Visualising program dataflow with string diagrams

S-REPLS 13 / Fun in the Afternoon

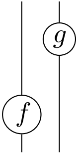
Nick Hu Alex Rice Calin Tataru Dan Ghica

Huawei Programming Languages Research Centre Edinburgh

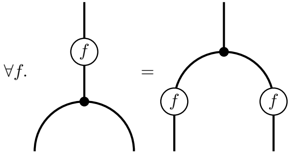
2023-11-01

String diagrams

- String diagrams are a graphical notation for terms in different types of monoidal categories
  - The term $(f \otimes \mathsf{id}) \circ (\mathsf{id} \otimes g)$ is represented by the string diagram:



  - Equations of terms arising from the monoidal structure are captured by isotopy of string diagrams
- Cartesian monoidal categories (i.e. $\otimes = \times$ and $I = 1$) admit a natural copy-delete comonoid:



`sd-lang`

- Toy language for programs
- Syntax: essentially lambda calculus with operations and recursive let
  - `bind x = v1 y = v2 … in v`
  - values are variables, thunks, or operations `op(v1, v2, …)`
    - `plus(x, y)`, `eq(x, y)`, `if(cond, tb, fb)`, etc.
- Semantics: hierarchical hypergraphs
  - a model of string diagrams for symmetric monoidal categories with copy-delete
- (D. R. Ghica, Muroya, and Ambridge 2021)

Example: factorial

```
bind fact = lambda(x .
  if(eq(x, 0),
    1,
    times(x,
      app(fact,
        minus(x, 1)
      )
    )
  )
)
in app(fact, 5)
```

Example: factorial

```
bind fact = lambda(x .
  if(eq(x, 0),
    1,
    times(x,
      app(fact,
        minus(x, 1)
      )
    )
  )
)
in app(fact, 5)
```

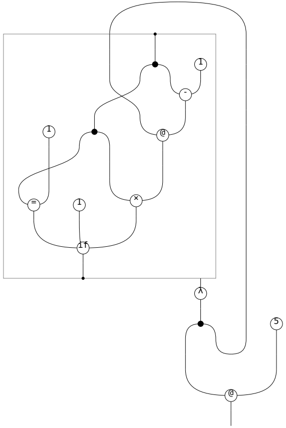


Figure 1: factorial as a string diagram

Representation of programs

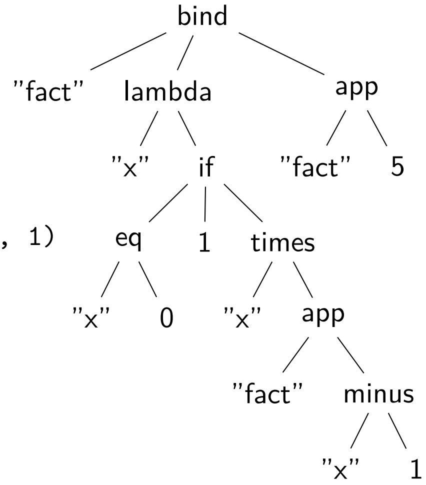
Traditional representation: abstract syntax tree

```
bind("fact",
  lambda("x",
    if(eq("x", 0),
      1,
      times("x",
        app("fact",
          minus("x", 1)
        )
      )
    )
  ),
  app("fact", 5)
)
```

Representation of programs

Traditional representation: abstract syntax tree

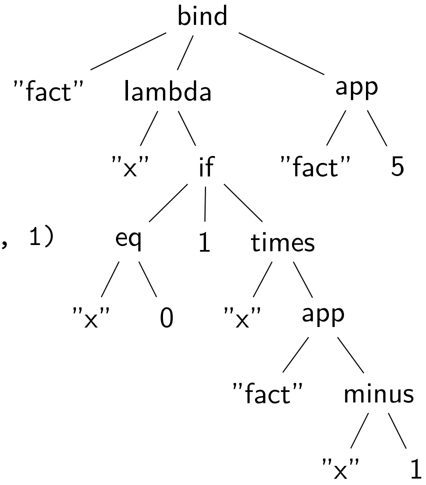
```
bind("fact",
  lambda("x",
    if(eq("x", 0),
      1,
      times("x",
        app("fact",
          minus("x", 1)
        )
      )
    )
  ),
  app("fact", 5)
)
```



Representation of programs

Traditional representation: abstract syntax tree

```
bind("fact",
  lambda("x",
    if(eq("x", 0),
      1,
      times("x",
        app("fact",
          minus("x", 1)
        )
      )
    )
  ),
  app("fact", 5)
)
```



Compiler optimisations are described by semantic-preserving transformations on these ASTs given by rewrite rules.

