

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 [6], 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 [20]. Higher-level frameworks such as MapReduce [5] 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, we¹ 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 [9]).

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 `enums` or `option`

¹ 'We' refers 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 [10], 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 [27,30], 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 [24]. 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 as well as the general MPS language design philosophy is explained in [26]. We analyze MPS’ suitability for modular language composition based on experience with mbeddr in [28] (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 [29], 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 [31] lead to good editor productivity and user acceptance [2].

2 KernelF Overview

2.1 Language

In this section I point out the most important language features of KernelF. For all of them, [25] provides more details and code examples; for many of these features we also show examples part of the case studies in Section 5.

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

² <https://www.jetbrains.com/mps/>

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 be 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>]`. Members 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 [14]). 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 [25]. 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 by [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).

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:

- **A:** required because of higher-order functions (`where`, `map`) on collections
- **B:** path navigation usually also has 1:n paths, which requires collections

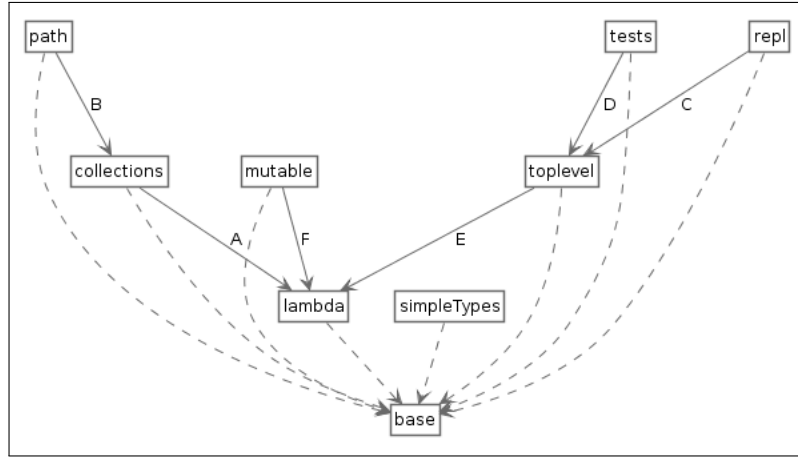


Fig. 1. Dependencies between the language modules in KernelF.

- **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, `boxes` 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 `Expressions` 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)`).

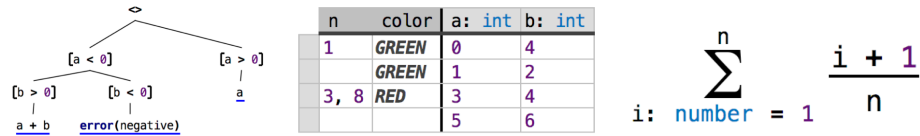


Fig. 2. `utils` extension: decision trees, multi-valued decision tables and math symbols.

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

Healthcare Voluntas’ 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 built 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 supports 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 their 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 : Kommulationswerte		
Ergebnistyp:	Laufvariable:	Parameter:
number{3}	x	i geschlecht q
$D_x := l_x * \frac{1}{(1+i)^x}$		

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

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 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 Convection³ 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://convection.io>

