EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Database Systems Research Group

Bachelorthesis Computer Science

Implementing Abstract Machines in SQL

Romain Carl

29.03.2023

Examiner

Prof. Dr. Torsten Grust

Supervisor

Louisa Lambrecht

Romain Carl:
Implementing Abstract Machines in SQL
Bachelorthesis Computer Science
Eberhard Karls Universität
From 01.12.2022 to 29.03.2023

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorthesis selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorthesis wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum

Romain Carl

Abstract

Abstract machines are a useful tool in understanding how a program's complex high-level language representation can be broken down to low-level execution steps. They feature environments and other data structures that are important building stones of interpreters. Hence, implementing them in SQL can provide valuable insights into how these crucial structures can be represented in relational databases, which can be used to inform source-to-source language translation into SQL.

This thesis develops SQL realizations of the *SECD* and the *Krivine machine*, that both operate on λ-calculus. Six machine implementations are obtained, tested and compared with regards to performance. Among these SQL-driven interpreters of λ-calculus, a PostgreSQL implementation relying on a hash table extension shows to be best-performing.

Contents

Acronyms

ACID Atomicity, Consistency, Isolation, Durability
ADT Algebraic Data Type
CTE Common Table Expression
DAG Directed Acyclic Graph
DBMS Database Management System
JSON JavaScript Object Notation
LIFO Last In First Out
UDF User-Defined Function
WHNF Weak Head Normal Form
WNF Weak Normal Form

1

Introduction

Upon reading this thesis' title, one might rightfully ask what, if any, relation between *abstract machines* and *SQL* exists and how it could possibly give sufficient reason to engage in implementing the first in the latter. Indeed, the notion of abstract machines, which by itself only implies something that enables step-by-step execution of programs (*machine*) while not being detailed in how it is actually built (*abstract*) [1], seems to have little in common with the *Structured Query Language (SQL*, originally *SEQUEL)* [2], the standard language for accessing and manipulating data on relational databases. Yet, the two might surprisingly be connected by the topic of *compilation*.

Characterized as "any set of data structures and algorithms which can perform the storage and execution of programs" [3, p. 2], abstract machines are inherently linked to compilation, as they "bridge the gap between the high level of a programming language and the low level of a real machine" [1, p. 1]. Such a machine takes a program p – expressed in a formalized language L – as input, loads it into its internal data structure and executes it.

A conceptually similar approach can be found in recent publications of the Database Systems Research Group – where SQL and an underlying query engine fill the role of the abstract machine in executing the code of a foreign language L. The attempt to move computation from outside to inside of database systems and thus *close to the data* [4] has motivated a series of compilation pipelines that translate code from a variety of source languages or SQL variants – be it PL/SQL [5], Python [6] or recursive UDFs [7] – into plain SQL.

Every one of these translations banks on the use of SQL:1999's `WITH RECURSIVE` construct[1] [9]. This is not a coincidence, since, by virtue of its expressive power that enables looping, `WITH RECURSIVE` makes SQL a full-fledged, Turing-complete programming language[2] [5, p. 2] (subsequently, we can expect our code to heavily rely on it and to structurally resemble interpreters as in [7]).

For these reasons, implementing abstract machines in SQL might not be such a far-fetched endeavour after all. We shall discuss two machines designed to interpret lambda calculus; their SQL realization will therefore allow the evaluation of any λ-term – or, by extension, a program written in any functional language – *by* the SQL engine. The goal of this thesis is to prove the feasibility of these implementations in different Database Management System (DBMS), first, as a pure *proof of concept*, and second, to compare their performances. Incidentally, our code might uncover patterns in the compilation pipelines developed so far and inform future work

[1] Whose semantics admittedly requires getting used to, since it specifies a very narrow form of (tail-)recursion, for which the term of *iteration* would be more adequate. It is best understood in [8].

[2] For proof, also see this Postgres realization of a Cyclic Tag System: https://wiki.postgresql.org/index.php?title=Cyclic_Tag_System&oldid=15106 (visited on 21.3.23).

on source-to-source translation into SQL.

In the following subsection, the DBMS we are to implement the machines in, as well as the motivation behind the choice of them, will be briefly presented.

1.1. Relational Database Management Systems

In 1970, British IBM computer scientist Edgar F. Codd proposed the *relational model*, by which data is expressed in terms of *tuples* or *records* grouped into *tables* or *relations* [10]. Since then, this model has served as the theoretical foundation for a variety of DBMS, i.e., "software system[s] that enable[...] users to define, create, maintain and control access to the database" [11, p. 64]. Among these *relational* DBMS, the use of SQL to write and query data is common. Although all derive from the same standard, each DBMS extends and syntactically adapts the language in its own way, effectively creating different dialects of SQL.

1.1.1. PostgreSQL

The free and open-source relational DBMS *PostgreSQL* (also referred to as *Postgres*) is among the most widely used.[3] In addition to performance and robustness, its strong compliance with the SQL standard and the wide array of features it offers, including a plethora of data types with accompanying function libraries, it has the advantage of being very extensible by virtue of its many APIs and language extensions.[4] These reasons, along with its extensive documentation,[5] make it the preferred DBMS at the Database Systems Research Group.

