible and Embeddable Functional Language

Markus Voelter

independent/itemis
voelter@acm.org • http://voelter.de

**Abstract.** KernelF is a functional language built on top of MPS. It is designed to be highly extensible and embeddable in order to support its use at the core of domain-specific languages, realising an approach we sometimes call Funclerative Programming. "Funclerative" is of course a mash-up of "functional" and "declarative" and refers to the idea of using functional programming in the small, and declarative language constructs for the larger-scale, often domain-specific, structures in a program. We have used KernelF in a wide range of languages including health and medicine, insurance contract definition, security analysis, salary calculations, smart contracts and language-definition. In this paper, I illustrate the evolution of KernelF over the last two years. I discuss requirements on the language, and how those drove design decisions. I showcase a couple of the DSLs we built on top of KernelF to explain how MPS was used to enable the necessary language modularity. I demonstrate how we have integrated the Z3 solver to verify some aspects of programs. I present the architecture we have used to use KernelF-based DSLs in safety-critical environments. I close the keynote with an outlook on how KernelF might evolve in the future, and point out a few challenges for which we don't yet have good solutions.

**Keywords:** Domain-specific Languages, Language Modularity, Function Programming, Language Engineering, Meta Programming

## 1 Introduction

### 1.1 Funclerative Programming

Functional programming is suitable for programming in the small [8], for compact algorithms. It is not ideally suited for programming in the large. Reasons include the lack of means for grouping functions into interfaces, hiding information, and defining contracts. To compensate for this, languages combine the functional paradigm with other paradigms, such as object-oriented programming in Scala [22]. Higher-level frameworks such as MapReduce [7] also provide more coarse-grained control over program execution that goes beyond the typical building blocks of functional languages: function calls, higher order functions and monads.

KernelF combines functional programming in the small, and declarative structures and behaviours in the large, in an approach we sometimes call "funclerative programming". Instead of using one particular paradigm for providing

coarse-grained behaviors and structure to programs, we extend a functional core language with custom, domain-specific abstractions.

### 1.2 Domain-Specific Languages

The need for custom abstractions on top of a functional core arises from domain-specific languages (DSLs). In our industry work, we[1] develop many different DSLs in a wide variety of different domains (we show a few examples in Section 5). All except the very trivial ones require a "calculation core": arithmetics, comparison, logical expressions, as well as functions, records and enums. Functional programming is perfectly suited for this task, because the lack of side-effects makes programs easy to analyse, and hence, safe to integrate into a DSL.

However, for most real-world DSL, functional abstractions alone are not sufficient. Instead, higher-level abstractions for the coarse-grained, often stateful behaviors are required, such as state machines, data flow or imperative programming. Finally, these DSLs operate on domain-specific data structures such as treatment logs in healthcare, insurance products or contract definitions in logistics. Constructing these from functional abstractions (or classes/objects) alone is not practical, since the result would be too limiting in terms of notation, static analyzability and IDE support. Thus, a three layer architecture for DSLs is typical in our work:

- Layer 1: Functional abstractions
- Layer 2: Higher-level behaviors, based on established paradigms
- Layer 3: Domain-specific data structures

### 1.3 A Reusable Functional Kernel Language

The domain specificity resides mostly in layers two and three, so there is potential for reuse of the functional abstractions of layer one. KernelF, the language discussed in this paper, is a functional language optimized for reuse as layer one. To make this feasible, it must be extensible, restrictable and configurable.

**Extension** Extension refers to adding additional language constructs to the language. For example, if KernelF is used to express guard conditions in the transitions of state machines, it must be possible to add new expressions that refer to the parameters of the events that trigger the transition. This must be possible *without* invasively modifying the definition of KernelF itself, and the extension must comprise structure, notation, scoping and type systems. To further enhance the potential for reuse, independently developed extensions should be combinable, again without invasive modification of the definition of any of the used languages (a feature called extension composition in [11]).

**Restriction** This refers to the ability to not expose certain language concepts to the end user; for example, a DSL might not need support for `enum`s or option

[1] 'We' referes to the team of languages engineers at itemis Stuttgart.

types, so it must be possible to remove all traces of those concepts from KernelF when it is used in a particular DSL. In particular, the associated keywords should not be recognised and the IDE should not propose all related concepts in the code completion menu.

**Configurability** In KernelF, this refers specifically to the ability to replace the primitive types. Often, a DSL will come with its own notion of numbers or strings, and those must then be used by KernelF. This is not exactly the same problem as restriction or extension because the type system will internally rely on those primitive types. Consider a `size` operation; the type system must type this operation to whatever (positive) integral type used by the surrounding DSL, so the *primitive types used by built-in operators* must be configurable.

### 1.4 Design Guidelines for the use in DSLs

KernelF is intended to be used as the calculation core of DSLs. Many of the users of these DSLs may not be programmers – most will certainly not be experts in functional programming. To make the language suitable for this purpose, it should adhere to the following guidelines, in addition to being extensible, restrictable and configurable, as discussed above.

**Simplicity** Users should not be surprised or overwhelmed. Thus, the language should use familiar or easy to learn abstractions and notations wherever possible. Advanced concepts, such as function composition or monads are not suitable. More generally, the ability to allow users to define their own (structural or behavioral) abstractions in their programs can be limited (in the service of the goal of simplicity), because those can be provided in domain-specific language extensions. A subrequirement of simplicity is **readability**; it is particularly relevant because many of the potential users who write KernelF code will start out by *reading* KernelF code when reviewing code written by other users. Scaring prospective users away during the reading phase is not helpful.

**Robustness** Since the users of the DSLs that embeds KernelF may not be experienced programmers the language should not have features that make it easy to make dangerous mistakes (such as pointer arithmetics, unbounded strings or overflow for numbers). To the contrary, the language should make ”doing the right thing” easy. 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, to detect unhandled cases in `switch`-style constructs.

**IDE Integration** DSLs must come with good an IDE, otherwise they are not accepted by users. This means 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. IDE support is a way of achieving **writability**, i.e., the ease with which code can be written. Writability is often at odds with readability, which is why we optimize the syntax, once written, for readability, and use IDE support to simplify writing code. In addition, programs should be executable with a short turnaround, to support end users to “play”

with the programs. Seeing what a program does is often easier for inexperienced users than imagining a program's behavior based on the program code.

**Portability** The various languages into which KernelF is embedded use different means of execution such as code generation to Java and C, direct execution by interpreting the AST 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.

### 1.5 Language Engineering and MPS

KernelF, and all the DSLs discussed in this paper, are built with Jetbrains MPS.[2] MPS is a language workbench [12], a tool for developing ecosystems of languages. MPS has been used for many interesting and significantly-sized languages over the last years, the biggest one probably being mbeddr [29,32], a set of C extensions optimized for embedded programming. MPS supports a wide range of modular language composition, in particular, extension and restriction are supported directly [26]. This is possible because of two fundamental properties of MPS. First, it relies on a projectional editor. Because projectional editors do not use parsing, no syntactic ambiguities arise when independently developed languages are combined. Second, MPS has been designed to not just develop one language, but ecosystems of collaborating languages. The formalisms for defining structure, type systems and scopes have all been designed with modularity and composition in mind; some details on language development with MPS are shown in Appendix D, and the general MPS language design philosophy is discussed in Appendix E. We analyze MPS' suitability for modular language composition based on experience with mbeddr in [30] (the paper also evaluates MPS more generally). MPS' projectional editor also allows the use of a wide range of different notations such as tables, diagrams, math symbols as well as structured ("code") and unstructured ("prose") text [31], a feature we exploit extensively in the construction of DSLs. Projectional editors have historically had a bad reputation regarding usability. However, recent advances as implemented natively in MPS and in an extension called grammar cells [33] lead to good editor productivity and user acceptance [3].

## 2 KernelF Overview

### 2.1 Language

In this section I point out the most important language features of KernelF. For all of them, Appendix A provides more details and code examples; for many of these features we also show examples part of the case studies in Section 5.

[2] https://www.jetbrains.com/mps/

**Purity and Effects** At its core, KernelF is a pure language. All expressions are effect-free. There are no variables, only named (local and global) values. All values, including collections are immutable. Of course, no sensible program can be written this way; but it is expected that the hosting DSL has domain-specific means of dealing with state. The core language thus supports effect tracking; each expression can describe whether it performs a read or modify effect.

**Types, Literals and Operators** KernelF comes with Boolean and string types which work as one would expect. Numeric types comprise int and real, even though they are constrained out of the language in most of the DSLs. Instead, the number[min|max]{decimals} type is used, where the range and precision are explicitly specified. The type system performs range calculations for added type safety, and a change of the number of decimals has to be performed explicitly. The usual operators are defined on those types. No null values are supported, instead, the language supports option types (written as opt<T> for any type T). Type checking is static, and most types can be inferred (exceptions are function arguments, record members and return types for recursive functions). Finally, KernelF supports type definitions written as type <name>: <OriginalType>, useful for numbers with ranges/precisions, collections, and constraints (see below).

**Loops and Conditionals** KernelF has no loops (except higher-order functions on collections). The basic if <cond> then <expr-1> else <expr-2> distinguishes between two cases, whereas alt | <cond-1> => <expr-1> ... <cond-n> => <expr-n> |, laid out vertically, evaluates to expr-i if cond-i holds. if is also used to test options: if isSome(v) then v else w returns a T if v is of type opt<T> and v actually contains a value; it returns w if v contains a none. Various additional conditionals, in particular, decision tables and decision trees, are supported as part of a language extension.

**Functions and Blocks** Functions use the usual syntax. Argument types have to be specified, the return type can be inferred except for recursive functions. The block expression, which is used instead of let, is written as

{<expr-1> ... <expr-n> <expr-ret>}, laid out vertically. The block

evaluates to <expr-ret>, and all other expressions must either have an effect or must be local values that are referenced downstream, written as val v = <expr>. Function types are written as (T-1, T-2, ... T-n => T). Values of function types can executed using the () operator. Currying is supported via f.bind(v) if f is a function value. Lambdas are written as |a-1: T-1, ... a-n: T-n: <expr>| or, for lambdas with one argument which is then named it, as |... it ...|. References to functions (which can be used as values for function types) are written as :f for any function f. KernelF also supports extension functions where the first argument can be written as the left side of a dot expression.

**Error Handling** Language support for error handling relies on attempt types. Typically used with functions, if the function returns a T plus one of several errors, then the return type is attempt<T|E-1,... E-n> where the Es are error literals. Error values can be returned using error(E); clients can react to errors using

```
try <e> => <s> error <E-1> => <e-1> ... error <E-n> => <e-n>
```

, where `<expr>` has an attempt type, and the overall `try` evaluates to `<success>` if `<expr>` does not represent an error, or one of the `<expr-i>` if `expr` evaluates to an error literal `E-i`.

**Collections** Lists, sets and maps are supported, together with the usual higher-order functions. Collections specify their element type, plus an optional size constraint, e.g. `list<T>[min|max]`. Literals use the same keyword; for example, `set(1, 2, 3)` or `map("Joe" => 12, "Jim" => 100)`.

**User-defined Types** KernelF supports `enums`, both plain and with associated values. Tuples are supported as well, their types are written as `[<T-1>, ... <T-n>]` and their values are written as `[<expr-1>, ... <expr-n>]`. Member are accessed positionally, using array-access notation (`tuplevalue[p]`). Records are declared using a Pascal-like notation, record values are constructed via `#T(<expr-1>, ..., <expr-n>)` or a semi-graphical `build<T>` expression. Members are accessed using dot notation.

**Constraints** KernelF supports constraints that are checked at runtime. They appear in several places, usually after the `where` keyword. `type` definitions can constrain the values; `records` can constrain their members, function can define pre- and postconditions, which typically constrain parameters or return values.

**Boxes and Transactions** KernelF makes the notion of mutable state explicit through boxes. A value `v` of type `box<T>` represents an immutable reference to a mutable "memory location", of type `T` (similar to `refs` in Clojure [16]). The box contents can change over time, but each value in the box is immutable. `v.val` accesses the value inside the box, `v.update(<expr>)` sets the contents of the box to `expr`. Inside the `update`, the `it` expression represents the current value; this way, evolutions of the box contents can be written in a compact form, as in this example for a box `lb` of type `box<list<string>>`, where an additional value is appended to the contents of the box: `lb.update(it.plus("additionalEntry"))`. To make working with boxes safe, `.val` has a `read` effect, and `update` has a `modify` effect. Modifications to multiple boxes can be grouped into transactions. An failed update to any box, for example, because of a violation of a type constraint, rolls back the updates on all boxes.

**State Machines** Once we had boxes to store evolving state, it was obvious that we need first-class support for expressing behavior that depends on state, i.e., state machines. KernelF state machines declare states, one of them initial, and the states can also be nested. Machines also declare events, which can optionally have arguments. State machines are passive, i.e., they have to be actively triggered by passing an event (and optionally, arguments) into an instance. A state owns transitions which, reacting to an event, bring the machine into a new target state. There are also automatic transitions that can be triggered by timeouts or other implicitly occurring events. State machines support entry and exit actions on states as well as transition actions.

### 2.2 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. 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 (relevant, functional) semantics are identical.

### 2.3 Tooling

Similar to the previous subsection, this one provides an overview over the tooling provided for KernelF; details are in Appendix B. Tooling is crucial for the acceptance of DSLs with their users, and all tooling discussed here for the core of KernelF is also available for the DSLs built on top of KernelF.

An **IDE**, implicitly provided by MPS, supports the usual editor features (syntax coloring, formatting, error markup, code completion, go to definition, find usages, tooltips) as well as version control integration including diff/merge support for arbitrary syntax. An **interpreter** is integrated directly into the IDE, supporting live execution of (suitably structured) programs. The interpreter is implemented in Java. A **code generator** to Java is available because most of the DSLs we build are ultimately mapped to Java code. To make semantic alignment with the interpreter easier, the generated code relies on the same persistent collections library as the interpreter, and also uses Java's `BigInteger`/`BigDecimal` for numbers. A **read-eval-print-loop** (REPL) is available for interactive use of the language. A **debugger** is available, it relies on rendering the execution trace as a tree, and overlaid directly over the code. One language module of KernelF supports writing **tests**, and, relying on the interpreter, they can be executed directly in the IDE, leading to the usual red/green visual feedback, directly in the code. Taken together, the REPL, tests, interpreter and debugger lead to a very "live" programming experience with quick feedback. To ensure test quality, KernelF supports **coverage measurement**, both structural (are all language features used, and how) and relative to the interpreter (are all parts of the interpreter executed). KernelF's test infrastructure also supports **test case generation** for language constructs that take arguments lists (functions, records) as well as **mutation testing** with interactive visualisation of the mutated code. Finally, we are in the process of integrating KernelF with the Z3 **solver** to provide advanced error checking.

## 3 Design Decision

Based on the goals for KernelF outlined in Section 1, we have made the design decisions outlined in this section.

### 3.1 General Design Decisions

**Static Types** KernelF is statically typed. This means that every type is known by the IDE (as well as the interpreter or generator). If a user is interested 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 [ IDESUPPORT ], but also with [ ROBUSTNESS ], because more aspects of the semantics can be checked statically in the IDE. For example, the number ranges discussed below are an example of such advanced checks.

**Numeric Types** Instead of `int` and `real` types known from programming languages, KernelF uses the `number[min|max]{prec}` type. This is motivated primarily be [ ROBUSTNESS ] because it supports more end-user relevant checks. 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 over approximation (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 over approximation.
- 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.

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 supports both [ READABILITY ]and [ WRITEABILITY ]. 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 to infer the type.
- 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. Domain-specific extensions 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. Options force client code to deal with the possibility of `null` (or `none`) values in programs. Similarly, attempt types deal systematically with errors and force the client code to handle them (or return the `attempt` type to its own caller). The design of the syntax for testing for and unpacking options was subject of a lot of fine-grained design discussion. See Appendix C for details.

**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.

**No Reflection or Meta Programming** By deciding to rely on the language engineering capabilities of MPS, the language does not require an elaborate reflective type system (like Scala) or meta programming support to enable extension and embedding.

**No Function Composition and Monads** 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 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 `effectDescriptor` method correctly. While this is simpler for the language implementor or extender, a drawback of this approach is an over approximation 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
```

**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 at the end of Section 1.2: 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.

There are also no algebraic data types. Option types and attempt types can be seen as a special case of algebraic data types, but we decided against implementing the general case for two reasons. The first reason is the general non-need for building abstractions. And second, by making attempt and option types first class, we can support 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).

**Keyword-rich** In contrast to the tradition of functional languages, KernelF is keyword-rich; 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.

### 3.2 Extension and Embedding

Here is a quick overview of the typical approaches used for extension of KernelF. We illustrate all of them in our case studies in Section 5.

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

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

**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 1 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 replaced by a different set of primitive types (discussed below). We briefly discuss the dependencies (other than those to `base`) between the languages and explain why they exist and/or why they do not hurt:

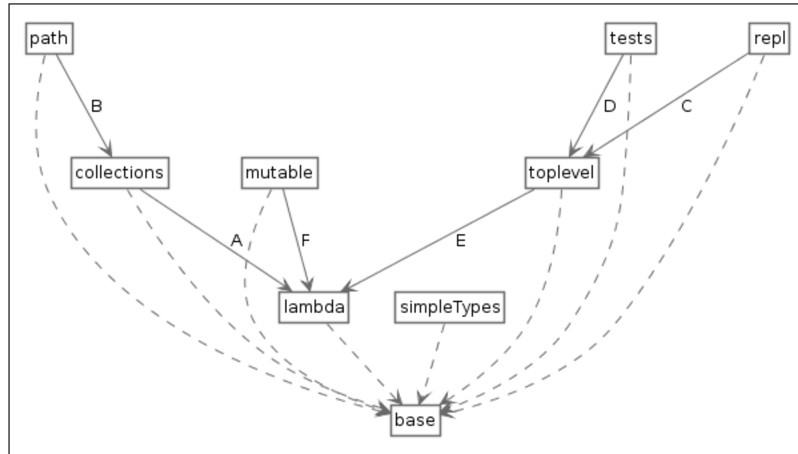


**Fig. 1.** Dependencies between the language modules in KernelF.

- **A:** required because of higher-order functions (`where, map`) on 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:** functions in `toplevel` require generic function-like support from `lambda`
- **F:** the transactions in `mutable` require the blocks 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 actually removed from the language.

**Exchangeable Primitive Types** Many DSLs come with their own primitive types, so it is crucial that it is possible to *not* use `kernelF.primitiveTypes` when KernelF is embedded into a particular DSL. Preventing the user from entering a particular type into the program can be achieved with the approach described in the previous paragraph. However, the type system rules in the `kernelF.base` language rely 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.

**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 embedding KernelF into a language that uses German keywords: the keywords of KernelF must now be adapted. MPS' support for multiple editors for the same concepts makes this possible.

## 4 Evolution

**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 numeric 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, this evolution did not involve any migration of existing, real-world models of customers, making the evolution process very simple.

**Transparent Options and Attempts** Initially, option types and attempt types were more restrictive than what has been described in this paper. For example, if a value of `option<T>` is expected, users had to return `some(t)` instead of just `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.

**Enums with Data** Originally, enums 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 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
}
```

**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 bad idea: 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.

The internal implementation 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 features such as inheritance. This extension hook has been used in several KernelF-based DSLs by now.

