EFFECTS AS IMPLICIT CAPABILITIES

Martin Odersky

SNF Proposal, Scientific Part
September 2015

1 SUMMARY

The purpose of running a computer program is to have some effect on the outside world – paint pixels on a screen, enter a record in a database, control a robot, to give some examples. We also talk of an effect if a procedure interacts with its environment in a way different from just taking some arguments and producing a result. Examples of such internal effects are updating or reading a mutable store, or throwing an exception. In the presence of effects, order of evaluation matters; in fact that's how Sabry defines effectfulness [Sab98]. Consequently, in order to understand a program that makes use of effects one must take its execution history into account.

It seems quite natural that one should track effects by means of a static typing discipline, similarly to what is done for arguments and results of functions. After all, the possible effects of a procedure are just as important as the types of its arguments and results in order to understand the procedure's contract and how it can be composed. However, despite 30 years of research on the subject, very little of it has been applied in practice. A central difficulty is how to express *effect polymorphism*, the ability to write a function once, and to have it interact with arguments that can have arbitrary effects.

We propose to investigate a new approach to effect checking that is fundamentally different from previous research. In particular, it has an elegant solution to the effect polymoprhism problem, and therefore promises to be quite usable in practice.

The central idea is that instead of talking about effects we talk about capabilities. For instance, instead of saying a function "throws an IOException" we say that the function "needs the capability to throw an IOException". Capabilities are modeled as values of some capability type. For instance, the aforementioned capability could be modeled as a value of type `CanThrow[IOException]`. A function that might throw an `IOException` needs to have access to an instance of this type. Typically it takes an argument of the type as a parameter.

It turns out that that the treatment of effects as capabilities gives a simple and natural solution to the problem of effect polymorphism. But there are two areas where work is needed to make capabilities as effects sound and practical.

First, when implemented naively, capabilities as parameters are even more verbose than effect declarations such as throws clauses. Not only do they have to be declared, but they also have to be propagated as additional

arguments at each call site. We propose to make use of the concept of implicit parameters to cut down on the boilerplate. Implicit parameters make call-site annotations unnecessary, but they still have to be declared just like normal parameters. To avoid repetition, we propose to investigate a way of abstracting implicit parameters into implicit function types.

Second, there is one fundamental difference between the usual notions of capabilities and effects: Capabilities can be captured in closures. This means that a capability present at closure construction time can be preserved and accessed when the closure is applied. Effects on the other hand, are temporal: it generally does make a difference whether an effect occurs when a closure is constructed or when it is used. We propose to address this discrepancy by introducing a "pure function" type, instances of which are not allowed to close over effect capabilities.

The proposed work will explore the idea of effects as capabilities in detail. We will work on minimal formalizations for implicit parameters and pure functions and study encodings of higher-level language constructs into these calculi. Based on the theoretical modelization we will develop a specification and implementation for adding effects to Scala. A high-quality implementation in Scala will require considerable effort, but will give an excellent means to validate the ideas and promote them if they are deemed successful.

As most languages, Scala currently does not track effects. So adding an effect system to the language raises the problem of migration. How can we add effects gradually to an existing codebase and library ecosystem? How can we extend these techniques to a multi-language environment? The project will devise new strategies for migration which make use of the "effects as capabilities" principle.

2 RESEARCH PLAN

2.1 State of the art

2.1.1 *Effects*

Computational *side-effects*, or simply *effects* describe observable behaviors of computations, e.g. whether the computation performs an IO operation, modifies mutable state or throws an exception. Typically, a "pure" computation, i.e. one with no effects, is easier to reason about. Conversely, an effectful computation (being hard to reason about) might exhibit unexpected behaviors compromising the *safety* or *performance* of a program. For example, an unchecked exception could lead to unexpected program termination, and the presence of mutable, shared state might compromise safety in a concurrent application by introducing race-conditions. Similarly, hidden data dependencies due to mutable state can obstruct useful optimizations such as loop fusion or parallel scheduling which may re-order computations.

