Inductive data types and pattern matching - Literature survey of implementation techniques of type systems

Pau Faraldos Pijoan
P.FaraldosPijoan@student.tudelft.nl

Responsible Professor
Jesper Cockx

Supervisor
Bohdan Liesnikov

1. Background

- Type systems are essential as they:
  - Help discover errors at compile time
  - Simplify refactoring
  - Make code self-documenting
- Implementing pattern matching in compilers can be complex:
  - Advanced patterns
  - Optimizations
  - Effects on type checking & inference
  - Exhaustiveness and redundancy checking
- Not a lot of relevant literature comparing different techniques of implementing pattern matching and inductive data types.

Research Question: ***What are the different implementation techniques for type systems regarding inductive data types and pattern matching that have been proposed in the literature?***

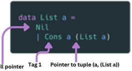
2. Method

- Dimensions of analyzing resources:
  - Code Generation Strategy
    - Decision trees
    - Backtracking finite state automata
    - Term Decomposition
    - Other compiler-specific approaches
  - Type system / Programming language Features
    - Type safety (statically & strongly typed vs. dynamically & weakly typed)
    - Complex types (e.g. inductive data types, coinductive data types, mutually inductive data types)
    - Evaluation techniques (lazy vs. eager)
    - ...
- Compare the resources based on where they fit into these dimensions, find concise and clear way of showing results.

3. Inductive data types

Inductive data types

- Data type defined in terms of itself, allowing recursive structures.
- In specific cases some memory indirection can be eliminated. [2]



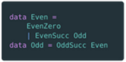
```
data List a =
    Nil
  | Cons a (List a)
```

Coinductive data types

- Add support to non-terminating, infinite data structures.

Mutually inductive data types

- Circular dependencies between data types.
- Important consideration: Forward declarations and dependency analysis[3]



```
data Even =
    EvenZero
  | EvenSucc Odd
data Odd = OddSucc Even
```

4. Pattern matching

Decision trees

- Each decision is a test of a pattern matching condition.
- Correct order is essential to performance -> **NP-Hard** problem. [1]

Backtracking finite state automata

- Each state is a test of a pattern matching condition.
- More flexible and complex patterns (e.g. guards, or-patterns)

Example: (Cons 1 tail)

Term decomposition

- Recursively decomposes terms to avoid unnecessary evaluation.[4]
- Useful with **lazy evaluation**.

5. Discussion and advice

- Understanding the context and usage of language is essential.
- Which features should the language have?
  - More features are **not always** better: can over-complicate the code, lead to performance issues, can become confusing to users...
- Applying memory indirection optimizations to the representation of inductive data types can make code more complex, but improves run-time performance.
- **Decision trees** can be more efficient at run-time when the order of tests is carefully optimized, **but** extensions of simple patterns can be more complicated to integrate compared to in **FSAs**.
- **Term decomposition** can be integrated into pattern matching techniques when lazy evaluation is used to avoid unnecessary computation.

6. Future work

- Deep-dive into commonly-used programming languages and analyze the techniques used.
- Combination of multiple techniques for better performance.
- Extensions for more complex data types (e.g. dependent types, polymorphic types).

7. References

[1] Marianne Baudinet and David MacQueen. Tree pattern matching in ML, 1985

[2] Luca Cardelli. Compiling a functional language. *In Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, page 208–217, New York, NY, USA, 1984. Association for Computing Machinery

[3] Jean-Baptiste Jeannin, Dexter Kozen, Alexandra Silva, David Baelde, Arnaud Carayol, Ralph Matthes, and Igor Walukiewicz. Cocaml: Functional programming with regular coinductive types. *Fundam. Inf.*, 150(3–4):347–377, jan 2017

[4] Laurence Puel and Ascander Suarez. Compiling pattern matching by term decomposition. *Journal of Symbolic Computation*, 15(1):1–26, 1993