Type Safety for Sizes without Hindrance [1]

Marc Pouzet

ENS/INRIA, Paris

Workshop SYNCHRON
Aussois, France
Nov. 2025

[1] Joint work with Jean-Louis Colaço, Baptiste Pauget, Loic Sylvestre.

Vivre sans temps mort, jouir sans entraves [2]

[2] Mustapha Khayati, 1968

The problem

- Write a generic definition $f\langle p \rangle$ of a system that depends on a parameter p that is ultimately known at compile-time.
- The definition may be partial, i.e., undefined in some cases.
- When should the compiler detect it and complain?

Such examples exist when the target is hardware and software.

Hardware target: E.g., a n-bit adder, a memory decoder, an FFT can be defined as a recursive function parameterized by a size.

What happen if the recursion does not terminate? if the sizes of the two bit vectors of an adder are not equal?

Software target: E.g., operations on arrays using map/fold and update take sized arrays; the modeling of a railway interlocking system in TECLA/HLL is parameterized by an array of railway components.

What to do if an array is accessed out-of-the-bound? A recursion does not terminate?

Several solutions exist.

Templates, compilation by evaluation, meta-programming:

Templates and macros: mark expressions that must be evaluated at compile time; evaluate them; then compile.

E.g., Lisp macros, C++ templates.

Reduce at compile time expressions that only depend on compile-time values. E.g.,VHDL and Verilog.

The compile-time evaluation may fail and produce a invalid circuit.

Define a DSL (Domain Specific Language) embedded in ML.

E.g., Lava, Hawk, Chisel.

Type safety for the elaboration phase; this elaboration may stop.

Meta-programming: the elaboration phase is well typed and produce a well typed program.

E.g., meta-ML, meta-OCaml.

Dependent properties (e.g., related to sizes) are not ensured unless a dependent type system is used for the meta-language.

Pros and Cons

Pros: very expressive! the evaluation language can be turing complete; statically typed (e.g., Haskell, Ocaml, Scala).

Cons: the elaboration phase may fail.

Errors are detected very late.

A function definition may have no use at all.

```
let f = fun i -> let x = [||] in x.(i)
```

Dependent properties are difficult to ensure on the generated program.

Exploit features of the type system of the source language.

E.g., polymorphism, GADT, high-rank polymorphism, polymorphic recursion.

```
val map : ('a -> 'b) -> ('a, 'n) array -> ('b, 'n) array
```

```
let a = [|1; 2; 3|] : (int, zero succ succ succ) array
```

Use a more expressive language for meta-programming. E.g., Coq, Lean, Agda, F*, Liquid Haskell.

You may need to write the proof that a function is total, that two types are equal, e.g., $[n + m]int \stackrel{?}{=} [m + n]int$.

Fragile w.r.t changes of the program.

Compilation is done through maximal expansion (function calls, arrays) may not be satisfactory (e.g., if the target is software).

Restricting the expressiveness of the surface layer

The language for static expressions is purposely restricted.

E.g., a limited set of operations on bounded types (integer range, +, *, if/then/else, etc.; enumerated types).

E.g., Scade 6, HLL [3].

```
(* Example in Scade 6 *)
node f<<n>>(x: int^n; y: int^n) returns (o: int^n)
  let o = map<<n>>*+*(x, y); tel;

node ff<<n>>(y: int^n) returns (o: int^n)
  var c: int^2;
  let c = 1^2; o = f<<n>>(x, y); tel;

node fff(x: int^42) returns (o: int^42)
  let o = ff<<42>>(x); tel;
```

Where should the size error be detected?

[3] https://hal.science/hal-01799749v1

Scade 6: the case of arrays

Richard Bird'map/fold operators applied to sized arrays; array slices; functional update. Cf. Lionel Morel's PhD. thesis.

Almost 20 years of use.

Sizes are bounded integers: $n + 1 \not\leq n$; $n + m$ may not be representable.

Size expressions: multi-variable polynomials (+, -, *); but also `if/then/else`, `mod`, `max`, `min`, etc.

