Memory optimisation in a first-order dataflow synchronous language

Cédric Pasteur

PARKAS team, Laboratoire d'Informatique de l'Ecole Normale Supérieure, Paris

November 29th, 2010

Contents

Contents

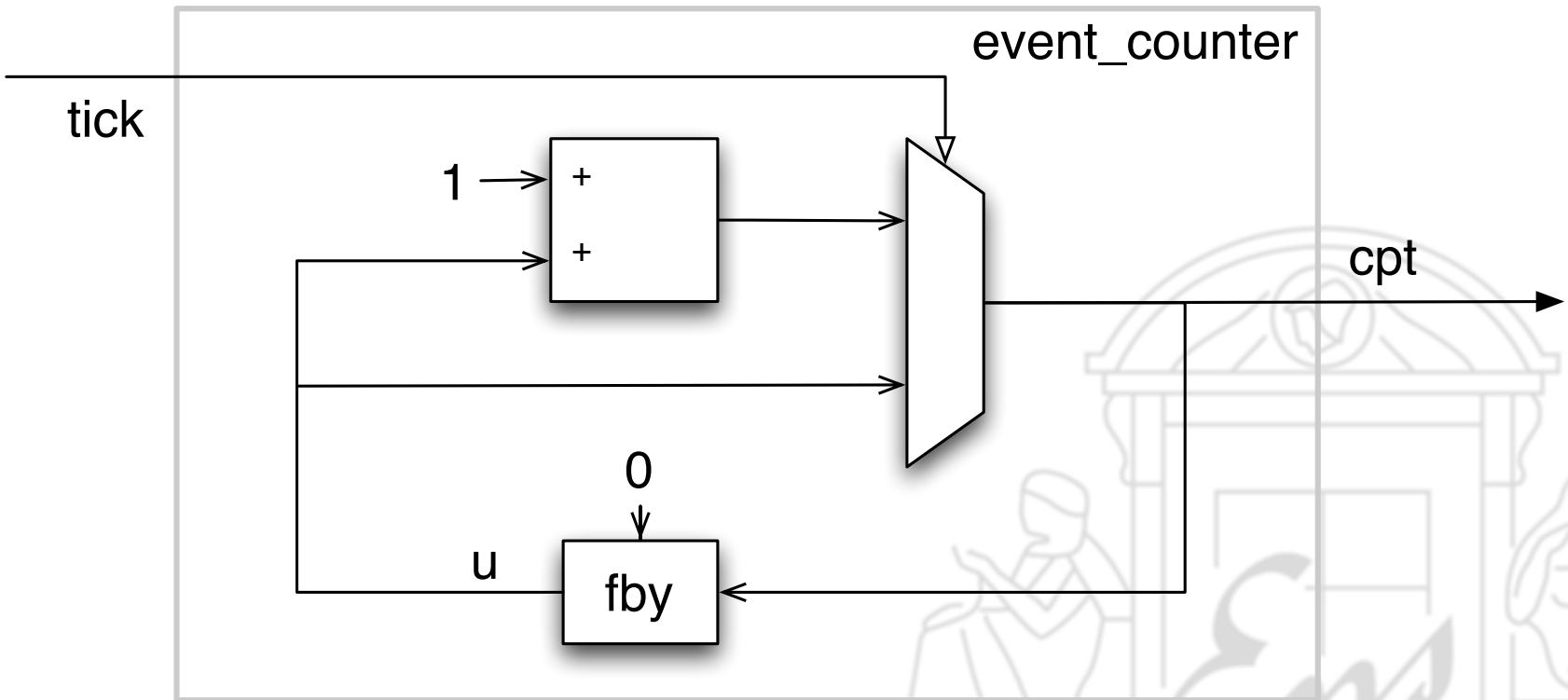
Dataflow synchronous languages

- Academic languages : Lustre, Signal, etc.
- Industrial languages : Scade, LabView, etc.
- Example : a counter

```
node event_counter(tick:bool)
 returns (cpt:int)
var u : int;
let
  u = 0 -> pre cpt;
  cpt = if tick then u + 1 else u;
tel
```

tick	true	false	true	true	...
cpt	1	1	2	3	
pre cpt	nil	1	1	2	
0 -> pre cpt	0	1	1	2	