**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 |
```

These are not really expressions, because, for example `< 10` does not directly specify on which value the check has to be performed; that argument is implicit from the context. This is why these range qualifiers can only be used under expressions that have been 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.

**Enhanced Effects Tracking** Originally, there was only one `effect` flag: an expression either has an effect or it does not. However, when extending KernelF with mutable data, it became clear that we must distinguish between `read` and `modify` effects because, for example, a function's 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.

**Mutable State** The initial plan for KernelF was to build a purely functional language and leave all state handling to extensions. While this is still fundamentally the case, it turned out that a general framework for dealing with state (beyond the declaration of effect discussed in the previous paragraph) is useful. In particular, `box`es enable the use of all functional/immutable data structures in a mutable way, and transactions handle the coordinated modification of multiple box-style values. The functionality is implemented as a framework (with interfaces such as `IBoxValue` or `ITransactionalValue`), and even if DSLs define their own abstractions and syntax for dealing with state, the use of those interfaces joins it together in a common semantic framework. This is why the `kernelf.mutable` language extension is now part of KernelF.

## 5 Case Studies

In this section I will present languages we built that extend and/or embed KernelF. Basically, they are all used in real-world customer projects, even though I took some liberty in assigning features to languages to make the discussion here more compact. We will discuss three of them in detail in the next subsections.

**Utilities** A reusable language extension that supports decision tables of various shapes (actually rendered as tables), decision trees (actually rendered as trees), math notation (sum symbols, fraction bars, roots). Examples are in Figure 2. All of these are `Expression`s and can (and are) used in many different languages. The language also supports range specifiers (`> 3` `4..8`) as well as type tags (useful to, for example, track tainted data or required confidentiality levels, as in `fun publish(d: Data<!secret>, receiver: Address)`).

**Solver Language** Many language concepts benefit from various checks with a solver. For example, the decision trees and tables mentioned above can be checked for completeness and overlap-freedom. To simplify the integration of the solver with (domain-specific) language constructs, we have built an intermediate language that abstracts over the solver API. It provides was of defining constrained variables, as well as typical tasks for the solver, such as checking completeness, consistency, equality, progressive refinement or subsetting of expressions. The

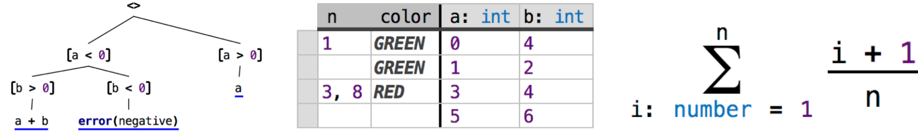


**Fig. 2.** `utils` extension: decision trees, multi-valued decision tables and math symbols.

intermediate language itself makes use of KernelF to represent the expressions, but uses different primitive types.

**Healthcare** Voluntis' mobile apps help patients with therapies and treatments. The apps let users log data and they recommend actions such as taking a medication of a particular dose, behaving in particular ways or calling their medical team. The algorithms in these apps are "programmed" by doctors and healthcare professionals (HCP) using a KernelF-based DSL. The language reuses decision tables and trees and supports component-based behavioral modules, in particular, state machines. A second language supports expressing test and simulation scenarios. We discuss this language in detail in Section 5.3.

**Salary/Tax Calculation** The purpose of this language is the specification of algorithms for salary and tax calculations based on German law. We have build extensions for ER-style data modeling as well as for calculation rules that re-compute the data in a reactive way. The calculation rules and other declarations can be polymorphic regarding their validity periods (the `tax` must be calculated with rule `A` between until 2017, and then using rule `B` from 2018 onwards). Finally, the language support temporal arithmetics, with operators overloaded to work with data whose value changes over time. Details about this language are presented in Section 5.1.

**Smart Contracts** We have developed a set of language extensions for efficiently and reliably defining smart contracts that emphasize multi-party collaborative processes. The language extensions comprise state machines (which are not specific to smart contracts), declarative abstractions for multi-party decisions, agreements and auctions, as well as ways of declaratively preventing several game-theoretical attack scenarios. This language also relies on boxes and transactions to manage a contract's state. Section 5.2 provides details.

**Public Benefits** This system uses form-style syntax with embedded KernelF expressions to let legal experts formalize German public benefits law (unemployment payments, social welfare, old-age care support). In addition to the forms, the system has domain-specific expressions for representing idioms in public benefits payments. Finally, systematically representing the variability in law between Germany's 16 states is another challenge for which this language provides custom-built abstractions and syntax.

**Insurances** Insurance mathematicians use many conventions when writing down there heavily numerical, recursive functions. For example, they distinguish between iterator variables and parameters, where parameters remain constant in (recursive) calls to functions that declare the same parameters (see Figure 3).

```
D : Kommutationswerte
Ergebnistyp:   Laufvariable:   Parameter:
  number{3}      x               i
                                 geschlecht
                                 q
D_x := l_x * 1 / (1 + i)^x
```

```
l : Lebende im Jahr x
Ergebnistyp:   Laufvariable:   Parameter:
  number{0}      x               geschlecht
                                 q
l_0 := startwertLebende
l_x := l_x - 1 * (1 - q.lookup(x, geschlecht))
```

**Fig. 3.** Definition of numerical, iterative insurance math formulas. Notice the calls to `l` and `D` that pass the parameters implicitly. The type of `q` as defined in the data dictionary (not shown) is a lookup table, which is why the `lookup` method is available.

Sameness is established by relating them to a common data dictionary definition, which is why the parameters do not declare types when they are used in functions; those types are in the data dictionary. The language also relies heavily on various forms of lookup tables.

**Cloud-based App Development** Our customer uses a proprietary object-oriented programming language to develop and customize cloud-based applications. The language provides first-class support for their particular style of UIs and persistence layer. KernelF is used as the functional core, the object-oriented abstractions and a module system is built around it. Execution is based on their own, existing cloud-based interpreter infrastructure, so KernelF (and their embedding language) is transformed to their interpreter's byte code format.

**Systems Engineering** Several customers use MPS-based DSLs for systems engineering, focusing on different aspects (such as structural modeling, performance prediction, and security analysis). All reuse a common, hierarchical component modeling language and a feature modeling language, both rendered in their natural graphical notations. KernelF expressions are embedded in various places, to define define type constraints on interfaces, to compute aggregate attribute values, to propagate configuration values and to navigate over component structures.

**Meta Languages** As part of the Convecton[3] project, a new browser-based language workbench, we have developed a set of new meta languages which all rely on KernelF regarding their functional core. The interesting challenges here is the delineation between expressing behaviors functionally and domain-specific declarative abstractions.[4] The former are straight forward to build (and debug), but the latter have advantages in terms of forward execution (for example, to automatically derive quick fixes for errors). The code below illustrates a scope definition that determines the valid targets for a reference. Note how it separates the language feature (`from`) and `path` from the `filter` that selects targets; the former two can be reused for the `create` parts.

```
scope FunCall::function -> pick from Module::declarations
                                path (node, prnt) = node.container<Module>.imported()
                              filter (node, prnt, candidate) = candidate.isPublic()
                              create (node, prnt, futureParent, prefix) = mkLabel(prefix)
```

[3] http://convecton.io

[4] https://languageengineering.io/thoughts-on-declarativeness-fc4cfd4f1832

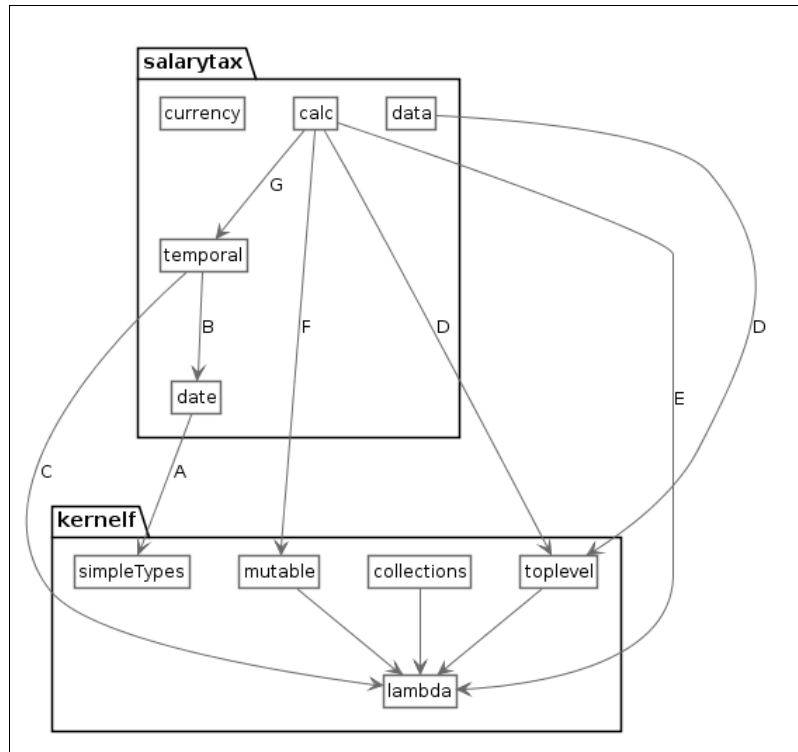


**Fig. 4.** Overview over the languages created in this system and their dependencies. The reasons for the dependencies are as follows: A: mixed arithmetics between dates and numbers; B: temporal slices use `date` values; C: temporal values support higher-order operations that contain lambdas; D: both contain top level declarations; E: treats the `BlockExpression` specially; F: requires `ILValue` for data field assignments; G: special typing rules for dealing with temporal values in calculation rules. Note that all of them depend on `kernelf.base`; the dependency has been elided to declutter the diagram.

```
at (node, prnt) = before(node.ancestor<Declaration>)
```

### 5.1 Salary/Tax Calculation

For this project, we have created several languages, as shown in Figure 4. `date` is a language or representing dates and some of their arithmetics. `currency` contains types, literals and arithmetics for working with EUR currency. `data` provides entities and their relationships. The core of the system is in the `temporal` and `calculation` extensions. `temporal` contains temporal data types, literals and arithmetics, and `calculation` contains to-be-computed data structures as well as the rules to calculate them. In particular, the language supports the evolution of calculation rules over time, a core feature for representing the changing tax law. We discuss each of these languages in the remainder of this section.

**Date Types** The system has to deal a lot with dates: people get married at particular dates, their salary changes at particular points in time and a salary calculation is valid for a particular month. So we need data types for dates, plus arithmetic operations for adding time periods to dates or finding the number of days between two dates. In the `date` extension we introduced a `date` type, as well as date literals, written as `/yyyy mm dd/`. They can be used like any other primitive type in KernelF, and the literals are expressions whose type is `DateType`, the concept behind `date`. The following is then valid:

```
fun printDate(d: date) {...}
val today = /2018 01 23/  // date type inferred
{ printDate(today) }
```

The reason for the unusual notation for date literals is to retain [ WRITEABILITY ] despite a particular drawback of the projectional editor: eager binding. If we were to use `yyyy/mm/dd` then, when you enter the slash behind the year, MPS interprets this as a division binary operator. Since there is no context by which to distinguish these two cases, the user would have to disambiguate manually, which is tedious. We could use the German notation: `dd.mm.yyyy`. Even though `dd.mm` would be initially interpreted as a number with a decimal point, entering the second dot could be used to trigger a further transformation to a date literal. However, using the `/yyyy mm dd/` notation is just the simpler solution, despite its slightly worse [ READABILITY ] and domain alignment.

We have overloaded a few operators to work with date types, in particular `+` and `-`. The former can be used to add days (the base unit of time in this system), and the latter can be used to compute the number of days between two dates, i.e., to subtract two dates. The following is valid:

```
val nextWeek: date         = today + 7
val lastYear: date         = today - 365    // ignoring leap years for now :-)
val howLongIsAYear: number = lastYear - today
```

To make this valid KernelF, no structural changes are required, since the operators already exist. However, the type system and the interpreter have to be adapted. Both of these, however, can be done modularly, in the `date` language. For the type system, we add a new overloaded operations rules, an MPS concept that supports polymorphic typing, typically used with operators:

```
overloaded operations for PlusExpression
  left argument  :==: <date>
  right argument :<=: <int>
  result type    { <date> }

overloaded operations for MinusExpression     overloaded operations for MinusExpression
  left argument  :==: <date>                    left argument  :==: <date>
  right argument :<=: <int>                     right argument :==: <date>
  result type    { <date> }                     result type    { <number> }
```

MPS executes overloaded operations by searching for all of those contributed by the set of languages used in a particular model, and then executing the first one that matches; since core KernelF has no rules that involve date types, the ones defined by the extension language apply.

The interpreter extension works in a similar way: we define a new interpreter that lives in the `date` extension that contains the two evaluators for `PlusExpression` and `MinusExpression`. Both first perform a check of the types of the arguments, and if they don't fit, return `tryAnotherInterpreter`, which triggers the the interpreter framework to continue its search for a matching evaluator. Otherwise we use the JDK's date API for the respective arithmetics.

**Currency Types** Another primitive type we have introduced for this system is `currency`. It is fundamentally a number with two decimals, the literals are written as `NN.DD EUR`. Their implementation is essentially identical to the date types discussed above, so we do not discuss it any further.

**Temporal Types** A more interesting extension concerns temporal types. The notation `TT[U]` represents a temporal version of a base type `U`. Temporality means that a variable `ttu: TT[U]` does not represent a single value; instead, `ttu` is a sequence of `(date, U)`-pairs, expressing when the particular value of `ttu` changed to a particular `u: U`. The following example states that on Jan 1, 2017 the `salary` became 5.000 EUR, and on May 1 it changed to 6.000 EUR.

```
val salary : TT[currency] = TT | /2017 01 01/  =>  5.000 EUR |
                               | /2017 05 01/  =>  6.000 EUR |
```

The reason for adding temporal types is that this customer's system is bitemporal [18], which means that the system manages two dimensions of time for each data item. The first one represents a data item's evolution over time, also known as its *validity time*. The above `salary` is an example, and it is readily obvious why this is useful: almost all quantities in (database-style) systems change as time passes. Representing this as a first class concept in a language makes computation with these values simpler, as we shall see. The second dimension of time is the *transaction time*, i.e., the time at which something became known to the system (and was stored). In a bitemporal system, the database stores both.

```
val salary#/2017 10 07/ = TT | /2017 01 01/  =>  5.000 EUR |
                             | /2017 05 01/  =>  6.000 EUR |
val salary#/2017 11 05/ = TT | /2017 01 01/  =>  5.000 EUR |
                             | /2017 05 01/  =>  5.500 EUR |
```

The example here essentially says that, on Oct 7, 2017, we knew that the salary was as in the previous example; but on Nov 05 we changed the second value to 5.500 EUR; we probably corrected a mistake. The database now contains both states of knowledge, the one from October, and the one from November. A typical use case in the context of our customer's system is to calculate the resulting tax for both perspectives, and then issue compensating transactions. In the example, the person would probably get some money back.

A fully bitemporal system is quite complex, not just in terms of the database and the implementation, but also from the perspective of the user, i.e., the person who uses the DSL to create the salary/tax calculation rules. This is why, in the interest of [ SIMPLICITY ], we only represent the first dimension (validity time) in the DSL programs, and handle the second one as part of a surrounding framework; we will not discuss it any further.

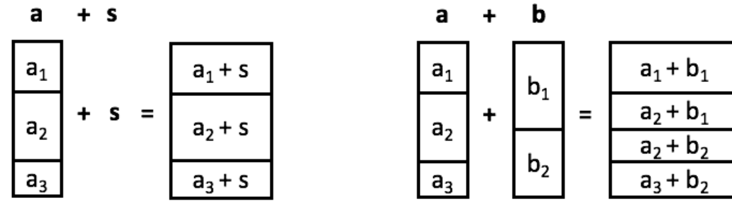


**Fig. 5.** Reslicing of temporal values; `a` and `b` are temporal values, `s` is a regular scalar. When a temporal value is "operatored" with a scalar, the slices remain the same, but their values change. In the case of two temporal values, the slices intersect, and the values are computed per intersection.

The temporal types support overloaded operators. Their most important characteristic is that they "reslice" the temporal periods according to what is shown intuitively in Figure 5; except for the slicing, the semantics of the operators regarding the basic types remain unchanged.

In the implementation, we once again had to overload the typing rules for the operators, this time for `TemporalType`s. In this overload we fell back on the typing rules of the base type. For an operator `op`, `TT[U] op TT[V]` is allowed if `U op V` is allowed by the existing KernelF type system. The interpreter was built similarly to the one for data types, except that the implementation of the arithmetics is more complicated. Lots of test cases helped us get it right.

The overloaded operators let users write arithmetic code that works with temporal data as if it was regular, scalar data. Being able to do this was one major goal of this extension. But to effectively work with temporal data, more support is required, as illustrated below (values beginning with `d` are dates, and values beginning with `tt` are temporal):

- `always(v)` transforms a value `v: U` into a temporal value `ttv: TT[U]` with exactly one slice that is dated to a predefined "beginning of time" date.
- `ttv.add(d, v)` adds a new slice to `ttv` begins at `d` and has the value `v`.
- `ttv.valueAt(d)` returns the scalar value at time `d`.
- `ttv.between(d1, d2)` cuts the slices to within the range `d1 .. d2`. In addition, `ttv.after(d1)` and `ttv.before(d2)` are also supported.
- `ttv.reduce(S, r)` where `r: daterange` (a type that represents time periods) reduces a temporal value back to a scalar. The operation takes into account the slices within the time period `r` (for example, the month for which taxes are calculated) and a reduction strategy `S`. The strategy includes `LAST` (the value of the last slice in `r`), `SUM` (sums up all slice values), and `WEIGHTED_AVERAGE` where the sum is weighted with the relative lengths of each slice value. We will see examples of `reduce` below.

**Basic Data** To model the basic data with which the system works (employee, address, employer, employment), the language supports another extension for data modeling. It supports entities with members which have either primitive, temporal, or other entities as type. In the latter case, cardinalities can be specified,

as well as whether the relationship is containment or reference. Constraints, i.e., Boolean validation rules, are supported for entities as well. The language supports a textual and a (fully editable) graphical notation that can be switched on demand. Since this is "just another entity language" we do not discuss it any further.

**Result Data** Result data are part of the resulting salary or tax calculations. Once computed, they are persisted in the database. The result data structures are similar to basic data entities in that the have a list of members. However, they are different in two important ways: first, they are always keyed by one or more basic data entities. For example, a `SalaryCalculation` result data item is always associated with an `Employment` entity, or the `TaxBill` result is associated with a `Person` entity. Second, result data items are time-indexed (which is different from being temporal). A time index identifies a discrete point in time and is typically `year` or `month`: the `SalaryCalculation` is indexed `monthly`, and `TaxBill` is indexed `yearly`. The association with the basic data entity and the time index uniquely identify each result data record. In the end, it is the purpose of the system to compute all result data item for all valid entity/time combinations.

**Calculation Rules** A calculation rule's purpose is to compute a result data item for a given entity/time pair; so each rule is thus associated with one result data item. The rule also declares which other result data items it uses in its calculation. Consider the following example:

```
result data [monthly] Salary {                 result data [monthly] Tax {
  employment -> Employment // basic data         person -> Person // basic data
  amount     : currency                          amount : currency
}                                              }

calculation for Tax
  depends Salary foreach person.employments     // depends on Salaries of all employments
          as salaries                           // of the Tax bill's person
                                                // in the respective time
  calculate [monthly] {
    val factor = // do some weird tax math
    val total   := salaries.amount.sum        // sum up all salaries in current month
    amount      := total * factor             // populate fields of the result data item
    employment  := ctx.employment             // ctx is available in all calculations
  }
```

Here, the calculation of the `Tax` relies on the calculation of the `Salary`. More specifically, it depends on all `Salary` calculations for the current `Tax`'s person's employments. Because these dependencies are explicit, they can be exploited during the execution. They can be used eagerly, like a function call: when the user requests the `Tax` for a particular person and month, the corresponding calculation rule is triggered, which in turn, when it calls `s.amount`, triggers the calculation of the `Salaries`. While this style of execution is good enough for in-IDE testing with the interpreter, a scalable engine for the data center will work in a reactive style. If a data item is changed, the dependencies are used in the reverse direction, and all dependent, upstream data is recalculated and persisted. This way, data is accessible to the user without the calculation delay incurred by the functional style. This is an example of [ PORTABILITY ] in the sense that different execution engines with different requirements in terms of performance and scalability can

use the same specification. Importantly, the dependencies can also take into account the time index. Consider the next example:

```
calculation for SalaryReport // data structure indexed to an Employment
  depends Salary             as s
          Salary[month.prev] as s_last
  calculate [monthly] {
    currentSalary    := s.amount
    lastMonthsSalary := s_last.amount
    delta            := s.amount - s_last.amount
  }
```

The monthly salary report contains data from the previous month, as a means of providing context for the employee. In the example above, the `SalaryReport` data structure that stores this difference, has a dependency on the current months's `Salary` and on the one from the previous month, expressed with a little sub-language for expressing dependencies that take time into account. Since it is declarative (not full expressions!), it can also be evaluated in reverse order; it works with the reactive execution engine.

The salary and the report calculation rules are also marked as `monthly`. This is automatically derived from the data structure, which is time indexed `monthly` as well. This way it is clear that the execution of the `Salary` calculation rule always happens for a given time period, or `increment` (a month in this example). This leads to various syntactic simplifications. Consider the following:

```
calculation for Salary
  depends ...
  calculate [monthly] {
    val e                = ctx.employment
    val totalHoursWorked = e.workedHours.reduce(SUM)
    val averageWage      = e.wage.reduce(WEIGHTED_AVERAGE)
    val religion         = e.person.religion.reduce(LAST, increment.year)
  }
```

As we have seen above, the `reduce` operator requires the specification of a `daterange`, the time period for which the reduction applies. Because we are in a time-indexed context (`monthly`), this time period is implicit (the particular month) and we do not have to specify it. However, it can be specified if we need a different time period, as shown in the religion example, where we want to get the last slice's value in the current increment's year. Not having to specify the date range explicitly helps with syntactic [ SIMPLICITY ], but also [ ROBUSTNESS ] because of the reduced potential for errors.

Note that in the code above, we use five KernelF extension languages together: the data language (the `employment` reference), the currencies (in the wages), the date extension (as part of the temporal types), the temporal types themselves as well as the main extension for result data and calculation rules. Except for an explicit dependency from temporal types to dates, there is no language-level coordination code (composite grammars, disambiguation logic); the extensions

are independent, but still used together in the same program. Please refer to Figure 4 to recap the dependencies between the various languages.

**Variants and Validity** Calculation rules depend on result data items, not on particular calculation rules for items. This is because there can be many calculation rules for a single result data item. There are two primary reasons for this. First, different calculations might apply for various context, such as different states, for married or unmarried people or for weekly vs. monthly pay. Instead of making all of these distinctions with conditionals in one rule, we can define a set of rules where each rule declares its applicability up front. Conditionals vs. multiple rules allow different tradeoffs regarding modularity, understandability and duplication, and thus help with [ SIMPLICITY ] and [ ROBUSTNESS ]. The second reason is that the algorithms embodied by the rules change over time, usually because of changes in the law that forms the basis for the calculation. Thus, a calculation specifies applicability and validity:

```
calculation for Tax
  depends Salary as s
    valid from /2017 01 01/
  calculate [monthly] {
    ...
  }
```

```
calculation for Tax
  depends Salary as s
          SomeOtherThing as t
    valid from /2017 07 01/
       if ctx.employment.person.homeAddress.state == BW
  calculate [monthly] { .. }
```

In the example above, we define a generally applicable `Tax` calculation that is valid from Jan 1, 2017. From Jul 1, a special rule has to be used if the employee lives in the `BW` state. If some other calculation rule declares a dependency on `Tax`, then, during execution, a dynamic dispatch will be performed that takes `valid` and `if` into account. The reason why this works is that all rules for a given result data item have the same signature (no arguments), so a transparent runtime dispatch is feasible – just as in object-oriented programming. However, the data structure can also change:

```
result data [monthly] Salary {
  employment -> Employment
  amount     : currency
}
```

```
result data [monthly] Salary from /2017 10 01/ {
  employment -> Employment
  amount     : currency
  taxFree    : boolean
}
```

In this example, from Oct 1, we have to populate a Boolean flag that determines if that salary is tax free. In this case the IDE has to be aware of the new version, because the code that the user writes must now populate this field; instead of this version being a runtime dispatch only, it now has to be taken into account by the scoping rules and the IDE.

**IDE Features** To keep track of the validity and applicability, we have implemented several IDE features, illustrated in Figure 6. First, through a drop down box in the toolbar, users can optionally select a date for which they want to see the rules. If a date is selected, the editor evaluates the validity expressions and shows only those calculation rules that are valid at this point. In addition, if the user selects a data item in the editor, a palette shows all the rules that apply to this item. If the user selects a calculation rule, the palette shows the other rules for the same data item, as well as all (directly) downstream dependencies. There

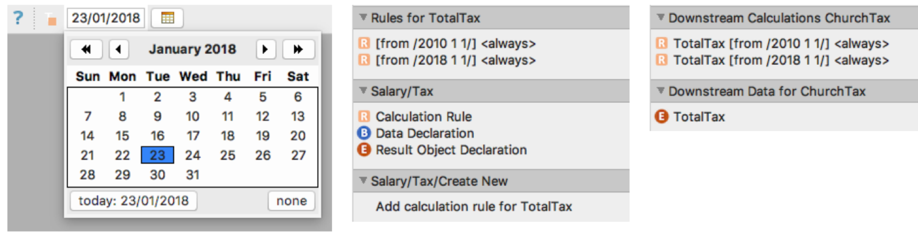


**Fig. 6.** IDE features that support the language: date chooser to adapt the code to show only those parts that are valid at that date; context buttons for the currently selected result data item and for the currently selected rule.

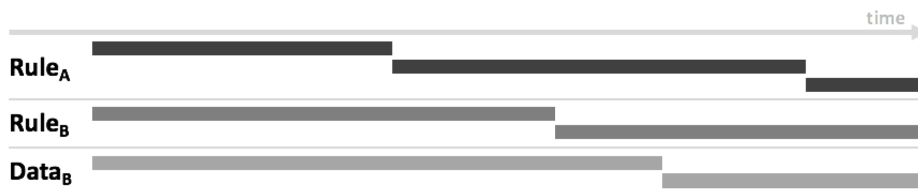


**Fig. 7.** Example of allowed (`Rule-A/Rule-B`) and non-allowed (`Rule-B/Data-B`) misalignment between validity periods.

is a synergistic relationship between the language design and [IDESUPPORT]: if the applicability and validity were implemented as conditionals in the body of a rule, if would be *much* harder to provide this kind of tool support.

The IDE also helps with consistency. Since the validity is only specified using a `from` date, there is no need to check for consistency and completeness. However, we will implement a graphical timeline that shows how the periods of various rules and their dependencies align. However, there is one particular aspect that must be verified for the combination of runtime polymorphism (as used in calculation rules) and static polymorphism (as used for the result data structures). Consider the scenario in Figure 7. Assuming that `A` depends on `B`, it is *not* a problem that the validity periods for the variants for `Rule-A` and `Rule-B` do not fully overlap, because the runtime dispatch is transparent to the programmer. However, the validity period misalignment between `Rule-B` and its result data structure `Data-B` is an error because the same rule would have to work with different data structures, in the IDE. This is not possible.

**Generalised Variability Framework** Based on the experience with this project, we are working on generalised variability support for KernelF. It will be realized as a framework to make it easy to define variants of any top-level declaration. The existing declarations (functions, records, enums, values, and the like), will be hooked into the framework so they can be varied by default. This will be an invasive change, because it will be necessary for those concepts to implement additional interfaces defined by the framework.

The framework will support runtime variability and static variability. For the former, the dispatch will have to be integrated into the interpreter (and other

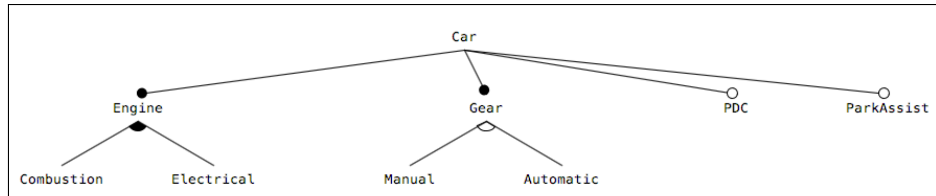


**Fig. 8.** An example feature model that describes a car. Notice the graphical notation the exactly reflects the feature modeling standard.

execution engines). At the same time, the signature must not change. For static variability the signature is allowed to change, and the variability is handled in the scoping logic.

A second important aspect of this framework is that the actual variability clause, i.e., the condition that expresses when the varied declaration is valid, is pluggable; the validity period condition discussed in this case study will be one default implementation, a presence condition expressed over features from a feature model will be another one. Feature models have already been implemented as part of our DSLs for systems engineering and can be plugged in easily; Figure 8 shows an example. The pluggable variability clause will also bring its own consistency checking. For example, for feature model-based presence conditions, a solver-based check for completeness and consistency would be supplied.

### 5.2 Smart Contracts

Blockchains [24] and smart contracts promise trusted, distributed execution of arbitrary programs. Ethereum [34] is currently the most relevant platform as a consequence of its flexible VM, expressive languages, comparatively mature infrastructure and adoption rate. Several languages, all compiling to EVM bytecode, exist, the most widely used one is Solidity. Solidity[5] is essentially a general-purpose programming language that also has some features that are specific to Ethereum's VM and distributed execution model.

However, Solidity does not provide first-class support for the typical patterns found in the distributed, multi-party contrats for which blockchains are supposedly ideally suited. Such abstractions are critical if we consider that a lot of the interest in blockchains and smart contracts comes from non-technical people in domains such as finance [23], logistics [20] or (computational) law.[6] They are very likely also the people who are interested in the specific behaviors encoded in the contracts. So, while ensuring the correctness of the EVM and blockchain infrastructure is crucial [17,2], a concise, understandable and (functionally) verifiable specification of contracts is also crucial. The language introduced here has this goal, but is of course not the only [13] one.[7]

[5] `http://solidity.readthedocs.io/`
[6] `https://www.artificiallawyer.com/2018/01/19/welcome-to-the-first-computational-law-blockchain-festival/`
[7] `https://runtimeverification.com/blog/?p=496`

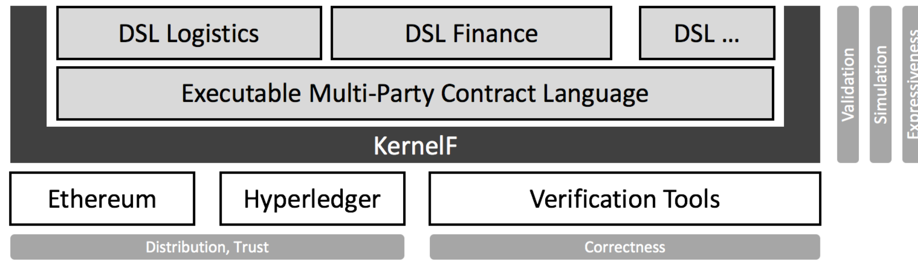


**Fig. 9.** A language architecture consisting of industry-specific DSLs, a set of common abstractions in EMPCL, the KernelF foundation, plus generation to blockchain-based execution infrastructures and verification tools for ensuring functional correctness.

Figure 9 shows stack of languages that potentially achieves this goal. At the top we envision DSLs that are specific to the business domains for which contracts should be specified. We have some exposure to finance, logistics and law, and the requirements are quite different. On the level below we envision a language (dubbed EMPCL) that has the basic abstractions for executable multi-party collaborative processes. This language, in turn, extends and embeds KernelF. For execution, the contract behaviors are generated to suitable blockchain technologies, and for verification, an integration with model checkers and solvers is useful. In this chapter we focus on a prototypical implementation of EMPCL.

**Processes** A contract's evolution over time is inherently stateful. The work on smart contracts drove many of KernelF's extensions for stateful programs, such as boxes and transactions (see Boxes and Transactions in Section 2 as well as Appendix A.14 and Appendix A.16). Before we illustrate those features, let me introduce the notion of a process. A process is a declarative description of a stateful, potentially long-running behavior. The process definition configures the behavior and determines how programs interact with it in terms of commands (that trigger changes in the process) and values (how the environment can observe the process state). Processes are a good baseline for representing the idiomatic behaviors expressed in Smart Contracts such as decisions, agreements or auctions. We performed a preliminary domain analysis for decisions and identified the following variations points: which parties are involved in the decision, and can that list of parties be changed dynamically during the execution of the decision process, what is the decision procedure (unanimous, majority, specific threshold or completely custom), is a minimum turnout required, is there a time limit for making the decision, and can votes by a particular party be revoked. Figure 10 shows the notation used for `MultiPartyDecisions`, and a few example configurations (ignore the code completion menus for now). Once defined this way, processes can be instantiated and used; the code below uses the leftmost process in Figure 10.

```
val s = run(Unanimous)
s.vote(bernd)
s.vote(bernd)
// continued
s.vote(markus)
assert(s.decisionTaken)
```

The process above has one command `vote(party)` and one Boolean value, `decisionTaken`. Which commands and values are available, depends on the configuration of the process. For example, if we were to configure `dynamic` parties, an additional command `addParty(party)` would be available (an example of [ IDESUPPORT ]). This is interesting in two respects. From a language design perspective, the fact that available commands and values depend on the process configuration prevents the user from making certain mistakes; a degree of correctness-by-construction is guaranteed, helping with [ ROBUSTNESS ]. As a point of comparison, this feature could not be provided by an OO framework, because it requires an IDE's awareness of the program's semantics, specifically for the process abstraction.

Second, it is interesting from a language implementation perspective. Normally, a method call is an actual reference (in terms of the MPS AST) to the method declaration. Here, no method `vote` or `decisionTaken` is available to act as reference targets. This is why we have implemented a "reflective" mechanism for commands and values. The process declares and registers them with a descriptor, depending on the process's configuration:

```
final IDCommand VOTE = new IDCommand("vote", new IDArg("who", <PartyType()>));
final IDValue DEC_TAKEN_BOOL = new IDValue("decisionTaken", PTF.createBooleanType());
final IDCommand ADD_PARTY = new IDCommand("addParty", new IDArg("who", <PartyType()>));

public void populateDescriptor(ProcessDescriptor d) {
  d.add(VOTE);
  d.add(DEC_TAKEN_BOOL);
  if (this.dynamic) {  // this queries the dynamic flag in the process definition
    d.add(ADD_PARTY);
  }
}
```

The invocation syntax (`process.value` and `process.command(args)`) is also generic: the node on the right side of the dot is not a reference, as mentioned above, instead it only stores the string that represents the name of the value or command. Code completion proposes only those strings that correspond to the currently active values or commands on the target process, and the type checker also relies on the descriptors to check for valid names and arguments. A language user cannot tell the difference; it behaves exactly like "native" references.

**Meta Functions** This is also a good place to demonstrate how to "escape from declarativeness": what to do if you want to provide a declarative means for configuring something (supporting [ SIMPLICITY ] for the simple cases), but still allow the option of injecting arbitrary code. We will illustrate this with the process' decision procedure: in terms of structure, the process has a child `proc` that is a `DecisionProcedure`, which is an abstract concept. I has three subconcepts: `UnanimousDecProc`, `MajorityDecProc` and `CustomDecProc`. The first two are just keywords, whereas the last one looks as follows:[8]

```
procedure: custom (voted, participated) = voted.size > (2/3) * participated.size
```

[8] The syntax of meta functions is generic, and it can be adapted in terms of its level of detail as shown in Figure 40.

The custom decision procedure embeds a meta function. A meta function has a number of parameters as well as an expression that computes a value from the parameters. Meta functions are a generic utility, they can be configured and executed easily: in terms of structure, `CustomDecProc` only has to implement `IMetaFunctionContext`. In its behavior, it overrides the `createMetaFunction` method to create the meta function structurally; in particular, it specifies the name, return type and arguments:

```
public node<MetaFunction> createMetaFunction()
  createNew(PTF.createBooleanType(), "custom_procedure")
    .addArg("voted", <ImmutableSetType(baseType: PartyType())>)
    .addArg("participated", <ImmutableSetType(baseType: PartyType())>);
}
```

Execution is just as straightforward. The `DecisionProcedure` declares a behavior method `isDecided` that returns true of false, depending on whether the decision has been made or not. The custom procedure implements it as follows:

```
public boolean isDecided(PSet parties, PSet whoVoted,
                         IContext ctx, ComputationTrace trace)
  (boolean)(new MFI(ctx, this.function).run(whoVoted, parties, trace));
}
```

This code instantiates the meta function interpreter (`MFI`), passing the interpreter context and the to-be-executed function (the `function` child is inherited from `IMetaFunctionContext`). Calling `run`, we pass values for the two arguments defined for the function, `voted` and `participated`. The return value is the Boolean flag that indicates whether the decision is successfully taken or not.

**State, Boxes, and Transactions** The primary benefit of boxes is that existing immutable data structures and their APIs can be reused in a mutable way in the sense that the box stores an evolving sequence of immutable values. All immutable data structures can immediately be used this way. In addition, boxes allow a straightforward implementation of transactions:

- The user marks the start of a transaction in the program code; a `Transaction` object is put into the interpreter context
- For any `update` of a box, the new value is stored in a `map<box, value>` that lives inside the transaction object; the box contents are not actually modified.
- Inside a transaction, a box read is redirected to a lookup in the map[9] (which we can find out by looking for a `Transaction` object in the context)
- When we commit the transaction, the actual box contents are updated based on the map inside the `Transaction`
- If the transaction is cancelled (for whatever reason), the map is discarded and the boxes stay unchanged

There are also language constructs that make sense only in a stateful context. The processes introduced above, as well as the state machines we will discuss below, are examples. For them, there is no point in defining an immutable API,

[9] Note that this also works if multiple transactions run in a concurrent context; isolation is maintained because the boxes themselves are not updated.

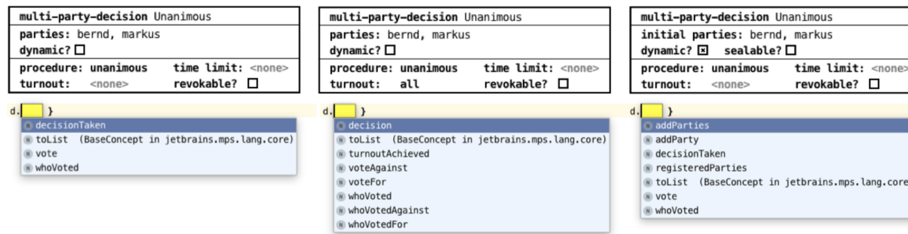


**Fig. 10.** A couple of different configurations of the `multi-party-decision` process and the resulting available entries in the code completion menu.

and consequently there is also no benefit in using boxes to be able to reuse an immutable API in a mutable context. This is why the decision's `vote` or `addParty` commands directly change the state of the process; they are a mutable API. However, internally, objects that are mutable in this way still rely on immutable data. In other words, a change to the state of the process internally sets a new, updated state object.