⁴ <https://languageengineering.io/thoughts-on-declarativeness-fc4cfd4f1832>

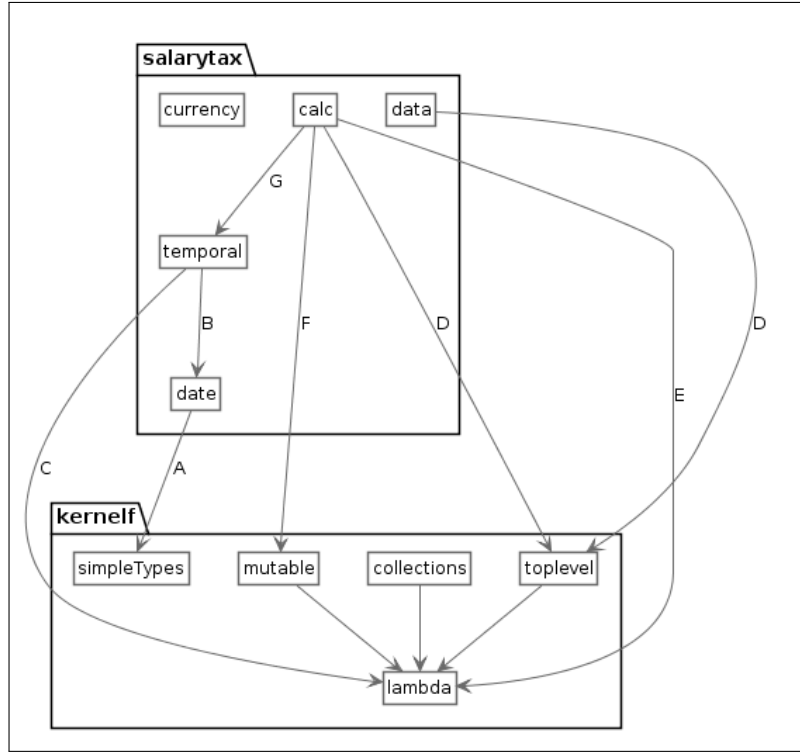


Fig. 4. Overview over the languages created in this system and their dependencies. The reasons for the dependencies are as follows: A: mixed arithmetics between dates and numbers; B: temporal slices use **date** values; C: temporal values support higher-order operations that contain lambdas; D: both 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 **kernel.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 for 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
left argument  ::= <date>
right argument <=: <int>
result type    { <date> }

overloaded operations for MinusExpression
left argument  ::= <date>
right argument ::= <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 [16], 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.

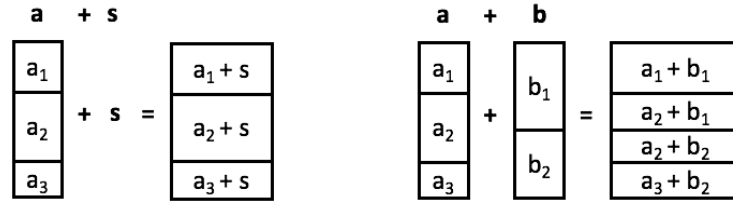


Fig. 5. Reslicing of temporal values; **a** and **b** are temporal values, **s** is a regular scalar. When a temporal value is “operated” with a scalar, the slices remain the same, but their values change. In the case of two temporal values, the slices intersect, and the values are computed per intersection.

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 **TemporalTypes**. 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 **ttv** 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 they 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 {
  employment -> Employment // basic data
  amount      : currency
}

result data [monthly] Tax {
  person -> Person // basic data
  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 month'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:

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

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:

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

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

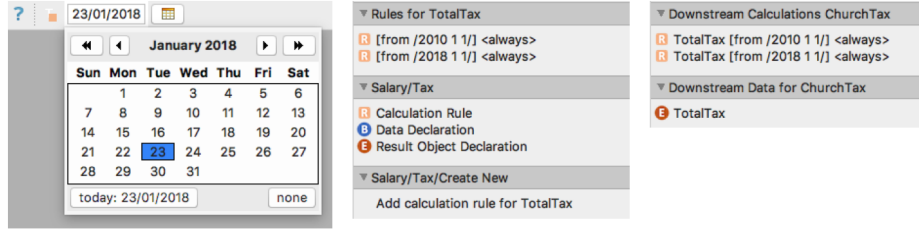


Fig. 6. IDE features that support the language: date chooser to adapt the code to show only those parts that are valid at that date; context buttons for the currently selected result data item and for the currently selected rule.