It is therefore useful, for programmers and compilers alike, to identify and reason about effects. *Effect systems* track the effects of computations, statically, much in the same way that type systems track the type of computations. Effect systems were originally introduced by Gifford and Lucassen [GL86, LG88] to track memory effects but have been extended over the years to provide type-and-effect inference [TJ92b, TJ92a] and to cover a wide variety of effects, such as exceptions [GJS+13], purity [Pea11],

atomicity [ABHI08] and even access to widgets in graphical user interfaces [GDEG13]. Various attempts have also been made to modularize and generalize effect systems [WT03, MM09, Fil10].

The static effect discipline with the most widespread use is no doubt Java's system of checked exceptions. Ominously, they are now widely regarded as a mistake [Whi15]. One frequent criticism is about the notational burden they impose. Throws clauses have to be laboriously threaded through all call chains. All too often, programmers make the burden go away by catching and ignoring all exceptions that they think cannot occur in practice. In effect, this disables both static and dynamic checking, so the end result is less safe than if one started with unchecked exceptions only. Another common problem of Java's exceptions is lack of polymorphism: Often we would like to express that a function throws the same exceptions as the (statically unknown) functions it invokes. Effect polymorphism can be expressed in Java only at the cost of very heavy notation, so it is usually avoided.

The Koka programming language [Lei14] features an effect system that can express effect-polymorphism and also functions like exception handlers that mask effects. Combining effects of multiple domains is straightforward; effects are represented as simple labels, and each function has a set of effects, which can be either annotated by the programmer or inferred. The system does not feature subeffecting, i.e., ordering between effect labels.

A somewhat different approach to tracking effects is prevalent in purely functional languages such as Haskell: in *monadic-style* programming, effects are directly represented by corresponding monadic datatypes and thus tracked by the type system, without the need for a separate effect system [PJW93]. The use of monads to model effects goes back to Moggi's seminal work on the semantics of effectful computations [Mog91]. While monadic programming and effect systems are related [WT03, Fil10], the former being at least as expressive as the latter, monadic programming differs from other type-and-effect systems in that it mixes the interface of effects (i.e. their syntax) with their implementation (i.e. their semantics) and exposes at least some of the latter to the programmer [KLO13]. Consequently, monadic programming is not a perfect fit with the built-in effects of most imperative languages and impure functional languages such as Scala or ML. The fact that monadic programming exposes low-level implementation details of effect handling to the programmer also makes it rather heavy-weight to use: a common example in Haskell is that computations with multiple effects require manual composition of the corresponding monads through monad transformers [LHJ95]. The composition introduces an ordering on the effects in question which further complicates their usage.

A third issue is that introducing side effects into an existing function requires refactoring that function to monadic style, and also other code that uses it has to be adapted. As described in Chapter 1 of the thesis of Lippmeier [Lip10], the fact that monadic and pure code have incompatible types leads to code duplication: for example the function *map* in Haskell can only apply a pure function over the elements of a list. To apply an effectful function, the language provides a second implementation *mapM*.

To alleviate these issues, there has been a recent uptake of *algebraic effects* [PP09, BP15] in the purely functional programming community. Algebraic effects can express common monadic effects such as mutable state, IO and exceptions (but not continuations). The definition of new algebraic effects is lightweight and unlike monads, combining effects is straightforward, which

encourages fine-grained effect definitions. Algebraic effects can be implemented in suitable functional languages [Bra13, KLO13, KI15, Bja14] using monads to implement effect handlers: the resulting systems provide flexible abstractions that do not compete with monads but build on them.

2.1.2 *Implicits*

Implicit mechanisms allow compilers to complete certain parts of the source code automatically, reducing boilerplate and improving automation. Implicit mechanisms have seen important use in generic programming [MS89, GJL+03]: although data structures and algorithms share the same definition across a wide range of data types, some of the information, such as how to compare two generic values, is only available at the last moment, when the data structure or algorithm is instantiated for a certain type – ideally, the compiler should implicitly fill in the information when necessary.

This problem has seen several solutions: type classes have been introduced in Haskell by *Wadler* in [WB89] as early as 1989 and have proven a very versatile mechanism. Similarly, ML module systems [MTHM97] have been shown to be equally expressive as type classes [WC08]. More recently, the work on concepts allowed more precise generic definitions in C++ [DRS06, GJS+06] and G [SL11]. However, these approaches are second-class citizens in their respective languages, using separate syntax and not allowing the same abstraction mechanisms as the other types in the language. This is a desirable property, as shown by *Lëmmar* in [LJ05].

