Verified Compilation of Functional Arrays for Safety-Critical Embedded Control Software

PhD thesis proposal, supervised by T. Bourke and M. Pouzet

February 2025

The dataflow synchronous languages Lustre [22] and Scade 6 [15] are domain-specific functional languages. Source programs transform streams representing the time-varying values of inputs and outputs, and embedded code that executes in bounded time and memory is generated automatically. Scade 6 is routinely used to develop embedded safety-critical software for rail (train tracking and brake control), aerospace (fly-by-wire, motor control, and landing gear control), energy generation (emergency shutdown in power stations), and automotive (ant-lock braking and battery management) applications.[1] Its compiler is qualified to the strictest industrial standards. This greatly facilitates the traceability arguments required to certify systems since they can be made between specifications and Scade 6 source programs, rather than to embedded code.

Over the last few years, we have been developing a formalization of, and verified compiler for, a subset of Scade 6 [6, 7, 8, 10] in the Coq proof assistant [18]. The Vélus project[2] has three main goals. (*1*) The first goal is to prove end-to-end compiler correctness from a subset of Scade 6 into Clight for further compilation by the CompCert verified C compiler [28, 5]. We follow the standard approach [14] of rewriting block-based constructs like state machines into simpler dataflow operators, which are normalized [8], before being translated into sequences of imperative commands [6]. It has been necessary to adapt the standard compilation scheme and intermediate languages [4] to facilitate formal proofs while still generating convincing code. The resulting correctness theorem links the functional stream semantics of the source language to the imperative semantics defined in CompCert for the generated assembly code. (*2*) The second goal is to formalize, in a proof assistant, the source semantics of a language of streams and functions on streams. This is necessary for proving compiler correctness, but a machine-checked semantics is independently interesting and useful. We build on existing formal definitions [22, 17, 14], but the constraints imposed by the proof assistant have led us to new definitions for modular reinitialization [7, 12], state machines [10, 31], and a denotational model that takes operator errors into account [9, 24]. (*3*) The third goal is to permit proof-assisted reasoning about systems that incorporate control software written in Scade 6, such that properties verified on the source language are transferred directly to the generated executables via the compilation correctness theorem. Recent thesis work is a step in this direction [9, 24].

A significant limitation of Vélus is that it only considers scalar types, that is, integers, floats, and user-defined enumerations, whereas Scade also provides arrays and they are essential in many applications for programming filters (FIR/IIR/Kalman), matrix algorithms (state-space representations/Cholesky decomposition), model predictive control, neural networks, etcetera. Compiling arrays correctly into efficient embedded code is difficult and a real source of errors. The challenge in this thesis project is to address this problem using a proof assistant: to verify compilation correctness, to express semantic models, and to lay the groundwork for interactive program proof. Ultimately, the aim is to complement a rigorous and expressive programming language for combining dataflow control algorithms, hierarchical state machines, and compute-intensive workloads on arrays, with the state-of-the-art in machine-assisted formal methods.

[1] `https://www.college-de-france.fr/site/gerard-berry/seminar-2013-04-23-11h00.htm`
[2] `https://velus.inria.fr`

State of the Art

Programming language features and compilation techniques for arrays have been developed and studied for decades. The particular problems posed by dataflow synchronous languages, like Lustre and Scade 6, come from combining a functional semantics, that includes parallel composition and stream operators, with the generation of embedded code, that allocates memory statically and is amenable to worst-case execution time analysis. It is these very features that make these languages ideal for specifying and reasoning about control algorithms while still allowing automatic code generation. From the perspective of specification and verification with a proof assistant, it is instructive to compare the treatment of arrays in the CompCert [5] and CakeML [27] projects. In the following text, we will also sketch the treatment of arrays in Lustre and Scade 6, and also the Heptagon and Zelus compilers.