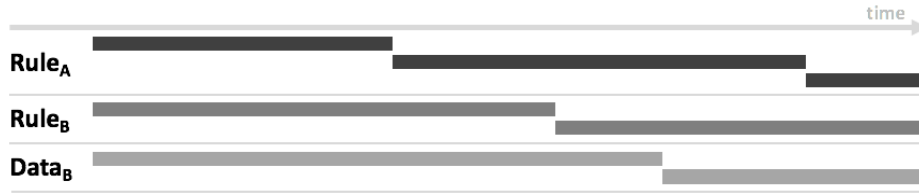


Fig. 7. Example of allowed (**Rule-A/Rule-B**) and non-allowed (**Rule-B/Data-B**) misalignment between validity periods.

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, it 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.

5.2 Smart Contracts

Blockchains [22] and smart contracts promise trusted, distributed execution of arbitrary programs. Ethereum [32] 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. Solidity⁵ is essentially a

⁵ <http://solidity.readthedocs.io/>

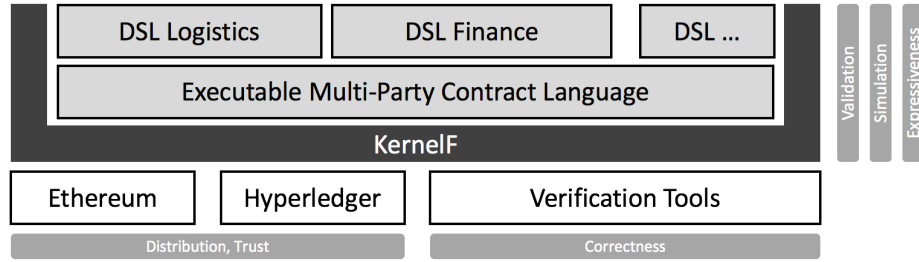


Fig. 8. A language architecture consisting of industry-specific DSLs, a set of common abstractions in EMPCL, the KernelF foundation, plus generation to blockchain-based execution infrastructures and verification tools for ensuring functional correctness.

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 contracts 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 [21], logistics [18] 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 [15,1], 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 [11] one.⁷

Figure 8 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 [25]). 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)

⁶ <https://www.artificiallawyer.com/2018/01/19/welcome-to-the-first-computational-law-blockchain-festival/>

⁷ <https://runtimeverification.com/blog/?p=496>

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 9 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 9.

```
val s = run(Unanimous)           // continued
s.vote(bernd)                   s.vote(markus)
s.vote(bernd)                   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. It has three subconcepts: `UnanimousDecProc`, `MajorityDecProc` and `CustomDecProc`. The first two are just keywords, whereas the last one looks as follows:

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

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 or 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.

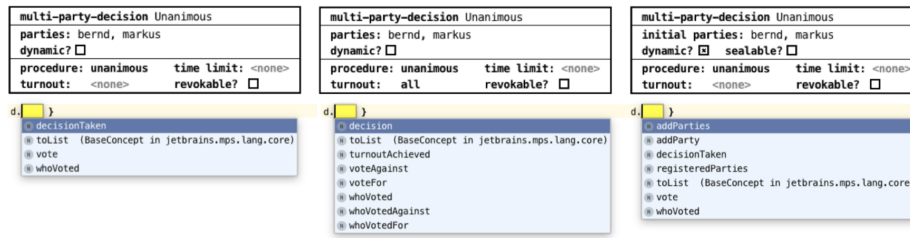


Fig. 9. A couple of different configurations of the `multi-party-decision` process and the resulting available entries in the code completion menu.

- Inside a transaction, a box read is redirected to a lookup in the map⁸ (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, 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)
val process = run(Unanimous)

fun voteAndCount(Party whoVotes)
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.

⁸ Note that this also works if multiple transactions run in a concurrent context; isolation is maintained because the boxes themselves are not updated.