Scala implicits [Ode14] are first-class language citizens, allowing the compiler to complete any argument based on the expected type and the current scope. This allows Scala implicits to model type classes [Ode06, OMO10]. The "Implicit Calculus" [OSC+12, OSC+14] provides a formalization close to the intent of Scala implicits, with two important differences: first, unlike Scala, the formalization describes a resolution mechanism that has to be triggered explicitly using the ? operator, so implicits have to be queried and then applied explicitly. Second, the formalization sidesteps subtyping and the challenges with respect to ambiguity and divergent searches that arise from it.

2.2 Work by the proposer

The proposer is the principal designer of the Scala programming language. Scala occupies a fairly unique spot in the spectrum of programming language designs in that it represents a fusion functional and object-oriented programming. With an estimated 400'000 developers and ongoing rapid increase in industrial adoption, it is one of the most widely used languages to have come out of academia. Companies with large Scala investments (typically, hundreds of developers) include IBM, Twitter, Apple, Amazon, Verizon, Goldman Sachs, Morgan Stanley, Workday, Zalando, Dwango, Unicredit. Scala is also the implementation language of Spark, which is well on its way to become the standard platform for data science.

2.2.1 *Implicit Parameters*

Particularly relevant to this proposal is Scala's implicit parameter design, described in [Ode06], [OMO10]. In a nutshell, a parameter section may be marked implicit, like this

```
def min[T](x: T, y: T)(implicit ord: Ordering[T]): Boolean
```

The `min` function computes the minimum of its arguments `x` and `y`, given an ordering `ord` over the domain type `T`. The `ord` parameter is passed implicitly, depending on the actual argument types. So a call like `min(1, 2)` would be completed by the Scala compiler to `min[Int](1, 2)(Ordering.Int)` where `Ordering.Int` is a predefined value that implements comparisons on integers.

Used in this way, implicits resemble Haskell's type classes. But since implicit values can also be defined locally they have other uses as well. For instance, implicit parameters are an excellent way to model contexts or capabilities.

One shortcoming of implicit parameters as they are currently implemented is that they cannot be abstracted: Every function that takes an implicit parameter has to have it declared explicitly; one cannot leave out the implicit parameter section. But the following idea, which we want to work out in the proposed research, can cut down on the boilerplate: We introduce an implicit function type, such as

```
implicit Context => Boolean
```

together with an automatic conversion rule: If an expression `e` has expected type[1] `implicit T => U` then `e` is rewritten to the anonymous function `(implicit x: T) => e`. This simple scheme allows to abstract implicit parameters in a type. For instance, instead of a function like

```
def f(x: T)(implicit ctx: Context): T
```

one can simply write

```
def f(x: T): Contextual[T]
```

using the abstraction

```
type Contextual[T] = implicit Context => T
```

The savings in boilerplate get the more pronounced the more implicit parameters are added, because all implicit parameters can be subsumed under a single type definition.

2.2.2 *Effects*

Previous work by the proposer and his group on effect systems is reported in the PhD thesis of Lukas Rytz [Ryt14], and the papers [ROH12], [RAO13]. Major contributions of this work are a modular way to compose effects through subtyping, and some improvements in dealing with effect polymorphism.

The fundamental problem of effect polymorphism can be illustrated by considering the `map` function. Without effects, a reasonable signature for `map` would be:

```
def map[T, U](xs: Seq[T], f: T => U): Seq[U]
```

[1] The notion of "expected type" is well defined in the local type inference system [OZZ01] employed in Scala.

That is, `map` applies a function of type `T => U` to elements of a sequence of type `Seq[T]`, yielding a sequence of type `Seq[U]`. `T` and `U` are type parameters that can be instantiated to arbitrary types. Adding effects, we note that `map` would have any effect that the given function `f` has. A way to express this in a classical effect system is by introducing an additional type variable `E`, as in:

```
def map[T, U, E](xs: Seq[T], f: T => U @eff E): Seq[U] @eff E
```

However, this does not scale. In actual programs, the amount of effect variables that need to get added grows quickly out of hand. The problem gets exacerbated if instead of single closures arguments are objects with side effecting methods. Each object can contain many side effecting methods, and each effect of each method that might be called from a higher-order function has to be recorded in the effect signature of the function. In the end, the necessary type annotations can grow proportionally to the size of the call graph. Our work improved the problem with effect polymorphism somewhat by introducing shorthand syntax for common cases [ROH12] and by allowing dependent effects. E.g., `map` in our system could be written like this:

```
def map[T, U](xs: Seq[T], f: T -> U): Seq[U] @eff(f)
```

This indicates that `map` has the same effects as `f` and obviates the need for an additional effect type parameter. The system was implemented as a plugin to the Scala compiler. However, there was no large uptake by Scala users, presumably because the notational savings were not large enough. Effect types were still too cumbersome to be proposed as a standard feature of Scala.