Therefore, we will primarily focus on Postgres (version 14.6) in our implementation of abstract machines.

1.1.2. DuckDB

DuckDB [12] is an open-source, relational DBMS that was initially developed at the Database Architectures Group[6] at the Centrum Wiskunde & Informatica in Amsterdam. It is currently still evolving, its source code being available for inspection and contribution on GitHub.[7] One of its main features is a *columnar-vectorized query execution engine* [13], which processes data in batches instead of row by row and thus considerably reduces performance overhead compared to traditional systems like Postgres.[8]

This performance advantage makes DuckDB of interest to us and we will therefore attempt to implement some abstract machines in it (version 0.6.1), for which the system's thorough documentation[9] should be of help.

[3] Ranking fourth, see https://db-engines.com/en/ranking/relational+dbms (visited on 21.3.23).
[4] See https://www.postgresql.org/about/ (visited on 21.3.23).
[5] See https://www.postgresql.org/docs/14/index.html (visited on 21.3.23).
[6] See https://www.cwi.nl/en/groups/database-architectures/ (visited on 21.3.23).
[7] See https://github.com/duckdb/duckdb (visited on 21.3.23).
[8] See https://duckdb.org/why_duckdb (visited on 21.3.23).
[9] See https://duckdb.org/docs/sql/introduction (visited on 21.3.23).

1.1.3. Umbra

The relational DBMS *Umbra*, which is currently being developed at the Chair of Database Systems[10] at the Technical University of Munich, offers yet another axis of performance improvement: designed to synthesize in-memory and disk-based approaches (i.e., storing data on main memory compared to storing it on external disks), it uses a combination of low-overhead buffering and variable-size pages to benefit from the efficiency of in-memory database systems while simultaneously avoiding their size limitations [14].

Like DuckDB, Umbra suggests itself as a possible host system for abstract machine implementations due to its promising performance; but, although it is designed to be similar to PostgreSQL,[11] its being in development and closed-source, as well as the lack of any documentation, may hinder implementation work.

A brief overview of database fundamentals having been given, Chapter 2 will introduce into λ-calculus basics and broadly present the theory of the implemented machines. In Chapter 3, we will develop different SQL implementations of these machines, whose performance will be assessed and compared in Chapter 4. Chapter 5 will conclude this thesis and discuss future possible work.

Throughout this thesis, implementation details will be referred to by linking to the corresponding files and folders in the GitHub repository available under `https://github.com/j-w-moebius/abstract-machines-to-sql`.

[10] `https://db.in.tum.de/` (visited on 21.3.23).
[11] See `https://umbra-db.com/` (visited on 21.3.23).

2

Abstract machines

Since one of the most influential abstract machines, the *SECD machine*, which we are set out to implement in this thesis, was designed to interpret λ-calculus, we will first discuss the language (in a section that can be skipped by readers already familiar with it), before addressing the machine itself, as well as an interesting alternative to it, the *Krivine machine*.

2.1. Overview on Lambda calculus

The origins of the formal system of λ-calculus date back to the 1930s, where it was introduced by Alonzo Church [15] as part of his work on formal logic. Indeed, its significance is not to be underestimated, for it has since be proven to be Turing-complete [16] and serves as theoretical foundation for functional programming languages [17, p. 5].

At the heart of the calculus is a formal language based on function abstraction and application, which is easily definable in an inductive fashion. A λ-term is either:

1. a *numeric literal* n,[1]
2. a *variable* x,
3. a function *abstraction* $(\lambda x.t)$ or
4. a function *application* $(t_1 \ \ t_2)$,

where t, t_1, t_2 are λ-terms and x any variable.

Some further terminology needs to be briefly introduced for use in later sections:

- The occurrence of a variable x in an abstraction $(\lambda x.t)$ is said to *bind* all free occurrences of x in t. An occurrence of a variable x is said to be *free* in t if it is not bound by any λ-abstraction in t.
- A term is *closed* if it does not contain any free variables.
- β*-reduction* is the process of performing an actual function application, i.e., reducing the term $((\lambda x.M)\ N)$ to $M[x := N]$ by substituting all free occurrences of x in M by N [17].
- An *evaluation strategy* is a set of rules determining to which part of a complex λ-term β-reduction is applied first. If a term cannot be further reduced by a particular strategy, it is said to be in *normal form*. Literature abounds with different strategies and the types of normal forms they reduce to [18], which we will not further explore in this thesis. If referenced in the following sections, individual strategies will be explained. Let it suffice

[1] Numeric literals are not present in *pure* λ-calculus, where one instead expresses them with functions, called *Church Numerals* [17, p. 13]. The calculus presented here is *extended* for the sake of implementation.

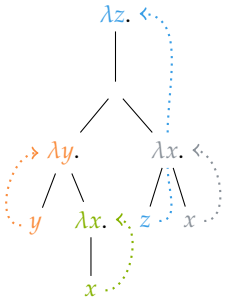


Figure 2.1.: Tree-ish representation of the term $(\lambda z.(\lambda y.(y\ (\lambda x.x)))\ (\lambda x.(z\ x)))$. The arrows link every bounded variable occurrence to its binding occurrence.

to name the Weak Normal Form (WNF) and the Weak Head Normal Form (WHNF), who differ in the absence, or presence, of reducible expressions in non-head, i.e., inner, positions.[2] Some terms, like $\Omega = (\lambda x.(x\ x))\ (\lambda x.(x\ x))$, do not reach a normal form by any reduction strategy.