Visual representation



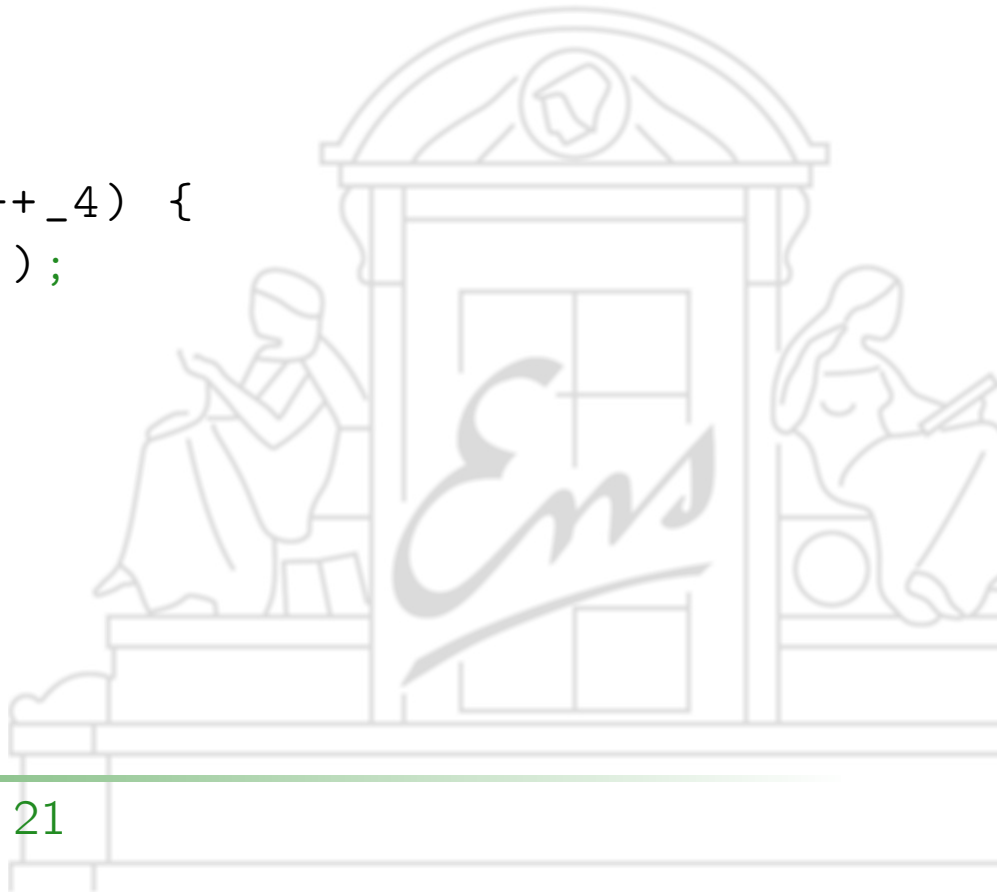
Example : Iterators

The following equations :

```
u = map inc <<n>>(a);
x = mean(u);
```

are compiled to :

```
for (int _4 = 0; _4 < n; ++_4) {
    u[_4] = inc_step(a[_4]);
}
x = mean_step(u);
```



Example : one modification at a time

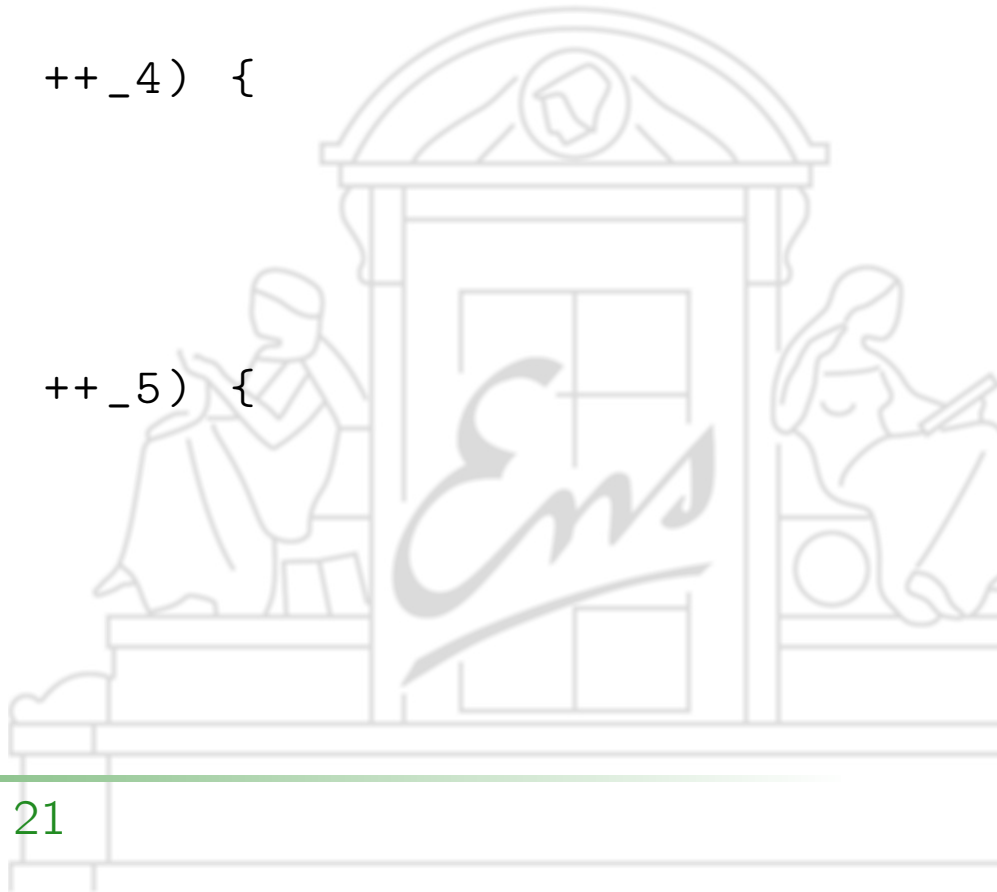
The following equations :

```
u = a with [0] = 0;
v = u with [99] = 0;
```

are compiled to :