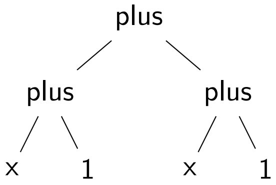
ASTs do not support sharing, or α-equivalence I

Consider the expression $(x+1)+(x+1)$ (where x is free).

This is represented by the `sd-lang` expression

```
plus(plus(x, 1), plus(x, 1))
```

Its AST is



ASTs do not support sharing, or α-equivalence II

Problem: The term obtained by the α-invariant substitution $[x \mapsto y]$ is represented by a different AST.

Consequence: The optimisation $\texttt{plus}(x_1, x_2) \to \texttt{times}(x_1, 2)$ needs to do a non-trivial computation to be valid, namely checking that $x_1 \equiv_\alpha x_2$.

- Can leverage de Bruijn indices, nominal techniques...

String diagrams do support sharing, and α-equivalence

Our string diagrams are equipped with a natural copy-delete comonoid.

This allows for a more meaningful representation of this program as the string diagram:

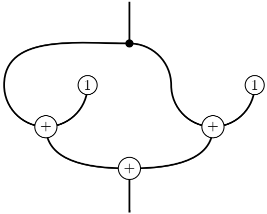


Figure 2: $(x+1)+(x+1)$ — observe that x does not appear in the diagram!

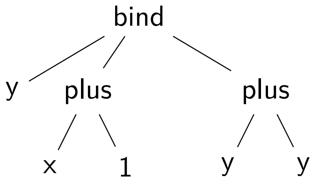
Nodes represent operations, and edges represent dataflow (e.g. of values)!

ASTs do not support binding and shadowing

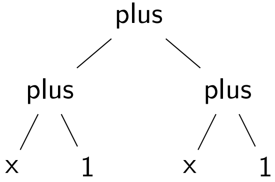
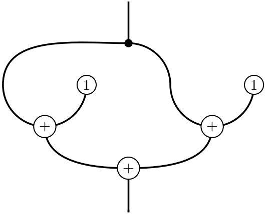
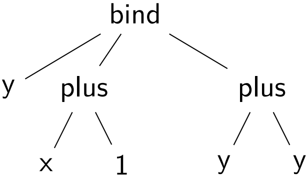
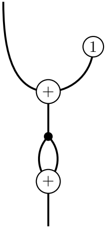
Another way to write this program:

```
bind y = plus(x, 1) in plus(y, y)
```

AST:

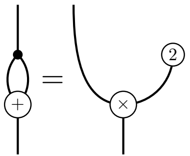


ASTs vs string diagrams

AST	String diagram
	
	

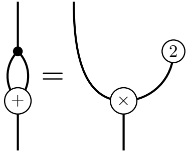
Compiler optimisations as string diagram rewriting

The optimisation we care about is

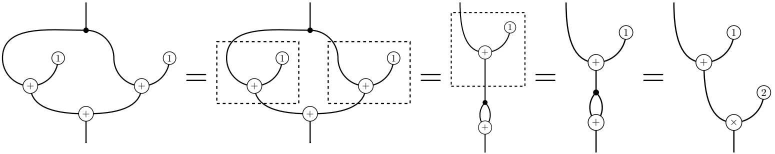


Compiler optimisations as string diagram rewriting

The optimisation we care about is



Derive



An aside on graphs

- Graphs are also used in production compilers to sidestep these issues

An aside on graphs

- Graphs are also used in production compilers to sidestep these issues
  - ✓ They also convey information efficiently, and naturally support sharing