Equality and inequality constraints, e.g., $n - m.k - 1 \leq 0$.

Very little symbolic reasonning during type checking.

Do everything possible; keep underground constraints in type signatures (e.g., check that $n + m$ is representable); propagate and evaluate them when constraints are closed.

Error messages come very late and are difficult to understand.

Size information not exploited in code generation.

Some functions definitions have no correct use.

Baptiste Pauget's PhD thesis [5]

An ML-like inference type system ("Polymorphic Types with Polynomial Sizes" [4]).

Size expressions are multi-variable polynomials.

Only consider equality constraints between polynomials, not inequalities.

Examples

```
let dot = fun u v -> fold (+)<_> 0 (map2 (*) <_> u v)
```

```
val dot : 'l. ['l]int → ['l]int → int
```

```
val window : 'l, 'k. α. <'k> → ['l+'k-1]α → ['l]['k]α
```

```
let convolution = fun k i -> map (dot k) <_> (window <_> i)
```

```
val convolution : 'l,'k.['k]int → ['l+'k - 1]int → ['l]int
```

[4] https://www.di.ens.fr/~pouzet/bib/array23.pdf

[5] https://gitlab.inria.fr/parkas/baptistepauget

Paptiste Pauget's proposal [6]

First order unification + a decision algorithm for $\wedge_i(P_i = 0)$.

Some of the size parameters are inferred, as for polymorphism in ML.

Size types are used in the generation of code.

Baptiste designed a code generation for array operations. The code is way better than what existing compilers do (e.g., Scade 6, Heptagon, Lustre V6).

Still, inequality constraints are unvoidable, in particular because sizes are bounded integers.

What happen if the size language is not limited to multivariate polynomials?

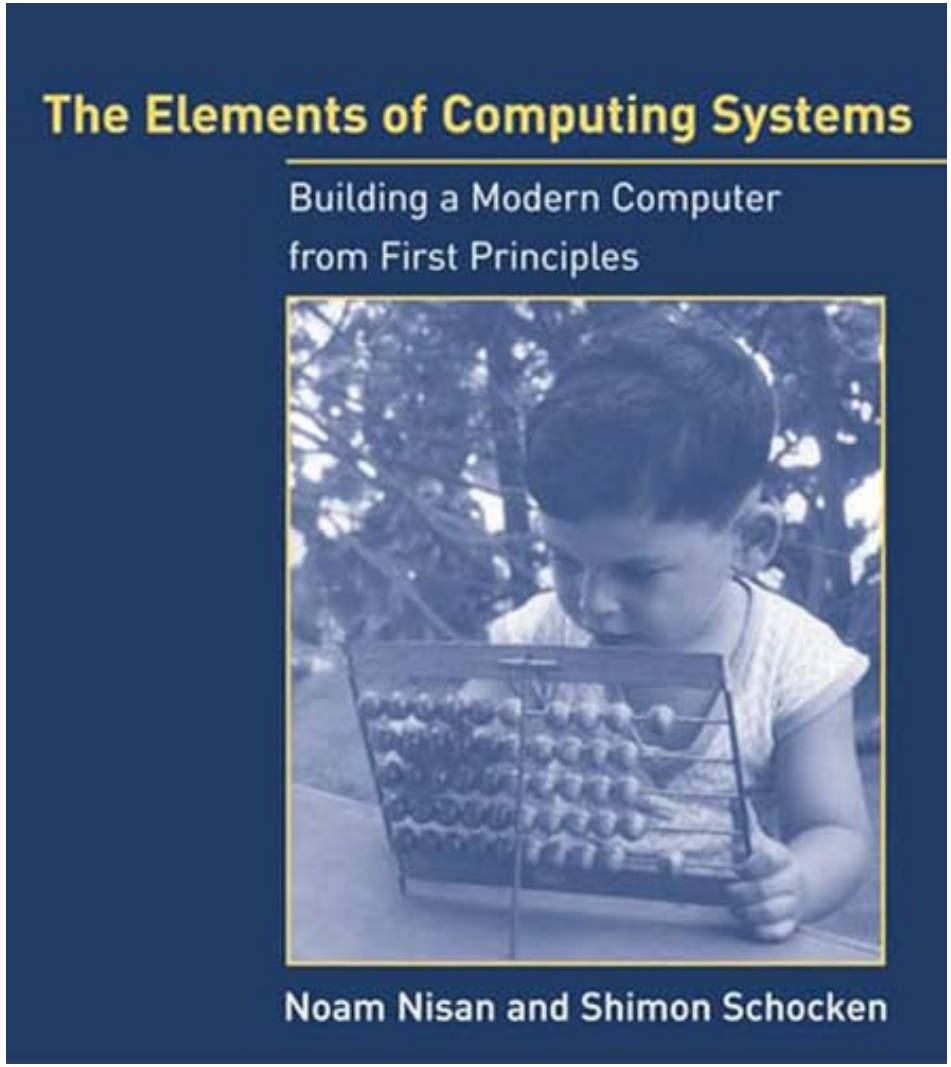
[6] It is "simple et de bon goût" (Paul Caspi)