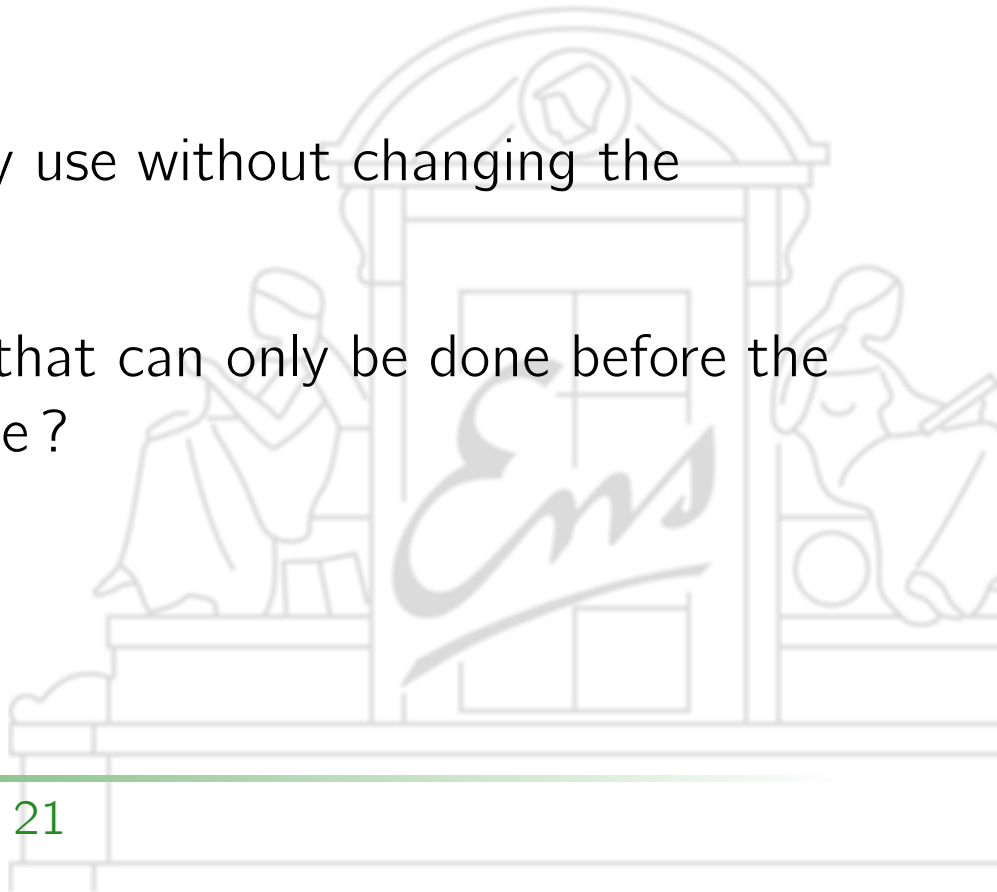
```
for (int _4 = 0; _4 < 100; ++_4) {
    u[_4] = a[_4];
}
u[0] = 0;

for (int _5 = 0; _5 < 100; ++_5) {
    v[_5] = u[_5];
}
v[99] = 0;
```



Eliminating copies

- Problem raised by Esterel Technologies engineers
- Generation of code for safety-critical embedded real-time systems (Scade)
- How do we optimise memory use without changing the language ?
- What are the optimisations that can only be done before the translation to sequential code ?



Traditional solutions

- Problem shared by all functional languages (aka *aggregate update problem* or *update-in-place problem*)
- Static optimisation : deforestation [Gill et al., 1993], operator fusion[Morel, 2005], live range analysis, heuristics [Schnorf et al., 1993],[Abu-Mahmeed et al., 2009]
- Typing : linear or quasilinear types [Wadler, 1990], [Barendsen and Smetsers, 1995], [Kobayashi, 1999]
- Runtime data structures : persistent arrays [Chuang, 1992], garbage collector, reference counting [Scholz, 2003]