```
public void handleCommand(IDCommand command) {
  if (command.is(VOTE)) {
    string party = (string) payload.first;
    state = state.voteFor(party); // old state is cloned with a new vote
  }
  ...
}
```

Effectively, this makes a process (and other similar construct) a kind of "implicit box". Explicit and implicit share the runtime API through which they interact with a transaction. This way, they can be used together:

```
val voteCount = box(0)                         fun voteAndCount(Party whoVotes)
val process = run(Unanimous)                     newTx { process.vote(whoVotes)
                                                         voteCount.update(it + 1) }
```

When calling the transactional function `voteAndCount`, and if the `vote(whoVotes)` fails (for example, because the party `whoVotes` is not a valid voter), then both the `process` and the `voteCount` remain unchanged.

**Live Values and the REPL** The values published by the processes provide a peek into its internal state, based on a generic, reflectively-defined API. In addition, using a `LiveValue` wrapper, the processes provide these values in a structured way, suitable for display in the REPL. Because of this homogeneous structure, the REPL highlights the diffs between the current state and the previous one. Figure 11 shows a REPL session.

The generic interaction mechanisms of processes (and their generalised version, `IInteractors`) are a good starting point for building simulators or other end-user oriented UIs (roughly similar to [5]). For example, commands can rendered as buttons, and the values can be rendered as text labels or other widget. Because the internal state is a sequence of mutable values that can be retained in such

```
[0]     run(Unanimous)  : Unanimous, effects[reads]
          org.iets3.core.expr.process.plugin.MultipartyBooleanDecisionValue
[1, x]  live($0)  : live<Unanimous>
          MultipartyBooleanDecisionValue                (snapshot)
          whoVoted ->  (collection|0)
          isSealed ->  false
          registeredParties ->  (collection|2)
                                 @[09583503534]
                                 @[lfd0g98d09g8sdf]
          decisionTaken ->  false
[2]     x.addParty(klaus)  : void, effects[modifies]
          MultipartyBooleanDecisionValue                (snapshot)
          whoVoted ->  (collection|0)
          isSealed ->  false
          registeredParties ->  (collection|3)
                                 @[dsfdslfd0g98d09g8sdf]
                                 @[09583503534]
                                 @[lfd0g98d09g8sdf]
          decisionTaken ->  false
[3]     x.vote(markus)  : void, effects[modifies]
          MultipartyBooleanDecisionValue                (snapshot)
          whoVoted ->  (collection|1)
                        @[09583503534]
          isSealed ->  true
          registeredParties ->  (collection|3)
                                 @[dsfdslfd0g98d09g8sdf]
                                 @[09583503534]
                                 @[lfd0g98d09g8sdf]
          decisionTaken ->  false
[4]     x.vote(klaus)  : void, effects[modifies]
          MultipartyBooleanDecisionValue                (snapshot)
          whoVoted ->  (collection|2)
                        @[dsfdslfd0g98d09g8sdf]
                        @[09583503534]
          isSealed ->  true
          registeredParties ->  (collection|3)
                                 @[dsfdslfd0g98d09g8sdf]
                                 @[09583503534]
                                 @[lfd0g98d09g8sdf]
          decisionTaken ->  true
```

**Fig. 11.** A REPL session where a processes is wrapped in a `LiveValue` to support structured rendering of the internal state and diffs that highlight its evolution.

```
multi-party-decision Sale
initial parties: bernd, klaus
dynamic? [x]   sealable? [ ]
procedure: unanimous     time limit: <none>
turnout:   <none>        revokable? [x]
```

```
multi-party-decision AccessControl
initial parties: bernd, klaus
dynamic? [x]   sealable? [ ]
procedure: majority      time limit: 20000
turnout:   <none>        revokable? [ ]
```

**Fig. 12.** Two decision processes used in the complex example contract.

a simulator, it is easy to build "time travel" functionality [4] or even branching, where users can interactively explore back and forth the behaviors of contracts.

**More Complex Contracts** We decided on declarative abstractions for the core decision, agreement, auction and sales processes because those are ubiquitous in smart contracts. In some sense they can be seen as the building blocks of contracts. In addition, it is feasible to capture the vast majority of real-world variants into a set of configuration parameters. However, the overall contracts that make use of these building blocks show more variability, which is why it is

more useful to use a less specific language for those: state machines are obvious candidates. Consider the following requirement for a non-trivial smart contract:[10]

*An online community has to continuously maintain a (selling) decision; it can be revoked or granted as time passes. A group of individuals, called the deciders, vote for and against this decision. The vote has to be unanimous. In addition, additional people can be allowed into the group of deciders. The existing deciders vote for new candidates, by simple majority, but with a time limit. Once allowed into the group of deciders, the new member can participate in the sell/no-sell decision. Multiple member approval processes can go on at the same time. While a member request is pending, the sales decision cannot be changed.*

The implementation of this contract relies on two declarative decision processes, `Sale` (to maintain the sales decision) and `AccessControl` (one is instantiated for each allow-in of a potential new decider). Both configurations are shown in Figure 12. The remaining state machine-based implementation of this contract is as follows (we omit the `state machine` declaration itself). First, we define the events which we want to use to control the contract:

```
event openAccess  // go to the mode where we allow new guys to request to join
event requestAccess(newGuy: party) // a new guy wants to join the deciders
event terminateAccessRequest(who: party, newGuy: party) // kill a decision procedure
event voteForAccess(voter: party, newGuy: party) // vote for a new guy to become decider
event letsSell    // go to the state where we maintain the sell/no-sell decision
event voteForSelling(who: party)     // vote for the sale decision
event voteForStopSelling(who: party) // vote against the sale decision
```

Next, we instantiate one `Sale` process in the state machine, and define a map from `party` to `AccessControl` where we store all pending access requests. We also define a query (essentially a parameterless Boolean function) that reports whether the selling decision is currently true or false. The `observable` flag means that the query can be invoked from outside the state machine:

```
var sale                          = run(Sale)
var pendingAccess                 = box(map<party, AccessControl>())
observable query currentlySelling = sale.decisionTaken
```

The similarity between processes and state machines is not coincidental: in fact, the state machine also implements `IInteractor`, the events act as commands and the observable queries or variables correspond to values. Thus, state machines can be used in the same interactive way (for example in the REPL) as the processes in the previous paragraph.

Next we define a few helper functions used inside the state machine; the `/R` or `/RM` flags indicate the kind of effect they have (`read` only, or `read-modify`):

```
fun isDecider/R(who: party) = sale.registeredParties.contains(who)
fun isPending/R(who: party) = pendingAccess.val.keys.contains(who)
fun hasPending/R()          = pendingAccess.val.size != 0
```

The core logic is implemented in the next few states. The first one represents the phase where the contract is gathering new members. The following code handles the `requestAccess` event, where a new party can request access to the group:

[10] Another extensively documented example can be found in [25].

```
on requestAccess(newGuy) [!isDecider/R(newGuy)] : {
  val acc = run(AccessControl)
  pendingAccess.update(it.put(newGuy->acc))
  acc.addParties(sale.registeredParties)
}
```

The transition only fires if the `newGuy` is not yet among the existing deciders (see guard condition); then we create a new `AccessControl` process and store it in the map that keeps track of the currently pending membership requests. Before that new `AccessControl` process can work, we have to populate it with the existing deciders, because it is them who make the decision about the membership of the `newGuy`. Note that this transition has no target state, so it remains in the current one; its only purpose is to perform the action associated with the transition.

The second transition terminates an existing access request if one of the deciders chooses to do so. The event has two arguments, the party who request termination and the party whose membership request should be terminated. The guard condition checks that these two parties actually play the respective roles. If everything is in order, we just delete the corresponding `AccessControl` process from the map of pending accesses.

```
on terminateAccessRequest(who, newGuy) [isDecider/R(who) && isPending/R(newGuy)]
   : pendingAccess.update(it.remove(newGuy))
```

Next we deal with a current member (`voter`) voting for a new guy. Again, we use the guard condition to establish the roles. We then get the `newGuy`'s `AccessControl` from the pending list and submit our vote. If after the voting the decision has been taken, we add the `newGuy` to the parties of our `Sale` process and remove their `AccessControl` from the list of pendings.

```
on voteForAccess(voter, newGuy) [isPending/R(newGuy) && isDecider/R(voter)] : {
  val acc = pendingAccess.val[newGuy]
  acc.vote(voter)
  if acc.decisionTaken then {
    sale.addParty(newGuy)
    pendingAccess.update(it.remove(newGuy))
  } else none
}
```

The last thing we do in the `requestAccess` state is to handle the request to move to the `selling` state, which is only possible if there are no pending requests (which is why current deciders can terminate pending requests by force):

```
on letsSell [!hasPending/R()] -> selling
```

The `selling` state is really simple. It handles voting for and against the sales decision maintained by the contract, as well as the `openAccess` event which gets us back into the state where we accept new members. Note how the actual logic of making the sales decision, independent of its own complexity, is handled completely by the `Sale` process.

```
state selling {
  on openAccess -> gatheringMembers
  on voteForSelling(who) [isDecider/R(who)] : sale.vote(who)
  on voteForStopSelling(who) [isDecider/R(who)] : sale.revoke(who)
}
```

**Game Theory, Interceptors and Context Arguments** Game theory [14] looks at how rules in cooperative processes ("games") impact the outcome, and also how the parties taking part in the game can cheat, i.e., exploit the rules for their own benefit. Smart contracts are cooperative processes, which is why they are susceptible to "game-theoretical" exploits.

For example, a sybil attack [9] is one where a reputation-based system is subverted by one (real-world) party creating loads of fake (logical) identities who then behave in accordance with the real world party's goals. For example, consider a decision that is based on majority vote. An attacker could create lots of additional parties and thereby taking over the majority, leading to a decision in the interest of the attacker. While there are many potential ways how such attacks can be thwarted, one approach is to limit the rate at which new parties can request to join the process. Instead of requiring users to implement this manually, the state machine language supports a declarative way: the rate at which events come into a state machine can be limited (helping with [ ROBUSTNESS ] without compromising [ WRITEABILITY ]). The following code expresses that while the machine is in state `requesting`, only three commands per second are allowed. If more requests come in, they are rejected.

```
state requesting [rate(3/1000|commands-only)] {
  ...
}
```

The code between the brackets registers an interceptor (the term is inspired by CORBA [21]). Interceptors see every incoming event before transitions have an opportunity to react to them. They can then let it pass through, change parameters in the event, or discard it. Interceptors can maintain their own internal state. They can be seen as a guard condition that applies for a whole state (or substates), and not just a particular transition. The rate interceptor discards events if the rate exceeds the one specified.

Looking at the example, you can see that many events take the sender as an argument, usually in order to check that the event is authorised (the sender is among the current deciders). This is typical for smart contracts, and in fact, every message sent into an Ethereum contract carries an implicit sender address. Implicit arguments, called context arguments, are also available for interactors. Together with an interceptor, this can be used for authorization:

```
state playing [senderIs(players)] {
  on offerBid(money) : bids := bids.put(sender->money)
  ...
}
```

The `senderIs()` interceptor checks whether the context argument `sender` is supplied by the client (and rejects the event if not), and verifies that the sender is in the collection passed as an argument to `senderIs` (and rejects the event if not). In addition, because any transition in the state will only be executed if a sender is given, the interceptor makes the `sender` variable available inside the state. It can be used just like an explicitly given argument. In the example above we use it to create an entry in the `bids` map that is keyed by the `sender`.

The last interceptor worth mentioning in the context of smart contracts and game theoretical exploits is the `takeTurns` interceptor. Many "games" require a fair allocation of opportunities to participating parties. One way of achieving this is to run a game turn-by-turn, where each party can make one "move" in every "round". Consider the bidding process example:

```
state playing [senderIs(players)] {
  state bidding [takeTurns(players|ordered|after 1000 remove)] {
    on offerBid(money) : bids := bids.put(sender->money)
    if [timeInState > 2000] -> finished
  }
  ...
}
```

The `takeTurns` interceptor can be configured regarding the strictness of the turn-by-turn policy. `Unordered` means that in each round, every party has to make a move, but the order within each round is not relevant. `ordered` means that the order given by the list of parties passed to the interceptor is strictly enforced. A violation leads to a rejection of the command. The interceptor also provides access to the list of allowed next movers; this could potentially be used to notify parties that it is their turn.

There is a risk of a denial-of-service attack in the case of `ordered` turn taking: if the next party `p` does not make its move, the whole process is stuck. Nobody else can make a move because it is `p`'s turn. This is why a turn-by-turn game should always include a timeout, 1000 in the example above. If the next party does not make their move within 1000 time units, that party is permanently removed from the list of participants; alternatively, it can also be `skip`ped.

### 5.3 Healthcare

Like all the other case studies, the set of languages built for this system builds on top of KernelF and extends it with new expressions (see Figure 13). High-level domain-specific behaviors are expressed as state machines, as explained below. However, this system is interesting because it removed about two thirds of KernelF (by constraining it out of program written in the context of this system). For example, attempt types, option types, some of the advanced operators as well as some of the collection operations are not accessible to the users of this system.

**Reactive Algorithm** The main algorithm controls notifications and reminders submitted to the mobile operating system and reacts to a user's data submissions. It also makes high-level decision as to the execution of the algorithm and manages data collected from the user (in what one could call databases).

The top level abstraction is the component, a unit of behavior. Components can be instantiated and then started by other components, hierarchically. When a parent component starts a child component, it supplies values to the parameters defined by the child components (just like an operating system that starts a process). The child then runs concurrently with the parent; it communicates with the parent by sending output data events. The parent component can react to those events. By waiting for particular events (see below), the parent can synchronise with (wait for completion of) a child it started.

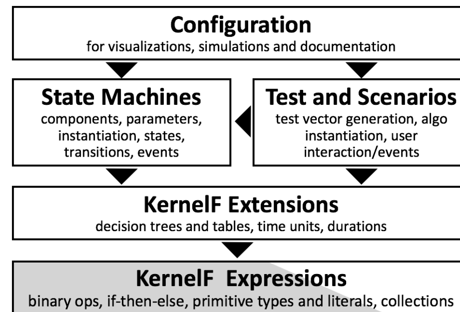


**Fig. 13.** The core of this system is a restricted version of KernelF. On top, we have developed a set of functional extensions that help medical professionals make non-trivial (multi-criteria) decisions. The core of the medical algorithm is expressed through state machines, and validation is performed through a testing and scenario description language. At the top is a language for configuring generated visualisations and reports.

In addition, a component also provides other means of interaction with its environment, and in particular with the user, through the UI. A component can bring up a UI, for example, a questionnaire where the user can then select one of several options. A component can also register reminders: essentially, this is an entry in the phone's calendar. The framework that runs the applications on the phone keeps track of the created notifications, and retracts them if the user reacted, or if a timeout occurs.

The implementation of the behavior inside a component can potentially be done in many ways to be able to handle future styles of applications. For now, only a state-based implementation is supported: the content of a component is a hierarchical state machine. The abstractions are the usual ones: nested states, events, transitions, guards, actions .

Consider the following example: the application wants the user to measure their blood sugar at `08:00` the next morning. To this end, the application registers a reminder for `07:55`, `08:00` and `08:10`. Once a new blood sugar value is entered by the user at roughly 8am, the remaining reminders can be retracted. In contrast, if no value is entered by `08:10`, the process might have to react to that: for example, a message might have to be shown to the user reminding them of the importance of a timely blood sugar measurement, or, if things become more serious, their medical team might have to be notified by the app. To realize this behavior, a timeout event in the state machine is necessary.

The code to implement this behavior looks roughly like the following. We start by defining a helper function that computes the next time at which a blood sugar measurement should take place. The time literals, and the associated types use an addition data type `datetime` whose implementation is similar to the one defined in the salary/tax case study in Section 5.1[11]. Second, we define a `timeseries`

[11] We are currently consolidating both into a common datetime extension.

for the blood sugar measurements. Time series are essentially records with an index of type `datetime`, and are also defined specifically for this system:

```
fun nextTime() {
  alt | now in [08:01 .. 11:45]  =>  12:00 |
      | now in [11:46 .. 17:45]  =>  18:00 |
      | otherwise                =>  08:00 |
}

timeseries BloodSugarSeries {
  value: number[50|400]
}
```

The meat of the blood sugar measurement functionality is in a component `AcquireBloodSugar`. It has two configuration parameters; the time at which the next measurement should take place, as well as the time series in which to store the measurement.[12]

```
component AcquireBloodSugar

parameters t : datetime
           db: BloodSugarSeries
```

Next, we define the interface of the component; it handles events of type `BloodSugarMeasurement` from the UI layer. In addition, it emits the `ok` and `missed` events, both without arguments.

```
inputs  BloodSugarMeasurement(bs: BloodSugar)

outputs ok
        missed
```

Next we declare a reminder. A reminder is essentially a group of OS-level reminders which, as we will see below, are managed as a group from the perspective of the algorithm. The reminders are defined relative to the time `t` passed to the component as a parameter.

