Typst

un nouveau langage de composition de documents

Louis Gesbert

23 janvier 2024

Inria Paris

LaTeX ??

- Incontournable en maths et informatique
- LaTeX est un empilement de macros vieux de plus de 40 ans
- On nous promet que la relève est là tous les 3 ans depuis X années…
- … mais jusqu'ici irremplaçable.

Et Typst alors ?

L’outil

- Écrit en Rust
- Pas besoin de 1 000 packages
- Compilation rapide et incrémentale
- De bons messages d’erreur !!

Et Typst alors ?

L'outil

- Écrit en Rust
- Pas besoin de 1 000 packages
- Compilation rapide et incrémentale
- De bons messages d'erreur !!

Le langage

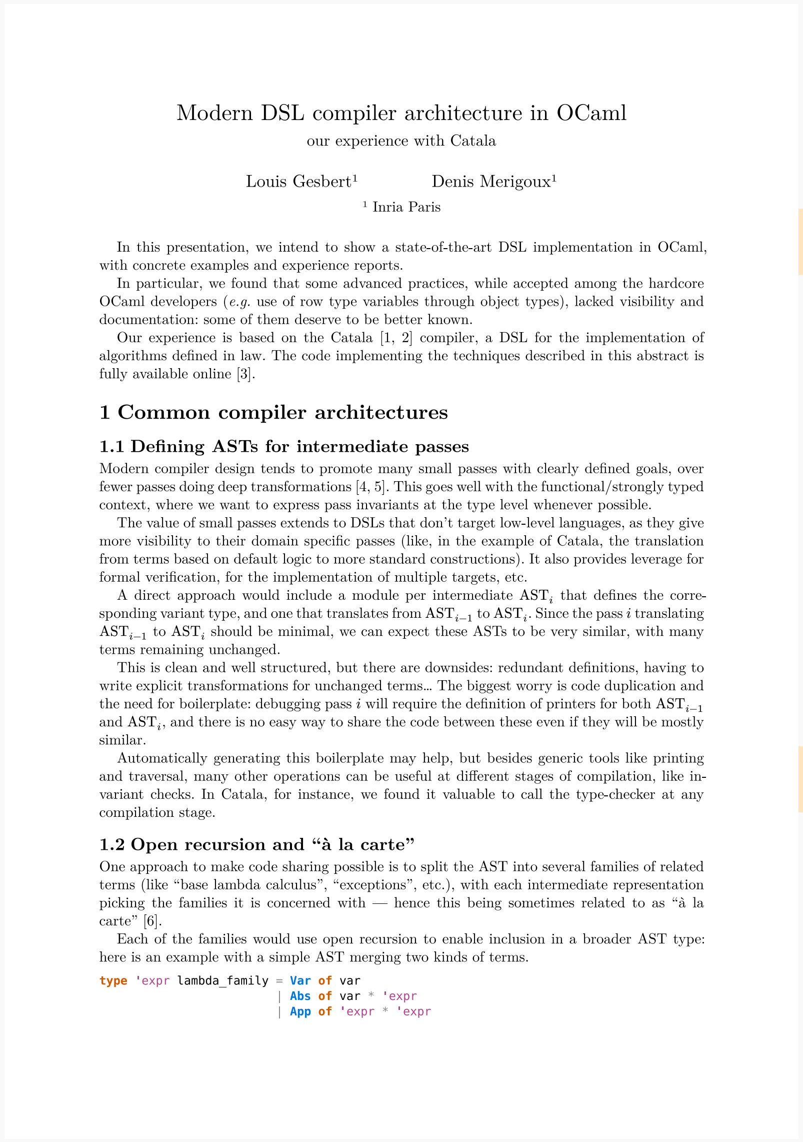
- Syntaxe légère (inspirée de markdown)
- Mais expressif et extensible
- Fonctionnel dans l'esprit (pas de blagues de scope)