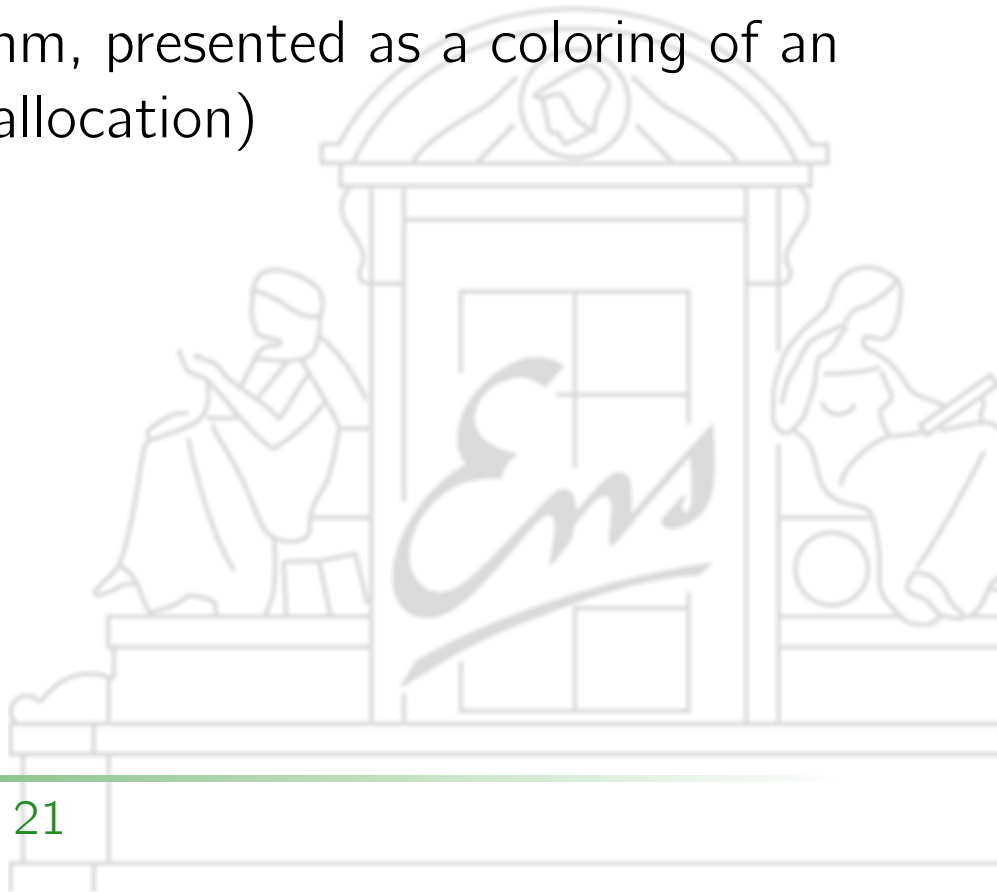
Traditional solutions

- Problem shared by all functional languages (aka *aggregate update problem* or *update-in-place problem*)
- Static optimisation : deforestation [Gill et al., 1993], operator fusion[Morel, 2005], live range analysis, heuristics [Schnorf et al., 1993],[Abu-Mahmeed et al., 2009]
- Typing : linear or quasilinear types [Wadler, 1990], [Barendsen and Smetsers, 1995], [Kobayashi, 1999]
- Runtime data structures : persistent arrays [Chuang, 1992], garbage collector, reference counting [Scholz, 2003]
  $\Rightarrow$ Unusable in real-time systems

Our approach

We combine the first two techniques :

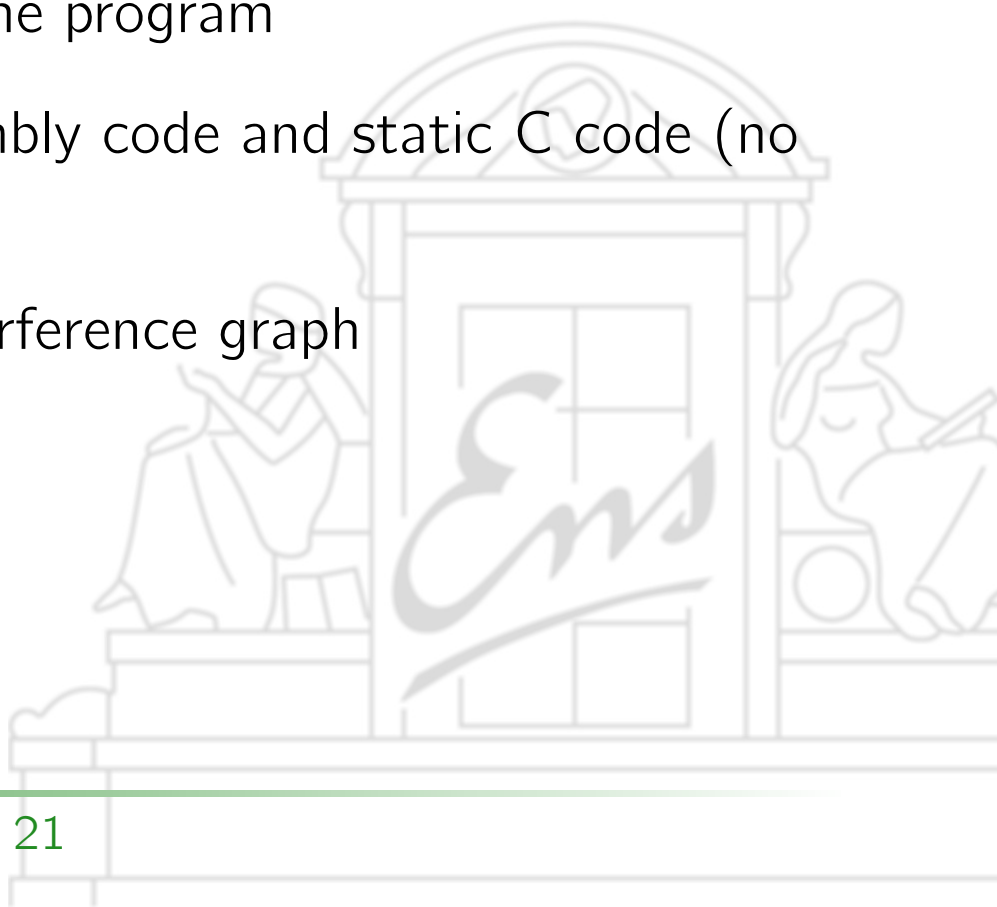
- A static optimisation algorithm, presented as a coloring of an interference graph (register allocation)
- A *semilinear* type system