```
val r = reminders at t - 15 : "Please enter blood sugar in 15 minutes"
                  at t - 5  : "Please enter blood sugar in 5 minutes"
                  at t      : "Please enter blood sugar now"
                  at t + 10 : "URGENT: Please enter your blood sugar"
```

Finally, we define the actual behavior of the component. Note that, because on mobile phones the app is passive when not in focus, and because it cannot actively push content to the user (except through reminders), the app is reactive. This is why a state machine is a very good fit. The `start` block is executed after the component is started (essentially a constructor). We create the reminders and unconditionally transition to the `waiting` state. In that state, as the name suggests, we passively wait for input events, i.e., the `BloodSugarMeasurement`. If one occurs, and the current time is within 20 minutes of the scheduled time `t`, then we store the measurement in the time series, and `terminate` with the `ok` event (`terminate(<evt>)` is a shorthand for sending an event (`send(<evt>)`) and then just terminating the execution of the component). If we do not receive

[12] The actual syntax relied a little bit more on boxes and other semi-graphical elements; we use text here so we do not have to resort to images.

the event within `t + 20`, we terminate with a `missed` event. In any case, once the `waiting` state is left, the OS-level reminders associated with `r` are all cancelled.

```
start: createReminders(r)
       -> waiting

state waiting:
  exit: cancelReminders(r)
  on BloodSugarMeasurement
    when now in [t - 20 .. t + 20]
      store now, bs in db
      terminate(ok)
    when now < t - 20
      message "too early, please submit around {t}"
  if now > t + 20
    terminate(missed)
```

Here is the (simplified) main state machine for the diabetes application that uses the `AcquireBloodSugar` component above. It creates and starts the `Acquire-`ıBloodSugar in its `running` state. It then keeps track of the missed measurements and, if too many are missed, notifies the medical team.

```
component DiabetesApp

val db: = createDatabase<BloodSugarData>

val missed: counter = 0

state running:
  val abs = AcquireBloodSugar.start(nextTime(), bloodSugarDB)
  on abs.ok
    abs.start(nextTime(), bloodSugarDB)
  on abs.missed
    missed.increment(1)
    abs.start(nextTime(), bloodSugarDB)
  if missed > 5
    -> error

state error:
  notifyMedicalTeam("missed blood sugar too often")
```

**Decision Support** As part of the overall reactive algorithm, many complex decisions have to be made. To represent those as intuitively as possible, we have implemented a decision support language. All abstractions in that language, at a high-level, can be seen as functions: based on a list of arguments, the function returns one or more values. Plain functions are available for arithmetic calculations. However, it is typical of medical decisions that they depend on the interactions between several criteria. To improve the [ READABILITY ] of a function call for non-programmers, we support a style of signature that reads like a sentence fragment. For example, the function in Figure 14 can be annotated with a syntax template that allows the following function call:

```
val riskScore = blood pressure risk for systolic <expr-1> and diastolic <expr-2>
```

The code completion and type checks for `expr-1` and `expr-2` work as usual, but this notation provides more context for the two values a plain function call `BpScopeDecisionTable(<expr-1>, <expr-2>)`.

To improve [ READABILITY ] of the actual decision algorithm (and thus make it easier to validate), they are often represented as decision trees (Figure 14) or

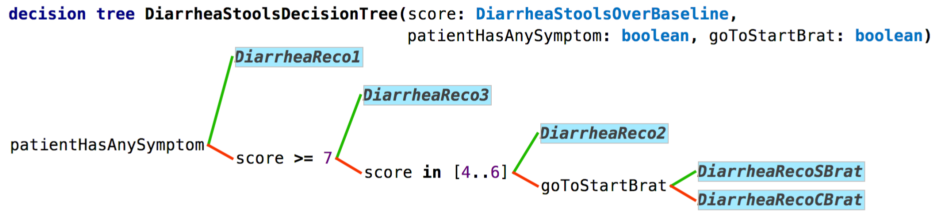


**Fig. 14.** A decision tree; the green/up edges represent `yes` answers to the preceding node, the red/down edges represent `no`.

```
decision table BpScoreDecisionTable(sys: bpRange, dia: bpRange) =
```

| | | dia | | | | | |
|---|---|---|---|---|---|---|---|
| | | <= 50 | [51..90] | [91..95] | [96..100] | [101..109] | >= 110 |
| sys | <= 90 | 1 | 1 | 3 | 4 | 5 | 6 |
| | [91..140] | 2 | 2 | 3 | 4 | 5 | 6 |
| | [141..150] | 3 | 3 | 3 | 4 | 5 | 6 |
| | [151..160] | 4 | 4 | 4 | 4 | 5 | 6 |
| | [161..179] | 5 | 5 | 5 | 5 | 5 | 6 |
| | >= 180 | 6 | 6 | 6 | 6 | 6 | 6 |

**Fig. 15.** A decision table that specifically works on ranges of values. Note the compact syntax for range representation.

decision tables. As mentioned in Section 2, basic tables and trees are available in KernelF's utility language. However, special forms are needed (and have been built specifically for this project). An example is a table that splits two values into ranges and returns a result based on these ranges. Figure 15 shows a table that returns a score; scores represent standardised severities or risks that are then used in the algorithm. KernelF's number types with ranges, and their associated static checking, is also an important ingredient to being able to improving the [ ROBUSTNESS ] of the algorithms.

**Testing** Testing is an important contributor to the success of this project, and we put significant effort into defining a suitable set of languages. For testing functions and function-like abstractions, regular JUnit-style function tests are supported; Figure 17 shows an example. The first of the tests in Figure 17 tests a function with one argument, the second one passes an argument list, and the last one shows how complex data structures, in this case, a patient's replies to a questionnaire, are passed to the test. The table notations for testing based on equivalence partitions in shown in Figure 16.

Scenario tests (Figure 18) are more involved because they take into account the execution of the reactive main algorithm over time. They are expressed in the well-known given-when-then style,[13] which is, for example, also supported by Cucumber.[14] To express the passage of time and occurrences at specific times,

[13] `https://martinfowler.com/bliki/GivenWhenThen.html`
[14] `https://cucumber.io/`

```
equivalence partition for BPStateMachine

inputs
  EventNewBpMeasure(sm: BPStateMachine)
```

| Input | Type | Partitions |
|---|---|---|
| inputBaselineDBP | bpRange | <= 90, ]90..100], >= 99 |
| inputBaselineSBP | sbpBaseline | <= 150, > 150 ] |
| systolic | bpRange | <= 90, > 90 |
| diastolic | bpRange | <= 90, > 90 |

**Fig. 16.** Equivalence partitions help test complex structures with relevant combinations of values.

```
PASS
function test gradeStools
  given 7 expected 3
  given 6 expected 2
  given 5 expected 2
  given 4 expected 2

PASS
function test DiarrheaStoolsDecisionTree
  given false, 1, true, false   expected   DiarrheaUSRecoLevel1Symptom
  given false, 9, false, false  expected   DiarrheaUSRecoGrade3

PASS
function test checkScreeningQuestion
  given   answers to DiarrheaScreeningQuestionnaire {   expected   true
            dietarySupplements: false
            medication        : true
            hospitalized      : false
          }
```

**Fig. 17.** Function tests call a function (or something function-like, such as a decision tree or table) with the arguments specified after `given`, and then check that the `expected` valued is returned. The `answers` construct represents a user's reply to a questionnaire; it can be seen as an instance of a record.

the `at` notation is used. The execution of the tests is based on a simulation. The number of steps and the time resolution is derived from the scenario specification.

**Simulation** The purpose of the simulator is to let healthcare professionals "play" with an algorithm. To this end, the in-IDE interpreter executes algorithms and renders a UI that resembles the one on the phone (Figure 19, right). A set of DSLs is available to structure the UI; lower-level styling support is available through Javascript and CSS. A control panel lets users configure a particular simulation and also fast-forward in time (Figure 19, left). There is also a debugger that, while relying on the same interpreter, provides a lower-level view on the execution of algorithms. It is not used by HCPs.

**Documentation Generation** An important output is the medical protocol, a visualisation of the complete algorithm for review by HCPs, associated medical personnel not trained in the use of the PLUTO DSLs, as well as external reviewers.

```
scenario scenario_8
  global timeout:    1 hours
  time granularity: 60 seconds
given
  inputPainBaseline         = 1
  inputPainMedicineDuration = Six
when
  at 0 min: EventInPainMeasure answers to PainMeasureQuestionnaire {
                                 measure: 4
                               }
            EventInPainSymptoms1 answers to PainSymptoms1Questionnaire {
                                   interferingDailyActivities: false
                                   newSite                   : false
                                   interferingAbilityToWalk  : false
                                 }
then
  at 0 hours: assert parent sent message Recommendation(PainRecoSymptom1, Six)
  at 29 min:  assert parent in state     PainMeasure.Ask
  at 30 min:  assert parent in state     PainInitial.Ask
```

**Fig. 18.** Scenarios follow the established given-when-then style: *given* preconditions, *when* something happens, *then* a set of assertions must hold. Scenarios express the passage of time, as well as points in time when something happens or is asserted.

The outputs are too large to show in the paper; they are essentially graphviz-style flow charts with a couple of special notational elements. It is often necessary to highlight specific aspects on the overall algorithm, so the generation of the flow chart can be configured using a DSL (Figure 20). It supports:

- The level of detail (`Deep` in the example)
- The tags that should be included and excluded. Model elements can be tagged, for example, whether they are part of the default flow or whether they are relevant for complications in the treatment. A visualisation might highlight specific tags.
- Color mappings for tags (e.g., render the case for complications in red)
- Human-readable labels for states or messages in order to make them more understandable for outsides.

The reason why these configurations are represented as models (expressed in their own DSL) as opposed to just configuring a particular visualisation through a dialog is that many such configurations exist, and they must be reproduced in bulk, automatically, as the algorithm evolves.

**Execution** We provide two separate execution infrastructures, which is important for quality assurance, as discussed below. The first one is an in-IDE interpreter. It reuses the existing KernelF interpreter. For the functional abstractions developed in this project, we have built additional interpreters using the same interpreter infrastructure also used in KernelF. For the reactive, state-machine based part of the system, an interpreter was built using plain Java code that works on the MPS AST. It drives the overall execution and invokes the functional interpreter. A similar approach has been taken for the scenario testing DSL. The in-IDE interpreter provide short turnaround times for the users of the DSL and are an example [ IDESUPPORT ].

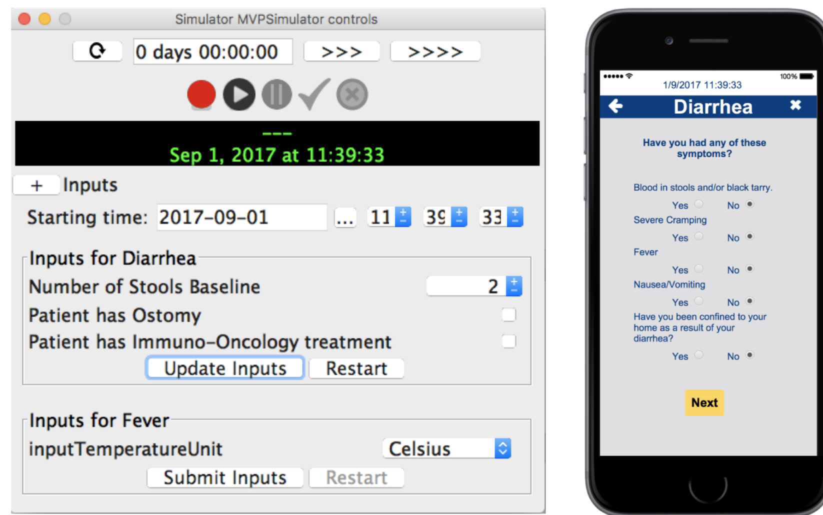


**Fig. 19.** Control panel to configure and execute simulations.

```
medical protocol for: ComposedStateStoolsOngoingIn24hr
  depth:     Deep
  included: everything
  excluded: #background #log
  colors
  labels:    StoolsOngoingIn24hr TITLE -> 'Diarrhea within 24 h'
             AppRecommendation -> 'Display recommendation "${message}"'
```

**Fig. 20.** Configuration for the generation of medical protocol flow charts.

The execution on the mobile phone is based on a second interpreter. It is implemented in C++ so it can be used on iOS and Android platforms. A platform adapter provides unified access to the necessary operating system services, such as the system clock, reminders and notifications, as well as networking APIs. The C++ interpreter works on an XML representation of the AST, essentially a generic serialisation format for the MPS AST structure. Directly using the AST is infeasible, because MPS is written in Java, and the runtime needed to be C++ for performance and portability. The reason why an interpreter was used in the first place (as opposed to generating C++ code from the algorithm) was because of the required update times: if a problem is found with the algorithm, an update has to be delivered as soon as possible. Waiting for the the clearance of Apple's review team was not an option.

**Quality Assurance** Ensuring the correctness of the algorithm models (validation) as well as their correct execution on the mobile phones (verification) was a major aspect of this project. Both because the well-being of human beings is directly at stake, and because the approach has to get FDA approval; otherwise

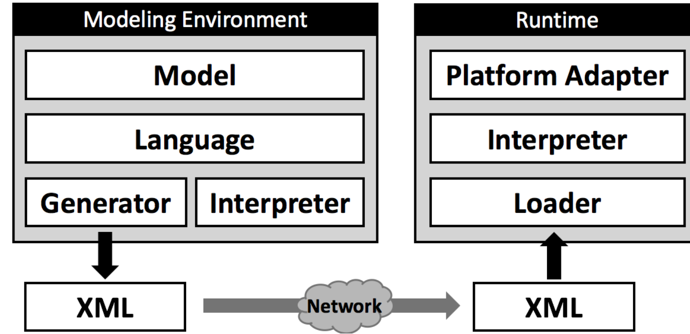


**Fig. 21.** Execution architecture of the languages: an IDE-interpreter plus an interpreter on the phone implemented that works on an XML representation of the algorithms.

the applications cannot be legally sold, jeopardising the business case. While a detailed discussion of our verification and validation approach is beyond the scope of this paper, here are the steps we took, based on a systematic risk analysis:

- Improved review-ability of the models because of the domain-oriented abstractions and notations
- Further validation of the model by healthcare experts using the simulator
- Extensive set of unit and scenario test cases that reach very high coverage of the algorithms
- Test generation to improve coverage
- Mutation testing [19] (aka fuzzing) to ensure sensitivity of tests
- Coverage measurement also of the language structure, the Java interpreter, and the C++ interpreter implementation, and 100% coverage for those.
- Redundant [15] execution of all tests in the two interpreters to find random errors in each
- The two interpreters were implemented by different (teams of) developers to avoid systematic errors
- Architectural safety mechanisms such as runtime watchdogs [15] based on independently specified invariants.

~ ~ ~

This concludes our case studies. Figure 22 summarizes the extensions, aligning them with the three layers introduced at the end of Section 1.2. Both, the salary/tax and healthcare case studies contribute to all layers, as suggested by Section 1.2. The smart contracts case study is a little bit different: because it is an "experimental" set of languages, there are no domain-specific data structures or types; we used the built-in ones. Based on our experience in the logistics domain, a fully fledged contract language would need schemas, mappings to actual documents, types for money and time, as well as physical units.

As a concluding remark of this chapter, the case studies should have given the reader a good illustration of the philosophy of MPS-based language design

| | Salary/Tax | Smart Contracts | Healthcare |
|---|---|---|---|
| IDE | Alternative Rules<br>Rules for Data Items<br>Projection for a given Date | Live Values<br>Diff of Live Values | Simulator with Phone UI<br>Debugger for State Machine |
| | Language-Specific Extensions to Functional Debugging/Tracing and Testing | | |
| Structure | Basic Data<br>Result Data. | | Time Series<br>Components<br>Test Scenarios |
| Behaviors | Calculation Rules & Dependencies<br>Variants<br>Validity | Interactors & Processes<br>State Machines & Interceptors<br>Context Arguments<br>Live Values | (Special) State Machines<br>Test Generation & Mutation<br>Coverage Measurement Utils |
| Functional & Types | Date Types<br>Currency Types<br>Temporal Types | | Time & Duration Types<br>Decision Trees and Tables<br>Sentence-like Function Calls |
| KernelF | | | |

**Fig. 22.** Overview of the extensions to functional abstractions, higher-level behavior, structures and IDE extensions for the three case studies.

introduced in Section E. It is really more like "libraries with syntax and type system", with lots of first-class concepts aligned closely with the application domain.

# 6 Challenges and Open Issues

In terms of language engineering, the development of KernelF is relatively similar to the development of mbeddr, which we have evaluated extensively in [30]. This is why this paper focuses on the *language* design in the development of KernelF. However, a couple of issues are worth pointing out specifically in the context of KernelF, even though they have been mentioned generally in [30].

## 6.1 Type System

The type system was the biggest challenge in the current implementation. I will point out two problems that both relate to subtyping.

**Number Types** The first one relates to number types. Normally, MPS determines subtype relationships via *subtyping rules*. For a given type, a subtyping rule returns the list of direct supertypes. MPS uses those to build a type hierarchy, and also uses it during type checking in situations like `val v: T = <expr>` with `expr: U`, where `U` must be a subtype of `T`. Now consider the situation where `T` is `number[0|100]` and `U` is `number[5|10]`. Cleary, the range `5..10` is a subrange of `0..100`, so the subtyping holds. But it is impossible to enumerate all supertypes of a number type, because there are infinitely many. MPS has *replacement rules* for this case. They are called as a last resort: if a type check fails, the engine tries the suitable replacement rules and sees if, by performing the specified type

replacements, the type check can be made to succeed. For number types, we have defined the following replacement rule (slightly simplified):

```
replacement rule for supertype :==: NumberType as super
                 and subtype   :==: NumberType as sub {
  applicable if { sub.range.isSubrangeOf(super.range); }
  replace {}
}
```

The rule applies if two `NumberType`s are tested for a subtype relationship. It then checks if the ranges of the two types are in the required relationship. If so, the rule executes, which means the original type equation is replaced with the one given in the `replace` part. Since this is empty here, the original typing rule is effectively discarded. Since there's nothing to fail, no error is shown.

We use replacement rules for a few other reasons as well, for example, in the context of `type` definitions. Here is the catch: replacements are only executed once during the solver's attempt at solving the type system equations. So if the replacement rules create a new set of equations *which can only be solved by applying more (different) replacement rules*, this does not work. As of now, we have not found a way to solve this problem. Sprinkling explicit `cast`s over the affected programs helps, but of course this is unintuitive for the end user.

**Options and Attempts** The second problem relates to the computation of supertypes in the presence of option and attempt types. Consider the following program. What is the type of `alt`?