In the proposed work we want to study the ramifications of the idea to treat effects as capabilities, which are in turn represented as implicit parameters. For instance, instead of writing a function with a throws clause like this:

```
def readLine(f: File): String @throws IOException
```

we reformulate `readLine` to say that it needs the capability to throw an exception, like this:

```
def readLine(f: File)(implicit c: CanThrow[IOException]):
    String
```

(To keep the presentation simple, we gloss over the fact that `readLine` has other effects besides throwing an exception). Using the implicit functions presented above, we can introduce the abbreviation

```
type throws[T, E] = implicit CanThrow[E] => T
```

which lets us abbreviate the signature of readLine to

```
def readLine(f: File): throws[String, IOException]
```

or, since binary type constructors can be written inline in Scala:

```
def readLine(f: File): String throws IOException
```

So we are notationally back to throws clauses, even though the underlying meaning is one of implicit capability parameters. This makes a crucial difference for effect polymorphism. In fact, the "effects as capabilities" paradigm often lets us side-step the problem of expressing effect polymorphism altogether. For instance, the map function would still have the simple effect-less signature given above:

```
def map[T, U](xs: Seq[T], f: T => U): Seq[U]
```

Even though `map` does not mention effects at all in its definition, the application `map(files, readLine)` would correctly be diagnosed to throw an `IOException`. After all, `readLine` needs a capability of type `CanThrow[IOException]`, and since `map` does not pass one along to its argument this capability must come from the context. So the required capabilities of the whole expression `map(files, readLine)` include the capabilities required by `readLine`.

As long as functions just propagate effects from the arguments they invoke they do not need effect annotations at all. I believe this will drastically cut down on the notational overhead of effects, and could be the necessary ingredient to making effect systems practical in future programming languages.

However, the whole story is not so simple. Something is clearly amiss in our treatment of effects as capabilities. The problem becomes apparent if we want to express a *parallel map* `pmap`. Unlike `map`, `pmap` should work only with functions that are pure, i.e. that do not have any side effects. However, what should the signature of `pmap` be? If it is the same as the one of `map` then

```
pmap(files, readLine)
```

would be a legal expression, even though `pmap` is not prepared to deal with side-effecting functions like `readLine`! The problem here is that the `readLine` argument captures the implicit capability `CanThrow`. That is, after implicit argument expansion, the call `pmap(files, readLine)` would be rewritten to `pmap(files, readLine(c))` where the capability `c` to throw an exception is taken from the environment of the `pmap` call. This scenario illustrates a fundamental difference between capabilities and effects. Effects happen at specific moments in time, but capabilities may travel in time by being captured in a closure. The proposed solution which we would like to work out in detail is to introduce a pure function type, written `A -> B`. Instances of a pure function type are not allowed to close over effect capabilities. We already studied function types with capturing restrictions in our work on Spores [MHO14], but the proposed variant is somewhat different, in that it singles out a particular class of values that may not be captured over. Using a pure function type, `pmap` can now be declared as follows:

```
def pmap[T, U](xs: Seq[T], f: T -> U): Seq[U]
```

What if we were to require a pure function, except that some particular effect should still be allowed? A realistic example would be a version of `pmap` that was prepared to handle and propagate exceptions, but would not allow any other side effects. This can be declared as follows:

```
def pmap[T, U, E]
      (xs: Seq[T], f: T -> implicit CanThrow[E] -> U)
      (implicit t: CanThrow[E]): Seq[U]
```

Or, using the abbreviations introduced above:

```
def pmap[T, U, E](xs: Seq[T], f: T -> U throws E): Seq[U]
    throws E
```

This looks like a return to the classical notion to effect checking with effect type variables, but note that explicit effect annotations are only needed when particular effect bounds are required. In a well-design program, the vast majority of functions would be like `map` in that they simply propagate argument effects up the call chain. Such functions can be declared without regards to effects altogether.

