

Small or medium-scale focused research project (STREP)  
Short proposal  
ICT FET Open Call  
FP7-ICT-2007-C

Certified Complexity  
CerCo

# 1 Scientific and/or technical quality, relevant to the topics addressed by the call

## 1.1 Concept and Objectives

The project aims to the construction of a formally verified complexity preserving compiler from a large subset of C to some typical microcontroller assembly, of the kind traditionally used in embedded systems. The work comprise the definition of cost models for the input and target languages, and the machine-checked proof of preservation of complexity (concrete, not asymptotic) along compilation. The compiler will be open source, and all proofs will be public domain.

## 1.2 Progress beyond the state-of-the-art

Automatic verification of properties of software components has reached such a level of maturity to allow complete correctness proofs of entire compilers, that is of the semantic equivalence between the generated assembly code and its source program. For instance, in the framework of the Verifix Project a compiler from a subset of Common Lisp to Transputer code was formally checked in PVS. Strecker and Klein certified bytecode compilers from a subset of Java to a subset of the java Virtual Machine in Isabelle. In the same system, Leinenbach formally verified a compiler from a subset of C to a DLX assembly code. Maybe, the most advanced project is Compcert, leaded by Leroy, based on the use of Coq both for programming the compiler and proving its correctness. In particular, both the back-end and the front-end of an optimising compiler from a subset of C to PowerPC assembly have been certified in this way.

Still, however, very little is known about the preservation of intensional properties of programs, and in particular about their (concrete) complexity. The theoretical study of the complexity impact of program transformations between different computational models has been so far confined to very foundational devices, while we address a concrete case of compilation from a typical high-level language to assembly. It is worth to remark that, between foundational models, it is very unlikely to have constant-time transformations: for instance coding a multitape Turing machine into a single tape one could introduce a polynomial slow-down. So, the fact that complexity is architecture independent, and that you may always pass from a language to another preserving the performance of your algorithms must be taken with the due caution.

In particular, as surprising as it may be, very little is known about the complexity behaviour of a compiled code with respect to its source; as a matter of fact, most industries producing robots or devices with strong temporal constraints (such as, e.g. photoelectric safety barriers) still program such devices in assembly.

The tacit assumption that the complexity of algorithms is preserved along compilation (even if plausible, under the suitable assumptions) is not supported by any practical or theoretical study. For instance, a single register is usually used to point to activation records, implicitly delimiting their number; you may take more registers to this purpose, but unless you fix a priori their number (hence fixing the size of the stack), you cannot expect to access data in activation records in constant time. In particular, the memory model assumed by Leroy prefigures an infinite memory, where allocation requests always succeed, that clearly conflicts with the reality of embedded software, where one has to work within precise (and often relatively small) memory bounds. If working in a restricted space on one side allows us to properly weight memory access with a unit cost, on the other it introduces a subtle interplay between space complexity, time complexity and correctness that will be one of the crucial issues of the project.

Even admitting (as we hope to prove) that in a confined universe we may actually preserve complexity, the actual interest of the project is in producing a (certified) computational cost for the instructions (or basic blocks) of the input program, giving the possibility of computing time constraints reasoning directly on the high level input language. It is also worth to observe that, from this point of view, the cost model of the input depends from the target architecture: for instance some microcontrollers lack the multiplication instruction as a primitive operation, preventing to count such an operation with a unit cost. Moreover, if we are interested in a

really tight complexity measure, we cannot expect to have a uniform cost for input instructions since, due to instruction selection and optimisations, their actual cost usually depends by their surrounding context. Hence, the compiler should return an annotated version of the input program, with a precise (and trustable) cost for each basic block. This would allow for instance to assert strong temporal constraints on the behaviour of input program, proceeding then to prove that the executable respects such constraints by reasoning directly on the input (e.g. by the help of an interactive prover).

### 1.3 S/T methodology and associated workplan

The methodology follows the promising research direction of Compcert, consisting in using a proof assistant both for programming the compiler and for proving its correctness. To this aim, a proof assistant supporting a primitive notion of reduction and some mechanism of program extraction is particularly suited. From the point of view of complexity, the approach consists in defining cost models for the source and target languages, as well as for all intermediate languages used along the compilation process, proving that each transformation phase preserves the complexity. On the other side, we are not concerned with the formal complexity of the compiler itself, although we aim to obtain a tool competitive with typical non-optimising compilers used for critical embedded software.

Due to the intensive study they have been subject to, compilers are highly modular tools whose development is particularly suited to follow a precise, well articulated and easily verifiable workplan. Moreover, the optimising nature of most of the transformations, does not preclude the possibility to obtain a valuable final product even in case of failure or delays in some of the workpackages, due to unexpected difficulties.

#### 1.3.1 Source and Target Languages

The choice of the source and target languages is in some way dictated by our objectives. For the source language, the practically unpredictable behaviour of the garbage collector, preventing any tight performance measure, automatically excludes most functional and object oriented languages. Programming constructs with dynamical aspects (exceptions, dynamic method dispatching, dynamic binding, etc.) are another potential source of trouble; moreover, typical applications running over microcontrollers do not usually require programming paradigms conceived for programming "in the large". The natural choice is a purely imperative language, that is also the closest one to the underlying physical architecture, e.g some subset of C inspired by Leroy's Cminor or Peyton Jones' C-. As for the target, we shall mostly focus on 8 and 16 bit microcontrollers, of the kind used in control systems for safety critical devices requiring responsiveness to asynchronous events within strong temporal constraints.