Contents

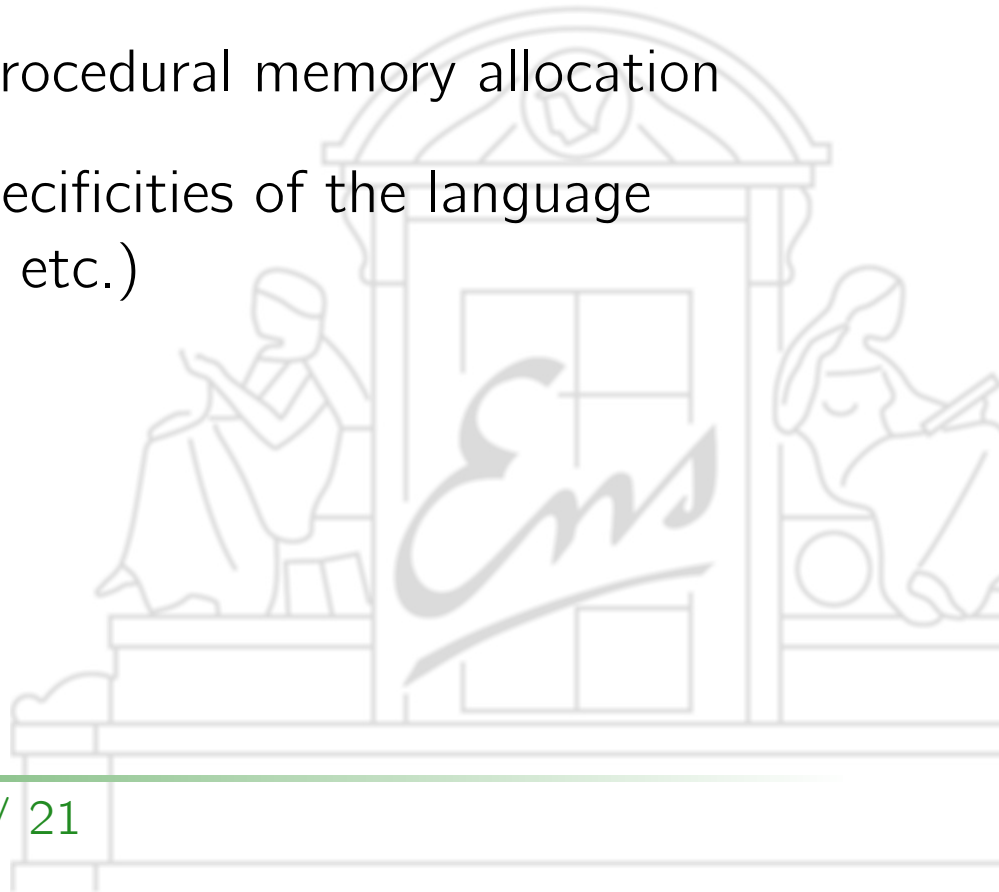
Register allocation and memory optimisation ?

- Same goal : share variables as much as possible without changing the semantics of the program
- Similar constraints for assembly code and static C code (no dynamic memory allocation)
- Method : coloring of an interference graph



Memory allocation

- Definition of interference for a clocked dataflow language
- Calling convention for interprocedural memory allocation
- Algorithm adapted to the specificities of the language (registers, iterators, records, etc.)



Contents

Motivation

The semilinear type system makes the user aware of hidden copies and gives her means to influence the memory allocation algorithm.

Property : No hidden copies

All variables with the same semilinear type correspond to a single memory location.