Here is an application of the new version of `pmap`.

```
pmap(xs, x => x * x)
```

By the proposed rules of implicit expansion, this would expand to

```
pmap(xs, x => implicit c: CanThrow[E] => x * x)
```

which matches the required argument type of `pmap`. The argument could of course also make use of the `CanThrow` capability provided by `pmap`. But it could not use any other capabilities.

2.2.3 *Interoperability and Migration*

Scala as it is now is not effect-safe, even though Scala programming conventions favor purely functional code and try to avoid effects where possible. Scala is also tightly integrated with its host platform (supporting Java or JavaScript), so even if Scala programs were effect-safe the platform on which they run would not be.

This poses the dual problems of interoperability and migration. How can effect-safe and effect-unsafe parts of a program be combined? How can effect declarations be gradually added to an initially effect-unsafe program? It is here where the "effects as capabilities" principle shines because it offers a simple escape hatch: To start, include a capability for every effect in the `Predef` module, which is automatically imported in every Scala file. This lets programs type check as before, at the price of not giving any guarantees about effect-safety. Then, on a module per module basis, use Scala's way of hiding selective Predef imports to replace the automatic global capability by capabilities with tighter scope. Once general adoption of effect annotations reaches some threshold, the default could then be switched to require an explicit import of the global "escape-hatch" capability into code where it is required.

Call a module that does not make use of the global effect capability *effect-safe*. The safety guarantee provided by the proposed interoperability and migration scheme is that effects are accounted for at all program points `P` where all modules on the call chain from `P` to an effect-producing construct are effect-safe.

In summary, the proposed scheme has the following advantages for migration:

- Purely functional sequential code carries over as is.
- Code that requires absence of some effects needs to be changed; often this is as simple as requiring pure functions `A -> B` instead of side-effecting functions `A => B`.

- Code that produces effects needs a capability. Initially that capability can be provided by a global automatic import.

This means that Scala programs as they are today can continue to compile. Furthermore, the burden of making programs effect-safe is least for programs that are predominantly functional. One goal of the proposed research is to find out whether these advantages are enough to make effect annotations an attractive proposition for mainstream programming.

2.3 Work plan

The aim of the project is to flesh out and validate the ideas developed in the last section. The elaboration will consist of both theoretical and practical parts. We will develop further the theory of implicits and effects as implicit capabilities by studying core calculi that embody these concepts and by developing mechanized proofs of their meta-theory. On the practical side, we will develop implementations as extensions of the Scala language, and will migrate core libraries and applications to the extended language. Validation will consist in, first, gaining experience ourselves, and, second, making these constructs available to the large audience of Scala programmers and studying their uptake. We plan to get to a state where successful concepts can be specified and added to the Scala standard.

We propose to divide this project into five work packages. Work packages A and B focus on the development of the theoretical foundations of implicits and a capability-based effect system for Scala. Packages C and D focus on the practical challenges related to the implementation of effect checking in the Scala compiler and to the migration of existing, pre-effects Scala code to the new effect system. We mark each of the following subsections with an estimate of the time required to complete the subproject in person-years (py).

2.3.1 *Package A – Implicit parameters (2py)*

TASK A1: FOUNDATIONS FOR SCALA IMPLICITS (1 PY). The first part of this package will consist in the development of a suitable formal system capturing the most important aspects of Scala's current implicit mechanism. The formalization will likely build on Oliveira et al.'s *implicit calculus* [OSC+12], adapting it to Scala's setting where necessary. The main focus will be on those features of Scala's implicit system that are not covered by the existing implicit calculus, or covered insufficiently for our purposes. These include, most importantly, support for subtyping and proper handling of divergent and ambiguous implicits. We plan to support our formalization with machine-verifiable proofs written using a proof assistant such as Coq [BCHPM04] or Agda [Nor07].

TASK A2: IMPROVED IMPLICITS (0.5 PY). The next phase of package A will consist in the further development of Scala's implicit system. Scala's implicit system currently suffers from a number of limitations such as the lack of implicit function types (i.e. types of functions taking implicit arguments), rigidity in the positioning of implicit arguments (always in the last argument list), or the absence of explicit call syntax. The goal of this task will be to eliminate these shortcomings while ensuring the consistency and usability of the language. The formal system developed in task A1 will be used to support and guide the developments in this task.