A fundamental problem in functional languages is maintaining a source semantics with referential transparency and without side-effects, while ensuring an efficient implementation. Consider, for example, the following array update, with, at left, the program syntax of Scade, and, at right, its semantics.

```
y = [ x with [2] = 7 ];
```

$$\forall i \in [0, \mathsf{length}(x) - 1], y.[i] = \begin{cases} 7, & \text{if } i = 2 \\ x.[i], & \text{otherwise.} \end{cases}$$

The fact that there are two arrays, `x` and `y`, implies that an implementation may need to copy the contents of `x` into `y` before updating the single element at 7, for instance, with the imperative code:

```
let y = copy(x) in
y.[2] = 7
```

A typical variation in a stream-based language would be to define `y` as an initial array "followed by" an updated version of its previous value:

```
y = y0 fby [ y with [i] = v ];
```

A naive implementation would perform two array copies: once for the array update subexpression and again to update the internal state associated with the `fby` operator. All of this becomes increasingly expensive, in terms of execution time and memory use, as the arrays or their elements become larger. This "aggregate update problem" is common to all functional languages and various static and dynamic techniques have been proposed to solve it [23]. Many of them, like, for example, shallow binding [3, 2], require garbage collection, which prevents, or at least complicates, their use in real-time applications.

Imperative Arrays One approach is to consider that `x` does not represent an array value, but rather refers to an object in memory, and to allow this object to be updated by an imperative command. For example, in C: `int *y = x; y[2] = 7;`. Or in OCaml: `let y = x in y.(2) <- 7`. Here, `y` is not a new array, but rather points to the same area of memory as `x` does. The meaning of such programs depends on the evaluation order and not just on dependencies between variables. A formal semantics must represent the indirection via memory and formalize execution order. This is the approach taken in the CompCert [5] and CakeML [27] verified compilers. The semantics of a language that permits parallel definitions, like `x1 = e1 and x2 = e2`, which may involve array updates, must either accept nondeterminism or mandate an evaluation order. Whereas in a synchronous language, like Lustre or Scade 6, just as in the original Kahn networks [25], the components of a program evolve in parallel to map inputs to outputs but nevertheless define a (determinate) function. For Vélus, adding imperative updates would complicate the semantics, in which each component is represented by a function on streams, and the ambition to reason about programs by rewriting using algebraic equivalence relations. It would also complicate the intermediate languages and compilation passes. For instance, the correctness proof for the scheduling pass relies on semantic invariance under reordering, and the invariants for the transformation to imperative code are stated solely in terms of data dependencies.

Monads Another approach is to use monads to represent the sequentiality constraints as data dependencies (the passing of a state from one monadic expression to another) and for the compiler to transform the monad into sequential commands. Using monads, a functional semantics is maintained and efficient code can be generated. This is the approach taken in Haskell and the subset of Haskell which is verified in the PureCake extension to CakeML [26]. In PureCake there is a single monad that provides *stateful arrays, exception-handling, and a foreign function interface* [26, §2]. The semantic model uses a version of interaction trees [35], which permits to encode sequentiality, divergence as infinite traces of silent steps, and interactions with the environment as non-deterministic choice (branching). As in process algebra or automata theory, reasoning involves bisimulations of one form or another, and a system evolves through synchronizations with its environment. This is not the approach taken by synchronous languages which prefer simpler notions of equivalence and composition that ensure determinism and permit a logical notion of time, both of which are important properties for safety-critical embedded control software.

Synchronous Languages In Lustre v4, arrays are eliminated by syntactic expansion into basic equations prior to circuit generation [32]. This approach not suitable for generating software. The Lustre v6 and Scade 6 languages instead provide a fixed set of iterators for manipulating arrays (map, fold, etcetera) [29, 15], fuse them where possible, and generate imperative code with loops. Arrays have a purely functional semantics and the compiler tries to minimize copying by updating arrays in place within a function. This approach is also taken in other "block diagram" languages [1]. An extension that permits in-place updates across function boundaries, and also provides typing annotations to guide optimization, was formalized using semi-linear types and implemented in the Heptagon compiler [21].[3] The types are "semi-linear" because while in-place writes to arrays must be linear, multiples reads may occur "simultaneously". Functional arrays and iterators are treated in another verified Lustre compiler [33, 34], but this work sidesteps the main scientific questions by simply extending the imperative semantics of CompCert and generating code with explicit copying at each array operation. It is thus not possible to reason about programs in terms of streams and stream functions and the generated code is too inefficient for real applications.

