# 1 Functional programming for the Web

## 1.1 Scientific context

### 1.1.1 What is functional programming?

Functional programming is a programming paradigm that emphasizes the modular decomposition of a program into mathematical functions from arguments and initial state to results and final state. Unlike conventional imperative programming, which works by elementary modifications of the whole machine state, functional programming emphasizes high-level, abstract descriptions of the computations to be performed, and makes state modifications more coarse-grained and explicit (as functions from the old state to the new state), as well as more local (by explicit specification of the parts of the state relevant to the computation). Combined with the fact that functions are themselves first-class citizens and can be manipulated like any other data by other functions, these features of functional programming greatly enhance the modularity and compositionality of large programs, ensuring non-interference between unrelated program parts.

Functional programming is particularly well-suited to the manipulation of complex, tree- or graph-shaped data structures. Transformations on these data structures are expressed concisely and precisely as recursive traversals combined with high-level pattern-matching notations. Data structures are naturally immutable, implying that transformations do not modify the input structure in place, but reconstruct a fresh structure as result. Besides matching mathematical specifications more closely, this approach also supports safe sharing of data, which is crucial for complex cross-linked data structures.

Most functional programming languages are strongly, statically typed: during compilation, programs are subject to extensive static analyses that detect large classes of programming errors, and ensure the integrity and consistency of data structures throughout the execution of the program. Thus, static typing greatly enhances the reliability of programs, especially those that are too complex to allow exhaustive testing. While static typing can be applied to any programming language, the semantic cleanliness and low reliance on state of functional languages allow the use of sophisticated, expressive static typing disciplines that catch many errors, yet do not hamper modular decomposition and code reuse. Beyond static typing, functional programming, for similar reasons, lends itself very well to the use of formal methods (proofs of programs, verification by model checking), as required for the highest levels of program assurance.

Functional programming has very solid theoretical foundations in formal semantics and mathematical logic ($\lambda$-calculus and type theory), and is still today an active area of research in principles of programming languages. At the same time, it is in no way a purely theoretical area of computer science: several high-quality implementations of functional programming languages (Haskell[1], Objective Caml[2]) are available, including high-performance compilers and comprehensive programming environments, and used for developing advanced software in demanding application areas[3], both in industry and in academia.

In this project, we will use the following two high-level, statically typed functional programming languages.

[1] `http://www.haskell.org`

[2] `http://www.caml.org`

[3] `http://caml.inria.fr/users_programs-eng.html`

**Objective Caml** (OCaml for short) is based on a functional core language, and includes classical imperative features, a class-based object layer and a powerful module system allowing safe and efficient management of real-world software applications. The Objective Caml implementation is open source and available for the most common computer architectures. It already has an important user base, is present in the major Linux distributions and receives industrial support through a Consortium[4] hosted by INRIA.

**Haskell** is a purely functional language featuring lazy evaluation, monadic encapsulation of imperative features, and a powerful type class system. Several open source implementations of Haskell are available, including the mature Glasgow Haskell Compiler, developed at Microsoft Research Cambridge. Haskell has an important user base in academia, and is also used in industrial R&D.

### 1.1.2 Functional programming of Web applications

Unlike simple Web sites that are composed of static pages written in advance, complex, interactive Web sites such as the author hypertexts for the Hyper-Learning project are organized around the on-the-fly generation of Web pages from various corpora and databases. Thus, these Web sites are really programs that act as a front-end to the databases, and nagivation through such a site is really an incremental, interactive execution of this program, with Web pages being generated as intermediate execution states. The design, development and testing of the corresponding programs are notoriously difficult and time- and labour-intensive, due to the immense number of navigation paths through a large Web site, and to several characteristics of the Web recalled below.

In the remainder of this section, we show that the functional programming paradigm has the potential to alleviate some of these problems, and is globally a good match for the programming of complex, interactive Web sites. The clean semantics, reliability, and conciseness of functional languages are of course important factors contributing to the quality of Web applications. More specifically, the following points of convergence between functional programming and Web applications can be identified.

**Stateless navigation** By nature, interaction with a Web site, as well as the supporting Web protocols (HTTP), is essentially stateless: when the user clicks on a link, the only information sent to the Web server is the selected URL and the contents of form fields from the current page, but not a complete history of the sequences of interactions that led to this link. Modest amounts of state can be maintained on the client side (Web "cookies") and on the server side (sessions), but such state is limited in size and not completely reliable: "cookies" can be erased by the user, server sessions expire after some time, ... Web applications written in conventional imperative languages tend to rely excessively on such state, and consequently fail to handle correctly several user interactions: nagivation though the browser history mechanism (the "back" and "forward" buttons); cloning of a page in two browser windows; bookmarking of an arbitrary page; etc.

