A Modular Memory Optimization for Synchronous Data-Flow Languages

Application to Arrays in a Lustre Compiler

Léonard Gérard Adrien Guatto Cédric Pasteur Marc Pouzet
DI, École normale supérieure, 45 rue d'Ulm, 75230 Paris, France
Firstname.Name@ens.fr

Abstract

The generation of efficient sequential code for synchronous data-flow languages raises two intertwined issues: control and memory optimization. While the former has been extensively studied, for instance in the compilation of LUSTRE and SIGNAL, the latter has only been addressed in a restricted manner. Yet, memory optimization becomes a pressing issue when arrays are added to such languages.

This article presents a two-level solution to the memory optimization problem. It combines a compile-time optimization algorithm, reminiscent of register allocation, paired with language annotations on the source given by the designer. Annotations express in-place modifications and control where allocation is performed. Moreover, they allow external functions performing *in-place* modifications to be safely imported. Soundness of annotations is guaranteed by a semilinear type system and additional scheduling constraints. A key feature is that annotations for well-typed programs do not change the semantics of the language: removing them may lead to less efficient code but will not alter the semantics.

The method has been implemented in a new compiler for a LUSTRE-like synchronous language extended with hierarchical automata and arrays. Experiments show that the proposed approach removes most of the unnecessary array copies, resulting in faster code that uses less memory.

Categories and Subject Descriptors C.3 [*SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS*]: Real-time and embedded systems; D.3.2 [*Language Classifications*]: Data-flow languages; D.3.4 [*Processors*]: Code generation, Compilers, Optimization

General Terms Algorithms, Languages, Theory

Keywords Real-time systems; Synchronous languages; Block-diagrams; Compilation; Optimization; Semantics; Type systems

1. Introduction

Synchronous data-flow languages [5] are widely used for the design and implementation of embedded systems. The generation of sequential imperative code was addressed more than twenty years ago in the early work on LUSTRE [9] and SIGNAL [6] and is routinely used in industrial tools such as SCADE. Its principle is to generate a transition function that computes a synchronous step of the system, which is then infinitely repeated. For tools like SCADE, code generation is done modularly, producing a single transition function per stream function, independently of the calling contexts [7].

Two critical optimizations have to be performed during the generation of sequential code: *control structure optimization* and *memory optimization*. Control optimization tries to reduce useless code at every reaction according to the value of certain boolean variables. Several methods have been proposed, ranging from local optimizations performed modularly [7] to more aggressive but non-modular ones [15]. In this paper, we focus on the memory optimization problem. It aims to minimize the allocated memory and the number of copy operations when computing a reaction. This becomes an important issue in production compilers like SCADE 6 due to the presence of functional iterators over large arrays [18]. Because these arrays are semantically functional — if t_1 and t_2 are arrays, $t_1 + t_2$ denotes a third array and the update $t_1\{i \leftarrow e\}$ returns a fresh copy whose i-th element is equal to e — the direct translation into sequential code is untenable for performance reasons. Arrays must be shared, with in-place modifications and useless copies eliminated as much as possible. Unfortunately, methods like register reuse [15] and iterator fusion [18] treat the problem only partially and locally to a block [21]. Recently, this problem was addressed by S. Abu-Mahmeed et al. [1] and applied on the data-flow language LABVIEW, but without proposing an interprocedural solution which is essential for good performance.

Contribution and organization of the paper: We address the problem in a different and more unified way by combining a static memory allocation algorithm together with language annotations. The memory allocation is presented as a graph coloring problem like the well-known problem of register allocation [10]. The main novelties are the extension to *clocked streams* and the handling of *synchronous registers*. As the optimization is necessarily fragile, it is coupled with language annotations that give the designer precise control over interprocedural memory sharing. These annotations also allow to safely import external functions performing in-place modifications on their arguments. These annotations do not change the semantics of programs, that is, removing them leads to the same behavior. The soundness of annotations is enforced by a semilinear type system [24] and additional scheduling constraints.

The method has been applied to a LUSTRE-like language, called HEPTAGON, which extends LUSTRE with hierarchical automata and arrays. The material presented here could nonetheless be adapted to similar languages such as SCADE and the discrete subset of SIMULINK.

This article presents the language and memory issues with examples in Section 2. Memory allocation is considered in Section 3.

Language annotations are described in Section 4 and the semilinear type system is formalized in Section 5. The changes induced on code generation are presented in Section 6 together with benchmarks. Future extensions and related work are respectively discussed in Section 7 and Section 8 and we conclude in Section 9.

2. Problem Statement

We informally introduce synchronous data-flow languages with a simple example, and then illustrate the memory issues tackled in the paper.

A simple example with two exclusive blocks. A program is made of a list of declarations of *nodes*, i.e., functions acting on streams. Each node is defined by a set of mutually recursive equations over streams. For example, consider the node `halfSum` given in Figure 1 that takes an input stream `x` and return an output `sum`.

The first equation defines the stream `half`. `fby` is a unit delay initialized with a constant value. It returns its first input concatenated with its second input. Thus, the value of `half` is the alternating sequence *tt* . *ff* . *tt* . *ff* . . . . In the remainder, a variable defined by an equation of the form *se* `fby` *e* will be called a *synchronous register* (to avoid confusion with registers in register allocation).

The `split` operator is used to filter the stream `x` according to the boolean stream `half` (it replaces the `when` operator of LUSTRE). `x1` (resp. `x2`) is the stream made of the values of `x` when `half` is true (resp. false). It is absent otherwise. The `merge` operator joins the complementary streams `sum1` and `sum2`: `sum` is equal to `sum1` (resp. `sum2`) when `half` is true (resp. false). The *clock* of e, written $clock(e)$, defines the instants when the value of e is present. Here, it is a boolean formula of the form [7]:

$$\begin{array}{lcl} ck & ::= & \texttt{base} \mid ck \ \texttt{on} \ c \\ c & ::= & x \mid \texttt{not}\ x \end{array}$$

where `base` stands for the base clock and is interpreted as the constant stream of true values, ck `on` c is true when ck is true and c is present and true. For example in Figure 1a, $clock(\texttt{half}) = \texttt{base}$ and $clock(\texttt{x1}) = \texttt{base on half}$. This notion of clock is also important for our memory allocation algorithm.

Figure 1c shows a simplified version of the sequential code generated from `halfSum` with a main simulation loop. Notice that synchronous registers require special care: their current value is the one computed during the previous activation. That is why their equation is set after the code computing the variables.

Control Optimization. During code generation, an equation $x = e$ is translated into an assignment which is executed only when the clock of e is true. It is important for efficiency to merge computations activated by the same clocks and not generate a separate conditional for each equation. Without this optimization, `x1`, `x2`, `sum1`, `sum2` and `sum` would have used one conditional each. The general form of this transformation is the basis of the compilation of SIGNAL [2].