```
fun f( ... ) = alt | <cond-1> => 42           |
                   | <cond-2> => 33.33        |
                   | <cond-3> => error(FAIL)  |
                   | <cond-3> => error(FATAL) |
```

A common supertype is typically calculated in the following way (see also Appendix D.5):

```
typing rule for AltExpression {
  var T;
  foreach alternative in node.alternatives {
    T :>=: typeof(alternative.then);
  }
  typeof(node) :==: T;
}
```

For each of the alternatives, this code submits a type equation to the solver which states that `T`, the to-be-calculated type of `alt`, is the-same-or-supertype of the type of the `then` part of the particular alternative. `T` ends up as the least common supertype of all the types of the `then`s. However, here the situtation is different, the correct type is `attempt<real|FAIL, FATAL>`, i.e., the least common supertype of all non-`error` values, wrapped in an `attempt` type that lists all the possible errors. A similar issue arises if you mix values with `none`, because this introduces an `option`. Now consider the following:

```
fun f( ... ) = alt | <cond-1> => 42           |
                   | <cond-2> => 33           |
                   | <cond-3> => error(FAIL)  |
                   | <cond-3> => none         |
```

There are two potentially correct types: `attempt<opt<number[33|42]>, FAIL>` and `opt<attempt<number[33|42], FAIL>>`, depending on the order of treating errors and options. We were not able to compute this type by using MPS' declarative type system DSL and resorted to imperative code. This code essentially treats attempts and options explicitly. This means, for example, that we could not implement options and attempt modularly: they are "baked into" the core type system. And one such baked in rule is that you cannot mix options and attempts; so the code above is flagged as illegal. For the DSLs we have built so far, this is an acceptable restriction.

### 6.2 Reactive Interpreter

Consider the following code, which might be part of a larger program (the functions) and test data (the values plus the `assertions`):

```
// test data for John
val j_last      = "Doe"
val j_first     = "John"
val j_birthYear = 1974
```

```
fun greet(f: string, l: string) = "Hello " + f + " " + l
fun age(y: int) = currentYear() - y
fun birthday(f: string, l: string, y: int) =
  "Happy " + age(y) + ". birthday, " + f + " " + l
```

```
test case Test_John {
  assert (1) greet(j_first, j_last)                 equals "Hello John Doe"
  assert (2) greet("Geddy", "Lee")                 equals "Hello Geddy Lee"
  assert (3) age(j_birthYear)                       equals 44
  assert (4) birthday(j_first, j_last, j_birthYear) equals "Happy 44. birthday, John Doe"
}
```

**The Status Quo** Our current interpreter works on-demand, always runs to completion. On-demand means that a recomputation is explicitly requested. The request can happen in two ways. One way is for the user to press `Ctrl-Alt-Enter` on a program node that has a manual check (indicated through an interface implemented by the node's concept). Alternatively, the execution of manual checks (and thus, the interpeter) can be triggered by the type system, in which case MPS uses heuristics to decide when to trigger the update. In the above example (and in the current KernelF implementation), the assertions implement the required interface, so users can reevaluate an `assert` this way. `Ctrl-Alt-Enter` also works for containers, so pressing it on the whole test case, or the surround (but not shown) tests suite recalculates all of them.

Once a recalculation is triggered it always recalculates everything, to completion. So, for example, when triggering the recomputation on the last `assert`, the interpreter for `assert` is invoked. It invokes the interpreter for the `actual` and `expected` slots. The string literal in the `expected` slot is trivial. The `actual` slot evaluates the function call. In turn, it evaluates the arguments (by calling the interpreter for the `val` references, and then, transitively, the interpreter for the init expressions on the `vals`) and then dispatches to the `birthday` function. Inside, among other things, evaluates the call to the `age` function.

**Reactivity** A more scalable way would work as follows:

– A change to `Geddy` would trigger assertion 2
– Changing any of the `j_` values would never trigger 2

– A change to `j_last` would trigger recalculation of 1 and 4
– A change to `j_age` would trigger recalculation of 3 and 4

We would also expect that, even if 4 is recalculated because `j_last` has changed, we would *not* execute the call to `age` inside `birthday`, because the argument to `age`, `j_birthYear`, did not change. Finally, we would also expect the on-demand recalculation for changes to the program: if we change the implementation of `age`, then 3 would have to be recalculated, but also 4, because it indirectly relies on `age`. This behavior would be just like in Excel[15]: you can imagine the `val`s as cells with user-entered values, the `assert`s as cells with formulas in them and the function calls as macros. To make this reactive architecture work, the following ingredients are required:

– Change Notifications: the engine that triggers the interpreter must be notified of changes to program nodes. Since MPS is a projectional editor, and changes to the AST are already performed essentially via an architecture that relies on events, those change events are easy to get.
– Reverse Dependencies: MPS maintains a fully resolved AST, i.e., even references such as `j_first` in assertion 1 or the reference to `age` in assertion 3 are maintained as fully resolved "object pointers". However, in order to find out which parts of the program must be recomputed, the reverse dependencies are required: if the string literal `"Doe"` is changed, then we have to follow the upstream tree of containment and reference dependencies (as indicated in Figure 23). MPS does not currently maintain (all of) these reverse dependencies. However, we assume we can maintain our own overlay data structure that is updated based on the same program change events just mentioned.
– Persistent Interpreter: Currently, the interpreter is restarted from scratch for every evaluation request (explicitly or by the type system). Restarting the interpreter means that the interpreter context, the data structure that maintains the interpreter's internal state, is also recreated, which means that all caches are empty. Thus, when a function is called with an argument for which it has been called before (and the function is pure), then the interpreter will recompute the function's result instead of reusing the one from the cache. So, again assuming a change to `"Doe"`, this triggers the recomputation of assertion 4, which calls `birthday`, which then calls `age`. Even though the argument to `age` did not change, the function is re-executed, because the (empty) cache does not know the previous result. To fix this issue, the interpreter's context (and thus, caches), would have to be maintained persistently during a user's interactive editing session.

All of these changes are absolutely feasible, and we will work on this architecture in the future. While the current implementation is not very scalable, we can, for now, live with the limitation because the in-IDE-interpreter is used for testing, and test cases are usually small and thus still run reasonably quickly. For systems that require larger integration test-style scenarios, we have explicit mocking features that act as "breakpoints" in the execution of the interpreter.

[15] An analogy that many of our users like to draw in more ways than is good for us!

```
// test data for John
val j_last      = "Doe"
val j_first     = "John"
val j_birthYear = 1974
```

```
fun greet(f: string, l: string) = "Hello " + f + " " + l
fun age(y: int) = currentYear() - y
fun birthday(f: string, l: string, y: int) =
  "Happy " + age(y) + ". birthday, " + f + " " + l
```

```
test case Test_John {
  assert (1) greet(j_first, j_last)                equals "Hello John Doe"
  assert (2) greet("Gedddy", "Lee")                equals "Hello Geddy Lee"
  assert (3) age(j_birthYear)                      equals 44
  assert (4) birthday(j_first, j_last, j_birthYear) equals "Happy 44. birthday, John Doe"
}
```

**Fig. 23.** The example code for reactive interpreters shown with the reverse dependencies relevant for a change to the value `"Doe"`. Solid lines represent containment, dashed lines represent reference dependencies.

### 6.3 Shadow Models

Many language extension add new abstractions on top of existing ones. This means that for their semantic definition, they can be "desugared" to more basic constructs. The `alt` expression is an obvious example:

```
alt | <cond-1>  => <val-1>  |
    | <cond-2>  => <val-2>  |
    ...
    | otherwise => <val-n>  |
```

desugars to

```
if <cond-> then <val-1> else
  if <cond-2> then <val-2> else
    ...
    else <val-n>
```

It is idiomatic for MPS generators to be stacked, and they can be scheduled to perform desugarings to a base language, before that language is processed further. Essentially, all of mbeddr's C extensions are translated this way. It would be nice if the same approach could be used with interpreters as well: programs are reduced to their most basic form, which is then submitted to the interpreter. This way, the interpreter only has to be defined for a minimal language. More importantly, the same desugaring could be used independent of what is done with the desugared, basic form of the program: it could be interpreted, submitted to a Java generator, or translated to the solver. You can see while this approach is very desirable for reasons of reduced effort and improved quality.

The reason why the approach works well with generators is that those are executed on demand; when the user requests a (re-)build of the model, the cascade of generators is executed according to their relative priorities ("higher" desugarings first). However, the interpreter is expected to run interactively, which means, very fast: as the user changes parts of the program, the interpreter should be executed and the results updated. The same is true for the checks performed with the solver. What we would need is an incremental maintenance of the desugared (or otherwise derived) models. While it is easy in MPS to receive fine-grained notification of changes to programs, we have not yet found a way of expressing the necessary incremental graph transformations. While we are actively working on this challenge, for now, every language concept requires a native interpreter, i.e., one that is specifically implemented for the (potentially desugarable) language concept.

# 7 Related Work

## 7.1 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.[16] In addition, the languages are all not purely functional and provide no support for explicit effects tracking. We discarded this option early and clearly.

## 7.2 Other Base Languages

**mbeddr C** mbeddr [32] is an implementation of C in MPS. It uses the same extension mechanisms as KernelF because it is built on MPS as well. Like KernelF, 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 generalising 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 standardised as part of Java 8). It also ships with a set of (modular) extensions for meta programming, supplying 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 what is relevant in any particular DSL.

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

[16] 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.

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 [10] is a functional language that ships with Xtext[17]. Similar to KernelF, its purpose is to be extended and embedded in the context of DSLs. Xtend[18] is a full programming language (with classes, modules and effects) that embeds Xbase expressions. Similar to Kotlin[19] and Ceylon[20], 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 significant 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 in general.

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

### 7.3 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[21] takes this to an extreme and allows significant syntactical flexibility for Lisp or extensions. We decided against this style of language for several reasons:

First, while, generally, it is a matter of taste (and of getting used to it) whether developers 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

[17] `https://www.eclipse.org/Xtext/`
[18] `http://www.eclipse.org/xtend/`
[19] `https://kotlinlang.org/`
[20] `https://ceylon-lang.org/`
[21] `https://racket-lang.org/`

the syntactic freedom to textual notations in general, and to the capabilities of the macro system more specifically. We needed more flexibility.

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).

### 7.4 Embeddable Languages

Lua[22] is a small 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 front-end 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 generate to Java or C, or implement reactivity (Section 6.2).

### 7.5 Other Language Workbenches

This paper is not about evaluating MPS' suitability as a language workbench; see [30] instead. Thus, a detailed evaluation about alternative implementation technologies for KernelF is outside the scope of this paper. Nonetheless, if, for some reason, we could not use MPS for KernelF and our customer projects, Racket would probably be the best alternative.

## 8 Conclusion

We have built KernelF as a base language for DSLs. 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 language concepts users do not need must be removable or replaceable. Our case studies show that we have achieved this goal. Since developing KernelF, we have used it

[22] `https://www.lua.org/`

in most customer projects that required expressions or a full-blown programming language as a basis.

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 extensibility 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 [30], MPS supports this kind of modular language engineering extremely well.

We continue to use KernelF as a basis for our DSL work. We are also using it as the core of a set of meta languages in our new web-based language workbench Convecton. Once it is expressive enough, we will implement KernelF in Convecton so we have it available as a base language for Convecton-based DSLs as well.

~~~

**Acknowledgements** I implemented most of KernelF myself. However, this would not have been possible without the team at itemis: they were sparring partners in design discussions, they helped mature the language by using and stressing it, they built some of the features in the case studies, and generally provided the fertile ground on which something like KernelF can flourish. I also want to thank our customers. Not just those of the particular systems described in the case studies, but all of them: without their trust in us and, ultimately, their money, none of what is discussed in this paper would have happened. Finally, I want to thank the MPS team at Jetbrains for building an amazing tool and for helping us use it productively over the years.
~~~

## References


1. Jetbrains MPS Documentation. https://www.jetbrains.com/mps/documentation.
2. S. Amani, M. Bégel, M. Bortin, and M. Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. *CPP. ACM. To appear*, 2018.
3. T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund. Efficiency of projectional editing: A controlled experiment. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 763–774. ACM, 2016.
4. S. P. Booth and S. B. Jones. Walk backwards to happiness: debugging by time travel. In *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*, number 001, pages 171–184. Linköping University Electronic Press, 1997.
5. E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale. Execution framework of the gemoc studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 84–89. ACM, 2016.
6. F. Campagne. *The MPS Language Workbench: Volume I*, volume 1. Fabien Campagne, 2014.
7. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
8. F. DeRemer and H. H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, (2):80–86, 1976.
9. J. R. Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.
10. S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. Xbase: implementing domain-specific languages for java. In *ACM SIGPLAN Notices*, volume 48, pages 112–121. ACM, 2012.
11. S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, page 7. ACM, 2012.
12. M. Fowler. Language workbenches: The killer-app for domain specific languages. 2005.
13. C. K. Frantz and M. Nowostawski. From institutions to code: Towards automated generation of smart contracts. In *Foundations and Applications of Self* Systems, IEEE International Workshops on*, pages 210–215. IEEE, 2016.
14. R. Gibbons. *A primer in game theory*. Harvester Wheatsheaf, 1992.
15. R. Hanmer. *Patterns for fault tolerant software*. John Wiley & Sons, 2013.
16. R. Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, page 1. ACM, 2008.
17. Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017.
18. C. S. Jensen and R. T. Snodgrass. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, 1999.
19. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.
20. K. Korpela, J. Hallikas, and T. Dahlberg. Digital supply chain transformation toward blockchain integration. In *proceedings of the 50th Hawaii international conference on system sciences*, 2017.

21. P. Narasimban, L. E. Moser, and P. M. Melliar-Smith. Using interceptors to enhance corba. *Computer*, 32(7):62–68, 1999.
22. M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical report, 2004.
23. G. W. Peters and E. Panayi. Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the internet of money. In *Banking Beyond Banks and Money*, pages 239–278. Springer, 2016.
24. D. Tapscott and A. Tapscott. *Blockchain revolution: how the technology behind bitcoin is changing money, business, and the world*. Penguin, 2016.
25. M. Voelter. A smart contract development stack. [Online; posted 6 Dec, 2017].
26. M. Voelter. Language and ide development, modularization and composition with MPS. In *GTTSE 2011*, LNCS. Springer, 2011.
27. M. Voelter. *Generic tools, specific languages*. Delft University of Technology, 2014.
28. M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering*. dslbook.org, 2013.
29. M. Voelter, A. v. Deursen, B. Kolb, and S. Eberle. Using c language extensions for developing embedded software: A case study. In *Proceedings of OOPSLA 2015*, pages 655–674. ACM, 2015.
30. M. Voelter, B. Kolb, T. Szabó, D. Ratiu, and A. van Deursen. Lessons learned from developing mbeddr: a case study in language engineering with mps. *Software & Systems Modeling*, Jan 2017.
31. M. Voelter and S. Lisson. Supporting Diverse Notations in MPS' Projectional Editor. *GEMOC Workshop*.
32. M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20(3):1–52, 2013.
33. M. Voelter, T. Szabó, S. Lisson, B. Kolb, S. Erdweg, and T. Berger. Efficient development of consistent projectional editors using grammar cells. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 28–40. ACM, 2016.
34. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.

# A KernelF Reference

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 operations are not mentioned. They can be found out easily through code completion in the editor.

## A.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]>
}
```

**A note on brackets** We try 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.

## A.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 3.1). 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>
```

## A.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 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 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 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 val 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
```

## A.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 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 below), 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:

```
// try will have error b/c error404 is not handled
val toDisplay = 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]]>
```

### A.5 Functions and Extension Functions

Even though the notion of "callable units of program functionality" is often 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). The return type is optional as it can be inferred. For recursive functions it is mandatory.

```
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>
```

### A.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 `type` definition 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 `type` 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 delineation.

```
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`; note the two different syntaxes to “call” function values:

```
fun doWithTwoInts1(x: number, y: number, op: INT_BINOP) =
                                     op.exec(x, y)
fun doWithTwoInts2(x: number, y: number, op: INT_BINOP) =
                                     op(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>
```

## A.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]`[23].

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 side effect; it "performs" the side effect and then returns the original list. We discuss effects in Section A.13.

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.).

### A.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>
```

### A.9 Records and Path Expressions

Like Tuples, records are structured data, but they are explicitly declared and the members are named. KernelF has them primarily for completeness and for prototyping; experience tells us that most data structures are domain-specific and thus contributed by a language that embeds KernelF.

```
record Company {
  offices: list<Office>
  emps   : list<Person>
}
record Person {
```

[23] 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.

```
  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 24. It can be used for any hierarchical structure, not just records, if a suitable adapter is provided.

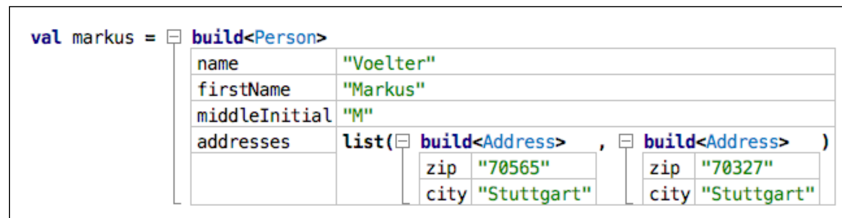


**Fig. 24.** The builder expression uses collapsible trees to build hierarchical structures.

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 25 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 access 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(". ")
```

**Fig. 25.** 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 25, we create one that has two fields, `author` and `cats`.[24] `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 25 shows an example.

### A.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
```

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

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 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]
```

## A.11 Unit Tests and Constraints

**Tests** Built-in support for unit tests is important, because, as we describe in subsection 2.2, 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; even our target audience of "programming domain experts" realizes the importance of testing and use the feature extensively.

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 `assertion` 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. Figure 32 shows an example.

**Constraints** KernelF supports checking of runtime constraints. Several forms exist, all illustrated in Figure 26:

- 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 initialising a value, when returning from a function, and when passing an argument into a function. For chained `type` definitions, the constraints are joined in a conjunction ("`and`ed" 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.

– 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 reaction. 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.

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

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

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

**Fig. 26.** 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.

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 using the `check` expression, 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[25] 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)
```

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

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. There is no way to avoid *all* static and runtime checks; KernelF always at least provides runtime safety.

### A.12 Type Tags

A type tag 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 [**?**]. 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
// sanitise is a special operator that cleans up the string,
// and them marks it as sanitized; passing to storeInDB works
val valid = storeInDB(sanitise[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)
```

### A.13 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 standardise/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 `standardise` 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.

## A.14 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 conclusion, 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 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 concise, 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.

```
public class BoxValue {
  private Object value;
  public BoxValue(Object initial) { this.value = initial; }
  public void set(Object newValue) { this.value = newValue; }
  public Object get() { this.value; }
}
```

### A.15 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 27 shows minimal one that represents a (slightly contrived) counter:

It defines two events, one to initialise 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:

```
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
  }
}
```

**Fig. 27.** A state machine is an example of natively mutable data structure.

```
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:

```
Object rvalue = #right; // recursively call interpreter
Object lvalue = #left;  // on the two arguments
if (lvalue instanceof ILValue) { // must be an ILValue
  // which has update method
  ((ILValue) lvalue).updateValue(rvalue);
} else {
  throw new InvalidValueException(node.left, "not␣an␣ILValue");
}
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`)

## A.16 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 of state machines and other boxes, as shown in the example below where the state machine modifies other global data.

```
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()
  }
}
```

**Fig. 28.** An example of using transactions with different mutable data structures.

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 transactions 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.

### A.17 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).

### A.18 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.

# B KernelF Tooling

## B.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 that the IDE support MPS provides is *the* IDE support for KernelF. Like for any other language, 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.