A better approach is to eliminate this reliance on local state altogether, and structure the Web application in such a way that all information needed for navigation is explicit in the URLs and fields. Besides addressing the issues above, this approach supports replication of the Web application across several servers, a crucial feature for reliability and performance. This approach

[4] `http://caml.inria/fr/consortium/`

fits very well within the functional paradigm: hyperlinks then correspond exactly to functions from URLs to pages. Recent research work shows that this explicit encoding of navigation state can greatly benefit from well-known functional programming techniques such as continuation-passing style, which relies on the ability of page-generating functions to take other functions as arguments.

**Static type-checking of HTML and XML generation and transformations** Web applications communicate with the user via on-the-fly generation of Web pages in HTML[5] format. Internally, they often use semi-structured data formats such as XML[6] to store and access persistent data. Both HTML and instances of XML are complex data formats that must obey strict formal constraints in order to guarantee interoperability (e.g. display properly on all browsers) and database integrity. These formal constraints are precisely defined in so-called DTDs or Schemas, and many tools exist to validate an HTML page or XML document against a given DTD. However, when the page is generated dynamically by a program, these validators can only establish the correctness of one run of the program. Ensuring once and for all that all runs of the program always generate valid HTML or XML documents is highly desirable, yet much harder.

A promising approach, pioneered by the XDuce system[7] from University of Pennsylvania, applies the general framework of static typing to this problem: by reflecting DTDs in the type algebra, it becomes possible to ensure statically that a given function always returns an HTML or XML document conforming to a DTD — just like conventional static type-checking can ensure that a given function always returns a well-formed list of integers, say. The semi-structured nature of HTML and XML data makes their static typing much harder than that of familiar data structures, however. Sophisticated type algebras and extended pattern-matching language constructs are required to capture and check DTD invariants properly.

Beyond generation of XML/HTML documents, XDuce also promotes statically-typed transformations of these documents. A typical application is on-the-fly consolidation of several Web sites, *e.g.* to build "portals" that provide unified access to several related authors hypertexts, or to integrate results of Web search engines such as Google in the hypertexts, via Google's XML-based programmatic interface.

**Data persistance** As mentioned earlier, a characteristic feature of functional programming is to shun in-place updating of data structures and favor instead the construction of new, modified data structures, sharing large parts of the original data. This "enrich, don't overwrite" model is a good match for the controlled update of the databases underlying a front-end Web application: by leaving the original data unchanged, the update cannot disturb interactions in progress, including long-term interactions arising from bookmarked intermediate pages. In contrast, in-place modification is problematic, as it is not possible to notify all clients of the data of the update. These considerations are especially true for databases intended for knowledge consolidation, such as the corpora of documents that are the focus of the Hyper-Learning project. In particular, the "enrich, don't overwrite" model naturally provides the versioning of documents, allowing not only the latest version of a document to be accessed, but all intermediate versions as well.

[5] `http://www.w3.org/MarkUp/`
[6] `http://www.w3.org/XML/`
[7] `http://xduce.sourceforge.net/`

## 1.2 Contributions to the Hyper-Learning project

The "functional programming for the Web" part of the Hyper-Learning project sets out to develop programming concepts, abstractions, methodologies and tools for efficient and reliable programming of complex Web applications such as the ones arising in the Hyper-Learning project. Challenges to be addressed include highly non-linear navigation, high degree of cross-referencing and hyperlinking between documents, unified access to large corpora of documents of many different nature and formats, and robustness in the presence of frequent addition of documents and links. We will address these challenges by leveraging research results on functional programming and static type-checking, and extending them towards semi-structured data and persistent data repositories.

The primary goal of this part of the Hyper-Learning project is to advance the state of the art in programming complex Web applications; it is not to develop the software infrastructure for the HyperServers supporting the author hypertexts – the latter is the goal of the "software development" part of the project. However, we expect fruitful interactions between these two parts. Concrete programming problems encountered during the development of the HyperServer software will identify important directions and guide design choices for the research carried out in the "functional Web programming" part. Conversely, the techniques and tools developed as part of the "functional Web programming" research effort will be tested and experimentally validated on corpora originating from the author hypertexts; some of these tools could then be eligible for integration within the HyperServers.