Exemple

```
// Utilise Polylux depuis le dépôt officiel (pour les slides)
#import "@preview/polylux:0.3.1": *
#import themes.metropolis: *

// Définitions de base du document
#set text(font: "Cabin", size: 20pt)
#set page(background: image("watermark.svg", width: 35%))
#set list(marker: ([●], [○]))

// Définition de variable
#let title = [Typst]

// Définition de fonction
#let small(body) = text(size: 0.8em, body)

// Quelques paramètres pour le thème
#show: metropolis-theme.with(
    aspect-ratio: "16-9",
    footer: {
        h(100fr)
        box(baseline:-0.4em, width:2em, height:3pt, m-progress-bar)
    }
)
```

```
// Appel de fonction
#title-slide(
    title: title,
    subtitle: [un nouveau langage de composition de documents],
    author: [Louis Gesbert],
    date: [23 janvier 2024],
    extra: [Inria Paris],
)

#slide(title: "LaTeX ??")[
    - Incontournable en maths et informatique
    - LaTeX est un empilement de macros vieux de plus de 40 ans
    - On nous promet que la relève est là tous les 3 ans depuis
      X années...
    - ... mais jusqu'ici irremplaçable.
]
```

… et pour un papier



Modern DSL compiler architecture in OCaml

our experience with Catala

Louis Gesbert[1] Denis Merigoux[1]

[1] Inria Paris

In this presentation, we intend to show a state-of-the-art DSL implementation in OCaml, with concrete examples and experience reports.

In particular, we found that some advanced practices, while accepted among the hardcore OCaml developers (*e.g.* use of row type variables through object types), lacked visibility and documentation: some of them deserve to be better known.

Our experience is based on the Catala [1, 2] compiler, a DSL for the implementation of algorithms defined in law. The code implementing the techniques described in this abstract is fully available online [3].

1 Common compiler architectures

1.1 Defining ASTs for intermediate passes

Modern compiler design tends to promote many small passes with clearly defined goals, over fewer passes doing deep transformations [4, 5]. This goes well with the functional/strongly typed context, where we want to express pass invariants at the type level whenever possible.

The value of small passes extends to DSLs that don't target low-level languages, as they give more visibility to their domain specific passes (like, in the example of Catala, the translation from terms based on default logic to more standard constructions). It also provides leverage for formal verification, for the implementation of multiple targets, etc.

A direct approach would include a module per intermediate AST_i that defines the corresponding variant type, and one that translates from AST_{i-1} to AST_i. Since the pass i translating AST_{i-1} to AST_i should be minimal, we can expect these ASTs to be very similar, with many terms remaining unchanged.

This is clean and well structured, but there are downsides: redundant definitions, having to write explicit transformations for unchanged terms… The biggest worry is code duplication and the need for boilerplate: debugging pass i will require the definition of printers for both AST_{i-1} and AST_i, and there is no easy way to share the code between these even if they will be mostly similar.

Automatically generating this boilerplate may help, but besides generic tools like printing and traversal, many other operations can be useful at different stages of compilation, like invariant checks. In Catala, for instance, we found it valuable to call the type-checker at any compilation stage.

1.2 Open recursion and "à la carte"

One approach to make code sharing possible is to split the AST into several families of related terms (like "base lambda calculus", "exceptions", etc.), with each intermediate representation picking the families it is concerned with — hence this being sometimes related to as "à la carte" [6].

Each of the families would use open recursion to enable inclusion in a broader AST type: here is an example with a simple AST merging two kinds of terms.

```
type 'expr lambda_family = Var of var
                         | Abs of var * 'expr
                         | App of 'expr * 'expr
```

```
#set page(paper: "a4")
#set heading(numbering: "1.1")
#set par(justify: true, first-line-indent: 1.2em, leading: 0.55em)
#set text(font: "New Computer Modern")
#show par: set block(spacing: 0.55em)

#align(center, {
  text(17pt)[Modern DSL compiler architecture in OCaml]
  v(1em, weak:true)
  text(12pt)[our experience with Catala]
  v(2em, weak:true)
  text(13pt)[
      #h(2fr) Louis Gesbert#super[1]
      #h(1fr) Denis Merigoux#super[1]
      #h(2fr)
  ]
  v(1em, weak:true)
  text[#super[1]~Inria Paris]
  v(1.5em)
})

In this presentation, we intend to show a state-of-the-art DSL implementation in
OCaml, with concrete examples and experience reports.
[...]
```