Recent work on arrays in synchronous languages [16, 30] focuses on type systems for representing array sizes, forbidding array copying unless specifically permitted in the source code (allowing programmers to better understand and control the resource use of generated code), and aggressive optimizations based on index manipulations and sharing. The source language in this approach is enriched with constructs inspired by those of Sisal [20], a general-purpose functional language for large-scale scientific applications with similar performance to imperative languages [19, §1]. These constructs have been implemented in a prototype [30], added to the Zelus language and compiler [11], and given a semantics in a coiterative interpreter written in OCaml [13]. An G. Bussone's ongoing thesis continues this line of work for a compiler that generates Rust code.

Objective

This thesis aims to treat functional arrays in a stream-based language within an end-to-end formalization in a proof assistant. The desired result is a high-level language for programming embedded software with a formally verified link to efficient implementations in assembly code.

Source language The source language must allow to freely mix dataflow primitives, control structures, and array operations. It must permit the programming of realistic control applications and integrate naturally into the functional semantics in Coq [9, 24] to provide both a specification for the compiler and a base for interactive reasoning about programs. The language will include array updates, as in the previous example, and array iterators like `map`, `fold`, etcetera. Recent work [30, 13] introduces a new primitive to express array computations using stream operators: the body of a loop over an array processes the stream of individual elements. It remains to be

[3] https://gitlab.inria.fr/synchrone/heptagon

seen how well this idea translates into a formalisation in a proof assistant. Does it allow more modular rules? Does it facilitate proofs? The candidate will build on previous and ongoing work in the PARKAS team and strike a compromise between the expressivity offered to programmers and the complexity of development in a proof assistant.

Compilation A synchronous language compiler transforms the source program step-by-step, first by rewriting into simpler subsets, and then by transformations through a sequence of intermediate languages. This approach is formalized in the Vélus compiler, where the details of the transformations and intermediate languages additionally have an important effect on the invariants, and thus the complexity, of correctness proofs. A central scientific question is to determine the compiler organization and form of intermediate languages (syntax, semantics, and invariants), such that the proofs are as concise and extensible as possible. This is not just important for our academic project, but also for transferring the results to an industrial context.

Rewriting Some operations on arrays can be expressed and verified based on rewriting with equivalence relations in the source language. This is the case with nestings of iterators, like fold, map, etcetera, for example, the "fusion" expressed in the semantic relation

```
map f (map g x) = map (f o g) x
```

potentially allows for the elimination of a loop in generated code. Rewriting can simplify the language subset to be treated by the compiler backend and proving its correctness only requires a single semantics, not different source and target semantics, and local invariants. Its effectiveness, however, depends on finding the right core language subset that carries through to intermediate languages. For instance, the `o` operator in the previous rule implies some way of expressing stream function composition, but the source language only provides certain higher-order operators.

Intermediate Languages Vélus has two main intermediate languages. The first introduces a notion of state, the second fixes the order of computations. This division facilitates the correctness proofs. The question now is how best to generalize each of the languages to handle operations on arrays. It will be necessary to account for array size and type parameters, to express sharing and copying, and to confront the interrelated constraints of memory allocation and scheduling. The compiler transforms the stream functions of the source program into equivalent transition functions with explicit state. In an expression like `map f x`, the `f` is a stream function, for which there is one instance, with its own internal state, for each element of the array `x`. The intermediate languages and the compilation passes must thus account for arrays of internal state. Dynamic allocation, using `malloc`/`free`, is avoided in safety critical systems, which means that specialized allocation schemes must be used within the compiler. It will be crucial to draw on existing work on Lustre, Lucid synchrone, Scade, and Zelus to find a scheme that works well with the intermediate languages to simplify invariants and reduce the number of proof obligations.

The compilation pass of Vélus transforms the second intermediate language into Clight. The current Vélus compiler is defined using the `Vint`, `Vlong`, `Vfloat`, `Vsingle` scalar values from CompCert,[4] adding arrays will inevitably require using the `Vptr` value and reasoning about indirections through the memory model of CompCert. Such questions have previously been resolved on a smaller scale using separation logic predicates [6, §4.2]. It remains to be seen how well this approach can be adapted for the compilation of arrays. It is not yet clear when best to introduce `Vptr` values and the associated memory model. What is the easiest way to reason about shared memory in a verified compiler, knowing that actual memory addresses, alignment, aliasing, and byte-level representations must be introduced at some stage?