- If two terms t_1, t_2 can be converted into each other by simple variable renaming (like $(x\ x)$ and $(z\ z)$), they are called *α-equivalent*.

An alternative representation of λ-terms exists, where variables are denoted by natural-numbered *indexes* instead of names. The index corresponds to the number of λ-bindings in scope between the variable and its own λ-binding, which is best understood if one renders the tree-ish structure of terms visible. For example, the term $(\lambda z.(\lambda y.(y\ (\lambda x.x)))\ (\lambda x.(z\ x)))$ can be represented as shown in Figure 2.1. Replacing every bound variable occurrence in the original term with the number of λ-bindings that lie between it and its binder in the tree (1 for z and 0 for all the other variables), as well as omitting all binding occurrences, yields $\lambda(\lambda\ 0\ (\lambda\ 0))\ (\lambda\ 1\ 0)$, which is called *De Bruijn notation*[3] of the original term. It has the advantage of abstracting over α-equivalence and will be needed for the Krivine machine.

2.2. The SECD machine

In 1964, British Computer Scientist Peter John Landin (1930-2009), since known as a key figure in the development of functional programming languages, described a "mechanical process for obtaining the value, if it exists, of any given A[pplicative] E[xpression][4] [...]" [20, p. 316]. He formalized this process by first stating rules on how to evaluate a given λ-term – as a compositional evaluation function – before proposing a data structure with a transition function defining execution steps on it, that together implement these rules [20, p. 316]. Reminding ourselves of the definition from [3] stated in Chapter 1, it is not surprising that Landin's data structure and algorithm have since been labelled an abstract machine.

[2] See [18] for exact definitions of these two normal forms.
[3] Named after Dutch mathematician Nicolaas G. de Bruijn, who introduced it in [19].
[4] A λ-term.

Stack	Env.	Control	Dump		Stack	Env.	Control	Dump	
v	e	$\square$	$\square$	$\longmapsto$	[Computation halts with result v.]				(1)
v	e'	$\square$	$(\bar{s}, e, \bar{c}) : \bar{d}$	$\longmapsto$	$v : \bar{s}$	e	$\bar{c}$	$\bar{d}$	(2)
$\bar{s}$	e	$n : \bar{c}$	$\bar{d}$	$\longmapsto$	$n : \bar{s}$	e	$\bar{c}$	$\bar{d}$	(3)
$\bar{s}$	e	$x : \bar{c}$	$\bar{d}$	$\longmapsto$	$e(x) : \bar{s}$	e	$\bar{c}$	$\bar{d}$	(4)
$\bar{s}$	e	$(\lambda x.t) : \bar{c}$	$\bar{d}$	$\longmapsto$	$(\lambda x.t, e) : \bar{s}$	e	$\bar{c}$	$\bar{d}$	(5)
$\bar{s}$	e	$(t_1\ t_2) : \bar{c}$	$\bar{d}$	$\longmapsto$	$\bar{s}$	e	$t_2 : t_1 : \texttt{apply} : \bar{c}$	$\bar{d}$	(6)
$(\lambda x.t, e') : v : \bar{s}$	e	$\texttt{apply} : \bar{c}$	$\bar{d}$	$\longmapsto$	$\square$	$e'[x \mapsto v]$	t	$(\bar{s}, e, \bar{c}) : \bar{d}$	(7)

Table 2.1.: The SECD machine's state transition rules.[6]
$\square$ denotes an empty stack, n any numeric literal, x any variable, $e(x)$ the value obtained by looking up x in e and $e[x \mapsto v]$ the environment obtained by extending e with the binding $x \mapsto v$ or, if x is already bound in e, overwriting it. Furthermore, $a : \bar{s}$ denotes an element a on top of the remaining stack $\bar{s}$.

The proposed data structure has four components, which, in retrospective, have given the machine its name and are as follows:

- **S**tack:
  The stack $\bar{s}$ is a Last In First Out (LIFO) structure for intermediate results, holding any number of evaluated terms (*values*). A value is either any *primitive data* n – like numeric constants – or, in case the evaluated expression is a λ-abstraction, which cannot be further reduced, a *closure*. A closure $(\lambda x.t, e)$ is a pair consisting of an abstraction $\lambda x.t$ and the *environment* e it is to be evaluated in.
- **E**nvironment:
  The environment e is a map-like structure that associates *identifiers* with their values, i.e., consists of pairs $x \mapsto v$ of identifiers x and values v, where each x is unique in the environment.
- **C**ontrol:
  The control $\bar{c}$ is a LIFO structure for machine instructions, holding any number of *directives*. A directive is either a λ-term t to be evaluated or the special `apply` directive.
- **D**ump:
  The dump $\bar{d}$ is a LIFO structure representing a call stack and holding any number of machine state snapshots, where the topmost is the one to return to once the current function call is completed. These snapshots are described by triples $(\bar{s}, e, \bar{c})$, also called *frames*,[5] of a stack, an environment and a control.