Memory Optimization. Let us consider a more complex example to illustrate the memory problems that arise when using arrays in a data-flow language. The auxiliary function `swap` takes two indices and an array of size `n` (a global constant) and returns the same array with values at the given indices swapped. The `shuffle` node sequentially applies an array of permutations to an internal array and then returns the value at a given index:

```
const n : int = 100
const m : int = 3
const t_0 : float^n = 0.0^n

node swap(i, j : int; t_in : float^n)
       = (t_out : float^n)
var t_tmp : float^n;
let
   t_tmp = [ t_in with [i] = t_in[>j<] ];
   t_out = [ t_tmp with [j] = t_in[>i<] ];
tel

node shuffle(i_arr, j_arr : int^m; q : int)
          = (v : float)
var t, t_prev : float^n;
let
  t_prev = t_0 fby t;
  t = fold<<m>> swap(i_arr, j_arr, t_prev);
  v = t[>q<];
tel
```

`float^n` is the type of arrays of `float` of size `n` and `0.0^n` is the literal array filled with `n` `0.0` values. `[t with [i] = e]` returns an array equal to `t` except for the element at index `i` which is set to `e`. The language aims at critical systems so no out-of-bounds error is permitted, `t[>i<]` is thus the clipped index array access, returning the element of `t` at index $\min(\max(\texttt{i}, 0), \texttt{n} - 1)$. The `fold` iterator successively applies the `swap` function to the elements of `i_arr` and `j_arr`,[1] using an accumulator whose initial value is `t_prev`, the previous value of `t` initialized to a constant `t_0`.[2]

As the language has a functional semantics, each operation on an array creates a new array. This choice is compatible with block-diagram syntax and the inherently concurrent nature of the language, but makes efficient implementation harder. In `swap`, a naive implementation would allocate new arrays for `t_in` and `t_tmp` and copy the whole array twice. A common optimization in synchronous languages is to store a variable together with its previous value [15]. In `shuffle`, `t` and `t_prev` may thus be stored together removing one unnecessary copy. Finally, the calling convention states that inputs are passed by value, so `m` unnecessary copies are done in the `fold` which calls `m` instances of `swap`.

Memory allocation avoids all copies inside the `swap` and `shuffle` nodes, but keeps the copies induced by the calling convention. The optimal implementation, which allocates only one array for the synchronous register `t_prev` and updates it in-place, is achieved in Section 4 by combining memory allocation with annotations.

3. Memory Allocation

The memory allocation algorithm described in this section is presented as a graph coloring problem like register allocation. Indeed, both problems have similar goals and constraints: to share local variables without changing the semantics of a program. The main novelties of our approach are the extension to clocked streams and the handling of synchronous registers.

We recall the general definition of interference [10]:

Definition 1 (Interference (general)). *Two variables interfere if they cannot be stored in the same memory location.*

This notion has to be adapted to the clocked data-flow setting. We first define live ranges and interference on streams elements, following the usual definitions. The resulting notion of interference cannot be computed statically, so we adapt the definitions to streams, using clocks and the properties of synchronous registers, in order to get an abstract and easily computable definition.

In the following, we assume that synchronous registers are isolated in equations of the form x = v `fby` y by a normalization

[1] If f has type signature $\tau_1 \times ... \times \tau_p \times \tau \longrightarrow \tau$, then $\texttt{fold}\langle n\rangle\, f$ has type signature $\tau_1\texttt{\^{}}n \times ... \times \tau_p\texttt{\^{}}n \times \tau\texttt{\^{}}n \longrightarrow \tau\texttt{\^{}}n$. When `m` equals 2, $t = \texttt{swap}(\texttt{i\_arr}[1], \texttt{j\_arr}[1], \texttt{swap}(\texttt{i\_arr}[0], \texttt{j\_arr}[0], \texttt{t\_prev}))$.

[2] All these operators exist in SCADE 6.

```
node halfSum(x:int)=(sum:int)
var sum1, sum2, x1, x2 :int;
    half :bool;
let
  half = true fby (not half);
  (x1, x2) = split half x;
  sum1 = 0 fby (sum1 + x1);
  sum2 = 0 fby (sum2 + x2);
  sum = merge half sum1 sum2;
tel
```

(a) HEPTAGON code

half	*tt*	*ff*	*tt*	*ff*	*tt*	*ff*	*tt*	...
x	1	7	4	2	9	3	1	...
x1	1		4		9		1	...
sum1	0		1		5		14	...
x2		7		2		3		...
sum2		0		7		9		...
sum	0	0	1	7	5	9	14	...

(b) Chronogram

```
//hS synchronous registers
typedef struct {
  int sum1, sum2;
  bool half;
} hSMem;

//hS transition function
int hSStep(int x, hSMem* m){
  int x1, x2, sum;
  if (m->half) {
    x1 = x; //split
    sum = m->sum1; //merge
    m->sum1 = m->sum1 + x1; //update registers
  } else {
    x2 = x;
    sum = m->sum2;
    m->sum2 = m->sum2 + x2;
  }
  m->half = not m->half; //update registers
  return sum;
}

int main() {
  hSMem m = {0, 0, true}; //register initialization
  while(true)  //simulation loop
    out(hSStep(&m, in()));
}
```

(c) Simplified C code

Figure 1. The `halfSum` node and corresponding generated code

pass. In this case, x *denotes both the stream and the register* so that $is\_reg(x) = true$. We consider every equation of a program as an infinite set of equations, one for each step, defining instantaneous variables. We note $x = (x_i)_{i\in\mathbb{N}}$ for a stream and $eq = (eq_j)_{j\in\mathbb{N}}$ for an equation.

$$eq : x = e \Leftrightarrow \forall i \geq 0.\ eq_i : x_i = e_i$$

$$\begin{aligned} & eq : x = v \texttt{ fby } e \Leftrightarrow \\ & \qquad i = 0.\ eq_0 : x_0 = v \\ & \qquad i > 0.\ eq_i : x_{i+1} = \texttt{if } clock(x)_i \texttt{ then } e_i \texttt{ else } x_i \end{aligned}$$

At each step, the value of a stream is computed if its clock is active. A register is persistent, so its value remains the same even if its clock is not active.

We suppose given a *schedule*, noted $\preceq$, that is a total order on equations compatible with data dependencies. $eq \preceq eq'$ means that eq must be computed before eq'. We note $\prec$ the associated strict order. The rest of the paper does not depend on the precise schedule chosen. We now define the *live range* of a variable, used to compute interference. $\mathrm{def}(x_i)$ is the equation defining x_i and $\mathrm{use}(x_i)$ is the set of equations using x_i.

Definition 2 (Live range)**.** *We say that x_i is* alive *in the equation eq_j, denoted $live(x_i, eq_j)$, if:*

$$live(x_i, eq_j) \triangleq (def(x_i) \prec eq_j) \wedge \left(\exists eq' \in use(x_i).eq_j \preceq eq'\right)$$

Intuitively, a variable is alive between its definition and its last use. In particular, the live range of a register spans over two steps, as its equation defines the value for the next step. Interference can now be defined on streams in almost the same way as in register allocation:

Definition 3 (Interference (dynamic))**.** *Two streams x and y interfere if they are alive in the same instance of an equation:*

$$\exists i, j, eq, k.\ \ live(x_i, eq_k) \wedge live(y_j, eq_k)$$

This definition of interference cannot be computed statically as it depends on the actual values of clocks. For instance, x is never used in the equation $y = \texttt{merge}\ c\ 0\ x$ if c is always false. However, we can compute a static approximation of the live range of a stream, valid at every step, by using its associated clock. Let $def(x)$ be the equation defining the stream x and $use(x)$ the set of equations where x appears on the right-hand side. These two notions are over-approximations of the definitions given on streams elements. In particular, if there exist i and j such that $eq_j \in use(x_i)$, then $eq \in use(x)$, but the converse might not be true, as in the previous example.