A personal try.

- ML-like typing: Baptiste's approach.
- Add a limited form of dependences on sizes in types.
- Replace "resolution" by "computation": Scade 6's approach.
- Correctness is simpler to prove.
- Add resolution for "confort" (e.g., detect error at early stages).

A demo with Zélus.

Examples from Nisan and Schocken's book



Expressions and Equations:

$$
\begin{array}{rcl}
e & ::= & \lambda p.e \mid e\ e \mid (e, ..., e) \mid x \mid c \mid e\ \texttt{fby}\ e \\
& & \mid \texttt{let}\ E\ \texttt{in}\ e \mid \texttt{let rec}\ E\ \texttt{in}\ e \\
& & \mid \texttt{match}\ e\ \texttt{with}\ (P_i \to E_i)_{i \in I} \\
& & \\
& & \mid e\langle s, ..., s\rangle \mid \texttt{match size}\ s\ \texttt{with}\ (P_i \to E_i)_{i \in I} \\
& & \mid \texttt{def}\ (f\ \langle n, ..., n\rangle = e)_{i \in I}\ \texttt{in}\ e \\
s & ::= & s + s \mid s \times s \mid n \mid 42 \mid ... \\
E & ::= & p = e \mid E\ \texttt{and}\ ...E \\
p & ::= & x \mid x : t \mid p, ..., p \\
P & ::= & x \mid c
\end{array}
$$

$\lambda\langle n_1, ..., n_k\rangle.e$ is a short-cut for $\texttt{def}\ f\langle n_1, ..., n_k\rangle = e\ \texttt{in}\ f$ with $f \notin FV(e)$.

The type language

Types *a la ML* + a language of sizes + dependences on sizes.

Two basic refinement types:

$$[s] = \{v : int \mid 0 \leq v < s\} \qquad \langle s \rangle = \{v : int \mid v = s\}$$

Type schemes σ and types t:

$$\begin{array}{rcl} \sigma & ::= & \forall \alpha.\sigma \mid t \\ \\ t & ::= & b \mid \alpha \mid t \to t \mid t * ... * t \mid [s] \mid [s]t \\ & & \mid \forall n_1, ..., n_k.t \texttt{ with } C \\ \\ b & ::= & int \mid bool \mid ... \end{array}$$