The transition rules given in Table 2.1 might provide a better understanding of these four components' role in the reduction of a λ-term. Computation starts by injecting term t, which is to be evaluated, with the initial state $(\square, \varnothing, t, \square)$. It is easy to see that the topmost element of

[5] Terminology taken from [21].
[6] The presented rules are taken from [22, p. 8] with some slight modifications, i.e., the absence of rules concerning primitive

the control stack essentially determines which rule is applied, with the exception of rules (1) and (2), that are dispatched according to the dump. Assuming the injected term to be well-formed and closed, the proposed data structure with its transition rules define a correct interpreter for λ-calculus which, given an input term t, yields the WNF of t, if it exists, or never halts [23, p. 117].

- Rule (1) terminates computation when encountering an empty control and dump, returning the stack's topmost (and only) value as evaluation result.
- Rule (2) corresponds to the return from a function call. Since no directive is left on the control stack, the machine adopts the (caller's) stack, environment and control stored in the dump's topmost frame and the callee's result v is appended to the new stack.
- Rule (3) specifies what to do when the next directive on the control stack is a numeric literal: it is simply put on top of the stack.
- Rule (4) does the equivalent for terms consisting of a single variable. Its value is looked up in the current environment and again pushed onto the stack.
- In rule (5), the next directive is a term consisting of a λ-abstraction, which is simply pushed onto the stack as a closure, together with the current environment.
- Rule (6) handles terms which consist of function applications: the function and argument term are both separately pushed onto the control stack, together with the special `apply` directive.
- Rule (7) performs the actual function application. Upon encountering the `apply` directive on the control stack, as well as a closure and an evaluated argument v on the stack, the closure's environment is extended by binding the closure's variable to v and adopted as new environment, the stack reset and the closure's function body left to be evaluated as control directive. Besides, the current stack, environment and control are saved on the dump.

It is essential to note the order in which the terms are pushed on the control stack in rule (6), by which the argument is evaluated *before* the function body. This use of the *call-by-value* [22, p. 17] strategy makes the SECD machine an *applicative order* evaluator [23, p. 117].

2.3. The Krivine machine

When evaluating an expression $((\lambda x.t_1)\ t_2)$ with call-by-value, t_2 is always evaluated (exactly) once, before its value is bound to x and t_1 is evaluated. An alternative to this strategy consists of evaluating t_2 only during evaluation of t_1, doing so each time x occurs, which can be any number of times (including zero, if x does not occur in t_1). This strategy is called *call-by-name*; and an example of an abstract machine implementing it can be found in the *Krivine machine*, which was, more than twenty years after its original formulation, introduced by French mathematician Jean-Louis Krivine (*1939) as a "particularly simple lazy machine which runs programs written in λ-calculus" [24, p. 1]. Aside from the evaluation strategy, it mainly differs from the SECD machine

functions for the sake of simplicity.

Term	Stack	Env.		Term	Stack	Env.	
$\lambda\, t$	$\square$	$\overline{e}$	$\longmapsto$	[Computation halts with result $(\lambda\, t, \overline{e})$]			(1)
$(t_1\ t_2)$	$\overline{s}$	$\overline{e}$	$\longmapsto$	t_1	$(t_2, \overline{e}) : \overline{s}$	$\overline{e}$	(2)
$\lambda\, t$	$(u, \overline{e'}) : \overline{s}$	$\overline{e}$	$\longmapsto$	t	$\overline{s}$	$(u, \overline{e'}) : \overline{e}$	(3)
$n+1$	$\overline{s}$	$(t, \overline{e'}) : \overline{e}$	$\longmapsto$	n	$\overline{s}$	$\overline{e}$	(4)
0	$\overline{s}$	$(t, \overline{e'}) : \overline{e}$	$\longmapsto$	t	$\overline{s}$	$\overline{e'}$	(5)

Table 2.2.: The Krivine machine's transition rules.
Notation conventions are the same as in Table 2.1. t_1, t_2, t, u are arbitrary λ-terms and n any De-Bruijn index.

in that it is conceived to operate on λ-terms in De-Bruijn notation.[7] Furthermore, it operates on pure λ-calculus instead of on the extended one.

The Krivine machine's state is defined by the following three components:

- Term:
  The single λ-*term* t currently being evaluated.
- Stack:
  The *stack* $\overline{s}$ is a LIFO structure holding any number of *closures*, each of one containing a term that is still to be evaluated. A closure is a pair (t, e) of an arbitrary[8] λ-term t and an environment e.
- Environment:
  The current *environment* $\overline{e}$, which is a stack of closures.

The transition rules are presented in Table 2.2. Computation is started by injecting a term t via the initial state $(t, \square, \square)$. It can be shown that this machine computes the WHNF for an input term t, if it exists, or does not terminate otherwise [24].