Definition 4 (Live range (static))**.** *We say that a stream x is* alive *in eq, denoted $live\_s(x, eq)$, if:*

$$live\_s(x, eq) \triangleq (def(x) \prec eq) \wedge \left(\exists eq' \in use(x).eq \preceq eq'\right)$$

In the `shuffle` example, `t` is alive in the equations defining `t_prev` and `v`. The similarity with Definition 2 makes it easy to see that the live range of a stream is an approximation of the live range of its elements (i.e. $\exists i, k.\ \ live(x_i, eq_k) \Rightarrow live\_s(x, eq)$). We now define the inclusion of clocks and the notion of *disjoint clocks* that approximates the liveness information given by clocks:

Definition 5 (Inclusion of clocks)**.** *A clock ck is included in ck', which is denoted $ck \sqsubseteq ck'$, if:*

$$ck \sqsubseteq ck' \quad \Leftrightarrow \quad \forall i \in \mathbb{N}.\ ck_i = tt \Rightarrow ck'_i = tt$$

Definition 6 (Disjoint clocks)**.** *Two disjoint clocks are never both true during the same step:*

$$dis\_ck(ck_1, ck_2) \triangleq$$
$$\begin{cases} ck_1 = \texttt{base} \vee ck_2 = \texttt{base} & \Rightarrow false \\ ck_1 = ck \texttt{ on } c \wedge ck_2 = ck \texttt{ on not } c & \Rightarrow true \\ ck_1 = ck \texttt{ on } c \wedge ck_2 = ck' \texttt{ on } c' & \Rightarrow dis\_ck(ck, ck') \end{cases}$$

For instance, in the `halfSum` example, streams `x1` and `x2` have disjoint clocks and the clock of `sum1` is included in the clock of `sum`, which is equal to `base`. As a stream is only computed when its clock is true, two variables with disjoint clocks are never both needed during the same step so they never interfere. The value of a synchronous register must be kept even when its clock is not true as it may be used in a future step. As a consequence, a register interferes with any variable whose clock is not included in the clock of the register. Using these two ideas, we define an approximate notion of interference:

Definition 7 (Interference (static))**.** *We say that x and y statically interfere, noted $x \boxtimes y$, if one of them is a register and the clock of the other is not included in the clock of the register, or if they are alive in the same equation and do not have disjoint clocks. Formally:*

$$\begin{aligned} x \boxtimes y \triangleq\ & (is\_reg(x) \wedge clock(y) \not\sqsubseteq clock(x)) \\ & \vee (is\_reg(y) \wedge clock(x) \not\sqsubseteq clock(y)) \\ & \vee (\exists eq.\ live\_s(x, eq) \wedge live\_s(y, eq) \\ & \qquad \wedge \neg\, dis\_ck(clock(x), clock(y))) \end{aligned}$$

In the example `swap`, streams `t_in`, `t_tmp` and `t_out` do not interfere as they are never both alive in the same equation. The inputs and outputs of a node can be addressed similarly by adding two pseudo-equations, that are used only to compute the interference graph and never actually appear in the generated code:

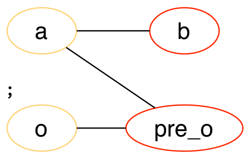
$$eq_{init} : a_1, \ldots, a_p = \mathtt{read\_inputs}()$$
$$eq_{return} : \_ = \mathtt{write\_outputs}(o_1, \ldots, o_q)$$

These equations state that inputs are alive at the beginning of node execution and that outputs are alive at the end of node execution.

Definition 8 (Interference graph)**.** *An* interference graph $G = (V, E, E_a)$ *is an undirected graph where each vertex is associated with one stream and* $(x, y) \in E$ *if and only if* $x \boxtimes y$.

In this example of an interference graph, the `map` operator applies a function to each element of an input array and returns an array of results. We suppose that the schedule $\preceq$ is the one given by the code on the left (i.e $eq_b \preceq eq_o \preceq eq_{pre\_o}$).

```
node p(a:float^n)
    = (o:float^n)
var pre_o, b:float^n;
let
  b = map<<n>>(-.)(a, pre_o);
  o = map<<n>>(+.)(a, b);
  pre_o = t_0 fby o;
tel
```



Normalization. In order to maximize sharing opportunities, memory allocation is done after a normalization pass which creates new equations for temporary results. Indeed, in the `swap` example, `t_in` and `t_tmp` interfere as they are both used to define `t_out`. However, after the normalization, `t_in` and `t_tmp` no longer interfere:

```
  v = t_in[>j<];
  v_2 = t_in[>i<];
  t_tmp = [ t_in with [i] = v ];
  t_out = [ t_tmp with [j] = v_2 ];
```

Algorithm. The algorithm to compute interferences closely follows the definitions. The list of live variables in each equation is computed using Definition 4, then the interference graphs (one for each type) are built using Definition 7. The DSATUR [8] algorithm is used to color the graphs with a minimal number of colors. The result of the algorithm is a set of equivalence classes, where two variables in the same class have the same color and should be stored together.

Memory Allocation, Scheduling and Control Optimization. We defined liveness according to a given schedule $\preceq$: the compiler performs memory allocation after the scheduling pass. The trade-off between scheduling and register allocation is a well-known problem in classic compilation (see [14] for instance). In our setting, the order of these passes is not an issue. Indeed, performing memory allocation before scheduling would be possible by considering only data-flow (or *true*) dependencies, but would always yield inferior results as any valid schedule respects these dependencies and thus induces strictly shorter live ranges.

However, in a clocked data-flow language, scheduling usually optimizes the control structure of the generated program by clustering equations with the same activation clock. This transformation may expand live ranges of streams, thereby limiting memory sharing. We modified the existing scheduling heuristic to favor memory optimization of arrays over control, by greedily minimizing the number of arrays alive at the same time. This heuristic is quadratic in the number of equations.

Memory allocation successfully avoids unnecessary copies within a node, for instance sharing `t_in`, `t_tmp` and `t_out` in `swap`. But the call-by-value convention states that inputs are passed by value, so one copy is made each time `swap` is called. These copies can be avoided if the input `t_in` is passed by reference and modified in-place in the generated code. This can be enforced by using the language annotations presented in the next section.

4. Language Annotations

4.1 Presentation

Location annotations enable the designer to express the in-place update of some inputs. If an input and an output are annotated with the same location, then the generated code will update the input in-place and return nothing.

Example. In the example of Section 2, the designer can express that `t_in` should be modified in-place by annotating `t_in`, `t_tmp` and `t_out` with the same location `r`. This is done using the notation `at r` beside the type declaration:

```
node swap(i, j:int; t_in:float^n at r)
       = (t_out:float^n at r)
var t_tmp : float^n at r;
let
   t_tmp = [ t_in with [i] = t_in[>j<] ];
   t_out = [ t_tmp with [j] = t_in[>i<] ];
tel
```

Only located variables can be given to a function that expects located arguments. To obtain a located variable `t_prev` from a non-located expression `t_0`, the programmer needs to explicitly initialize a new location `r` with the `init` construction:

```
node shuffle(i_arr, j_arr : int^m; q : int)
          = (v : float)
var t, t_prev : float^n at r;
let
  init t_prev = t_0 fby t;
  t = fold<<m>> swap(i_arr, j_arr, t_prev);
  v = t[>q<];
tel
```

As a result, the synchronous register `t_prev` will be updated in-place by `swap` and shared with `t`, so that no unwanted copy occurs.

The memory allocation algorithm is readily adapted to incorporate annotations: all variables with the same annotation are ensured to be stored in the same memory location (i.e., they correspond to the same vertex in the interference graph). However, the algorithm may still choose to share variables even if they are annotated with different locations.

Annotations may express that a function modifies its argument in-place even if it is not returned by the function, e.g.:

```
node f(mat:int^n^n at r) = (o:int)
```

which states that the body of `f` is allowed to overwrite `mat`.

Calling External Functions. Location annotations are also used to safely import external functions that may modify their inputs in-place. For instance, we may import an efficient sorting function (e.g., written in C), with signature `void sort(int a[100])`, that modifies its input in-place. The usual way to achieve this is to add fake variables to enforce the necessary dependencies. Using location annotations, the function can safely be imported as:

```
fun sort(a : int^100 at r) = (o:int^100 at r)
```

Annotations* vs *side-effects. In this proposal, we have chosen to maintain the block-diagram formalism, using annotations that can always be erased. Annotations are only used to control the efficiency of generated code. The semantics of a program with correct annotations remains the same if all annotations are removed. An alternative could have been to introduce mutable imperative variables, explicit side-effects and sequence in the source language, and take side-effects into account in the semantics.