```

[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)
                           @[dsfslfd0g98d09g8sdf]
                           @[09583503534]
                           @[lfd0g98d09g8sdf]
      decisionTaken -> false
[3] ☐ x.vote(markus) : void, effects[modifies]
      MultipartyBooleanDecisionValue (snapshot)
      whoVoted -> (collection|1)
                  @[09583503534]
      isSealed -> true
      registeredParties -> (collection|3)
                           @[dsfslfd0g98d09g8sdf]
                           @[09583503534]
                           @[lfd0g98d09g8sdf]
      decisionTaken -> false
[4] ☐ x.vote(klaus) : void, effects[modifies]
      MultipartyBooleanDecisionValue (snapshot)
      whoVoted -> (collection|2)
                  @[dsfslfd0g98d09g8sdf]
                  @[09583503534]
      isSealed -> true
      registeredParties -> (collection|3)
                           @[dsfslfd0g98d09g8sdf]
                           @[09583503534]
                           @[lfd0g98d09g8sdf]
      decisionTaken -> true

```

Fig. 10. 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	multi-party-decision AccessControl
initial parties: bernd, klaus	initial parties: bernd, klaus
dynamic? <input checked="" type="checkbox"/> sealable? <input type="checkbox"/>	dynamic? <input checked="" type="checkbox"/> sealable? <input type="checkbox"/>
procedure: unanimous time limit: <none>	procedure: majority time limit: 20000
turnout: <none> revokable? <input checked="" type="checkbox"/>	turnout: <none> revokable? <input type="checkbox"/>

Fig. 11. Two decision processes used in the complex example contract.

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 10 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 [4]). For example, commands can be 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 a simulator, it is easy to build “time travel” functionality [3] 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:⁹

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 11. 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

⁹ Another extensively documented example can be found in [23].

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:

```
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 [12] 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 [7] 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 [19]). 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 `skipped`.

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 12). 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.

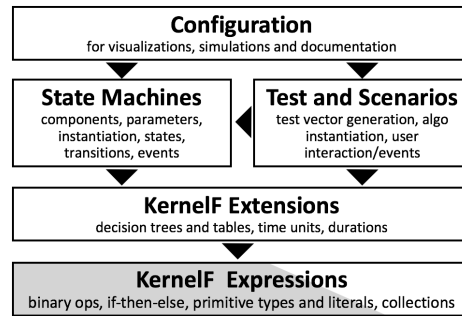


Fig. 12. The core of this system is a restricted version of KernelF. On top, we have developed a set of functional extensions that help medical professionals make non-trivial (multi-criteria) decisions. The core of the medical algorithm is expressed through state machines, and validation is performed through a testing and scenario description language. At the top is a language for configuring generated visualisations and reports.

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.

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 5.1¹⁰. Second, we define a `timeseries` 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.¹¹

```
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"
```

¹⁰ We are currently consolidating both into a common `datetime` extension.

¹¹ 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.

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 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 **AcquireBloodSugar** 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

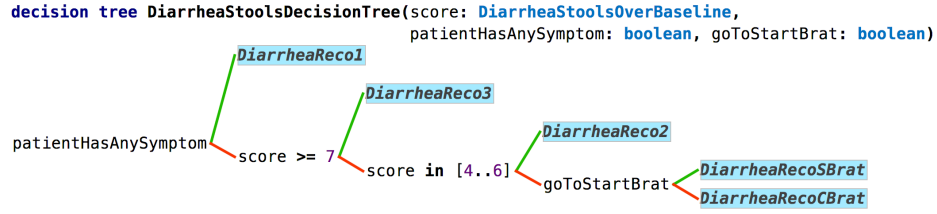


Fig. 13. A decision tree; the green/up edges represent **yes** answers to the preceding node, the red/down edges represent **no**.

```

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

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 13 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 13) or 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 14 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 16 shows an example. The first of the tests in Figure 16 tests

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. 15. 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. 16. 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.

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 15.

Scenario tests (Figure 17) 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,¹² which is, for example, also supported by Cucumber.¹³ To express the passage of time and occurrences at specific times, 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.

¹² <https://martinfowler.com/bliki/GivenWhenThen.html>

¹³ <https://cucumber.io/>

```

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. 17. 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.