- Rule (1) terminates computation when encountering an empty stack and a λ-abstraction in the term component, returning a closure consisting of the abstraction and the current environment as evaluation result.
- Rule (2) handles function applications: the argument term is pushed onto the stack, together with the current environment, and the machine proceeds with evaluating the function term.
- Rule (3) handles λ-abstractions by transferring the stack's topmost closure to the environment. Then, the function body is evaluated.
- Rule (4) handles any non-zero De-Bruijn index: it is decremented by one and the environment is popped.
- Rule (5) replaces a zero index with the term in the environment's topmost closure and proceeds to evaluate it in the environment stored in the same closure.

[7] Krivine uses a variant of De-Bruijn indixes, which shows to be easily convertible to standard De-Bruijn notation. This allows for a simpler representation of the machine structure and rules, as in [25, p. 66], on Table 2.2 is based.
[8] Differing from the SECD machine, where closures are based on λ-abstractions.

In these rules, the machine's call-by-name nature becomes apparent: any function argument is not evaluated at once (as the SECD machine does it), but instead saved on the stack by rule (2), from where it will be transferred to the environment by rule (3). Rules (4) and (5) will again fetch it from the environment every time it is needed.

2.4. Other machines

The Krivine machine is only one among a plethora of alternatives to the SECD machine. As illustrated by [22], many structural derivations of the SECD machine can be built by omitting one or several of its components or changing its evaluation strategy.

Furthermore, other machines may operate on richer languages, like the CESK machine [26], which interprets a language supporting mutation and first-class continuations, among others. Although this kind of machine could also have been addressed in this work as far as implementation is concerned, the generation of correct example programs in these more complex languages for testing purposes would have made it significantly more difficult, which is why we restrict ourselves to the SECD and the Krivine machine.

3 Implementation

Essentially, the machines presented in the previous chapter are characterized by the nested data types defining their structure and the transition rules pattern-matching on their structure's state. Hence, implementing them may seem an easy task if undertook in a language with natural support of recursion, algebraic data types and pattern matching. However, SQL supports these features only within limits and thus poses a greater challenge, whose difficulty furthermore varies with the particularities of different DBMS. Starting with PostgreSQL, we will develop the main ideas behind any SQL implementation of both the SECD and the Krivine machine, before showing that efficient representation of environments is difficult within the limits of Vanilla PostgreSQL. Therefore, we will develop a variant which makes use of a hash table extension, as well as some implementations in other, newer DBMS.

3.1. Vanilla PostgreSQL

3.1.1. Input format and data types

Before addressing their evaluation, one must first find a way to represent λ-terms as such in relational databases. Translating the definition in Section 2.1 into a Haskell-style type definition reveals that any sensible type for λ-terms must make use of product, sum and recursive types:

```
data Term = Lit Int
          | Var String
          | Lam String Term
          | App Term Term
```

For product types `T = (T₁,...,Tₙ)`, whose values consist of tuples `(t₁,...,tₙ)` with `tᵢ` of type `Tᵢ` for all `i` $\in \{1, ..., n\}$, PostgreSQL's *composite* or *row types* can be used.[1] These can be created with `CREATE TYPE t AS (column_name₁ T₁, ..., column_nameₙ Tₙ);`. Unfortunately, PostgreSQL does not offer any equivalent for sum types `T = T₁ | ... | Tₙ`, whose values can be of one of the types `T₁`, ..., `Tₙ`. Instead, they can be simulated by using composite types and setting all columns to `null` except for the one whose type the expressed value is of.

Recursive types pose still a greater challenge, since PostgreSQL forbids their definition. One viable solution is to represent recursion via intra-table reference: instead of holding a subterm itself, the corresponding column in the `terms` table will only hold a reference to the one of its other entries that represents the subterm. Listing 3.1 shows the definition of table `terms` and its accompanying types that result from such an approach. `FOREIGN KEY` constraints would be

[1] See https://www.postgresql.org/docs/current/rowtypes.html (visited on 21.3.23).

```
{"app":
 {"fun":
  {"lam":
   {"var": "x",
    "body":
     {"var": "x"}}},
  "arg":
   {"lit": 42}}}
```

terms

id	lit	var	lam	app
1	□	x	□	□
2	□	□	(x,1)	□
3	42	□	□	□
4	□	□	□	(2,3)

Figure 3.1.: JSON (left) and tabular (right) representation of the term $((\lambda x.x)\ 42)$. In table `terms`, the root term is highlighted and □ stands for `null`.

appropriate, but are difficult to enforce, since self-reference only takes place in nested sub-columns.

```
1  CREATE DOMAIN term AS integer;
2  CREATE DOMAIN var  AS text;
3  CREATE TYPE lam AS (ide var, body term);
4  CREATE TYPE app AS (fun term, arg term);

5  CREATE TABLE terms (id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY, lit int,
       var var, lam lam, app app);
```

Listing 3.1: PostgreSQL type and table definitions for the language of extended λ-calculus used for the SECD machine.