4.2 Checking Annotations

Location annotations given by the programmer are unsound if two streams associated with the same location interfere. Annotated equations must satisfy well-formedness rules expressed as a type system and be statically schedulable.

We use a *semilinear* type system, following the work of Wadler in [24]: *a value of semilinear type can be read multiple times and then updated once*. We call *update* an operator or function that deliberately modifies its argument in-place. For instance, physically modifying one element in an array (`[t with [i] = v]`) or calling the `swap` node are updates. A semilinear variable is defined either by updating a semilinear variable at the same location or by explicitly initializing a new location using the keyword `init`. The correctness of the annotations, that is, that two variables with the same semilinear type do not interfere, relies on the three following properties.

Property 1 (Init)**.** *A location is only initialized once.*

This is ensured by a simple syntactic check before typing that ensures that all the locations used in the inputs or with `init` are distinct.

Property 2 (Causality)**.** *If y results from an update of x, then the equation defining y is the last use of x with respect to $\preceq$.*

After its update, a semilinear variable cannot be read since its value has been overwritten, so it has to be dead (according to the schedule $\preceq$). The scheduling algorithm is modified in Section 4.3 to enforce this property.

Property 3 (Type soundness)**.** *If two streams are associated with the same location, either one is obtained by successive updates from the other or they are obtained by updates from two variables defined by the application of the* `split` *operator.*

This property relies on the type system presented in Section 5. In the first case, the streams are alive one after the other, while in the second, they have disjoint clocks. In both cases, they do not interfere. These three properties are essential to the correctness theorem for the system of annotations:

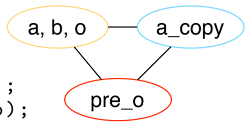
Theorem 4.1 (Annotation soundness)**.** *Two variables associated with the same location do not interfere.*

An annotated program is correct if it is well-typed (Property 3), schedulable (Property 2) and its locations are initialized only once (Property 1). A sketch of the proof of these properties and of Theorem 4.1 is given in Appendix B.

4.3 Scheduling

In order to ensure Property 2, static scheduling occurs after semilinear type checking. Extra dependencies between equations are added so that the update of a semilinear variable happens after all reads. This may introduce cycles, making scheduling impossible. For instance, in the node `p`, if variables `a`, `b` and `o` are annotated with the same location, the equation defining `o` reads `a`, so it should be scheduled before the equation defining `b` which updates `a` (i.e. $eq_o \prec eq_b$), but it also reads `b`, so it needs to be scheduled after the equation defining `b` (i.e. $eq_b \prec eq_o$). This can be fixed by introducing a copy of `a` (i.e., `a_copy = a`):

```
node p(a:float^n at r) = (o:float^n at r)
var b:float^n at r;
    a_copy, pre_o:float^n;
let
  a_copy = a;
  b = map<<n>>(-.)(a, pre_o);
  o = map<<n>>(+.)(a_copy, b);
  pre_o = t_0 fby o;
tel
```



The copy is not added automatically as we want to enforce the invariant that all streams at the same location are shared without any hidden copy.

5. Semilinear Type Checking

This section formalizes the semilinear type system that enforces the soundness property stated in Property 3. The system is used as a type checker, that is, with no type inference. For the sake of simplicity, we present a semilinear type system on a reduced version of the synchronous data-flow kernel used as an intermediate language during compilation.

5.1 Types

We define $\mathcal{R}_\top \triangleq \mathcal{R} \cup \{\top\}$, where $\mathcal{R}$ is the set of locations and $\top$ a special location representing the absence of information. We write by convention $r \in \mathcal{R}$ and $\rho \in \mathcal{R}_\top$. We denote by τ `at` r a semilinear type associated with the location r and by τ `at` $\top$ a non-linear type. Static expressions (se) are either values (v) or global constants (s). A plain type (τ) in the language is either a basic type or an array type. A type (μ) is given by a plain type and a location. We also define a node signature (σ):

$$se ::= v \mid s \qquad\qquad \tau ::= \texttt{int} \mid \texttt{float} \mid \texttt{bool} \mid \tau\texttt{\^{}}se$$
$$\mu ::= \tau \ \texttt{at} \ \rho \qquad\qquad \sigma ::= \forall r, \dots, r. \mu^P \longrightarrow \mu^Q$$

An *update* is a function having an input and an output with the same semilinear type τ `at` r. We write $P \triangleq [1\,..\,p]$, $\mu^P \triangleq \mu_1 \times \cdots \times \mu_p$ and $x_P : \mu_P \triangleq x_1 : \mu_1, \dots, x_p : \mu_p$ (likewise for any letter). We will also assume that $\mu_i \triangleq \tau_i$ `at` ρ_i.

5.2 Abstract Syntax

$$\begin{aligned}
n &::= \texttt{node}\ f(p; \dots ; p) = (p; \dots ; p)\ \texttt{var}\ p, \dots , p\ \texttt{let}\ D\ \texttt{tel} \\
eq &::= p = e \mid (p, \dots , p) = f(w, \dots , w) \\
&\quad \mid p = se\ \texttt{fby}\ w \mid (p, p) = \texttt{split}\ (x)\ x \\
&\quad \mid \texttt{init}\langle r \rangle\ p = se\ \texttt{fby}\ w \mid \texttt{init}\langle r \rangle\ p = e \\
w &::= x \mid se \\
e &::= w \mid op(w, \dots , w) \mid \texttt{merge}\ (x)\ w\ w \\
D &::= eq \mid D\ ;\ D \\
p &::= x : \mu
\end{aligned}$$

In this kernel, function arguments are extended values (w), either variables (x) or static expressions (se). A simple normalization pass can put any program into this form by introducing new local variables and equations. Although redundant, types also appear on the left-hand side of equations in order to simplify the presentation of the type system.

5.3 Typing Rules

The global and local typing environments, respectively written Δ and Γ are defined by ($\uplus$ stands for the union of multisets):

$$\Delta ::= \emptyset \mid \Delta \cup \{f : \sigma\} \mid \Delta \cup \{s : \tau\}$$
$$\Gamma ::= \emptyset \mid \Gamma \uplus \{x : \mu\}$$

The typing judgments are:

$$\Delta, \Gamma \vdash e : \mu \quad \Delta \vdash se : \tau \quad \Delta \vdash f : \sigma \quad \Delta, \Gamma \vdash b \quad \Delta, \Gamma \vdash D$$

which respectively mean that the expression e has type μ, the static expression se has type τ, the function f has signature σ and the block b or equations D are well-typed, in the global and local environments Δ and Γ.

The typing rules are given in Figure 2. The size of some rules comes from the presence of n-ary functions returning multiple values, as in most block diagram languages. The most important rules are the ones that express the linearity properties (Figure 2a).