TASK A3: IMPLICIT PARAMETER ERASURE (0.5 PY). The final task in this package concerns the elimination of "dead-code" implicit parameters. In Scala, implicit parameters are often used in type-level programming where they serve as (implicit) evidence for the existence of a given type. A prominent example of such type-level programming are *type classes* [OMO10]. Implicit evidence parameters may play a computational role, e.g. in the case of type classes where they serve as type class instances. However, sometimes the sole role of implicit evidence is to guide type checking and type inference, as is typically the case when one uses types such as `<:<` or `=:=` to unify type parameters. In the latter case, the implicit evidence parameter is discarded by the callee, but the compiler may nevertheless produce code for its allocation and dispatch. This introduces unnecessary computational overhead (dead code), possibly degrading performance. This problem will be further exacerbated by the introduction of implicit-based effect capabilities (cf. package B) which, by design, serve no computational role. Thus A3 consists in developing methods for tracking and eliminating such dead-code implicits in the Scala compiler.

2.3.2 *Package B – foundations of capability-based effect checking (2py)*

TASK B1: FOUNDATIONS OF CAPABILITIES (1 PY). The first part of this package will consist in the development of a formal, *capability-based* type-and-effect system. The effect systems we consider are arbitrary, as long as they satisfy the following two conditions. (1) The order of effects does not matter, for instance, we just record that a function may throw an exception and may change a store location, not that it has the two effects in a certain order. In other words, we only consider flow-insensitive effect systems. (2) Unless explicitly masked, effects propagate up the call graph. For instance if function `f` may call function `g` and `g` may throw exception `E`, then `f` may also throw `E`. The only exception to this rule is if there is an intervening effect masking construct. For instance a `try` expression that catches `E` can stop its propagation up the call graph. To summarize, we consider any set of properties that propagates up the call graph as a potential domain for static effect checking.

The formalization will likely be an extension of either the Girard-Reynolds polymorphic lambda calculus (System-F) [Rey74] or the Dependent Object Types calculus (DOT) [AMO12, ARO14] with two additional type-level constructs for *i)* effect types (capabilities) and *ii)* pure vs. impure function types. The main challenge will consist in developing the meta-theory of the calculus, i.e. proof techniques for establishing *type-soundness* w.r.t the formal semantics of the calculus. To this end, we plan to adapt the standard approach introduced by Wright and Felleisen [WF94, Pie02] or, if necessary, alternative techniques better suited to the underlying calculus [ARO14]. Note that, while existing effect systems based for System-F (and related calculi) [WT03, MM09, Fil10] or Scala [Ryt14] might serve as an inspiration for this task, their results are not directly applicable. Our capability-based approach is, in some sense, dual to the approach taken by these other effect systems: whereas they track the effects *resulting* from a computation (effects as output), we track the capabilities *required* by a computation (capabilities as input). As for task A1, we intend to support our formalization with machine-verifiable proofs.

TASK B2: CAPABILITIES AND SUBTYPING (0.5 PY). The next step in this work package will consist in the extension of the formal system obtained in task B1 with subtyping. Subtyping is an integral yet complex aspect of Scala's type system, so care must be taken to design the effect system so as to interact soundly with subtyping. In particular, the combination of multiple effects/capabilities gives rise to a lattice structure reminiscent of the subtyping lattice. Since capabilities are themselves types, it seems natural to combine the two lattices into one. The details of this merger will impact the soundness of the resulting system, as well as its ease of use. Of particular concern are the role of subsumption in strengthening resp. weakening capabilities, and that of upper resp. lower type bounds in controlling what capabilities are available in a given method. Due to the complexity of this interaction, it will be imperative to develop a machine-checked soundness proof for the resulting system.

TASK B3: INTEGRATION WITH IMPLICITS (0.5 PY). The last phase of this work package will consist in merging the effect system developed throughout tasks B1 and B2 with the improved implicit system developed in package A. The result will be a high-level type system where capabilities become implicit parameters, thus alleviating the notational overhead of the effect system and providing effect inference through implicit resolution. This will require the development of *i)* a suitable surface syntax for capabilities (as implicits) and for pure/impure functions, *ii)* a corresponding encoding into the underlying implicits, type and effect system, and *iii)* a soundness proof of the resulting system.