Although this design solves the problem of internal term storage for the evaluation process, it does however not provide a practical way of inputting terms: in order to set up this tabular representation, terms have to be decomposed into individual subterms, which must be inserted into the table in an order guaranteeing referential integrity. While this can be achieved by a recursive User-Defined Function (UDF), such a function still needs an input type allowing for the expression of recursive data structures which are nested to an arbitrary, but limited depth. Admittedly, one could opt for a textual representation, but we prefer to rely on the existing PostgreSQL parsers for JavaScript Object Notation (JSON)[2] and decide to encode terms in JSON for initial input. Figure 3.1 provides an example of this and the tabular encoding. Implementing the procedure `load_term(t jsonb)` is now relatively straightforward. The result can be seen in Listing 3.2.

```
1  CREATE FUNCTION load_term(t jsonb) RETURNS term AS
2  $$
3  INSERT INTO terms(lit, var, lam, app) (
4      SELECT new.*
5      FROM jsonb_each(t) AS _(type, content),
6           LATERAL (
7              -- Case 1: Term is literal
8              SELECT lit, null, null, null
9              FROM jsonb_to_record(t) AS _(lit int)
10             WHERE type = 'lit'

11               UNION ALL
12             --Case 2: Term is variable
13             SELECT [...]
```

[2] See https://www.postgresql.org/docs/current/datatype-json.html (visited on 21.3.23).

```
              UNION ALL
            -- Case 3: Term is λ-abstraction
            SELECT null, null, row(var, load_term(body))::lam, null
            FROM jsonb_to_record(content) AS _(var var, body jsonb)
            WHERE type = 'lam'

              UNION ALL
            -- Case 4: Term is function application
            SELECT [...]

          ) AS new(lit, var, lam, app)
  )
  RETURNING id
$$
LANGUAGE SQL VOLATILE;
```

Listing 3.2: Excerpt of a PostgreSQL recursive UDF that loads a term from JSON into tabular representation in table `terms`. Cases 2 and 4 are analogous to cases 1 and 3, respectively. The full code can be inspected in Vanilla-PSQL/SECD/definitions.sql:33.

The question of term representation and import being settled, we will proceed with our machine implementations.

3.1.2. SECD machine

All of the discussed machines adhere to the following simple control flow:

```
while current machine state is not of final form do
    apply transition rule
end
```

An implementation of such a loop in SQL can be achieved by using SQL:1999's `WITH RECURSIVE` and following the pattern shown in Listing A.1. In this design, the Common Table Expression (CTE) `r` models the machine's state and each of its iterations represents a single step in the evaluation process.

```
-- CREATE TYPE env AS ???
CREATE TYPE primitive AS ENUM('apply');
CREATE TYPE closure AS (v var, t term, e env);    -- Closure = (Var, Term, Env)
CREATE TYPE val AS (c closure, n int);            -- Val = Closure | Int
CREATE DOMAIN stack AS val[];
CREATE TYPE directive AS (t term, p primitive);   -- Directive = Term | Primitive
CREATE DOMAIN control AS directive[];
CREATE TYPE frame AS (s stack, e env, c control); -- Frame = (Stack, Env, Control)
CREATE DOMAIN dump AS frame[];
CREATE TYPE machine_state AS (s stack, e env, c control, d dump);
```

Listing 3.3: PostgreSQL type definitions for the SECD machine.

Next comes the question of representing the machine's architecture in terms of a set of rigorous type declarations. While the LIFO structures of stack, control and dump can be implemented using PostgreSQL arrays,[3] yielding the type definitions in Listing 3.3, environments are more difficult to grasp. Again, examining the data structure in Haskell-style can help uncover the structural complexity:

[3] See https://www.postgresql.org/docs/14/arrays.html (visited on 21.3.23).

```
data Env = [(String, Val)]
data Val = Closure
         | Int
data Closure = Clo Var Term Env
```

The mutual recursion between types `Env` and `Closure`, through `Val`, calls for a self-referential approach similar to the one in Subsection 3.1.1. However, unlike the static table `terms`, which is not modified after the initial term import, a potential `environments` table would be dynamic in that the machine's transition rules would require it to change *during* term evaluation. Furthermore, later rule applications would rely on the environment modifications made by earlier applications. Thus, the evaluation query would perform inserts on the external table `environments` while simultaneously relying on the visibility of these changes. Unfortunately, the Atomicity, Consistency, Isolation, Durability (ACID) properties all database transactions fulfill[4] [27] make such a query unfeasible.[5] This restriction leaves us with two options: either finding a representation of environments which works internally, i.e., *within* the recursive CTE, or working around ACID. We will start by pursuing the former.

So far, the recursive CTE `r` holds one single row per iteration, representing the current machine state. In addition to this single `machine_state` row, an arbitrary number of `env_entry` rows can be added, representing the entries of a *virtual* `environments` table. On account of `WITH RECURSIVE`'s semantics, by which the recursive query can only access the rows produced in the *previous* iteration, these environment rows must be passed from one iteration to the next, and modified according to the applied rule.

As we recall from the transition rules in Table 2.1, two functions on environments need to be considered: lookup and extension, needed in rules (4) and (7). Unlike variable lookup, which is read-only, extension *modifies* an environment by adding or replacing a binding. Additionally, one needs to consider that, in rules (2) and (7), environments which are saved in closures on the stack or in frames on the dump need to be *restored*.