1. The VAR rule is common to all linear type systems: it shows that each occurrence of x in the source corresponds to one and only one occurrence of x in the local environment, which is a multiset.
2. WEAKENING allows the removal of unnecessary elements from the environment.
3. The COPY and LINEAR COPY rules show the difference between semilinear and non-linear variables. For a non-linear variable x (of type τ `at` $\top$), we can duplicate its occurrence in the environment as much as we want, in order to use them as arguments for multiple reads. Conversely, the semilinear occurrence of a semilinear variable y (of type τ `at` r), cannot be duplicated. It is used in the typing rule of its only update. We can nevertheless create other occurrences of the same y with a non-linear type, in order to use them for multiple reads.
4. There are two ways to define a semilinear variable. The first one is to apply an update to another variable of the same type with the EQUATION rule. This is the case for instance for `t_tmp` and `t_out` in the `swap` example. The second one consists in initializing a new location from a non-linear variable with INIT.
5. The INITFBY rule ensures a correct use of semilinear synchronous registers. As two synchronous registers should always interfere, they can never have the same semilinear type. Except for the presence of `init`$\langle r \rangle$, this rule is the same as the application of an update, writing the value used in the next instant. The presence of `init` attests that the location r is initialized by the register with the value of the previous instant. Looking back at the `shuffle` example, it is clear that, in order to be able to modify the synchronous register `t_prev` in-place, it has to be defined as an update of `t`, which is itself an update of the previous value of `t_prev`. The location `r` is initialized at the first instant by `t_0`.
6. The MERGE rule uses a single local environment to type its arguments (unlike APP for instance) as we know that they have disjoint clocks. The SPLIT rule creates two variables with the same semilinear type, but it is safe since they have disjoint clocks.
7. The EQLIST rule conforms to the equational nature of our data-flow kernel: equation ordering does not matter and the type system is thus independent from scheduling.
8. The NODE rule states the constraints that a node signature must respect. Locations used in the inputs (resp. the outputs) must be distinct from each other (WF2) (resp. (WF3)). A node application cannot create a location: locations appearing in the outputs must appear in the inputs (WF1). Semilinear outputs can only be read (and not updated) as their value is needed at the end of the step, so they are added to the local environment with a non-linear type τ'_Q `at` $\top$.

Array Operators. Semilinear typing extends to array operators: ($\gamma \leq \rho$ stands for $\rho \neq \top \Rightarrow \gamma = \rho$)

ARRAYUPDATE
$$\frac{\Delta, \Gamma \vdash w : \tau\texttt{\^{}}n \texttt{ at } \rho \qquad \Delta, \Gamma_1 \vdash w_1 : \texttt{int} \qquad \Delta, \Gamma_2 \vdash w_2 : \tau \texttt{ at } \top}{\Delta, \Gamma \uplus \Gamma_1 \uplus \Gamma_2 \vdash x : \tau\texttt{\^{}}n \texttt{ at } \rho \texttt{ = } [w \texttt{ with } [w_1] = w_2]}$$

MAP
$$\frac{\begin{array}{c}\Delta \vdash f : (\tau_i \texttt{ at } \rho_i)^P \longrightarrow (\tau'_j \texttt{ at } \rho'_j)^Q \\ \sigma = (\tau_i\texttt{\^{}}n \texttt{ at } \gamma_i)^P \longrightarrow (\tau'_j\texttt{\^{}}n \texttt{ at } \gamma'_j)^Q \\ \texttt{well\_formed}(\sigma) \qquad \forall i.\ \gamma_i \leq \rho_i \qquad \forall j.\ \gamma'_j \leq \rho'_j\end{array}}{\Delta \vdash \texttt{map}\langle n \rangle\ f : \sigma}$$

FOLD
$$\frac{\Delta \vdash f : \tau_1 \texttt{ at } \top \times \ldots \times \tau_p \texttt{ at } \top \times \tau \texttt{ at } \rho \longrightarrow \tau \texttt{ at } \rho \qquad \gamma \leq \rho}{\Delta \vdash \texttt{fold}\langle n \rangle\ f : \tau_1\texttt{\^{}}n \texttt{ at } \top \times \ldots \times \tau_p\texttt{\^{}}n \texttt{ at } \top \times \tau \texttt{ at } \gamma \longrightarrow \tau \texttt{ at } \gamma}$$

The ARRAYUPDATE rule shows that modifying one element of an array can either be done in-place for semilinear variables (if $\rho \neq \top$) or possibly with a copy for other variables ($\rho = \top$).

The MAP and FOLD rules state all the possible signatures according to f. Map applies f to each element of its input arrays, so we can modify them in-place. However, if f modifies one of its inputs in-place, the corresponding array has to be modified in-place. Fold iterates f over the accumulator (the last argument), which may be modified in-place. It has to if f requires it. The other arguments are only read.

6. Implementation and Experiments

The material presented here on a kernel language has been implemented in the compiler of a richer synchronous language called HEPTAGON. The language allows the mixing of data-flow equations with hierarchical automata [11]. Automata are eliminated by a source-to-source translation into the data-flow kernel. The language also supports a comprehensive set of operators on arrays [18] and a simple form of parametricity compiled to C by macro-expansion. Apart from memory allocation, it implements two traditional optimizations for synchronous data-flow programs: iterator fusion [18] and data-flow minimization.[3] The type checker is implemented in a simple, syntax-directed manner.

6.1 Sequential Code Generation

Following [7], the data-flow kernel is first translated into a small imperative intermediate language called OBC. The translation from OBC to existing sequential languages is then straightforward. Backends for C and JAVA have been implemented.

In OBC, the transition function of a node f is encapsulated with its internal state which stores the values of the synchronous registers from f. This encapsulation, called `machine`, is made of a list of state variables (declared with the `registers` keyword), a list of instances of other machines used by the machine (introduced by `instances`) and a `step` method for the transition function. The body of the transition function is expressed in a simple imperative

[3] Data-flow minimization generalizes Common Subexpression Elimination. E.g., equations $x = 1\ \texttt{fby}\ x + 1$ and $y = 1\ \texttt{fby}\ y + 1$ reduce to a single one.

$$\textsc{Var}\quad \frac{}{\Delta, \{x : \mu\} \vdash x : \mu} \qquad\qquad \textsc{Weakening}\quad \frac{\Delta, \Gamma \vdash e : \mu'}{\Delta, \Gamma \uplus \{x : \mu\} \vdash e : \mu'}$$

$$\textsc{Copy}\quad \frac{\Delta, \Gamma \uplus \{x : \tau\ \texttt{at}\ \top\} \uplus \{x : \tau\ \texttt{at}\ \top\} \vdash e : \mu}{\Delta, \Gamma \uplus \{x : \tau\ \texttt{at}\ \top\} \vdash e : \mu} \qquad \textsc{Linear Copy}\quad \frac{\Delta, \Gamma \uplus \{y : \tau\ \texttt{at}\ r\} \uplus \{y : \tau\ \texttt{at}\ \top\} \vdash e : \mu}{\Delta, \Gamma \uplus \{y : \tau\ \texttt{at}\ r\} \vdash e : \mu}$$

(a) Linearity rules

$$\textsc{Const}\quad \frac{\Delta \vdash se : \tau}{\Delta, \emptyset \vdash se : \tau\ \texttt{at}\ \top} \qquad \textsc{Block}\quad \frac{\Delta, \Gamma \uplus \{x_L : \mu_L\} \vdash D}{\Delta, \Gamma \vdash \texttt{var}\ x_L : \mu_L\ \texttt{let}\ D\ \texttt{tel}} \qquad \textsc{EqList}\quad \frac{\Delta, \Gamma \vdash D \qquad \Delta, \Gamma' \vdash D'}{\Delta, \Gamma \uplus \Gamma' \vdash D\ ;\ D'} \qquad \textsc{Equation}\quad \frac{\Delta, \Gamma \vdash e : \tau\ \texttt{at}\ \rho}{\Delta, \Gamma \vdash x : \tau\ \texttt{at}\ \rho \texttt{ = } e}$$