The research carried out in this part of the Hyper-Learning project is subdivided in the following three work packages.

### 1.2.1 WP1: Unified approach for server- and client-side Web programming

Abstraction is one key to the successful construction of software systems. For Web-based applications, the popularly used tools do not yet offer a sufficient degree of abstraction and thus applications may incur subtle consistency problems. For example, those tools rely on string processing when they should really manipulate suitably designed abstract datatypes.

Functional programming languages are traditionally strong in providing orthogonal abstraction facilities. Taken together with their expressive type systems, many concepts arising in application domains can be captured satisfactorily. The WASH system[8] developed at Freiburg University is a promising first step towards making available the advantages of functional programming languages in the domain of Web scripting. WASH consists of a set of Haskell libraries that provide typed abstractions for programming server-side Web scripts. In particular, WASH provides a powerful session abstraction with its own dedicated state. Using the session abstraction greatly simplifies application development because programmers are freed from decomposing the application into small mutually dependent programs and from explicitly passing the session state throughout the application. Second, WASH guarantees that scripts only generate correct HTML by providing an abstract data type for documents. The same mechanism can be used to guarantee adherence of generated XML documents to a DTD. Third, WASH imposes Haskell's type system on the input fields of a form (which are strings in, *e.g.*, JSP and ASP). Violations of the type system by user input are detected and handled transparently to the application. Finally, there is an interface for type-safe, concurrent access to persistent data on client and server.

[8] `http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASH`

This workpackage aims at extending the scope of functional Web scripting to client-side Web scripting. The ultimate goal is a unified approach that allows the specification of server-side as well as client-side Web scripts without having to resort to multiple programming languages. Due to the functional programming style, where side effects are scarce and flagged by types, there is more flexibility in the choice of an execution location. Starting with a high degree of independence from the server host, parts of the scripting code might be executed on intermediate computing proxies that relieve end-user devices (*e.g.*, palmtops or cell phones) so that only code directy related to the user interface needs to run on the client itself. Hence, the code of the application should not make premature commitments to bind a computation to a particular execution location (client, server, a computing proxy). Rather, the decision where the code actually runs should be made as late as possible —potentially by the runtime system— depending on availability of computing power, bandwidth, whether support for JavaScript is available in the browser, and so on.

**Implementation.** We wish to leverage existing technology to ensure widespread usability on the client-side without installing additional software. Hence, we expect to translate a functional Web script into executables that run on the servers and on dedicated computing proxies as well as JavaScript or Java programs which are referenced from generated pages. Executables may generate HTML documents or —in the case of a server accessed by a computing proxy— XML documents that are further processed on the proxy.

**Calculi for specifying execution location.** Our implementation model implies the generation of code at runtime on the server. To ensure robustness, we must guarantee that the thus generated code does not crash. This kind of property has been formalized using program calculi for staging. Hence, we propose to develop a staging calculus that is suitable for modeling this situation. Promising candidates are multi-level calculi (to specify staging properties), region and ambient calculi (to specify spacial properties). The results drawn from such a calculus will be incorporated into the implementation using static analysis. Investigating this calculus and its static analysis constitutes a significant theoretical challenge.

**Types for client-side scripts.** We expect that we can roughly carry over the type system of the underlying functional language (OCaml or Haskell) to generated JavaScript code (JavaScript is a dynamically typed, object-based language which provides function objects). The challenging issues in this task are the identification of a sublanguage of the server-side scripting language that is amenable to translation into JavaScript, establishing a sound type discipline for this sublanguage, and the static analysis of potential boundaries between server-side execution and client-side execution (some computations *e.g.* database accesses *must* run on the server, some computations *e.g.* creating new windows *must* run on the client).

**Applications** We plan to develop prototypes for documents and structures from the Hyper-Learning project as examples for our techniques. Due to the high-level of abstraction, a functional Web script is readily and radically adaptable, so it is the ideal tool for exploring different ideas regarding presentation and structuring.

**Roadmap**

1. **Client-side user interfaces**

   - specification of a simple user-interface description language (UIDL) based on the facilities of client-side JavaScript
   - translation of UIDL to JavaScript
   - implementation of translation using code generation

2. **Staging calculus with locations**

   - develop calculus AbstractJavaScript (AJS) with explict constructs for staging and specification of location
   - establish its formal properties
   - investigate static analysis to translate AJS without staging and location to full AJS

3. **General client-side Web scripting**

   - specification of AbstractJavaScript (AJS), its server-side semantics, and its client-side semantics (translation to JavaScript)
   - implementation of the translation using code generation
   - validation: reimplementation of UIDL using AJS

4. **Implementation of staging calculus**

   - runtime system to distribute work between client and server
   - structure for computing proxy
   - extend runtime system to deal with proxies

5. **Documentation**

6. **Public release** under an open source license.