Executable constraints:

$$\begin{array}{rcl} C & ::= & true \mid false \mid s \leq s \mid s = 0 \mid s < s \mid f\langle s, ..., s \rangle \\ & & \mid C \& C \mid match\ s\ with\ (P_i \to C_i)_{i \in I} \\ & & \mid if\ C\ then\ C \quad \mid let\ n = s\ in\ C \\ & & \quad else\ C \\ & & \mid def\ (f\ \langle n, ..., n \rangle = C_i)_{i \in I}\ in\ C \end{array}$$

$$\frac{H(x) = t}{H \vdash x : t \mid true}\ (\textsc{var}) \qquad \frac{H \vdash f : t_1 \to t_2 \mid C_1 \qquad H \vdash e : t_1 \mid C_2}{H \vdash f\ e : t_2 \mid C_1 \& C_2}\ (\textsc{app})$$

$$\frac{H, H_1 \vdash p : t_1 \mid C_1 \qquad H, H_1 \vdash e : t_2 \mid C_2 \qquad FV(p) \cap FV(C_1 \& C_2) = \emptyset}{H \vdash \lambda p.e : t_1 \to t_2 \mid C_1 \& C_2}\ (\textsc{fun})$$

$$\frac{H \vdash p : t \mid C_1 \qquad H \vdash e : t \mid C_2}{H \vdash p = e : [p \mid t] \mid C_1 \& C_2}\ (\textsc{Eq})$$

$$\frac{\forall i \in [1..n].H \vdash E_i : H_i \mid C_i}{H \vdash E_1 \text{ and } ...E_n : H_1 + ... + H_n \mid C_1 \& ... \& C_n}\ (\textsc{And})$$

with $[x \mid t] = [x : t]$ and $[p_1, ..., p_n \mid t_1 * ... * t_n] = [p_1 \mid t_1] + ... + [p_n \mid t_n]$

(Let)

$$\frac{H \vdash E : H_1 \mid C_1 \qquad H, H_1 \vdash e : t \mid C_2}{H \vdash \texttt{let}\ E\ \texttt{in}\ e : t \mid C_1 \& C_2}$$

(Let-rec)

$$\frac{H, H_1 \vdash E : H_1 \mid C_1 \qquad H, H_1 \vdash e : t \mid C_2}{H \vdash \texttt{let rec}\ E\ \texttt{in}\ e : t \mid C_1 \& C_2}$$

(match)

$$\frac{H \vdash e : t \mid C \qquad \forall i \in [1..n].H \vdash (P_i \to E_i) : t \to H_i \mid C_i}{H \vdash \texttt{match}\ e\ \texttt{with}\ (P_i \to E_i)_{i \in [1..n]} : H' \mid C \& C_1 \& ... \& C_n}$$

(handler)

$$\frac{H, H_1 \vdash P : t \qquad H, H_1 \vdash E : H \mid C \qquad FV(P) = Dom(H_1)}{H \vdash P \to E : t \to H \mid C}$$

with $Dom([x_1 : t_1; ...; x_n : t_n]) = \{x_1, ..., x_n\}$

(FUN-SIZE)

$$\frac{H, n_1 : [n_1] \qquad ... \qquad n_k : [n_k] \vdash e : t \mid C \qquad n_1, ..., n_k \notin FV(H)}{H \vdash \lambda\langle n_1, ..., n_k \rangle.e : \forall n_1, ..., n_k.t \,\texttt{with}\, C}$$

(APP-SIZE)

$$\frac{H \vdash f : \forall n_1, ..., n_k.t \,\texttt{with}\, C}{H \vdash f\langle s_1, ..., s_k \rangle : t[s_1/n_1, ..., s_k/n_k] \,\texttt{with}\, let\, (n_i = s_i)_{i \in I}\, in\, C}$$

(MATCHSIZE)

$$\frac{H \vdash e : [s] \mid C \qquad \forall i \in I.H \vdash (P_i \to E_i) : [s] \to H_i \mid C_i}{H \vdash \texttt{match size}\, e\, \texttt{with}\, (P_i \to E_i)_{i \in I} : H' \mid C \& match\, s\, with\, (P_i \to C_i)_{i \in I}}$$

Function definitions

- Recursion w.r.t a list of sizes $\langle n_1, ..., n_k \rangle$.
- E.g., n-bit linear or dichotomic adder.
- The size vector $\langle n_1, ..., n_k \rangle$ and sizes are never negative.
- Lexicographic order.
- Express in C that recursion is bounded.