For the sake of example, let us consider the environment $\{(x, 42)\}$, first extend it by the binding $(y, 41)$ and then by $(x, 3)$, which overwrites the initial binding of x. After this, the initial environment $\{(x, 42)\}$ is restored from a closure and extended with $(y, 7)$. A first intuitive implementation approach suggests copying the environment which is being extended before adding a new binding to it, which would result in the `environments` table shown in Figure 3.2. However, this causes a significant performance overhead for extension. A second approach can be found in representing environments as a Directed Acyclic Graph (DAG), where extending e means adding a new binding which points to its ancestor e via self-reference in column `parent`, resulting in the tree and table shown in Figure 3.3 for our example.

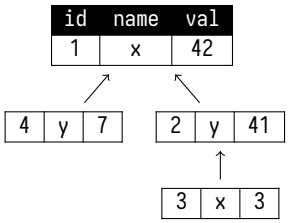
Yet, such a performance gain in extension may come at cost of a less efficient lookup, since instead of accessing the `environments` table once, it has to be accessed multiple times throughout graph traversal until the looked up variable is found. Testing, however, has shown that the

[4] In particular, PostgreSQL has been ACID-compliant since 2001, see https://www.postgresql.org/about/ (visited on 21.3.23).
[5] In the scenario [T_1 start … [T_2 start … T_2 end] T_1 end], where T_1 is the overall query transaction and T_2 a writing transaction caused by an invoked `INSERT INTO` statement, transaction isolation forbids T_1 to see changes introduced by T_2 (see https://www.postgresql.org/docs/14/transaction-iso.html, visited on 21.3.23).

environments

id	name	val
1	x	42
2	x	42
2	y	41
3	x	3
3	y	41
4	x	42
4	y	7

Figure 3.2.: `environments` table, as obtained when extending environments by copying in the discussed example.



environments

id	name	val	parent
1	x	42	
2	y	41	1
3	x	3	2
4	y	7	1

Figure 3.3.: Environments (left in DAG form, right in resulting tabular form), as obtained when extending them by reference and using a DAG representation in the discussed example.

second approach prevails nonetheless, as it is 50-100× faster than the first, which is therefore discarded. We can thus finalize the type definitions in Listing 3.3 by adding `CREATE DOMAIN env AS integer;` and `CREATE TYPE env_entry AS (id env, name var, val val, parent env);`.

Environment representation being settled, we can now modify the pattern in Listing A.1 in order to handle environments within the CTE. For this, we add the additional column `new_env_entry` to `step`, which contains the new entry by which an environment is to be extended, and augment the `WITH` expression by the CTE `new_envs`, whose purpose is to select the new environment entries from the ones of the previous iteration according to the rule applied in `step`. The resulting pattern is shown in Listing A.2. Substituting the specific details of the SECD rules for the placeholders $\texttt{injection\_expr}_i$, $\texttt{rule}_i\texttt{\_expr}_j$, etc., as well as implementing variable lookup with the recursive subquery shown in Listing 3.4 gives a correct Vanilla PostgreSQL implementation of the SECD machine.

Environment `id`s can be kept track of using PostgreSQL `SEQUENCE`s, as can be seen in Vanilla-PSQL/SECD/. Example terms in JSON representation can be found in `term_examples.txt` and an example evaluation is provided in Table B.4.

```
1  WITH RECURSIVE s(parent,name,val) AS (
2  -- traverse environment tree, starting from env
3    SELECT e.parent, e.name, e.val
4    FROM environments AS e
5    WHERE e.id = env

6      UNION ALL
```

```
  SELECT e.parent, e.name, e.val
  FROM s JOIN environments AS e(id,name,val,parent)
    ON s.parent = e.id
  WHERE s.name <> variable_name  ) -- stop as soon as variable_name is found
SELECT s.val
FROM s
WHERE s.name = variable_name
```

Listing 3.4: Recursive PostgreSQL subquery that looks up the value of `variable_name` in environment `env`, SECD machine.

3.1.3. Krivine machine

Implementing the Krivine machine requires several alterations to our SECD code: first of all, table `terms` needs to be adapted to the De-Bruijn notation and the absence of numeric literals. Second, type definitions need to be accommodated to the different machine structure. Since environments and closures are mutually recursive, a self-referencing, virtual `environments` table is going to be needed again. Additionally, the DAG approach shown in Figure 3.3 lends itself ideally to environments now being defined as stacks of closures. Listing 3.5 summarizes the ensuing definitions.

```
CREATE TABLE terms (id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY, i int, lam
     term, app app);

CREATE TYPE closure AS (t term, e env);    -- Closure = (Term, Env)
CREATE DOMAIN stack AS closure[];
CREATE TYPE machine_state AS (t term, s stack, e env);
CREATE TYPE env_entry AS (id env, c closure, parent env);
```

Listing 3.5: PostgreSQL type definitions for the Krivine machine.