2.3.3 *Package C – implementation of capability-based effect checking (2.5py)*

TASK C1: IMPLEMENTATIONS OF IMPROVED IMPLICITS AND CAPABILITIES (1 PY). The first part of this package consists in extending the Scala compiler with support for the improved implicit system developed in task A2, and the capability-based effect checking on top of those implicits, as developed in task B3. In particular, this will require support for implicit function types and pure function types and their consequences on type checking and type inference. Core language constructs such as **`throw`** will also need adaptations to require appropriate implicit capabilities in scope, and core/common capabilities should be added to the standard library.

TASK C2: PROGRAMMING METHODS FOR CAPABILITY-BASED EFFECT CHECKING (0.5 PY). Using the implementation of task C1 as a vehicle for experimentation, the second part of this package will consist in developing programming methods for practical effect checking using capabilities. Above all, idioms should be developed to keep the boilerplate down, so that the burden of effect annotations does not outweigh the benefits of effect tracking. In particular, the type-and-effect lattice developed in task B2 can be combined with intersection types to construct capabilities for multiple effects in a single type. Type aliases can then be used to abstract over sets of require capabilities. A second area of interest lies in effect shadowing, which can be modeled with the scoping rules of implicits. The details of these idioms will impact the practical applicability of capability-based effect checking.

TASK C3: INDIVIDUAL EFFECT DOMAINS (1PY). The third part of this package will consist in defining and refining individual effect domains to be represented. These include at least (checked) exceptions, mutation and I/O. We will use standard subtyping to more precisely define exceptions, and will investigate region typing for describing mutation more precisely. A topic of particular relevance is effect masking: Depending on the effect domain there can be ways to stop propagation of an effect up the call chain. For exceptions, these are `try` blocks, whereas for mutations we will investigate methods to limit mutations to local regions.

2.3.4 *Package D – Interoperability and Migration and Practical Validation (2.5py)*

TASK D1: INTEROPERABILITY (0.5PY). The first part of this work package consists in studying how effect-safe and effect-unsafe parts of a program can work together. The basic mechanism of global effect capabilities gets us off the ground here, but will need to be augmented by other techniques, such as providing effect annotations for existing code in Scala and its host language.

TASK D2: MIGRATION (1PY). The second part of this package will consist in migrating common libraries to be effect-safe. These will include Scala's standard library and selected community libraries. We will recruit open source community contributors to participate in this effort.

TASK D3: VALIDATION (1PY). The third part of this package will refactor common Scala applications to be effect-safe. These will include the Scala compiler, and selected open source applications such as potentially parts of the Play web framework and the Spark data analytics platform.

2.3.5 *Timeplan*

We plan to organize the work for two Ph.D. students and one post-doc. One of the two students will be focused on foundations, the other on implementation. The post-doc will collaborate with the students in both fields.

Here is a tentative time allocation for the planned work over the six semesters following April 2016:

Semester	Student A	Student B	Postdoc
April 2016 - Sept. 2016	Task A1	Task C1	Task A1
Oct. 2016 - March 2017	Task A2	Task C2	Task C1
April 2017 - Sept. 2017	Task A3	Task C3	Task C3
October 2017 - March 2018	Task B1	Task D1	Task B1
April 2018 - Sept. 2018	Task B2	Task D2	Task D2
October 2018 - March 2019	Task B3	Task D3	Task D3

2.4 Significance

If successful, this research will make Scala into a pure functional programming language, in the sense that side-effecting expressions and purely functional expressions are distinguished by means of their type. This will represent a significant advance of the state of the art and remove the embarrassement that despite obvious benefits and much research, static effect tracking is still very uncommon in practice. The proposed extensions are

fairly lightweight: essentially, we need implicit parameters and a pure function type. It is therefore likely that the techniques developed in this project can be transferred to other programming language designs. Over time, this could fundamentally change our expectations of what properties of programs should be statically tracked.

REFERENCES

[ABHI08] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 63–74, 2008.

[AMO12] Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*, FOOL '12, 2012.

[ARO14] Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 233–249, 2014.