## B.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.

## B.3 Read-Eval-Print-Loop

KernelF ships with a read-eval-print-loop (REPL; Figure 29 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`

```
repl Time_repl_0 for node artificialclock(0) in Time
downstream updates ☐
Reevaluate All   Clear   Clear Tail   Close & Return

[0]  artificialclock(0)   : artificialclock, effects[reads]
         artificialclock:5
[1]  $0.advance(5)   : artificialclock, effects[modifies]
         artificialclock:5
[2]  $0.time   : number[0|∞], effects[reads]
         5
[3]  <no expression>
       <no result found>
```

**Fig. 29.** An example of a REPL session on a clock expression.

button in the REPL, the REPL is deleted and the node from which it was opened is put back in focus.

## B.4 Debugger

One of the benefits of a functional language is that there is no program state to evolve; all computations can be 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.[26] The debugger, also known as the tracer, can be invoked for anything that has no upstream dependencies, i.e., test case `assertion`s, global `value`s and `fun`ctions 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 computation, but with different values, producing a different 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

[26] 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.

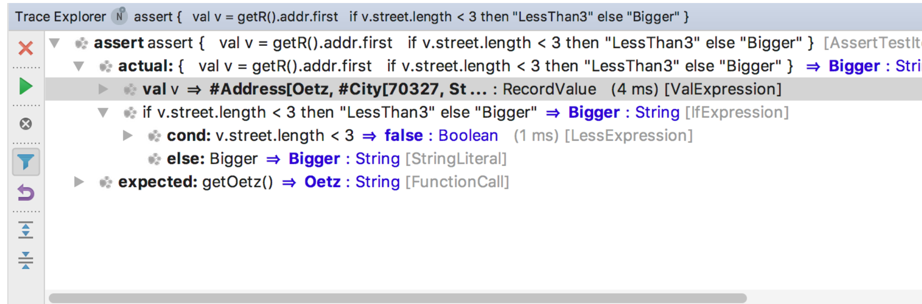


**Fig. 30.** The frame tree as shown in the debugger.

(for example, `cond` or `then`), the (abbreviated) syntax, the (abbreviated) value and the time it took to compute it[27]. 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 31, 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 governed 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.

**Fig. 31.** Decorated code that associates values with syntax nodes.

**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 re-executes the same root, if it is re-executable (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 re-executions. The little grey round X removes all code decorations created by the current tab. The blue filter icon toggles

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

between the regular 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 `[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.

### B.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.

```
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}>
}
```

**Fig. 32.** 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.

### B.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 statistics on the coverage, as shown in Figure 33.

*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.

### B.7 Test Case Generation

KernelF supports test case generation; an example is shown in Figure 34. 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[28], 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 selects “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.

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

```
assessment: InterpreterCoverage
query:      interpreter test coverage in current model
              problems only: ☐
              languages: language/org.iets3.core.expr.base/
              ignore:     {33 ignored concepts}
sorted: ☒     must be ok: ☐     hide ok ones: ☐
last updated: 2 minutes ago by markusvoelter
--------------------------------------------------------------------
org.iets3.core.expr.base
--------------------------------------------------------------------
| TupleValue                Covered.
| SomeValExpr               Covered.
| LogicalImpliesExpression  Covered.
| ErrorExpression           Partial. Missing: [generic]
| SuccessExpression         Covered.
| ParensExpression          Covered.
| TryExpression             Partial. Missing: [fallthrough, generic]
--------------------------------------------------------------------
total 38, new 38, ok 0
value ranges
  decimal.max       = 340.00
  decimal.min       = -80.01
  decimal.zero      = true
  integer.zero      = true
  integer.max       = 99383
  decimal.minusOne  = true
  integer.one       = true
  integer.minusOne  = true
  decimal.one       = true
  integer.min       = -99383
coverage 95 %
--------------------------------------------------------------------
```

**Fig. 33.** 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 analysis). 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.

```
test case TestFunctions [success] {
  vectors function add  -> producer: random 25
          results: true
```

| | a | a2 | b | c | s | res | status |
|---|---|---|---|---|---|---|---|
| 0: valid | 3 | -3.06 | false | BLUE | "" | 3 | ok |
| 1: valid | 2 | 2.74 | false | BLUE | "M/Yh-0I/ac" | 2 | ok |
| 2: valid | 1 | 0.22 | false | BLUE | "" | 1.22 | ok |
| 3: invalid input | 4 | -0.45 | true | BLUE | "7l:6h7sg!afLmULGU8wtI00H9" | | not executed |
| 4: invalid input | 1 | 1.38 | false | RED | "Mtoa7J66uuWTye2f2-fLhD§hj8C2K" | | not executed |
| 5: invalid input | 1 | -2.63 | false | BLUE | "n66r7E (f0J$aQMj§" | | not executed |

```
vectors function plus -> producer: eqclass
        results: true
```

| | a | b | c | res | status |
|---|---|---|---|---|---|
| 0: valid | 0 | -10 | GREEN | -10 | ok |
| 1: valid | 0 | -10 | BLUE | -10 | ok |
| 2: valid | 0 | 10 | GREEN | 10 | ok |
| 3: valid | 0 | 10 | BLUE | 10 | ok |
| 4: valid | 0 | -1 | GREEN | -1 | ok |

| | a | b | status |
|---|---|---|---|
| 0: valid | 7 | 2 | no expected value given; actual was 14 |
| 1: valid | 1 | 8 | no expected value given; actual was 8 |
| 2: valid | 5 | 2 | [POST] res == a * b |
| 3: valid | 7 | 8 | no expected value given; actual was 56 |
| 4: valid | 5 | 6 | [POST] res == a * b |
| 5: valid | 8 | 0 | [PRE] b > 0 |
| 6: valid | 5 | 8 | [POST] res == a * b |

**Fig. 34.** A couple of examples for test case generation; refer to the text for details.

### B.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 arithmetic operations with others, e.g. `+` with `*`, `-` with `/`
- Replacement of 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 35 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

```
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
}
```

**Fig. 35.** An example of mutation testing.

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

fun doodle(a: number[1|5]) = if true then a else a * 2
                                                     1

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

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

**Fig. 36.** Highlighting of code mutations. The mutated code is red, the original one is grey.

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 36. Note that the mutator can also touch indirectly used functions; the particular scope of the mutations is defined by the test subject adapter.

# C Language Design 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 value 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 recognise 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
```

## D Language Development with MPS

This section describes language development with MPS. As a running example, this section illustrates the development of the `alt` expression. This section is not a full tutorial for which we refer the reader to [28], [6] [27] and [1].

**Example** The running example we use in this section shows the implementation of one of the language features of KernelF: the `alt` expression. Its purpose it to return a value based on a set of conditions; it is roughly similar to a `switch` statement in Java. Figure 37 shows an example.It defines a function with two arguments, one for the pulse of a person, and the other one representing their age. Based on these values, we compute a risk factor for the person. To achieve this, we use an `alt` expression that looks at the two contributing factors and returns the risk.[29] Note that an expression must *always* evaluate to a value. So for `alt` to be correct, all possible combinations of the inspected values must be taken into account. To ensure this, KernelF includes a solver that checks all cases in the `alt` expression for completeness; we do not discuss the solver integration here. Alternatively, the user can also add a catch-all case using the `otherwise` condition. We use an attempt type to signal the error.

**Terminology** We introduce some of the most important terminology used in MPS. A *program* denotes code written by an end user. It is represented as the AST and projected in whatever notation is defined for the language in which the program is written. A program consists of a tree of *nodes* (the AST) with resolved cross-references between nodes (so it is effectively a graph). A *root* node is a node that has no parent; it is edited in its own editor tab in the MPS IDE and intuitively corresponds to a file in a classical IDE. The *model* is the granularity of physical storage in MPS. It is an XML file and contains a number of root nodes, each with its own tree/graph beneath it. Models are owned by *modules*, and a *project* is a collection of modules.[30] Modules come in three kinds: *Languages* are modules that contain language implementations. *Solutions* are modules that contain end-user programs, as well as support libraries for languages. Thus,

[29] In practice, we would probably use a two-dimensional decision table that splits the two ranges separately, but a table is too complex an example to use here.

[30] For Eclipse users: the project corresponds to the Eclipse Workspace, the modules correspond to projects and models correspond to files or packages.

```
val AGE_THRESH_1 = 40
val AGE_THRESH_2 = 60

fun determineRisk(pulse: number[0|200], age: number[0|120]): attempt<number[1|3]{0}|INVALID> {
  alt | pulse < 60                                  => 1              |
      | pulse.inRange[60..80] && age <= AGE_THRESH_1 => 2              |
      | pulse.inRange[60..90] && age <= AGE_THRESH_2 => 2              |
      | pulse >= 90                                 => 3              |
      | otherwise                                   => error(INVALID) |
}
```

**Fig. 37.** Example use of the `alt` expression in a function.

language modules are the meta level relative to solution modules. *Devkits* are groups of languages to simplify a solution's import of related groups of languages. Finally, the *BaseLanguage* is MPS' (slightly extended) version of Java. It can be used for Java programming (in solutions) and also plays a role in language implementation. We now discuss some details of the various languages used for language implementation.

A language definition consists of language *aspects* such as structure, editor or type system. Each of the aspects is implemented with an aspect-specific DSL. Some of these DSLs are declarative, others are rule-based, and yet others are imperative. However, all of them reuse MPS' BaseLanguage to some extent, typically by embedding BaseLanguage expressions or statements. All of them are also optimized for modular language definition. We discuss each language aspect below, each in a separate subsection. We start with a subsection on language modularity and dependencies in general.

## D.1 Created and Used Languages

The `alt` expression developed in this running example is part of KernelF's core. However, for the purpose of this tutorial, we assume that it is a modular language extension, whose development must not require changes to KernelF itself. Thus we start out by creating a new language `org.kernelf.ext.decisions`. Because we will reuse parts of the definition of KernelF, we make this new language extend `org.kernelf.base`; it is part of the modularised implementation of KernelF. Creating new languages and defining their dependencies is handled via menu items and property dialogs in MPS; we do not show these in this paper.

## D.2 Structure

A language contains a number of language concepts (known as meta class or AST type in other tools). Each of the language aspects mentioned before contributes to each concept's definition. In this sense, a language definition in MPS is a 2-dimensional matrix of concepts and aspects.

The definition of a language concept starts with its structure, because all other aspects refer to the structure of concepts in one form or another. For our example we need two concepts: the `AltExpression` itself as well as the `otherwise` expression used inside its body.

The `otherwise` expression is a keyword expression, i.e., it is an expression with a language-defined structure and syntax. It has no further substructure under it (in terms of the AST). Here is the structure definition:[31]

```
concept OtherwiseExpr extends Expression
  alias otherwise
```

[31] As long as MPS uses textual notations for language definition, we show the example code as text; when non-textual notations are used, such as in editor definitions, we use screenshots.

MPS uses an object-oriented style subtyping to implement structural compatibility: if the `OtherwiseExpr` is to be legal in places where KernelF expects an `Expression` (being a functional language, this is almost everywhere), then `OtherwiseExpr` has to extend KernelF's `Expression` concept. It is visible here because our new language extends `org.kernelf.base`, which contains the `Expression` concept.

The alias is the string a user has to type (or select from the code completion menu) to enter an instance of `OtherwiseExpr` when editing a program. It is good practice, though not technically required, to make the alias the same as the leading keyword of the concrete syntax of the concept; in case of `Otherwise` the two are identical.

The `AltExpression` also extends KernelF's `Expressions`, and defines an alias `alt`. In addition, it implements two interfaces that integrate the new expression with the effect tracking framework of KernelF.

```
concept AltExpression extends Expression
                      implements IMayHaveEffect
  alias: alt
  children:
    alternatives : AltOption  [0..n]
```

Importantly, the `AltExpression` also defines a child collection `alternatives` that represents a `<cond> => <value>` pair. The cardinality is `many`, and the concept is itself defined as below; this should be rather self-explanatory now.

```
concept AltOption implements IMayAllowEffect
  children:
    when : Expression  [1]
    then : Expression  [1]
```

### D.3 Editors

In MPS, editors play the role of the concrete syntax, or notation: they define how an instance of a concept is visually represented. They also define actions that customize how the user interacts with the instance of the concept when editing a program. We discuss both below.

**Notation** Each language concept has its own editor.[32] An editor consists of a collection of editor *cells*. There are many different kinds of cells available for use by the language developer: examples include constant cells, cells that contain child nodes, collection cells that act as containers for more cells as well as cells that render arbitrary strings. The editor for the `OtherwiseExpr` is the simplest one possible: it contains a single constant cell with the text "otherwise".

Figure 38 (left) shows the editor definitions of the `AltExpression`. It contains a horizontal collection (`[> .. <]`) at the top level, which in turn contains a constant cell with the value `alt` and vertical child collection (`(/ .. /)`) of the `alternatives`. Each of the `AltOptions` in the `alternatives` collection uses its own editor, as discussed next. Three more details are configured in the

[32] A concept can have several editors; they can be switched for each program.

```
<default> editor for concept AlternativesExpression
  node cell layout:
    [> alt (/ % alternatives %                    /) <]
              /empty cell:   <constant>
              /folded cell: * R/O model access *
```

```
<default> editor for concept AltOption
  node cell layout:
    [> wrap % when % => % then % <]
```

**Fig. 38.** The editor definition for the `AltExpression` (left), and the `AltOption` (right).

inspector[33]. First, the `alternatives` child collection is marked as `foldable`, i.e., a little minus signs show up in the gutter of the editor that let the user collapse the `alt` expression. In folded/collapsed mode, the editor shows a text instead of the options, that text is computed inside a `read-only model access` cell. In this example, it simply returns the string `"{N options}"`, where N is the number of options in the `alt`. Second, the layout of the collection is actually `vertical grid`, not just `vertical`. This makes sure that the `=>` symbols of all options are aligned underneath each other, leading to a cleaner syntax. The third setting done in the inspector sets the `draw-brackets` style attribute to true, which makes the surrounding large brackets show up in the editor.

For the `AltOption`, shown on the right side of Figure 38, we basically have a horizontal list with three elements: a child cell `%when%` that embeds the condition, the constant cell for the `=>` symbol, and another child cell for the `then` child. We discuss the special `wrap` cell next.

**Actions** In addition to the definition of the visual representation of concepts in the editor, the editor aspect also defines how users interact with the notation. Examples of such interactions include:

- Deletion: what happens when the user presses `Backspace` on a given cell.
- Side transformations: how can tree structures be entered linearly (entering `2+3` by typing `2`, then `+` and then `3`).
- Substitutions: allow a local variable declaration (such as `int32 x;`) to be created by entering the type `int32`.

Traditionally, in raw MPS, language developers have to implement all these actions with a very low-level DSL, which is very tedious and error-prone. In particular, it is easy to create inconsistent editing behaviors, which confuses users. To solve this problem, we have developed grammar cells [33]. They provide special cells that imply their editing actions. We always use grammar cells to define languages in MPS; it is a huge boost to productivity and language (editor) quality.

One of them is the `wrap` cell. It is placed around a child cell in an editor definition, and has the following behavior: in places where an `AltOption` is expected, the user can also enter an `Expression` (because the `wrap` cell is placed around a child of concept `Expression`, the `when` of an `AltOption`). If the user enters an `Expression`, the editor removes the just entered `Expression`, creates

[33] A kind of property view that allows the specification of additional details for editor cells; not shown in Figure 38.

an `AltOption`, sets the `Expression` as its `when` child, and inserts the thusly created `AltOption` at the location where the user entered the initial `Expression`.

## D.4 Constraints

As a first approximation, the validity of a program is determined by the structure: only nodes of a compatible concept (in terms of subtyping through the `extends` or `implements` relationship between concepts/interfaces) can be instantiated in a given program location. However, validity is further determined by typing rules (see next subsection) and constraints. A constraint is a Boolean expression that determines whether a structurally compatible concept can actually be instantiated in a given location, thereby further restricting the tree structure beyond pure structural compatibility.

**Tree Constraints** The `OtherwiseExpr` extends `Expression`, so, structurally, it can be used wherever an `Expression` is expected. However, from a semantic perspective, it is only valid directly in the `when` child of an `AltOption`. To enforce this, we define a `can be child` constraint for the `OtherwiseExpr`:

```
constraints for OtherwiseExpr
  can be child
    (childConcept, node, parentNode, containingLink)->boolean {
      parentNode.isInstanceOf(AltOption) && containingLink == link(AltOption/when)
    }
```

Note that constraints *prevent* the user from entering invalid code[34]. This means that they are executed *before* the node, in this case the `OtherwiseExpr`, has been created. This is why the constraint is expressed in terms of the `parentNode`: we check if the parent is an `AltOption`, and we check that we are in the `when` slot. Note that this enforces that `otherwise` is used as the top level expression in the `when` slot; if we could nest it in a deeper expression tree (a typical situation) we would write the constraint as follows:

```
constraints for OtherwiseExpr
  can be child
    (childConcept, node, parentNode, containingLink)->boolean {
      // grab the containing AltOption from anywhere above us
      // the + includes the parentNode itself in the search
      node<> ao = parentNode.ancestor<AltOption, +);
      // not allowed if no AltOption is above us
      if (ao == null) return false;
      return (containingLink == link(AltOption/when)  // direct containment
             || ao.when.descendants.contains(node);   // indirect containment
    }
```

**Scopes** Constraints are also used to express visibility rules, known as *scopes* in MPS. A scope is a constraint that determines which target nodes are visible to a *reference* (as opposed to the containment constraints discussed before). These targets are then made available to the user through the code completion menu, so they can be selected or just typed in (scopes also lead to error markers in the case

[34] Remember that a projectional editor only lets the user enter programs that are structurally valid.

where a (now) invalid reference was entered when the scope was (erroneously) still more permissive). Our language extension does not contain any references, but we can look at the `ConstantRef`s used to refer to the two age thresholds in Figure 37. The structure is defined as follows:

```
concept ConstantRef extends Expression
  references:
    constant : Constant [1]
```

This reference can refer to *any* global constant declaration; it is the responsibility of the scope to determine the set of valid targets. Since the reference is typed to be a `Constant`, the scope implementation must return a sequence of global variable declaration nodes (an `nsequence<Constant>`). The implementation of the scope shown below starts from the enclosing node, navigates up the tree to find the `Library` in which the current node resides, gets its contents and filters for `Constant`s. Since it is structurally ensured that KernelF code is written in `Library`s, we can assume that one of the ancestors is actually a `Library`.