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 18, 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 18, 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. 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 19). 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 outsiders.

The reason why these configurations are represented as models (expressed in their own DSL) as opposed to just configuring a particular visualisation through

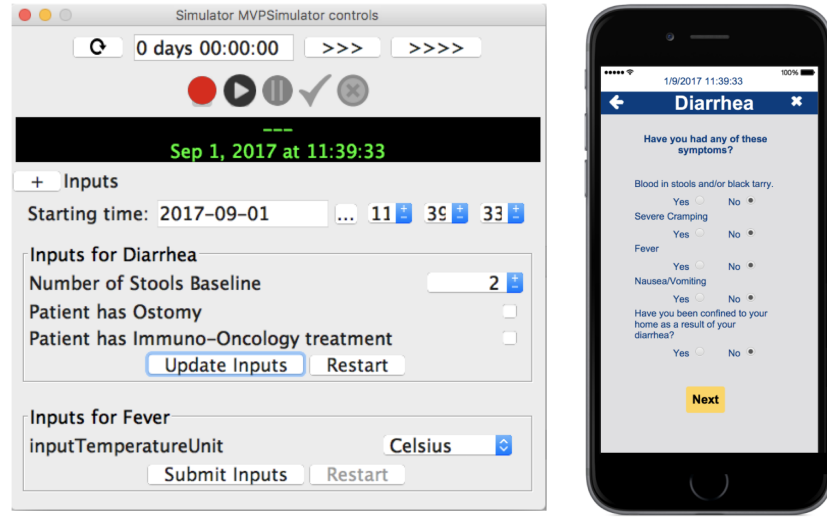


Fig. 18. Control panel to configure and execute simulations.

```

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

```

Fig. 19. Configuration for the generation of medical protocol flow charts.

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].

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.

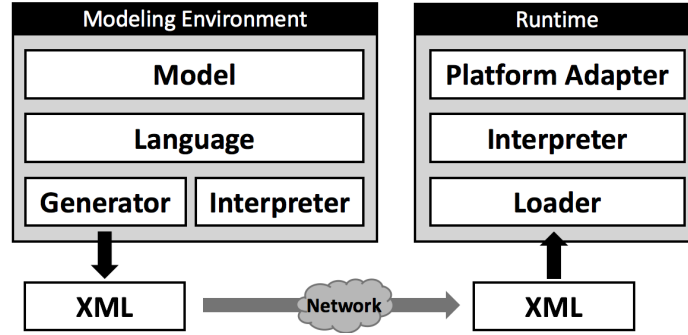


Fig. 20. Execution architecture of the languages: an IDE-interpreter plus an interpreter on the phone implemented that works on an XML representation of the algorithms.

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 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 [17] (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 [13] 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

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

Fig. 21. Overview of the extensions to functional abstractions, higher-level behavior, structures and IDE extensions for the three case studies.

- Architectural safety mechanisms such as runtime watchdogs [13] based on independently specified invariants.

~ ~ ~

This concludes our case studies. Figure 21 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 introduced in [26]. 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 [28]. 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 [28].