#### 1.3.2 Workpackages

**Work Package 1** aims at building a functional prototype of an untrusted cost annotating compiler. The compiler will be written in a high-level, comfortable programming language particularly tailored to compiler construction. Its innovative feature will be that of respecting the complexity of the source code and make it explicit by annotating the code with complexity annotations for every  $O(1)$  operation. Each annotation will reflect the exact time taken by the compiled code that better matches the input operation. Because of optimizations, different occurrences of the same high level instruction are likely to receive different annotations.

One important task of the WP will be the study of a suitable memory model that take space limitations into account. Indeed, as we already remarked, the assumptions relative to the memory model are likely to have a crucial impact on the possibility of obtaining a correct complexity preserving compiler. Other major issues to be weighted are about the adoption of a typed/untyped description (a typed one seems to be preferable only if the architecture of the microcontroller has

dedicated registers for e.g. floats versus integers), and the organisation of the stack frame, to minimise the effort of proving the absence of overlap between different areas.

**Work Packages 2 and 3** are devoted to the correctness proof of the front-end and the back-end of the compiler. The first step of the correctness proof consists in rewriting the code into the internal language of a proof assistant. Then the code is proved correct with respect to the appropriate invariants. In case of discovery of bugs, feedback will be immediately provided to WP 1 in order to update and test the untrusted compiler before proceeding in the formalization. The set of skills required for these WPs are largely orthogonal to those required for the previous WP. The division into two largely independent WPs will be helpful in maximizing parallelism of the tasks involved. The main expected outcome will be a trusted version of a cost annotating compiler.

**Work Packages 4** will develop a proof of concept prototype of a system to draw complexity assertions on the execution time of a C program. The system, which will be interactive, will present to the user the complexity annotations for  $O(1)$  operations produced by the annotating compiler and will help the user in stating and proving more complex costs annotations for the whole C program. It will be based on existing tools (like Caduceus) for the management of Hoare style pre and post-conditions and the synthesis of proof obligations. The main expected outcome will be the possibility of interactively proving the exact complexity of an high-level C program. All tasks of this WP will be independent of those of WP 2 and 3, allowing for a very high degree of parallelization in the second part of the project (after the first release of the untrusted annotating compiler). The skills required in this WP are clearly complimentary to those required to develop a compiler and prove its correctness. In particular, here we need expertise in semantics of high-level languages and complexity.

**Work Packages 0 and 5** will be devoted respectively to main management activities and to an intense work of dissemination of scientific results. In the final year, we will also assess the technologies developed in the project in order to ensure fulfilment of actual requirements of the targeted exploitation communities, which will be identified at the beginning of the project.

The annotating compiler architecture will be traditional and will comprise a front-end and a back-end that implement a multi-phase translation of a C like program into assembly code. Phases may be roughly divided into two categories: transformations (with different source and target), and optimisations (usually within a given intermediate representation). We shall privilege transformations first, in order to get, as soon as possible, a working and verifiable tool, leaving optimisations for a later stage. This will also allow to tune the number and kind of optimisation according to the actual progress of the project. Moreover, optimisations are likely to introduce, during compilation, a large degree of uncertainty in performance (especially on lower bounds) and, a priori, it is almost impossible to predict an expected behaviour (and hence verifiable results) for these phases (one of the goals of the project is also to obtain a better theoretical investigation of the many heuristics used for optimising the code).

The aim of the transformations is to progressively refine the notion of pseudo-register (temporaries) to eventually map them to hardware registers or memory slots in activation records (or in the heap), and to pass from a high level representation of control (e.g. by means of flow diagrams) to a sequential list of machine instructions with labels and jumps.

## 2 Implementation

## 3 Impact

### 3.1 Expected impacts listed in the work programme

The project responds to the increasing expectation for trustworthy, dependable and long-lasting systems by developing reliable compilers not only from the point of view of behaviour but also of time. Disposing of such a tool would allow e.g. to shift programming of critical systems with strong temporal requirements from assembly to a high-level language, with a major beneficial impact on reusability and maintainability. At the same time, it would provide an occasion for rethinking the nature of compilation, measuring the actual impact of the many optimisation phases, and providing a better theoretical status for the many heuristics of current use. The project lies at the intersection between complexity, formal checking and compilers, requiring a complex synergy between experts of the three fields, and providing an original forum for a radical interdisciplinary exploration of complexity issues from a computer assisted viewpoint.

### 3.2 Positioning with respect to the realisation of a long term vision in the ICT domain

In spite of the many recent, astonishing achievements in proof checking (proofs of the four colors theorem, prime number theorem, Kepler conjecture, Jordan curve theorem, ...) and program verification (let us just recall the formalization of the TCP/IP stack, in addition to the already mentioned results), the field of computer assisted reasoning, with its long term vision of a fully dependable, formally checked information society is still one of the most visionary field of computer science. The impressive results obtained in this area are only partially due to the advancements in the tools for assisted reasoning, but especially to a growing confidence in their potentialities. The only way to go past the proof-of-concept phase and to bring these technologies to the maturity level that could allow a substantial technology transfer towards industries, is by testing them on real, complex and challenging tasks, as the one proposed in this project.

Compilers, filling the gap between humans and machines from the programming point of view, are one of the main tools at the base of the information technologies. We cannot hope to build truly trustworthy, dependable and long-lasting systems over shaky grounds: the semantics of compilation must be better understood, and such a semantics cannot be restricted to the denotational aspects of computation. The possibility to provide space or time bounds of the source, carrying them to compiled code is still perceived by the compiler community as a highly visionary goal, that would provide a major milestone in this area.