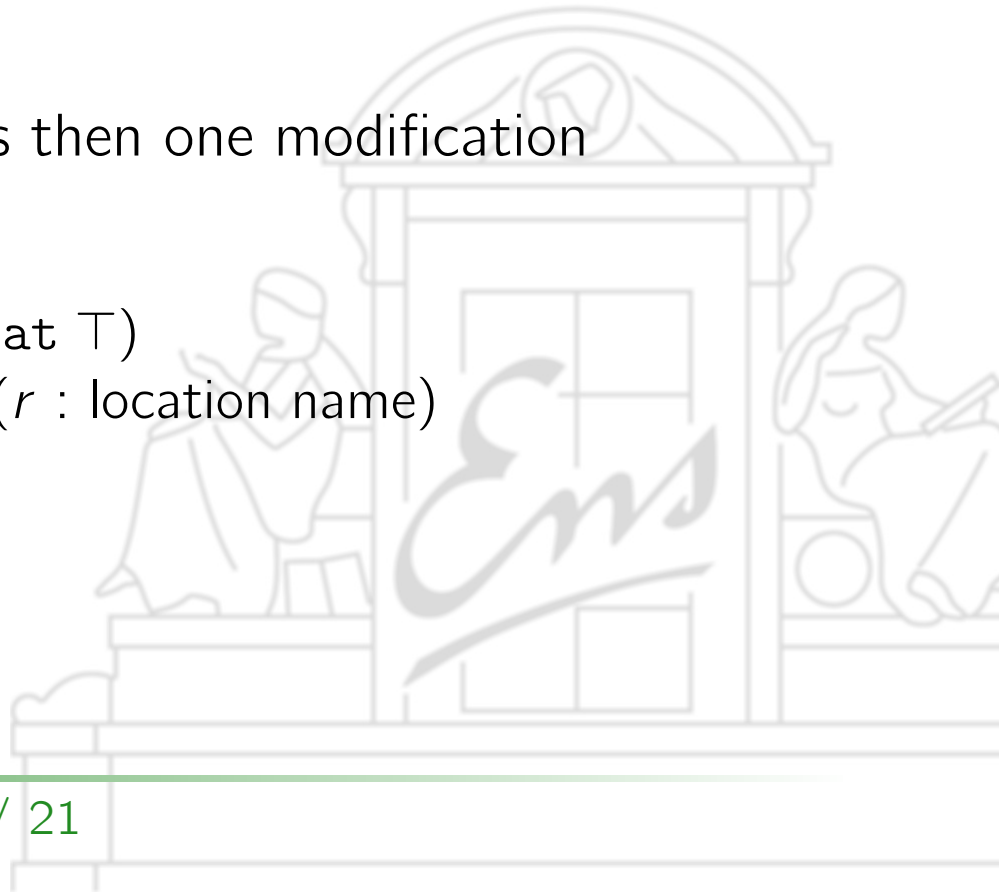
Property : Same semantics

The semantics of a program is the same if we erase all the typing annotations.

Overview

(Linear types [Wadler, 1990], Uniqueness typing
[Barendsen and Smetsers, 1995], Quasi linear types [Kobayashi, 1999])

- Linear type : one use
- *Semilinear* type : many reads then one modification
- Notation :
  - Not linear types : τ (or τ `at` $\top$)
  - *Semilinear* types : τ `at` r (r : location name)



Functions

- *Semilinear* types : many **reads** then one **modification**
- **Read** : function reading a variable
- **Modification** : function modifying (in place) a variable of semilinear type (can be an imported function with side effects)
- Examples :

```
(*read*)
x = t[idx] = select<<idx,n>>(t);
(*modification*)
q = [t with [idx] = v] = update<<n>>(t, idx, v);
```

$$\text{select} : [i, n : \texttt{int}].\ \tau[n] \longrightarrow \tau$$

$$\text{update} : [n : \texttt{int}, r : \texttt{loc}].\ \tau[n]\ \texttt{at}\ r \times \texttt{int} \times \tau \longrightarrow \tau[n]\ \texttt{at}\ r$$

- *Semilinear* type : **many** reads then **one** modification

Var

$$\frac{}{\Delta \mid x : \mu \vdash x : \mu \mid \emptyset}$$

Weakening

$$\frac{\Delta \mid \Gamma \vdash e : \mu' \mid S}{\Delta \mid \Gamma, x : \mu \vdash e : \mu' \mid S}$$

Copy

$$\frac{\Delta \mid \Gamma, x : \tau, x : \tau \vdash e : \mu \mid S}{\Delta \mid \Gamma, x : \tau \vdash e : \mu \mid S}$$

Linear Copy

$$\frac{\Delta \mid \Gamma, x : \tau \ \mathtt{at}\ r, x : \tau \vdash e : \mu \mid S}{\Delta \mid \Gamma, x : \tau \ \mathtt{at}\ r \vdash e : \mu \mid S}$$

Sub

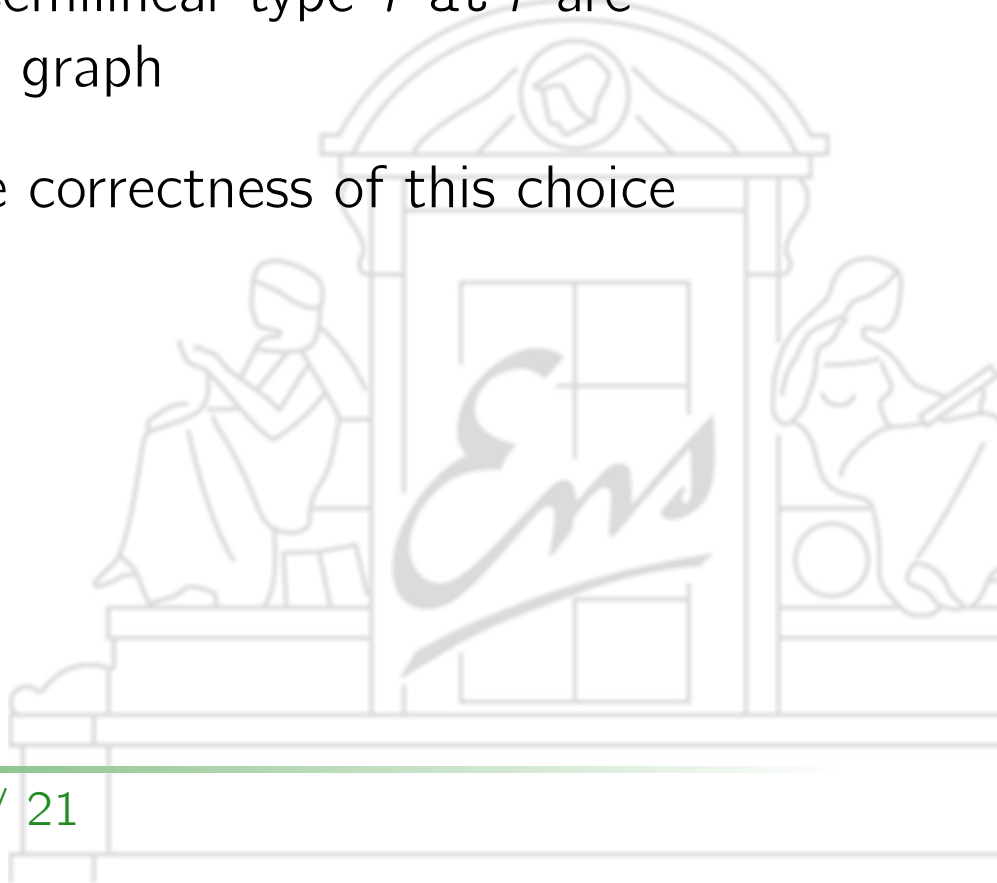
$$\frac{\Delta \mid \Gamma \vdash e : \tau \ \mathtt{at}\ r \mid S}{\Delta \mid \Gamma \vdash e : \tau \mid S}$$