$$\textsc{Init}\quad \frac{\Delta, \Gamma \vdash e : \tau\ \texttt{at}\ \top}{\Delta, \Gamma \vdash \texttt{init}\langle r\rangle\ x : \tau\ \texttt{at}\ r \texttt{ = } e} \qquad \textsc{Fby}\quad \frac{\Delta, \Gamma \vdash w : \tau\ \texttt{at}\ \top \qquad \Delta, \emptyset \vdash se : \tau\ \texttt{at}\ \top}{\Delta, \Gamma \vdash x : \tau\ \texttt{at}\ \top \texttt{ = } se\ \texttt{fby}\ w} \qquad \textsc{InitFby}\quad \frac{\Delta, \Gamma \vdash w : \tau\ \texttt{at}\ r \qquad \Delta, \emptyset \vdash se : \tau\ \texttt{at}\ \top}{\Delta, \Gamma \vdash \texttt{init}\langle r\rangle\ x : \tau\ \texttt{at}\ r \texttt{ = } se\ \texttt{fby}\ w}$$

$$\textsc{Merge}\quad \frac{\Delta, \Gamma \vdash w_1 : \tau\ \texttt{at}\ \rho \qquad \Delta, \Gamma \vdash w_2 : \tau\ \texttt{at}\ \rho \qquad \Delta, \Gamma' \vdash x : \texttt{bool}}{\Delta, \Gamma \uplus \Gamma' \vdash\ \texttt{merge}\ (x)\ w_1\ w_2 : \tau\ \texttt{at}\ \rho} \qquad \textsc{Split}\quad \frac{\Delta, \Gamma \vdash y : \mu \qquad \Delta, \Gamma' \vdash x : \texttt{bool}}{\Delta, \Gamma \uplus \Gamma' \vdash (y_1 : \mu, y_2 : \mu) \texttt{ = split } (x)\ y}$$

(b) Expression and Equation rules

$$\textsc{App}\quad \frac{\Delta \vdash f : \mu^P \longrightarrow \mu'^Q \qquad (\Delta, \Gamma_i \vdash w_i : \mu_i)_{i=1\ldots p}}{\Delta, \Gamma' \uplus \biguplus_{1\ldots p} \Gamma_i \vdash (x_Q : \mu'_Q) \texttt{ = } f(w_1, \ldots, w_p)} \qquad \textsc{Node}\quad \frac{\begin{array}{c}\Delta, \{x_P : \mu_P\} \uplus \{x'_Q : \tau'_Q\ \texttt{at}\ \top\} \vdash b \\ \sigma = gen(\ \mu^P \longrightarrow \mu'^Q) \qquad \texttt{well\_formed}(\sigma)\end{array}}{\Delta \vdash \texttt{node}\ f(x_P : \mu_P) \texttt{ = } (x'_Q : \mu'_Q)\, b : \sigma}$$

$$\textsc{Inst}\quad \frac{\begin{array}{c}\Delta(f) = \forall r_J.\mu^P \longrightarrow \mu'^Q \\ \forall i, j \in J.\ r'_i = r'_j \Rightarrow i = j\end{array}}{\Delta \vdash f : \mu^P \longrightarrow \mu'^Q\ [r_J \leftarrow r'_J]} \qquad \textsc{Gen}\quad \frac{\{r'_i\}_{i=1\ldots j} = \{\rho_i \mid i = 1 \ldots p \wedge \rho_i \neq \top\}}{gen(\mu^P \longrightarrow \mu'^Q) = \forall r'_J.\mu^P \longrightarrow \mu'^Q}$$

$$\texttt{well\_formed}(\forall r_J.\mu^P \longrightarrow \mu'^Q) \triangleq \forall i, j, k. \begin{cases} \rho'_j \neq \top \Rightarrow \exists i.\ \mu_i = \mu'_j & \text{(WF1)} \\ \rho_i = \rho_k \neq \top \Rightarrow i = k & \text{(WF2)} \\ \rho'_j = \rho'_k \neq \top \Rightarrow j = k & \text{(WF3)} \end{cases}$$

(c) Function-related rules

Figure 2. Semilinear Typing Rules

language. Figure 3a (respectively 3c) shows the OBC code (respectively C code) corresponding to the translation of node `shuffle`, without memory optimization.

Expressing sharing in the intermediate sequential code. In the original version of OBC [7], programs were forced to be in *Static Single Assignment* (SSA) form with all arguments of a method passed by value. In order to be able to share a location, we added *mutable* variables that can be assigned multiple times and *mutable* inputs that are passed by reference.

The result of the memory allocation described in Section 3 is a set of equivalence classes, where two variables in the same class must be stored together. Sharing is applied by a modular source-to-source transformation in OBC. A representative is chosen in each equivalence class (either an input or synchronous register if there is one, otherwise any variable). All other variables in the equivalence class are replaced by this representative and unused variables are removed. An input shared with an output becomes mutable (to express the in-place modification) and the output is removed. Finally, node calls have to take into account the removed outputs. For instance, in the `shuffle` node, `t_next` is chosen as the representative for `t` and `t_next`, and it is passed by reference to `swap`, that does not return anything after the transformation. Figures 3a and 3b show the OBC code before and after the transformation (the `swap` node without optimization is in Appendix A). In the end, all the updates are performed in-place in the synchronous register.

6.2 Experiments

The graphs in Figure 4 show both the effects of memory optimization alone and combined with annotations on the generated step function. The figures are given relatively to the unoptimized results.

We use the CompCert [17] 1.9.1 C compiler to generate PowerPC code, and compute worst-case execution times (WCET) with the Open Tool for Adaptative WCET Analysis.[4]

As the `shuffle` example showed, annotations are essential as many unnecessary copies are made when iterating over arrays. Thanks to them, the generated code performs no array copies and is thus much faster with memory optimization. The program is tested with an array of size 50.

The second example sorts an array of size n in n^2 steps by swapping two elements at each step. Here, although the coloring done by memory allocation is optimal in terms of the number of colors, i.e., in terms of memory used (as seen in the second graph), it awkwardly shares arrays. Annotations are used to force one coloring which removes one unnecessary array copy.

The third example is a simplified version of a radar control panel (about 1 kLOC), adapted from one of SCADE demos.[5] Even though the program uses only small arrays (of size 2 to 6) and records, the use of annotations still results in performance improvements.

[4] The tool is available at `http://otawa.fr`.

[5] The Mission Computer demo is available at: `http://www.esterel-technologies.com/technology/demos`

```
const n = 100
const m = 3
const t_0 = 0.0^n

machine shuffle =
  registers t_prev:float^n = t_0;
  instances swap:swap[m];

  step(i_arr, j_arr:int^m; q:int) = (v:float) {
    var t:float^n;
    t = this.t_prev;
    for i = 0 to m-1 do
       t = swap[i].step(i_arr[i], j_arr[i], t);
    this.t_prev = t;
    v = t[between(q, n)];
  }
```

(a) OBC without memory optimization

```
machine swap =
  step(i, j:int; mutable t_in:float^n) = () {
    var v_2, v:float;
    v = t_in[between(i,n)];
    v_2 = t_in[between(j,n)];
    if (i<n && i<=0) t_in[i] = v_2;
    if (j<n && 0<=j) t_in[j] = v;
  }

machine shuffle =
  registers mutable t_prev: float^n = t_0;
  instances swap: swap[m];

  step(i_arr, j_arr:int^m, q:int) = (v: float) {
    for i = 0 to m-1 do
      swap[i].step(i_arr[i], j_arr[i], this.t_prev);
    v = this.t_prev[between(q,n)];
  }
```

(b) OBC with memory optimization

```
#define between(idx, n)\
  (((idx)>=(n))?(n)-1:((idx)<0?0:(idx)))
static const int n = 100;
static const int m = 3;
struct shuffle_mem {float t_prev[100];};
struct swap_out {float t_out[100];};

float shuffle_step(const int i_arr[3], const int j_arr[3],
             int q, struct shuffle_mem* this) {
  float t[100];
  struct swap_out out;
  for (int i_1 = 0; i_1 < n; ++i_1)
    t[i_1] = this->t_prev[i_1];
  for (int i = 0; i < m; ++i) {
    swap_step(i_arr[i],j_arr[i],t,&out);
    for (int i_2 = 0; i_2 < n; ++i_2)
      t[i_2] = out.t_out[i_2];
  }
  for (int i_3 = 0; i_3 < n; ++i_3)
    this->t_prev[i_3] = t[i_3];
  return t[between(q, n)];
}
```