### 1.2.2 WP2: Statically-typed generation and transformation of HTML and XML data

The emergence of XML as a universal format for data and documents requires from high level programming languages to provide facilities for easy, efficient and reliable processing of XML documents. While the application area of XML-related tools is very broad and the techniques that are proposed in this work package are of general use, we concentrate here on developing and using extensions of the Objective Caml (OCaml) programming language with reliable XML support, with the World Wide Web as main application domain.

Extending a statically typed programming language such as OCaml towards the processing of XML documents is most beneficial if the extension is itself statically typed and provides a similar level of reliability as the programming language itself. Resulting from research on static typing of XML processing, XDuce[9] is a prototype of a functional programming language equipped with a sophisticated type system dedicated to XML as well as an expressive pattern-matching mechanism for traversing XML data in a type-safe way.

[9] `http://xduce.sourceforge.net/`

The aim of this work package is therefore to extend the OCaml system with XDuce, providing in a single system the flexibility, the efficiency and the rich library of OCaml and the type-safe XML support of XDuce. The resulting system will provide a flexible and efficient development environment for many XML-related applications. In the context of the Hyper-Learning project, the following two applications are particularly relevant: 1- programming dynamic Web pages whose well-formedness and validity will be guaranteed once and for all, at compile time; 2- facilitate data exchange and consolidation between multiple Web servers (related authors hypertexts, search engines, etc).

In order to preserve the integrity of the Ocaml system, the approach we will follow is to isolate the XML processing parts of the applications in dedicated XDuce modules (with their specific syntax, typing discipline and pre-compilation). Applications will therefore be composed of OCaml (regular) modules as well as XDuce modules, each processed by their specific front-end, but all using the OCaml optimized back-end. The XDuce modules will use a language that is an extension of the current XDuce language with OCaml primitives and constructs. The technical problems to be solved here are the design of this extension, its typing discipline (an extension of that of XDuce) and its properties (well-typed programs cannot generate run-time type errors), and its implementation (parsing, type-checking and compilation).

**Typing.** Type-checking is where the main challenges occur: at least, the XDuce type system must be made compatible with OCaml higher-orderness (passing functions as arguments and returning them as results) and polymorphism (the ability to have generic functions, operating uniformly on whole ranges of data, such as "lists of anything"). The current XDuce type system accepts only first-order functions (functions from XML data to XML data), and has no polymorphism but only subtyping. Extensions to higher-order functional types and polymorphism is important, because it allows the use of OCaml primitives or library functions (such as those operating on lists, arrays, hash-tables, etc.) that are naturally both polymorphic and receive functional arguments. This aspect involves a great deal of theoretical work, since it implies both a new formalization and correctness proofs of the resulting type system, as well as an implementation effort.

**Compilation** From the execution point of view, the XDuce modules could simply be executed by an interpreter written in OCaml: this is an easy way of linking both kinds of modules, using the OCaml runtime. However, more efficient execution can be obtained by designing a specific compilation route for XDuce modules. For instance, translating XDuce programs into OCaml programs, and reusing the OCaml code generator for producing either bytecode (for portability) or native code (for maximal efficiency). Special care has to be taken for the XDuce pattern-matching construct: pattern-matching is a fundamental feature of XDuce since it enables to extract XML components for processing, and it therefore must be extremely efficient.

**Applications** In the context of this project, our main target is the production of HTML or XML documents whose well-formedness and validity are statically guaranteed correct. We therefore plan to document specifically this kind of applications, and produce a sample Web site as a running example for this documentation.

**Roadmap**

1. **Design:**

   **Extension of XDuce with OCaml primitives and constructs:** design, type checking, formalization, correctness proofs (verify that the initial properties of XDuce typing are preserved in the extension).

   **Interfaces** between XDuce modules, from XDuce modules to OCaml modules. In the latter case, the goal is to verify that the properties of the OCaml typing discipline are preserved.

2. **Implementation** of the mechanisms above:

   **Typing.**

   - A parser for the extended XDuce.
   - A type checker for the extended XDuce.
   - Extension of the OCaml type reconstruction mechanism so that it can exploit XDuce interfaces (as previously designed).