[BCHPM04] Yves Bertot, Pierre Castéran, Gérard Huet, and Christine Paulin-Mohring. *Interactive theorem proving and program development – Coq'Art: the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004.

[Bja14] Runar Bjarnason. Compositional application architecture with reasonably priced monads. functionaltalks.org Blog, November 2014.

[BP15] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015.

[Bra13] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 133–144, 2013.

[DRS06] Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ Concepts. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 295–308, 2006.

[Fil10] Andrzej Filinski. Monads in action. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 483–494, 2010.

[GDEG13] Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. Java UI : Effects for controlling UI object access. In Giuseppe Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 179–204. 2013.

[GJL+03] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. A Comparative Study of Language Support for Generic Programming. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 115–134, 2003.

[GJS+06] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in c++. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 291–310, 2006.

[GJS+13] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 2013.

[GL86] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 28–38, 1986.

[KI15] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, Haskell 2015, pages 94–105, 2015.

[KLO13] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 145–158, 2013.

[KSS13] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: An alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 59–70, 2013.

[Lei14] Daan Leijen. Koka: Programming with row polymorphic effect types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP 2014, Grenoble, France, 12 April 2014.*, pages 100–126, 2014.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 47–57, 1988.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 333–343, 1995.

[Lip10] Ben Lippmeier. *Type Inference and Optimisation for an Impure World*. PhD thesis, Australian National University, 2010.

[LJ05] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: Extensible generic functions. In *Proceedings of the*

Tenth ACM SIGPLAN International Conference on Functional Programming, ICFP '05, pages 204–215, 2005.

[MHO14] Heather Miller, Philipp Haller, and Martin Odersky. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pages 308–333, 2014.

[MM09] Daniel Marino and Todd Millstein. A generic type-and-effect system. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 39–50, 2009.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.

[MS89] David R. Musser and Alexander A. Stepanov. Generic Programming. In P. Gianni, editor, *Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25. Springer Berlin Heidelberg, 1989.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. The Definition of Standard ML, revised edition. *MIT Press*, 1(2):2–3, 1997.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[Ode06] Martin Odersky. Poor Man's Typeclasses. Presentation to IFIP WG 2.8, 2006.

[Ode14] Martin Odersky. Scala Language Specification, 2014.

[OMO10] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type Classes As Objects and Implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, 2010.

[OSC+12] Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. The implicit calculus: A new foundation for generic programming. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 35–44, 2012.

[OSC+14] Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, Kwangkeun Yi, and Phil Wadler. The implicit calculus: A new foundation for generic programming. submitted, June 2014.

[OZZ01] Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 41–53, 2001.

[Pea11] David J. Pearce. JPure: A modular purity system for Java. In Jens Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 104–123. 2011.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 71–84, 1993.

[PP09] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer Berlin Heidelberg, 2009.

[RAO13] Lukas Rytz, Nada Amin, and Martin Odersky. A flow-insensitive, modular effect system for purity. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs, FTfJP 2013, Montpellier, France, July 1, 2013*, pages 4:1–4:7, 2013.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 408–423, London, UK, UK, 1974. Springer-Verlag.

[ROH12] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, pages 258–282, 2012.

[Ryt14] Lukas Rytz. *A Practical Effect System for Scala*. PhD thesis, EPFL, 2014.

[Sab98] Amr Sabry. What is a purely functional language? *J. Funct. Program.*, 8(1):1–22, 1998.

[SL11] Jeremy G. Siek and Andrew Lumsdaine. A language for generic programming in the large. *Sci. Comput. Program.*, 76(5):423–465, May 2011.

[TJ92a] J. Talpin and P. Jouvelot. The type and effect discipline. In *Logic in Computer Science, 1992. LICS '92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 162–173, 1992.

[TJ92b] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2:245–271, 6 1992.

[Wad90] Philip Wadler. Comprehending monads. In *LISP and Functional Programming*, pages 61–78, 1990.

[WB89] P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, 1989.

[WC08] Stefan Wehr and Manuel MT Chakravarty. ML Modules and Haskell Type Classes: A Constructive Comparison. In *Programming Languages and Systems*, pages 188–204. Springer, 2008.

[WF94] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994.

[Whi15] Thomas Whitmore. Checked exceptions, Java's biggest mistake. Literal Java Blog, "May" 2015.

[WT03] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1):1–32, January 2003.