An aside on graphs

- Graphs are also used in production compilers to sidestep these issues
  - ✓ They also convey information efficiently, and naturally support sharing
  - ✗ They are not algebraic: no inductive structure, hard to reason about and distil algorithms which preserve invariants

An aside on graphs

- Graphs are also used in production compilers to sidestep these issues
  - ✓ They also convey information efficiently, and naturally support sharing
  - ✗ They are not algebraic: no inductive structure, hard to reason about and distil algorithms which preserve invariants

An aside on graphs

- Graphs are also used in production compilers to sidestep these issues
  - ✓ They also convey information efficiently, and naturally support sharing
  - ✗ They are not algebraic: no inductive structure, hard to reason about and distil algorithms which preserve invariants
- String diagrams can be thought of as an intermediate representation between ASTs and graphs

An aside on graphs

- Graphs are also used in production compilers to sidestep these issues
  - ✓ They also convey information efficiently, and naturally support sharing
  - ✗ They are not algebraic: no inductive structure, hard to reason about and distil algorithms which preserve invariants
- String diagrams can be thought of as an intermediate representation between ASTs and graphs
  - ✓ Have enough graphical structure to support sharing and α-equivalence

An aside on graphs

- Graphs are also used in production compilers to sidestep these issues
  - ✓ They also convey information efficiently, and naturally support sharing
  - ✗ They are not algebraic: no inductive structure, hard to reason about and distil algorithms which preserve invariants
- String diagrams can be thought of as an intermediate representation between ASTs and graphs
  - ✓ Have enough graphical structure to support sharing and α-equivalence
  - ✓ They are algebraic, as they represent terms of some kind of monoidal category

An aside on graphs

- Graphs are also used in production compilers to sidestep these issues
  - ✓ They also convey information efficiently, and naturally support sharing
  - ✗ They are not algebraic: no inductive structure, hard to reason about and distil algorithms which preserve invariants
- String diagrams can be thought of as an intermediate representation between ASTs and graphs
  - ✓ Have enough graphical structure to support sharing and α-equivalence
  - ✓ They are algebraic, as they represent terms of some kind of monoidal category
  - ✓ Support a natural theory of rewriting via double-pushout graph rewriting (corresponding to equipping the monoidal category with equations)

An aside on graphs

- Graphs are also used in production compilers to sidestep these issues
  - ✓ They also convey information efficiently, and naturally support sharing
  - ✗ They are not algebraic: no inductive structure, hard to reason about and distil algorithms which preserve invariants
- String diagrams can be thought of as an intermediate representation between ASTs and graphs
  - ✓ Have enough graphical structure to support sharing and α-equivalence
  - ✓ They are algebraic, as they represent terms of some kind of monoidal category
  - ✓ Support a natural theory of rewriting via double-pushout graph rewriting (corresponding to equipping the monoidal category with equations)
  - ✗ (✓?) Not very well studied, **lack of tooling**(!)

How to draw a string diagram

- Hypergraphs quotient monoidal categories with copy-delete
- For each hypergraph, we need to pick a representative monoidal term
  - Involves (non-canonically) foliating the hypergraph into layers, and determining the order of operations (which determines how many ‘swaps’ are needed)
  - *Aesthetically-pleasing diagram heuristic*: minimise the number of layers, and the number of swaps (NP-hard)
- Given a monoidal term, we can construct a big LP to determine the coordinates of each node and positioning of edges (Tataru and Vicary 2023)

Demo

- Also available at https://sd-visualiser.github.io/sd-visualiser

Future work and references

- LLVM's Multi-Level Intermediate Representation (MLIR)

References

Ghica, Dan R., Koko Muroya, and Todd Waugh Ambridge. 2021. "A Robust Graph-Based Approach to Observational Equivalence." September 23, 2021. https://doi.org/10.48550/arXiv.1907.01257.

Ghica, Dan, and Fabio Zanasi. 2023. "String Diagrams for λ-Calculi and Functional Computation." October 19, 2023. https://doi.org/10.48550/arXiv.2305.18945.

Tataru, Calin, and Jamie Vicary. 2023. "A Layout Algorithm for Higher-Dimensional String Diagrams." May 11, 2023. https://doi.org/10.48550/arXiv.2305.06938.