(c) C code without memory optimization

```
struct shuffle_mem {float t_prev[100];};

void swap_step(int i, int j, float t_in[100]) {
  float v_2, v;
  v = t_in[between(i, n)];
  v_2 = t_in[between(j, n)];
  if (i<n && 0<=i) t_in[i] = v_2;
  if (j<n && 0<=j) t_in[j] = v;
}

void shuffle_init(struct shuffle_mem* this) {
  for (int i = 0; i < n; ++i)
    this->t_prev[i] = 0.000000;
}

float shuffle_step(const int i_arr[3], const int j_arr[3],
             int q, struct shuffle_mem* this) {
  for (int i = 0; i < m; ++i)
    swap_step(i_arr[i], j_arr[i], this->t_prev);
  return this->t_prev[between(q, n)];
}
```

(d) C with memory optimization

Figure 3. A node and the corresponding generated code

The last example is a simple downscaling image filter. It mainly consists of repeated vector-style computations on pixels, represented as floating-point arrays of size 4. Here, annotations give a small time boost, and provide negligible improvements in memory occupancy.

Note that the optimization is performed both on structured and scalar variables, but that the impact of the latter on execution times and memory use are negligible. However, the generated code is shorter and more readable, both in terms of instruction and variable counts. The last example, composed of multiple nested automata typical of the industrial use of SCADE, illustrates this point. As the program is composed of one complex monolithic node, our annotations did not give any extra benefits.

7. Discussion

Semilinear typing. A more natural approach to annotations would have been to treat location annotations as coloring instructions for the interference graph, without any constraints, and report an error if coloring fails. While it may appear simpler, the drawback is that fixing incorrect annotations would require an understanding of the interferences and the choices made by the scheduler. On the contrary, the type system we have considered is independent from memory allocation. It is useful even without memory allocation, e.g., to manually express in-place modifications or import external functions with side-effects.

Extensions. Most of the additional features of the HEPTAGON language are implemented by source-to-source transformations to a data-flow kernel close to the one presented in Section 5.2. Performing memory allocation on this kernel is simpler while retaining all possibilities for optimization. This is not surprising since this kernel shares much similarity with the SSA form used in compilers such as GCC [19]. The only change needed is a simple extension of the notion of disjoint clocks to share synchronous registers between states of an automaton separated by a reset. The compiler is also able to share fields inside a record, by treating them individually in the interference graph. The changes required to adapt the semilinear type system to the full language are also minimal.

8. Related Work

The problem of eliminating array copies, also known as the *aggregate update problem* [16], is shared by all functional languages. As a consequence, many solutions to this problem have been proposed. They can be divided into three families. The first relies on data structures at run time, such as a *garbage collector* or *reference counting*. In the special case of arrays, one can use *persistent arrays* [12], where only modifications to an array are stored instead of copying the whole array.

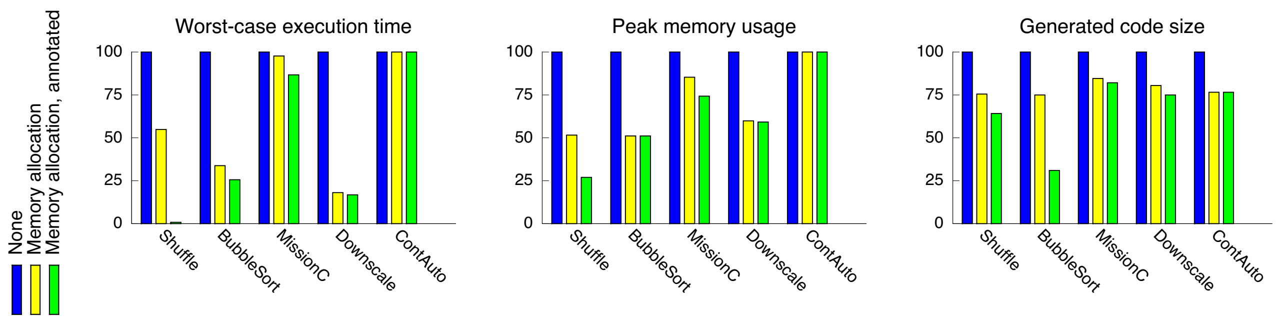


Figure 4. Experimental results: worst-case execution time, maximum memory use and generated code size (lower is better)

The second family of solutions tries to to tackle the problem at compile time. Static analyses try to find the last use of variables, to know when an update can safely be done in-place. This information on the live-range of variables can be found using heuristics [22], abstract interpretation [20] or through Hindley-Milner type inference [3]. All these methods are coupled with a dynamic system such as reference counting to deal with cases that cannot be decided statically. Another related method is *deforestation* [23], which eliminates temporary data structures by transforming the code. All these static optimizations are fragile and do not allow direct control on memory sharing, as it is possible with explicit annotations.

The third family of solutions uses type systems to only accept programs where memory can be reused. They are based on *linear logic* [13]: a variable of linear type can only be used once, and can thus be updated in-place. This restriction of a single use is too strong to be used in a real programming language. Many proposals have been made to relax it whilst maintaining strong enough invariants to enable memory sharing. One solution is to syntactically limit a scope where a linear variable can be considered as non-linear [24], or to mix linear and non-linear types in the language, as in *uniqueness* typing [4]. The type system presented here is based on the same principles, the main novelty being the use of locations, which is made necessary by the presence of n-ary functions.

The choices made here, in particular in terms of calling convention, are similar to those made in [22], although our formalism is more generic. The closest work is that of S. Abu-Mahmeed et al. [1] and applied to LabView. They propose a greedy algorithm that chooses successively, using a notion of cost, an operation to do in-place until dependencies make it impossible to choose another one. Our approach is more general in that it not only focuses on in-place modifications but that it can also share unrelated variables. In addition, we also propose a solution to interprocedural memory optimization. However, their notion of cost could be used to improve our greedy scheduling algorithm.

In the field of data-flow synchronous languages, a classic memory optimization consists in storing `pre` x and x in the same memory location [15]. This can be done if all the reads of `pre` x occur before the definition of x. In our formalism, it implies that x and `pre` x do not interfere, so this optimization is a particular case of the more general approach presented here.

9. Conclusion

This paper has presented a method for optimizing memory when compiling synchronous data-flow programs to sequential code. The method combines a static memory allocation algorithm with explicit language annotations. Memory allocation is expressed as a graph coloring problem, which links it to the classic register allocation problem. The soundness of annotations is checked by a semilinear type system and additional scheduling constraints. This ensures that annotations do not change the original functional semantics of the language but only its efficient code generation. A possible extension is the automatic inference of the annotations.

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 *CC '09: Proceedings of the 18th International Conference on Compiler Construction*, pages 204–219, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] T. Amagbegnon, L. Besnard, and P. Le Guernic. Implementation of the Data-flow Synchronous Language SIGNAL. In *Programming Languages Design and Implementation (PLDI)*, pages 163–173. ACM, 1995.

[3] H. G. Baker. Unify and conquer. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 218–226, New York, NY, USA, 1990. ACM.

[4] E. Barendsen and S. Smetsers. Uniqueness type inference. In *PLILPS '95: Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, pages 189–206, London, UK, 1995. Springer-Verlag.

[5] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, pages 64–83, 2003.

[6] A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[7] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 121–130, New York, NY, USA, 2008. ACM.