Proof complexity The Vélus specification has become quite complicated; for each line of specification there are 10 or more lines of proof. Simply extending the existing prototype with arrays

[4] https://compcert.org/doc/html/compcert.common.Values.html

in a brute force manner would not only be difficult, it would not be very interesting. Instead it will be crucial to restructure the compiler passes as a library of modular and reusable elements, whose formal definitions clarify and assist reasoning about the transformation of functional specifications to efficient embedded code. Just as separation logic provides a means to isolate proofs about aliasing and byte-level representations, so too could we hope to factor out some aspects related to the compilation of arrays. Such a backend would allow also eventually allow for the addition of records, tagged unions, and abstract data types. A good solution would provide a general and independent interface to CompCert, providing a compilation path for different source languages, not just the one formalized in Vélus. A very good solution would provide new solutions and techniques that could also be applied in other contexts.

References

[1] S. Abu-Mahmeed, C. McCosh, Z. Budimlić, K. Kennedy, K. Ravindran, K. Hogan, P. Austin, S. Rogers, and J. Kornerup. Scheduling tasks to maximize usage of aggregate variables in place. In O. de Moor and M. I. Schwartzbach, editors, *Proc. 18th Int. Conf. on Compiler Construction (CC 2009)*, volume 5501 of *LNCS*, pages 204–219, York, UK, Mar. 2009. Springer.

[2] H. G. Baker. Shallow binding makes functional arrays fast. *ACM SIGPLAN Notices*, 26(8):145–147, Aug. 1991.

[3] H. G. Baker, Jr. Shallow binding in Lisp 1.5. *Comms. ACM*, 21(7):565–569, July 1978.

[4] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *Proc. 9th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*, pages 121–130, Tucson, AZ, USA, June 2008. ACM Press.

[5] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *J. Automated Reasoning*, 43(3):263–288, Oct. 2009.

[6] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg. A formally verified compiler for Lustre. In *Proc. 2017 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 586–601, Barcelona, Spain, June 2017. ACM Press.

[7] T. Bourke, L. Brun, and M. Pouzet. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. *Proc. of the ACM on Programming Languages*, 4(POPL):1–29, Jan. 2020.

[8] T. Bourke, P. Jeanmaire, B. Pesin, and M. Pouzet. Verified Lustre normalization with node subsampling. *ACM Trans. Embedded Computing Systems*, 20(5s):Article 98, Oct. 2021. Presented at 21st Int. Conf. on Embedded Software (EMSOFT 2021).

[9] T. Bourke, P. Jeanmaire, and M. Pouzet. Functional stream semantics for a synchronous block-diagram compiler. In *submitted*, 2025.

[10] T. Bourke, B. Pesin, and M. Pouzet. Verified compilation of synchronous dataflow with state machines. *ACM Trans. Embedded Computing Systems*, 22(5s):137:1–137:26, Sept. 2023. Presented at 23rd Int. Conf. on Embedded Software (EMSOFT 2023).

[11] T. Bourke and M. Pouzet. Zélus: A synchronous language with ODEs. In C. Belta and F. Ivancic, editors, *Proc. 16th Int. Conf. on Hybrid Systems: Computation and Control (HSCC 2013)*, pages 113–118, Philadelphia, USA, Apr. 2013. ACM Press.

[12] L. Brun. *Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset*. PhD thesis, PSL University, June 2020.

[13] J.-L. Colaço, M. Mendler, B. Pauget, and M. Pouzet. A constructive state-based semantics and interpreter for a synchronous data-flow language with state machines. *ACM Trans. Embedded Computing Systems*, 22(5s):152:1–152:26, Sept. 2023. Presented at 23rd Int. Conf. on Embedded Software (EMSOFT 2023).

[14] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In W. Wolf, editor, *Proc. 5th ACM Int. Conf. on Embedded Software (EMSOFT 2005)*, pages 173–182, Jersey City, USA, Sept. 2005. ACM Press.