The case of a single function definition

$$\frac{H' = H, [f : \forall \vec{n}.t_1 \,\texttt{with}\, f\langle\vec{n}\rangle] \quad H' \vdash \lambda\langle\vec{n}\rangle.e_1 : \forall \vec{n}.t_1 \,\texttt{with}\, C_1 \quad H_f \vdash e : t \mid C_2}{H \vdash \texttt{def}\ f\ \langle\vec{n}\rangle = e_1\ \texttt{in}\ e : t \mid \mathit{def}\ f\ \langle\vec{n}\rangle = \mathit{ifneg}\ \langle\vec{n}\rangle\ C_1\ \mathit{in}\ C_2}\ (\textsc{def-rec})$$

with:

$$\begin{array}{lcl} \mathit{ifneg}\ \langle\rangle\ C & \stackrel{def}{=} & C \\ \mathit{ifneg}\ \langle n_1, ..., n_k\rangle\ C & \stackrel{def}{=} & \mathit{if}\ n_1 < 0\ \mathit{then\ false} \\ & & \mathit{else\ ifneg}\ \langle n_2, ..., n_k\rangle\ C \end{array}$$

This is not enough to ensure that recursion is bounded

Add a bound on size parameters.

$$t \quad ::= \quad \forall \vec{n}.t \,\texttt{with}\, C \mid \forall \vec{n} < \vec{s}.t \,\texttt{with}\, C$$

where $\vec{n} < \vec{s}$ is the strict lexicographical order.

Update the rule for recursion and application.

$$\frac{H' = H, [f : \forall \vec{m} < \vec{n}.t_1[\vec{m}] \,\texttt{with}\, f\langle \vec{m} \rangle] \quad H' \vdash \lambda \langle \vec{n} \rangle . e_1 : \forall \vec{m}.t_1[\vec{m}] \,\texttt{with}\, C_1[\vec{m}] \ldots}{H \vdash \texttt{def}\ f\ \langle \vec{n} \rangle = e_1\ \texttt{in}\ e : t \mid def\ f\ \langle \vec{n} \rangle = ifneg\ \langle \vec{n} \rangle\ C_1[\vec{n}]\ in\ C_2} \ (\textsc{def-rec})$$

$$\frac{H \vdash f : \forall \vec{n} < \vec{s'}.t \,\texttt{with}\, C}{H \vdash f\langle \vec{s} \rangle : t[\vec{s}/\vec{n}] \,\texttt{with}\, let\ (n_i = s_i)_i\ in\ ifless\ \langle \vec{s} \rangle\ \langle \vec{s'} \rangle\ C} \ (\textsc{app-size})$$

$$
\begin{array}{lcl}
\mathit{ifless}\ \langle\rangle\ \langle\rangle\ C & \stackrel{def}{=} & C \\
\mathit{ifless}\ \langle s_1, \ldots, s_k \rangle\ \langle s'_1, \ldots, s'_k \rangle\ C & \stackrel{def}{=} & \mathit{if}\ s_1 < s'_1\ \mathit{then}\ \mathit{true} \\
 & & \mathit{else}\ \mathit{if}\ s_1 > s'_1\ \mathit{then}\ \mathit{false} \\
 & & \quad \mathit{else}\ \mathit{ifless}\ \langle s_2, \ldots, s_k \rangle\ \langle s'_2, \ldots, s'_k \rangle\ C
\end{array}
$$

Conclusion

- This is work-in-progress.
- The ZRun interpreter [7] has been extended to deal with size arguments, sizes in arrays and bounded recursion on a size argument.
- Non decreasing recursion, access out-of-the-bound, negative sizes are detected at run-time.
- The new implementation of Zélus do have sizes and recursion;
- the type system is still limited: there is no inference mechanism for sizes.
- The use of Z3 and DV (Scade 6) has been tempted on a few simple examples only (e.g., n-bit adder, or, iteration on an array).

Is-it a publishable work? Is-it more than a direct and ad-hoc instance of HM(X) or a dependent type system (or type system with refinements)?

[7] `https://github.com/marcpouzet/zrun/tree/2025`