[8] D. Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, 1979.

[9] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM.

[10] G. J. Chaitin, M.A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47 – 57, 1981.

[11] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.

[12] P. F. Dietz. Fully persistent arrays (extended array). In *WADS '89: Proceedings of the Workshop on Algorithms and Data Structures*, pages 67–74, London, UK, 1989. Springer-Verlag.

[13] J.-Y. Girard. Linear logic. *Theoretical computer science*, 50(1):1–102, 1987.

[14] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 442–452, New York, NY, USA, 1988. ACM.

[15] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau, Germany, August 1991.

[16] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 300–314, New York, NY, USA, 1985. ACM.

[17] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52:107–115, July 2009.

[18] L. Morel. Array Iterators in Lustre: From a Language Extension to Its Exploitation in Validation. *EURASIP Journal on Embedded Systems*, 2007.

[19] D. Novillo. TreeSSA a new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, 2003.

[20] M. Odersky. How to make destructive updates less destructive. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–36, New York, NY, USA, 1991. ACM.

[21] B. Pagano. The optimization of iterators and updates for functional arrays in scade 6. Personal communication, June 2010.

[22] P. Schnorf, M. Ganapathi, and J. L. Hennessy. Compile-time copy elimination. *Softw. Pract. Exper.*, 23(11):1175–1200, 1993.

[23] P. Wadler. Deforestation: transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248, Amsterdam, The Netherlands, 1988. North-Holland Publishing Co.

[24] P. Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

A. `swap` without memory optimization

```
machine swap =
  step(i: int, j: int, t_in: float^n)
    = (t_out: float^n) {
    var v_2: float; v: float; t_tmp: float^n;
    v_2 = t_in[between(j, n)];
    v = t_in[between(i, n)];
    if (i<n && 0<=i) {
      for i_4 = 0 to i-1 do
        t_tmp[i_4] = t_in[i_4]
      t_tmp[i] = v_2;
      for i_5 = i+1 to n-1 do
        t_tmp[i_5] = t_in[i_5]
    } else {
      t_tmp = t_in
    }
    if (j<n && 0<=j) {
      for i_2 = 0 to j-1 do
        t_out[i_2] = t_tmp[i_2]
      t_out[j] = v;
      for i_3 = j+1 to n-1 do
        t_out[i_3] = t_tmp[i_3]
    } else {
      t_out = t_tmp
    }
  }
```

B. Proof of correctness of the annotation system

Definition 9 (Root). *We call x the* root *of location r if $x : \tau$ `at` r and x is either an input or a variable defined by* `init`$\langle r \rangle$ $x = e$. *We denote* `root`$(r) = x$.

Property 4 (Init). *There is only one root for each location r.*

Proof. By definition of `init`. □

Property 5. *If x is semilinear and is a register, then x is a root.*

Proof. See InitFby rule. □

Definition 10 (Update relation). *We say that x is an* update *of y, denoted $y \rhd x$, if $x, y : \tau$ `at` r and one of the following applies :*

- $(x_1, \ldots, x_q)$ = $f(w_1, \ldots, w_p)$ *with* $f : (\tau_i\ \texttt{at}\ \rho_i)^P \longrightarrow (\tau'_j\ \texttt{at}\ \rho'_j)^Q$, $x_i = x$, $w_j = y$ *and* $r_i = r'_j$.
- x = y
- x = `merge` $(c)\ w_1\ w_2$ *with* $w_j = y$
- $(x_1 : \mu, x_2 : \mu)$ = `split` $(c)\ y$ *with* $x_i = x$

It should be noted that there is no case corresponding to the InitFby rule.

Definition 11 (Update order). *We define the (partial) order $\trianglerighteq$ as the smallest reflexive transitive antisymmetric relation such that $y \rhd x \Rightarrow y \trianglerighteq x$.*

Property 6. *If x is semilinear, then x is either a root or there exists a y such that x is an update of y.*

Proof. By induction on the typing rules. □

Property 7. *If $x : \tau$ `at` r then* `root`$(r) \trianglerighteq x$.

Property 8 (Type soundness). *If two variables are associated with the same location, either one is obtained by successive updates from the other or they are obtained by updates from two variables resulting of a* `split`. *Formally, if $x, y : \tau$ `at` r then one of the following condition is true:*

- *$y \trianglerighteq x$ (resp. $x \trianglerighteq y$).*
- *There exists $z, z_x, z_y : \tau$ `at` r such that $z \trianglerighteq x$, $z \trianglerighteq y$, $z_x \trianglerighteq x$, $z_y \trianglerighteq y$, $z_x \not\trianglerighteq y$, $z_y \not\trianglerighteq x$ and (z_x, z_y) = `split` $(c)\ z$ (or the opposite).*

Proof. If $y \trianglerighteq x$ (resp $x \trianglerighteq y$), then y (resp x) is the maximum we are looking for. Otherwise, let's denote $S = \{z \mid z \trianglerighteq x \wedge z \trianglerighteq y\}$. We know that `root`$(r) \in S$. There exists z such that `root`$(r) \rhd z$. If z is still in S, we can iterate with z. Eventually, we find z_0 such that $z_0 \not\trianglerighteq x$ and $z_0 \trianglerighteq y$ (or reciprocally) (we know that we have not encountered x or y, otherwise we would have been in one of the first two cases). We also know that there exists $z_x, z_y : \tau$ `at` r such that $z_x \trianglerighteq x$ and $z_y \trianglerighteq y$ and (z_x, z_y) = `split` $(c)\ z_0$ (or the opposite), as this is the only case where two variables can be updates of a variable. □

Property 9 (Causality). *If x is an update of y, then the equation defining x is the last use of y:*

$$y \rhd x \Rightarrow \forall eq \in \mathrm{use}(y).\ eq \preceq \mathrm{def}(x)$$

Proof. If $y \rhd x$, then $\mathrm{def}(x)$ is the only update of y and its last use, as guaranteed by the modified scheduling algorithm (see Section 4.3). □

Property 10. *If $y \trianglerighteq x$ then x and y do not interfere.*

Proof. If $y \trianglerighteq x$ then there exists $y_1, \ldots, y_m$ such that $y \rhd y_1 \ldots \rhd y_m \rhd x$. Then $\forall eq \in \mathrm{use}(y).\ \mathrm{use}(y).\ eq \preceq \mathrm{def}(y_1) \preceq \mathrm{use}(y_1) \preceq \ldots \preceq \mathrm{def}(x)$ by applying m times Property 9 and by transitivity of $\preceq$. It means that $\forall eq.\mathrm{live\_s}(y, eq) \Rightarrow \neg\mathrm{live\_s}(x, eq)$, so x and y do not interfere. □

Theorem B.1 (Annotation soundness). *Two variables associated with the same location do not interfere:*

$$\exists \tau, r.\ x, y : \tau\ \texttt{at}\ r \Rightarrow \neg\,(x \boxtimes y)$$

Proof. Let $x, y : \tau$ `at` r.

Case 1 x and y are registers.
This is impossible as a semilinear register is the unique root of its location (Property 4 and 5).

Case 2 y is a register, x is not.
Then y is the root of location r so $y \trianglerighteq x$. By Property 10, we have that x and y do not interfere.

Case 3 x and y are not registers.

- Either $y \trianglerighteq x$ (or reciprocally) then x and y do not interfere (Property 10).

- Or there exists z, z_x, z_y such that $z, z_x \trianglerighteq x$ and $z, z_y \trianglerighteq y$ and (z_x, z_y) = `split` $c\ z$ by Property 3. Then we can show that x (resp. y) is on the same clock or a slower clock than z_x (resp. z_y), which proves that x and y have disjoint clocks. As they are not registers, it follows that x and y do not interfere.

$\square$