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]`. Clearly, 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 `NumberTypes` 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 `casts` 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 [26]):

```
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 `thens`. However, here the situation 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`):

<pre>// test data for John val j_last = "Doe" val j_first = "John" val j_birthYear = 1974</pre>	<pre>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</pre>
<pre>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" }</pre>	

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 interpreter) 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 Excel¹⁴: you can imagine the `vals` as cells with user-entered values, the `asserts` 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 22). 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

¹⁴ An analogy that many of our users like to draw in more ways than is good for us!

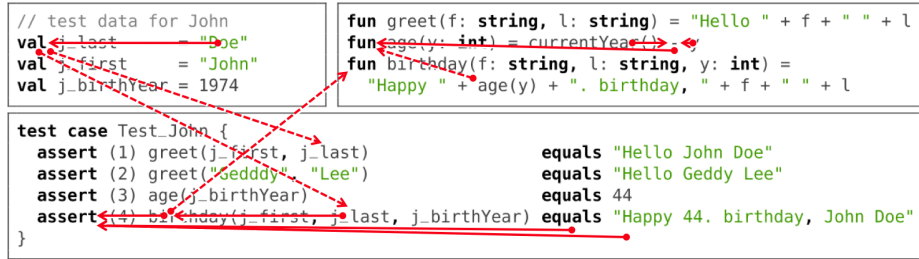


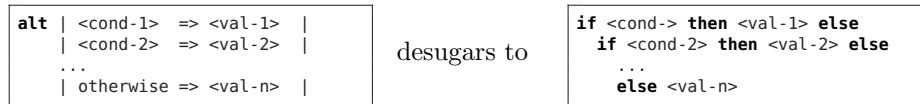
Fig. 22. The example code for reactive interpreters shown with the reverse dependencies relevant for a change to the value "Doe". Solid lines represent containment, dashed lines represent reference dependencies.

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.

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:



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.¹⁵ 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 [30] 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.

¹⁵ 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.

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 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 [8] is a functional language that ships with Xtext¹⁶. Similar to KernelF, its purpose is to be extended and embedded in the context of DSLs. Xtend¹⁷ is a full programming language (with classes, modules and effects) that embeds Xbase expressions. Similar to Kotlin¹⁸ and Ceylon¹⁹, 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.

¹⁶ <https://www.eclipse.org/Xtext/>

¹⁷ <http://www.eclipse.org/xtend/>

¹⁸ <https://kotlinlang.org/>

¹⁹ <https://ceylon-lang.org/>

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. Racket²⁰ 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 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 in 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

Lua²¹ 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).

²⁰ <https://racket-lang.org/>

²¹ <https://www.lua.org/>

7.5 Other Language Workbenches

This paper is not about evaluating MPS' suitability as a language workbench; see [28] 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 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 [28], 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 Convector. Once it is expressive enough, we will implement KernelF in Convector so we have it available as a base language for Convector-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. S. Amani, M. Bégel, M. Bortin, and M. Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. *CPP. ACM. To appear*, 2018.
2. 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.
3. 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.
4. 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.
5. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
6. F. DeRemer and H. H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, (2):80–86, 1976.
7. J. R. Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.
8. S. Eftinge, 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.
9. 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.
10. M. Fowler. Language workbenches: The killer-app for domain specific languages. 2005.
11. 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.
12. R. Gibbons. *A primer in game theory*. Harvester Wheatsheaf, 1992.
13. R. Hanmer. *Patterns for fault tolerant software*. John Wiley & Sons, 2013.
14. R. Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, page 1. ACM, 2008.
15. 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.
16. C. S. Jensen and R. T. Snodgrass. Temporal data management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, 1999.
17. 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.
18. 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.
19. P. Narasimban, L. E. Moser, and P. M. Melliar-Smith. Using interceptors to enhance corba. *Computer*, 32(7):62–68, 1999.

20. 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.
21. 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.
22. D. Tapscott and A. Tapscott. *Blockchain revolution: how the technology behind bitcoin is changing money, business, and the world*. Penguin, 2016.
23. M. Voelter. A smart contract development stack. [Online; posted 6 Dec, 2017].
24. M. Voelter. Language and ide development, modularization and composition with MPS. In *GTTSE 2011*, LNCS. Springer, 2011.
25. M. Voelter. The kernelf reference. 2018.
26. M. Voelter. Language development with mps - a quick overview. 2018.
27. 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.
28. 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.
29. M. Voelter and S. Lisson. Supporting Diverse Notations in MPS’ Projectional Editor. *GEMOC Workshop*.
30. 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.
31. 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.
32. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.