   **Shared back-end for OCaml and XDuce.** XDuce modules should use a specific front-end (parsing and type checking), but must use the same back-end as OCaml in order to be linked with OCaml modules. A minimal solution is to use an interpreter written in OCaml for the execution of the XDuce parts of applications. A more efficient solution, that currently remains to be explored, would be to translate XDuce modules into OCaml abstract syntax trees that will be sent to the OCaml code generator. We will implement the former solution, and explore the latter (and implement and optimize it if successful).

3. **Documentation** for OCaml+XDuce. A specific documentation of using this system in building dynamic web pages, with an example site demonstrating the various features of the system.

4. **Public release** under an open source license.

#### 1.2.3 WP3: Object-based persistent data repositories with distributed updates

The purpose of this work package is to develop a model and an implementation for a persistent data repository that matches well the functional programming paradigm, and supports directly the storing of complex data structures, including pointers from structures to sub-structures.

Conventional relational database systems, as used in most Web applications, do not support complex structured data natively, forcing them to be encoded as simple data plus many relations to represent the pointers. Moreover, relational databases promote a highly imperative programming style, where the data is naturally modified in place. All interactions between the program and the persistent data goes through explicit, low-level "read", "insert" and "update" operations. This is a poor match for the functional programming paradigm. In particular, significant encodings are necessary to implement the "immutable data with sharing" model typical of functional programming on top of a relational database.

We propose to break away from this traditional database model and follow the data persistence approach to obtain a model for data repositories that matches functional programming better. In the persistence approach, data stored in the repository has essentially the same structure and the same static types as the transient data natively supported by the functional programming

language. Thus, the program manipulates both persistent and transient data through a common set of language operations; the only difference between these two kinds of data is that creation and modification of the former persist beyond the end of the program execution when committed.

The data persistence approach has been well investigated in the context of object databases and semi-structured databases. Object database, while unsuccessful commercially, were found to be effective as back-ends for certain Web applications, such as Web indices and e-commerce sites. We believe that Web applications of the kind investigated in the Hyper-Learning project, would greatly benefit from a data repository based on the persistance model.

In this work package, we will therefore design and implement a persistent data repository for the OCaml functional programming language (the same language that is used for WP2). A Haskell interface for this repository is also planned. The resulting library and extension of the OCaml and Haskell systems will be distributed publically as open source software. The expected outcome of this work is the smooth integration of data persistance within the OCaml or Haskell programming languages and systems. Besides Web applications, this integration is expected to enhance the usefulness of functional programming languages for many application areas. Challenges to be addressed include:

- Design of a suitable persistent data model and of its OCaml application programming interface (API). Due to the structure of the OCaml system, persistent data can easily include all "passive" OCaml data structures (base types, sum and record types, tuples and arrays), but "active" data structures (functions and objects) are harder to handle. The API can either be presented as an abstract type with associated operations, similar to that currently used for lazy evaluation, or in an object-oriented style.

- Strong typing of accesses to persistent data. Since persistent repositories are not generally available at compile-time, fully static enforcement of the typing discipline is not possible, and some amount of dynamic (run-time) type checks is necessary. Advanced dynamic typing techniques can however reduce the frequency of such checks, e.g. from every access to once when the database is opened.

- Design of a suitable disk file format for storing the repositories. This format must support atomic transactions, dynamic allocation, and automatic reclaimation of unreferenced entries. Alternatively, an existing database management system can be used as a back-end for this implementation.

- Replication and distribution. According to the functional paradigm, most of the persistent data stored in a repository is never modified once created, and those that can be modified in place are distinguished by their static types. This offers ample opportunities for replicating immutable data while preserving the semantics of the program.

As soon as persistent repositories are distributed or replicated, the problem of ensuring the atomicity of updates becomes much harder than in a centralized setting, and requires distributed transaction mechanisms. A general approach to this problem is to extend the functional base language with a process orchestration layer in which a variety of distributed transaction policies can be expressed. Beyond transactions, such a general orchestration layer could also provide support for the definition and enforcement of high-level policies for modifying the database, such as obtaining the approval of an editor before adding a new document, waiting until copyright issues are resolved before publicizing the new document, etc.