- *Semilinear* types : many reads **then** one modification
- The modification must be done after the reads

```
node (t:int^n at r)
  returns (t2: int^n at r; v:int)
let
  t2 = t with [0] = 0;
  v = t[3] + t2[3];
tel
```

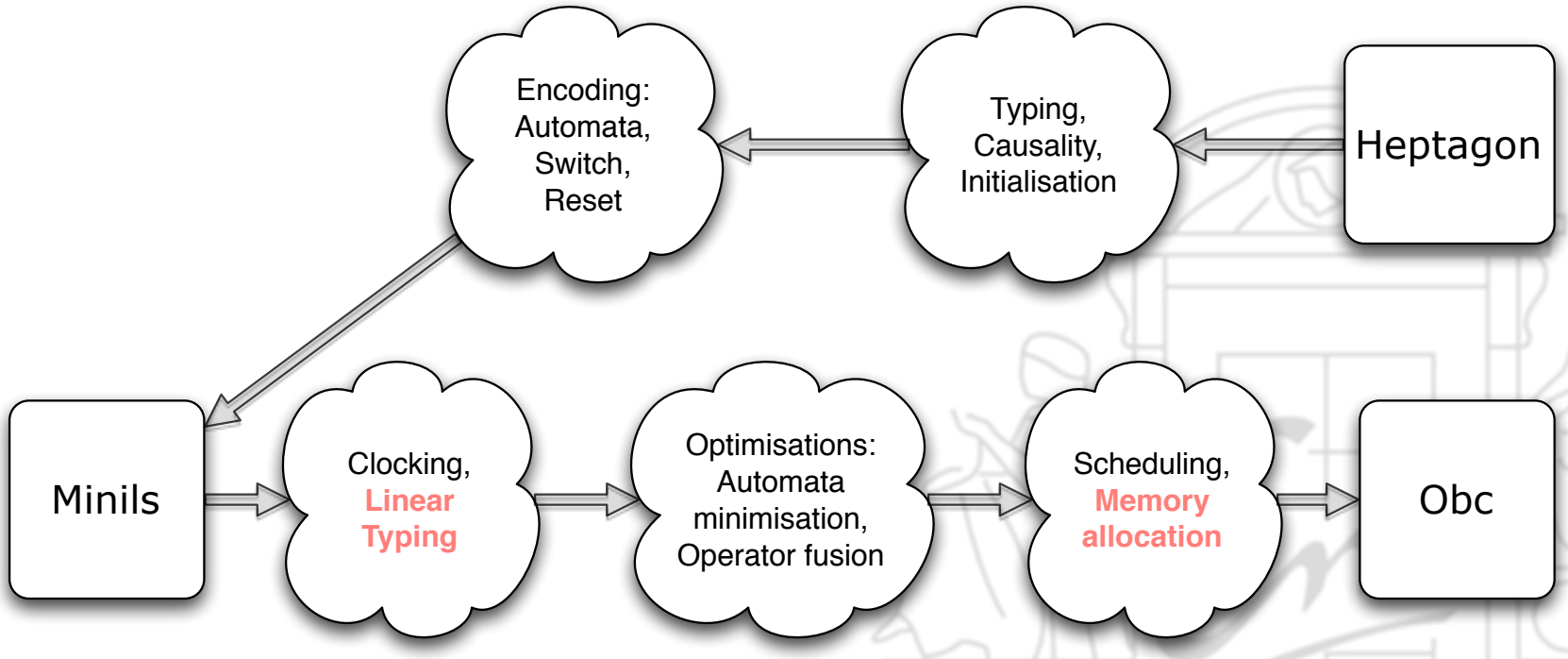
Relation with memory allocation

- All variables with the same semilinear type τ `at` r are coalesced in the interference graph
- The type system ensures the correctness of this choice



Contents

- Functional prototype on the Heptagon language (about 2000 lines of OCaml for typing and memory allocation)



Example : Code generated without optimisation

```
int compute_step(compute_mem* self) {
  int z;
  int x[100];
  int y[100];
  int o[100][100];
  for (int _4 = 0; _4 < 100; ++_4) {
    y[_4] = 0;
  };
  for (int _5 = 0; _5 < 100; ++_5) {
    x[_5] = y[_5];
  };
  for (int i_17 = 0; i_17 < 100; ++i_17) {
    f_step(self->m1[i_17], x, &self->_2[i_17]);
    for (int _6 = 0; _6 < 100; ++_6) {
      o[i_17][_6] = self->_2[i_17].o[_6];
    };
    for (int _7 = 0; _7 < 100; ++_7) {
      x[_7] = self->_2[i_17].acc_out[_7];
    };
  };
  for (int _8 = 0; _8 < 100; ++_8) {
    for (int _9 = 0; _9 < 100; ++_9) {
      self->m1[_8][_9] = o[_8][_9];
    };
  };
  z = o[2][2];
  return z;
}
```

Example : Code generated with optimisation

```
int compute_step(compute_mem* self) {
  int z;
  int r_16[100];
  for (int _4 = 0; _4 < 100; ++_4) {
    r_16[_4] = 0;
  };
  for (int i_25 = 0; i_25 < 100; ++i_25) {
    f_step(self->r_17[i_25], r_16, &self->_2[i_25]);
  };
  z = self->r_17[2][2];
  return z;
}
```

- Half the memory, twice as fast

Why don't we let gcc do it ?