[15] J.-L. Colaço, B. Pagano, and M. Pouzet. Scade 6: A formal language for embedded critical software development. In *Proc. 11th Int. Symp. Theoretical Aspects of Software Engineering (TASE 2017)*, pages 4–15, Nice, France, Sept. 2017. IEEE Computer Society.

[16] J.-L. Colaço, B. Pauget, and M. Pouzet. Polymorphic types with polynomial sizes. In T. Henriksen and A. Sinkarovs, editors, *9th ACM SIGPLAN Int. Workshop onLibraries, Languages and Compilers for Array Programming (ARRAY 2023)*, pages 36–49, Orlando, FL, USA, June 2023. ACM.

[17] J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. In R. Alur and I. Lee, editors, *Proc. 3rd Int. Workshop on Embedded Software (EMSOFT 2003)*, volume 2855 of *LNCS*, pages 134–155, Philadelphia, PA, USA, Oct. 2003. Springer.

[18] Coq Development Team. *The Coq proof assistant reference manual*. Inria, 2020.

[19] J. Feo. *Sisal*, pages 337–386. North Holland, 1992.

[20] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the Sisal language project. *J. Parallel and Distributed Computing*, 10(4):349–366, Dec. 1990.

[21] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet. A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler. In R. Wilhelm, H. Falk, and W. Yi, editors, *Proc. 13th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2012)*, pages 51–60, Beijing, China, June 2012. ACM Press.

[22] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proc. IEEE*, 79(9):1305–1320, Sept. 1991.

[23] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *Proc. 12th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 1985)*, pages 300–314, New Orleans, LA, USA, Jan. 1985. ACM Press.

[24] P. Jeanmaire. *Une sémantique dénotationnelle pour un compilateur synchrone vérifié.* PhD thesis, PSL University, Dec. 2024.

[25] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Proc. 6th Int. Federation for Information Processing (IFIP) Congress 1974*, pages 471–475, Stockholm, Sweden, Aug. 1974. North-Holland.

[26] H. Kanabar, S. Vivien, O. Abrahamsson, M. O. Myreen, M. Norrish, J. Åman Pohjola, and R. Zanetti. PureCake: A verified compiler for a lazy functional language. *Proc. of the ACM on Programming Languages*, 7:Article 145, 2023.

[27] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *Proc. 41st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2014)*, pages 179–191, San Diego, CA, USA, Jan. 2014. ACM Press.

[28] X. Leroy. Formal verification of a realistic compiler. *Comms. ACM*, 52(7):107–115, 2009.

[29] L. Morel. Array iterators in lustre: From a language extension to its exploitation in validation. In *EURASIP Journal of Embedded Systems*, volume 2007, page Article 59130. Springer, Mar. 2007.

[30] B. Pauget. *Memory Specification in a Data-flow Synchronous Language with Statically Sized Arrays.* PhD thesis, PSL Université, Paris, France, Dec. 2023.

[31] B. Pesin. *Verified Compilation of a Synchronous Dataflow Language with State Machines.* PhD thesis, PSL University, Oct. 2023.

[32] F. Rocheteau and N. Halbwachs. POLLUX: A LUSTRE based hardware design environment. In P. Quinton and Y. Robert, editors, *Proc. Int. Workshop onAlgorithms and Parallel VLSI Architectures II*, pages 335–346, Château de Bonas, France, June 1992. Elsevier Science.

[33] G. Shi, Y. Gan, S. Shang, S. Wang, Y. Dong, and P.-C. Yew. A formally verified sequentializer for Lustre-like concurrent synchronous data-flow programs. In *Proc. 39th Int. Conf. on Software Engineering Companion (ICSE-C'17)*, pages 109–111, Buenos Aires, Argentina, May 2017. IEEE Press.

[34] G. Shi, Y. Zhang, S. Shang, S. Wang, Y. Dong, and P.-C. Yew. A formally verified transformation to unify multiple nested clocks for a Lustre-like language. *Science China Information Sciences*, 62(1):article 12801, Jan. 2019.

[35] L.-y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. Interaction Trees: Representing recursive and impure programs in Coq. *Proc. of the ACM on Programming Languages*, 4(POPL):article 51, Jan. 2019.