**Roadmap**

1. **Design:** review the state of the art in persistent store systems; design of the persistent data model and of its OCaml application programming interface.
2. **First implementation** of a persistent repository for OCaml, without distribution nor strong typing.
3. **Addition of strong typing guarantees** to the implementation via the recourse to dynamic type checks.
4. **Experimental evaluation** on problems arising both from the "software development" branch of the Hyper-Learning and from the OCaml user community.
5. **Extension towards a distributed implementation**, including support for distributed transactions and user-defined update policies.
6. **Documentation and public release** as Open Source software.

## 1.3 Summary of work packages

### 1.3.1 WP1: Unified approach for server- and client-side Web programming

**Start date** Beginning of project.

**Participants** Freiburg (Peter Thiemann), PPS.

**Objectives** Development of the formal foundations for a unified functional approach to server- and client-side Web scripting. Implementation of the resulting design. Application to the construction of flexible user interfaces.

**Description of work (first 18 months)**
Review the state of the art for GUI toolkits for functional programming languages. Identify and document concepts that are expressible in HTML/JavaScript. From this information, develop an abstract specification language for Web-based user interfaces (a UIDL in terms of a combinator library). Specify the translation from the UIDL to HTML/JavaScript. Implement the translation in terms of code generation. Develop an example application.

**Deliverables (first 18 months)**
Prototype: GUI library for the WASH system
Report: design of the UIDL
Report: translation of UIDL to HTML/JavaScript and its implementation by runtime code generation.

**Milestones and expected results**
$T_0 + 9$ months: design of the UIDL completed.
$T_0 + 18$ months: prototype implementation of the GUI library for WASH publically distributed as open source software.

### 1.3.2 WP2: Statically-typed generation and transformation of HTML and XML data

**Start date** Beginning of project.

**Participants** PPS (Jérôme Vouillon), INRIA (Michel Mauny).

**Objectives** Design of an integrated environment for programming statically typed transformations of XML documents, based on the Objective Caml and XDuce systems. Implementation of the design, documentation and public release.

**Description of work (first 18 months)**
Design of an extension of XDuce with OCaml primitives: interfacing XDuce and OCaml modules, handling and using OCaml polymorphism and higher-order functions from XDuce modules.
Execution issues: interpretation *vs.* compilation of XDuce modules.
Prototype implementation of the design, using an OCaml interpreter as the XDuce execution model.
Documentation including a small web site as running example.

**Deliverables (first 18 months)**
Prototype: implementation of a programming system mixing OCaml and XDuce.
Report: a report on the design of the language.

**Milestones and expected results**
$T_0 + 9$ months: design of the extended XDuce
$T_0 + 18$ months: prototype implementation of the programming system, publically distributed as open source software, together with its documentation.

### 1.3.3 WP3: Object-based persistent data repositories with distributed updates

**Start date** Beginning of project.

**Participants** INRIA (Pierre Weis, Xavier Leroy); Bologna (Cosimo Laneve).

**Objectives** Developement of a model and an implementation for a persistent data repository that matches well the functional programming paradigm, and supports directly the storing of complex data structures in a type-safe manner. Application of this repository to Web applications. Extension towards a distributed implementation, using a process orchestration layer in which distributed transaction policies and user-defined update policies can be expressed and enforced.

**Description of work (first 18 months)**
Review the state of the art in persistent store systems.
Design and prototype implementation of a persistent repository for the OCaml functional language, without distribution nor strong typing.
Addition of strong typing guarantees via the recourse to dynamic type checks.
In preparation for a distributed implementation, design of a process orchestration layer for expressing transaction and update policies.

**Deliverables (first 18 months)**
Prototype: a (non-distributed) implementation of a persistent data repository for OCaml.
Report: a report on the design of this repository and the dynamic type-checking mechanisms used to enforce type safety.
Report: a report on the design and use of the process orchestration layer.

**Milestones and expected results**
$T_0 + 9$ months: design of the persistent data model and of its OCaml application programming interface completed.
$T_0 + 18$ months: prototype implementation of a persistent data repository for OCaml publically distributed as open source software.

## 1.4 Project effort

To help estimate the number of person-months for activities in which partners are involved, here is a break-down of the persons involved by site. These include both researchers and faculty on permanent positions, and temporary positions such as post-docs and research associates.

**Bologna**
Permanent positions: Cosimo Laneve (30%)
Temporary positions: 2

**Freiburg**
Permanent positions: Peter Thiemann (20%)
Temporary positions: 3

**INRIA and PPS**
Permanent positions: Xavier Leroy (20%), Michel Mauny (50%), Jérôme Vouillon (50%), Pierre Weis (20%).
Temporary positions: 4