```
constraints for ConstantRef:
  link constant scope:
    (enclosingNode, pos) -> nsequence<Constant> {
      enclosingNode.ancestor<Library>.contents.ofConcept<Constant>;
    }
```

Note that this implementation is slightly simplified compared to KernelF's actual implementation of this scope, because KernelF uses a set of library functions to take import relationships between modules into account.

### D.5 Type System

The type system aspect encodes the static semantics of a language; it ensures the consistency of the types in the program and also checks other arbitrary correctness rules beyond structure.[35] We start with the latter, because it is different in an interesting way from the constraints just discussed.

**Checking Rules** As discussed above, constraints use Boolean expressions to determine whether a node is valid in a given program location. If it is not valid, they *prevent the user from entering that node*. The type system's checking rules similarly use Boolean expressions to determine validity. However, they are evaluated *after the node has already been entered*. Instead of preventing invalid use, they flag invalid use with a red squiggly line and an error message after the fact. The following code shows a checking rule for the `Otherwise` expression: it is only legal in the *last* of the options.[36]

[35] In traditional, parser-based language definitions, the type system is typically considered to also include the name-based resolution of references. However, in MPS, references are not encoded by name resolution rules, but by actual references to the unique ID of the target node that are established upon *entering* the reference (by code completion or plain typing).

[36] The interpreter we will implement later will always evaluate the `otherwise` last, but to make this clear to the reader of the program, we enforce that the syntax reflects this.

```
checking rule for OtherwiseExpr {
  ensure node.parent == node.ancestor<AltExpression>.alternatives.last
    else error "otherwise can only be used in the last alternative"
         on node.when;
}
```

The `ensure` statements verifies that the `otherwise`'s parent `AltOption` is the last of the containing `AltExpression`'s alternatives. If it is not, it annotates the error message. In addition to `ensure` statements, checking rules can also use regular `if` statements to check more complex conditions, and report errors using the `error` statement.

**Typing Rules** Our example also makes use of actual typing rules: the first one ensures that the `when` part of an `AltOption` is of type `boolean`:

```
typing rule for AltOption {
  typeof(node.when) :==: <boolean>;
}
```

MPS' type system relies on typing equations: for every concept, the developer specifies one or more typing equations. MPS then instantiates all type equations for all program nodes and uses a solver [**?**,**?**] to infer types and detect typing errors. Here we declare a typing equation that expresses that the type of the current node's `when` child must be identical to an instance of `BooleanType` (written using the quotation syntax). Note that the `:==:` operator is not an assignment, but it establishes type equivalence. If one of the two arguments evaluates to an unbound type variable (based on all the other typing equations for the given program), the operator propagates the type to the unbound type variable, thus implementing type inference. If both arguments of `:==:` return actual types, then the operator acts as a constraint: if the two types are not the same, an error is reported. There are additional typing operators beyond `:==:`. For example, `:<=:` takes subtypes into account. A small detail worth mentioning here is that the actual typing rule looks slightly different:

```
typing rule for AltOption {
  typeof(node.when) :==: PTF.createBooleanType();
}
```

`PTF` is the factory for the primitive types, and it is used to realize the exchangeable primitive types feature required for KernelF's embeddability.

A more interesting typing rule is the one for the `alt` expression itself: the type of `alt` must be the "smallest common supertype" of all the `then` children of all alternatives. In principle, this could be written as follows:

```
typing rule for AltExpression {
  var T;
  foreach alternative in node.alternatives {
    T :>=: typeof(alternative.then);
  }
  typeof(node) :==: T;
}
```

We declare a type variable `T` which will become the type of the `alt` expression; see the last line where we equate the `typeof(node)` with `T`. To compute `T`, we

iterate over all alternatives and register an equation that expresses that `T` must be the same or a supertype of the type of the `then` part of each alternative. Again, note that the `:>=:` is not an assignment, it is a constraint on `T`; MPS solves the equation system created by registering all the equations for all alternatives with the type system solver.

If you look at the typing rule for the actual `alt` expression in KernelF, it looks quite different. This is because dealing with options, attempts and number types requires a lot of more differentiated treatment of super types. However, this is beyond the scope of this tutorial.

## D.6 Behavior

The behavior aspect allows the definition of methods on concepts. These act similar to Java methods and can be invoked from all other aspects (often called from constraints, the type system and generators). Methods are polymorphic, and they can also be declared on interfaces. For example, the `IMayAllowEffect` interface (implemented by `AltOption`) defines several abstract behaviour methods, among them `allowsEffect(node<> n)`. It has to be implemented by the concepts that implement `IMayAllowEffect` and return an error string if the argument node has an invalid effect, and `null` otherwise. `AltOption` implements it as follows:

```
public string allowsEffect(node<> n) overrides IMayAllowEffect.allowsEffect {
  if (n == this.when) {
    return EffectDescriptor.reads().allows(n, "only_read_effects_allowed_for_when");
  }
  null;
}
```

`EffectDescriptor.reads()` creates and effect descriptor that allows only `read` effects, and the `allows` call checks if the argument `n` either has no effect or a `read` effect; if it has a `modify` effect, then the methods returns the error message passed as the second argument.

`allowsEffect` is invoked generically by a checking rule which annotates the returned error message, if any, to the respective node.

## D.7 Generators

The name generator is a little bit misleading: in fact, generators are tree transformations that map a source AST to an output AST. Generators consist of various different kinds of transformations rules, which in turn make use of templates, i.e., fragments of target language code that determine how the source AST is transformed to the target AST. We skip the discussion of generators, because they are not discussed any further in this paper; we focus on interpreters. Refer to the tutorials referenced at the beginning of this chapter for details on generators.

## D.8 Interpreters

At the core, interpreters are essentially Java code blocks associated with language concepts. Each code block, called an evaluator, has two tasks: return a Java

```
alt [] {
        register branches aCase, otherwise;
        foreach caze in node.nonOtherwiseAlternatives() {
          if (((boolean) #(caze.when))) {
            branch aCase;
            return #(caze.then);
          }
        }
        node<AltOption> otherwise = node.findOtherwise();
        if (otherwise != null) {
          branch otherwise;
          Object neededForTracingAndCoverage = #(otherwise.when);
          return #(otherwise.then);
        }
        return null;
      }
```

**Fig. 39.** The evaluator for the `AltExpression` in the interpreter.

object that represents the value to which the AST node evaluates, and recursively invoking the interpreter for child nodes or reference targets, using their returned value in the computation of its own return value.

An interpreter is a collection of such evaluators – typically all the evaluators defined for the concepts in a particular language – with a declaration of the language to which the interpreter applies and priorities relative to other interpreters, the latter supporting shadowing of evaluators.

There might be several evaluators for one language concept (in the same or in different interpreters), which either use different type guards to support dispatching, or they return `tryNextInterpreter` if, after trying, they find out they are not able to evaluate a particular node.

Figure 39 shows the evaluator for the `AltExpression`. The evaluator uses the alias `alt` as a marker, has an empty type guard (it would be written between the brackets), and then has a Java code block to perform the evaluation. The Java code first iterates over all non-`otherwise` alternatives and recursively calls the interpreter for the `when` part of each `AltOption`; the `#(<expr>)` syntax recursively calls the interpreter on `expr`. We know that the type checker ensures that the `when` expression of `AltOptions` is a Boolean, and we rely on the fact that the Booleans used in KernelF are always represented as Java `boolean` in the interpreter, so we can perform a cast. If the `when` evaluates to true, then we recursively call the interpreter on the `then` child of that `AltOption` and return this value as the result of the evaluation of the `alt`. If no alternative is true, we end up where we check if there is an `otherwise`. If so, we evaluate and return its `then` part. If not, we have an inconsistent program. We return `null`, which is generically detected as an internal error and stops the interpreter.

The pink statements refer to the coverage analyzer. We declare two branches, and then "cover" them if we visit the `branch` statements as part of the execution of the interpreter.

### D.9 Intentions

Intentions are program transformations that change the program in the editor (as opposed to generators which transform programs during MPS' make process). Intentions are invoked by pressing `Alt-Enter` on a program node and then selecting a particular intention from the menu that pops up. MPS has a DSL for specifying such intentions, but the actual transformation is typically implemented procedurally using BaseLanguage. Since intentions are not very important for the remainder of the paper we do not discuss them any further.

### D.10 Refactorings

Refactorings are available from the context menu. They are also implemented procedurally. While they are important for the user, they are not particularly important for the rest of this paper, so we provide no more details.

### D.11 IDE Customisation

MPS supports the customisation of various aspects of the user interface of the tool itself, including buttons, menu items, customised project views as well as additional windows. These customisations are crucially important for building end user friendly products based on MPS. However, they are not relevant to language engineering per se, which is why we do not discuss them in this paper.

## E The MPS Language Design Philosophy

It is possible to use MPS to define programming languages that work like any other one: a relatively small set of language constructs designed for letting the user define their own abstractions plus a large standard library on which users can build. For example, MPS ships with an implementation of Java (called BaseLanguage) which is fundamentally similar to standard Java 7. The whole JDK is available for users to build on. However, when exploiting MPS' unique characteristics, the resulting languages look very different.

**Syntactic Forms** Because of the wide range of supported notations (mentioned in the previous subsection; code, prose, tables, diagrams, math), MPS-based DSLs often use more syntactic variety than languages built with other tools. It is also possible to define completely custom notations that do not fit any of these paradigms. Notations can also be mixed (nesting one in another, using them next to each other in the same "file"). Since MPS is unique in this respect among industry-strength language workbenches, it is not uncommon that MPS is specifically selected as the tool to implement a particular language because of this feature.

**Language Modules instead of Libraries** In general-purpose programming languages (GPLs), new abstractions are provided through libraries[37], developed

[37] In this context, we consider frameworks a form of library.

with the language itself. This is possible because GPLs are built for defining abstractions.[38] However, as a means of providing new abstractions for programmers, libraries are limited in the sense that they cannot meaningfully extend the language syntax, type system and IDE.

In idiomatic use of MPS, additional abstractions are provided through language extensions, defined outside the language, using MPS' language definition facilities. A language extension can be seen as a library plus syntax, type system and IDE support (and a semantics definition via an interpreter or generator). As Appendix D shows, the structure definition of languages is object-oriented, and many of the design patterns relevant for libraries and frameworks can also be found in MPS languages. Examples include the Adapter/Bridge/Strategy patterns or the separation of the construction of a data structure from its subsequent interpretation or execution. This approach fits extremely well with DSLs, which, because of their purpose and target audience, often do not come with sophisticated means of building custom abstractions.

One very nice feature of libraries is that, in general, they can be composed. For example, you can use the collections from the Java standard library together with the Joda Time library for date and time handling and the Spring framework for developing server-side applications. There is no need to explicitly combine the frameworks, the combination "just works". While this composability is not true for language composition in general (primarily because of syntactic ambiguities), it is true with MPS: for all intents and purposes, language extensions can be composed modularly, just like libraries. The approach also has the same limitations: one cannot statically prove that the composition will work, and the set of libraries/language extensions might not fit well in terms if their style. However, if language extensions are developed in a coordinated, but still modular way, as stack of extensions, these limitations do not apply. KernelF is such as stack.

To illustrate the difference between library and language extension, I will provide two examples. The first one concerns the collection in KernelF. Consider the following code:

```
val l1            = list(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) // type inferred to list<int>
val l2: list<int> = l1.where(|it > 5|).select(|it / 2|) // type inferred to list<real>
                                                         // results in type error for l2
```

The collections are generic: the collection type carries the type of the list elements, either explicitly specified (`l2`) or inferred (`l1`). However, KernelF does not generally support generic types. For example, users cannot write the following:

```
fun<type T1, type T2> typedPair(v1: T1, v2: T2): [T1, T2] = [v1, v2]
```

Generics are not generally necessary for DSLs. In fact, their exposure to the user will be often be confusing, and it will make the job of the language extender harder, because they have to take into account generics for all extensions. However, for collections, an explicit specification of their element type is useful and intuitive,

[38] In some languages, there are parts that are optimized for library developers, often, because they are more complicated to understand.

which is why the language extension for collections supports it. The `where` and `select` operators in the example above are also language extensions, available on `list` types. These could have been implemented with extension functions in a standard library. However, because they have to work with the collections' type parameters and because they use a particular kind of type inference not generally supported by KernelF, these are also built using language extension.

As a second example, take a look at the state machines towards the end of Section 5.2. They come with a rich syntax, specific type checks, and dedicated IDE support. It is hard to imagine how the state machine feature could be provided by a library, even in a language with meaningful meta programming facilities.

Because of the ease of developing languages in a modular way, we try to separate generally useful KernelF extensions from actual customer-specific extensions when we run projects; the generally useful parts become a customer-independent KernelF extension – if you will, the equivalent of a standard library, but as languages. For example, in the salary/tax cases study Section 5.1, the (independent) extensions for dates, currencies and temporal types are generic, and have been moved into the KernelF project. The languages for data and calculations lives under the customer's control.

The last point of comparison between libraries and language extensions is the effort to create them. For an experienced developer, the development of a language extension is not significantly more effort than the effort to write a library.[39] In addition, because language development and language use in MPS happens in the same environment, turn-around time is very quick, supporting iterative, and example-driven language development, just as if you develop a library together with representative examples of its use.

**More First-class Concepts** As a consequence of the increased reliance on language extensions, a (stack of) MPS language(s) will typically be more keyword-heavy than non-MPS languages. While this may offend the sense of style of some developers, this has two distinct advantages.

First, because more concepts are first-class, the IDE can know the semantics of those concepts and provide better support in terms of analyses. This, in turn, can be used to create meaningful error messages that align with the particular semantics of an extension. For example, in state machines, if the user creates a transition to the start state (assuming scoping allows this in the first place), an error message could read `Start states cannot be used as the target of a transition`, and in smaller font, below, `Start states are pseudo states that are only used internally during startup`. In a library-based solution, or one that relies on meta programming, very likely this problem cannot be determined statically at all, and would lead to a runtime error. Alternatively, the error message would perhaps be much more generic, as in `Type StartState is not a subtype of State` or something, which is also not very helpful to the end user.

[39] A sophisticated type system, notation or specific IDE support increases that effort, but then there is also a bigger benefit.

Second, the language is easier to explore, primarily because code completion has more sensible things to show. In a minimal language like Scheme, the contents of the code completion menu are essentially the completions for the basic syntactic forms such as atoms, lists or functions, plus calls to existing functions. This makes it harder for the user to explore the things they can do with a language.

**Focus on Evolution** Because languages and their extensions contain comparatively many first-class concepts, and many reflect a business domain that evolves, the languages we build with MPS also evolve quickly. Evolution in this context can mean one of two things. First, we may build additional languages on top of a core language, while keeping the core language stable; we grow the stack of languages into one or more domains. KernelF is designed to evolve like this, and this paper illustrates the process.

The other notion of evolution is the actual invasive evolution of the language itself (to make it concrete: you'll ship a new version of `kernelf.jar`, whereas in the extension case above you ship additional jars that rely on an unchanged `kernelf.jar`). If the new version is compatible with the previous version, this case is simple: just deploy the new version of the language, and users now have more features, while the existing programs remain valid. If the new version is not backward compatible, then existing programs become invalid. For this case, MPS supports explicit language versioning. As the language developer makes a breaking change to a language, they increment the version counter of the language and provide a migration script. When a language user opens an existing model after the new version has been deployed, the scripts run automatically, bringing the model up-to-date. If no algorithmic migration is feasible (because the user has to make a semantic decision not previously necessary), the recommended approach is to keep the old construct around, deprecate it, and output an error message that tells the user that a manual migration is necessary.

Note how this is a much more robust infrastructure for dealing with program migration than what is possible with libraries: an incompatible change prompts a generic error from the type checker or compiler, and automatic program migration is not available (outside of experimental systems). All in all, iterative development of languages becomes feasible, even when taking into account models that are "in the wild" with language users.

**Recasting Tools as Languages** Traditional programming systems consist of the language and libraries, the compiler and type checker, and an IDE. Many added-value services, for example, those for program understanding, testing, and debugging, are part of the IDE; more specifically, they rely on tool windows and other service-specific UI elements (buttons, menues, etc.). Because of MPS' flexibility in how editors can be defined, we use languages and language extensions for things that would be tool windows or other IDE addons in classical languages and IDEs. Examples include the REPL and its rendering of structured values (Figure 11), the overlay of variable values over the program code during debugging (Figure 31), test coverage and other assessment results (Figure 33), generated test vectors and their validity state (Figure 34) and the diffing of mutated programs vs. their original in the context of mutation testing (Figure 36).

```
A  pick from   StateMachine::nodes
        path   (node: Transition, parent: Any): collection<StateMachine> = parent.ancestors<StateMachine>
        filter (node: Transition, parent: Any, candidate: INode): boolean = candidate.isInstanceOf<IState>

B  pick from   StateMachine::nodes
        path   (node, parent) = parent.ancestors<StateMachine>
        filter (node, parent, candidate) = candidate.isInstanceOf<IState>

C  pick from   StateMachine::nodes
        path   = parent.ancestors<StateMachine>
        filter = candidate.isInstanceOf<IState>
```

**Fig. 40.** Three different levels of details for the projections of functions in Convecton meta languages: (A) types and names, (B) names only, with types available through an IDE action, and (C) nothing at all.

As a consequence, the notion of what constitutes a "language" is much broader in MPS, compared to the traditional understanding. A side-effect of this approach is that the chrome of the development environment – the set of windows, tabs, buttons, menus and such – can be reduced, because "everything happens in the editor, through typing, code completion and intentions". Since complaints about MPS' too cluttered tool UI is among the most-heard complaints among our users, we consider this side-effect an advantage.

**More Reliance on the IDE** There is no meaningful standard for the implementation of languages, which means that, once a language is implemented with one particular language workbench, it cannot be ported to another language workbench.[40] This is all the more true for MPS, which, because of its particular style of language implementation, is unique among language workbenches. Specifically because of its projectional editor, MPS languages cannot be used outside of the MPS tool. While this can be seen as a drawback, the flip side is that one can assume the IDE to *always* be present, and the language can be designed assuming the IDE and its services. I list a few examples below:

- Different projection modes: Instead of making a design decision on which level of detail should be used for function signatures, the user can switch (see Figure 40). This is useful because users with different levels of proficiency will prefer different styles: the newbie prefers the explicitly listed types, and once one gets more proficient, one appreciates the conciseness of alternative (C).
- Read-only editor contents: In many DSLs we use them to create a more form-style editor experience, with non-editable labels. In mbeddr, when a component `implements` an operation defined in an interface, we use a read-only projection of the operation's signature in the implementation.
- Intentions: These little in-place transformations of the program are available from a drop down menu activated with `Alt-Enter` (they are known as Quick Fixes in Eclipse). In some languages, especially non-textual ones, these are the *only* way to access certain constructs – you can't just type them. Many examples of this can be found in those languages that recast traditional IDE services (see previous paragraph). While replying on intentions might be

[40] Unless one implements it completely from scratch.

unintuitive for text-focused programmers, we teach our users to consider the intentions menu to be an integral part of the editor experience.

**A Hybrid** Domain-specific languages in general, and our approach in particular, are a hybrid between between modeling and software language engineering. From modeling we borrow declarativeness and high-level, domain specific concepts; multiple integrated languages; meta modeling for defining the structure of languages (named properties and links, inheritance, actual references); notational freedom, and in particular, diagrams. From the field of software language engineering we adopt a focus on behavior and integration of fine-grained aspects, such as expressions; actual type checking and not just constraint checks; powerful, productivity-focused IDEs; and textual languages. We like to think that the approach combines the best of these two worlds and leads to convincing outcomes. We will show this with our case studies in Section 5.