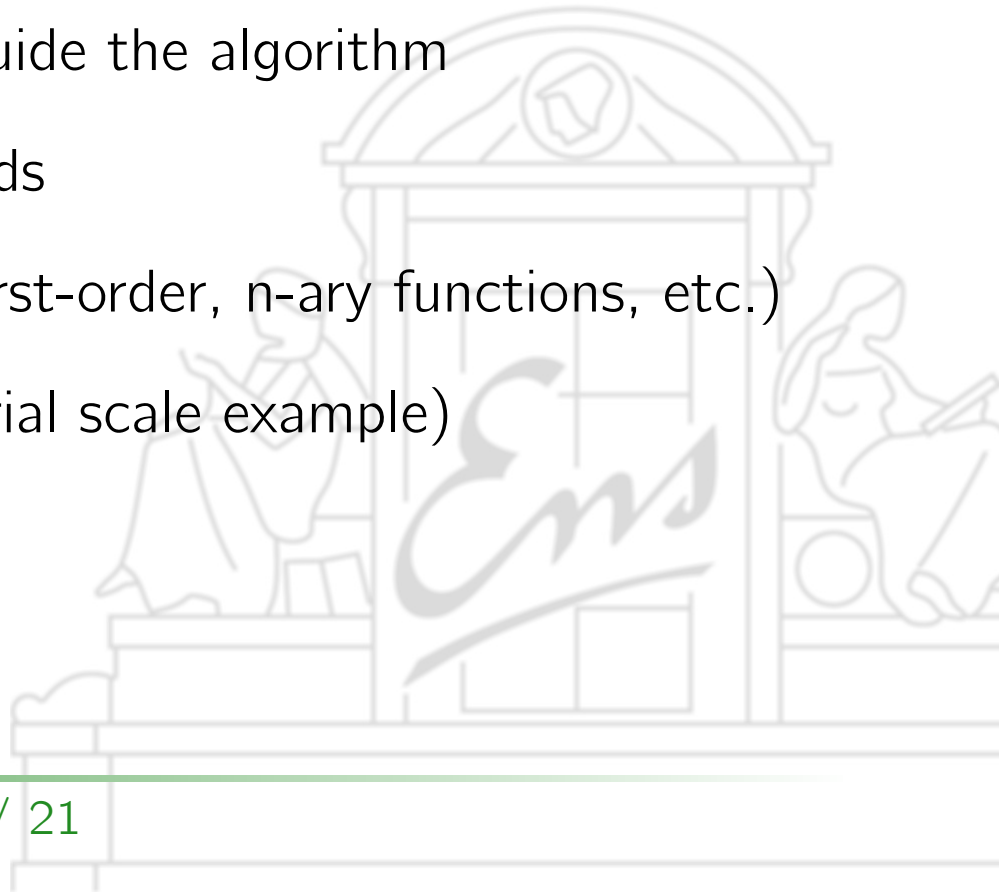
- Compilation hides information
- gcc does *register* allocation, but nothing on arrays
- Stack space is limited (about 8 MB)
- Compilers used for safety-critical systems are not as good as gcc

Table: Result with optimisation with respect to the result without optimisation (percentage)

	Memory	Time
Single threaded computation on an array	25	100
Iteration on a matrix	50	50
Cellular automata	32	0
Only base types (with -O0)	100	105
Only base types (with -O2)	100	160

Conclusion

- A static optimisation algorithm expressed as a coloring of an interference graph
- Semilinear type system to guide the algorithm
- Reuse of traditionnal methods
- Adapted to the language (first-order, n-ary functions, etc.)
- Good results (but no industrial scale example)



Bibliography I

Abu-Mahmeed, S., Mccosh, C., Budimlić, Z., Kennedy, K., Ravindran, K., Hogan, K., Austin, P., Rogers, S., and Kornerup, J. (2009).
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. Springer-Verlag.

Bibliography II


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


Chuang, T.-R. (1992).
Fully persistent arrays for efficient incremental updates and voluminous reads.
In *ESOP '92 : Proceedings of the 4th European Symposium on Programming*, pages 110–129, London, UK. Springer-Verlag.

Bibliography III


Gill, A., Launchbury, J., and Peyton Jones, S. L. (1993).
A short cut to deforestation.
In *FPCA '93 : Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232, New York, NY, USA. ACM.


Kobayashi, N. (1999).
Quasi-linear types.
In *POPL '99 : Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 29–42, New York, NY, USA. ACM.


Morel, L. (2005).
Exploitation des structures régulières et des spécifications locales pour le développement correct de systèmes réactifs de grande taille.
PhD thesis, Verimag.


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


Scholz, S.-B. (2003).
Single assignment c : efficient support for high-level array operations in a functional setting.
J. Funct. Program., 13(6) :1005–1059.


Wadler, P. (1990).
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.