Subsequently, one merely has to instantiate the pattern in Listing A.2 with expressions derived from the rules in Table 2.2. In doing so, particular attention needs to be devoted to rules (4) and (5): while all other rules write an *existing* subterm to the term component, rule (5) modifies an index value from n to $n-1$, thus creating a *new* term. This conflicts with our approach of representing terms in an *invariant* external table. Closer examination, however, reveals that, though presented separately, rules (4) and (5) jointly implement the Krivine-machine equivalent of variable lookup, i.e., the access of the environment's $n+1$-th closure upon encountering De-Bruijn index n in the term component. Therefore, they can be implemented jointly by a recursive subquery, as shown in Listing 3.6, avoiding the aforementioned issue of term alteration.

```
WITH RECURSIVE s(e,n) AS (
-- traverse the environment tree, starting from env
  SELECT env,i

    UNION ALL

  SELECT e.parent,s.n - 1
  FROM s JOIN environments AS e(id,c,parent)
    ON s.e = e.id
  WHERE s.n > 0   -- stop after i steps
)
```

```
10 SELECT s.e
11 FROM s
12 WHERE s.n = 0
```

Listing 3.6: Recursive PostgreSQL subquery that returns a reference to the `i+1`-th closure of environment `env`, Krivine machine.

The rest of the implementation is straightforward and can be inspected in `Vanilla-PSQL/Krivine/`. Table B.2 provides an example evaluation.

3.2. PostgreSQL with hash table extension

As we have seen in the previous section, the representation of environments poses a major problem, which hitherto we have solved within the recursive CTE, presumably at cost of performance due to the expensive copying of `env_entry` rows from iteration to iteration. Moreover, CTEs are temporary tables without indexes[6], which makes lookup comparatively slow. For this reason, [28] has explored the benefit of simulating CTE indexes with the help of a PostgreSQL hash table extension, which was developed by Denis Hirn at the Database Systems Research Group. Fortunately, this extension also provides a way to work around the restrictions imposed by ACID. The following section will show how relying on it considerably simplifies implementation of abstract machines and hopefully improves their performance.

3.2.1. Hash table extension

The Postgres hash table extension implements globally accessible hash tables using the type `TupleHashTable`.[7] As presented in [28, Ch. 3] in more detail, it provides the functions

- `prepareHT(tableId, numOfKeyColumns, colTypes)` to create a hash table,
- `insertToHT(tableId, override, rowToInsert)` to insert a row into a hash table or override it and
- `lookupHT(tableId, upsert, keyColumns)` to retrieve a row from a hash table.

Implemented in C, the stateful extension allows for circumventing transaction isolation. However, we shall not use it without the explicit mention that such an extension not only breaks ACID, but also SQL's declarativity [29, p. 2] (since programs now specify a form of control flow) . Yet, we consider these to be necessary concessions in view of the expected performance improvement, and proceed with implementing hash-table-based variants of the abstract machines.

3.2.2. SECD and Krivine machine

First, a hash table needs to be set up with column types corresponding to the `env_entry` type used in Section 3.1. For the SECD machine, e.g., this is done with `SELECT prepareHT(1, 1, null::env, null :: var, null :: val, null :: env);`. Since we can now rely on the external

[6] See `https://www.postgresql.org/docs/14/queries-with.html` (visited on 21.3.23).
[7] See `https://doxygen.postgresql.org/execnodes_8h_source.html#l00768` (visited on 21.3.23).

hash table to model the environments, the simpler pattern expressed in Listing A.1 suffices, which can be instantiated by using the same machine-specific expressions as in Section 3.1 for the most part, with the exception of all environment-interacting rules: having replaced the CTEs `environments` and `new_envs` with the external hash table, appropriate `insert`s and `lookup`s have to be formulated. Listings 3.7 and 3.8 show the result for the affected SECD rules, which yield the hash-table-based SECD implementation in Hashtables/SECD/.

```
1  WITH RECURSIVE s(parent,name,val) AS (
2    SELECT e.parent, e.name, e.v
3    FROM lookupHT(1,false,env) AS e(_ env, name var, v val, parent env)
4      UNION ALL
5    SELECT e.parent, e.name, e.v
6    FROM s,
7    LATERAL lookupHT(1,false,s.parent) AS e(_ env, name var, v val, parent env)
8    WHERE s.name <> variable_name
9  )
10 SELECT s.val
11 FROM s
12 WHERE s.name = variable_name
```

Listing 3.7: Hash-table-based lookup of `variable_name` in environment `env`, SECD machine.

```
1  SELECT [...],
2         (SELECT new_env
3          FROM (SELECT nextval('env_keys')::env) AS _(new_env),
4          LATERAL insertToHT(1, true, new_env, closure.v, arg, closure.e)), [...]
5  FROM machine AS ms,
6       LATERAL (SELECT ms.s[1].c.*) AS closure(v,t,e),
7       LATERAL (SELECT ms.s[2]) AS _(arg),
8       [...]
```

Listing 3.8: Hash-table-based rule (7), SECD machine, excerpt. The `LATERAL FROM`-entry in l. 4 performs the desired environment extension.

The Krivine machine can be implemented in the same fashion, as can be seen in Hashtables/Krivine/.

3.3. Other DBMS

Having found that implementing abstract machines in PostgreSQL requires quite a few clever workarounds, one might wonder about the situation in other DBMS. Furthermore, the question of performance is of particular interest to us. We therefore decide to attempt implementations in DuckDB and Umbra – both of which are DBMS that are in the making and promising with regards to performance – although we are well aware that their being in development will hinder implementation work.

Since the difficulties regarding the representation of environments encountered in Section 3.1 are rooted in DBMS-independent design principles – like ACID – we can expect to encounter them again and therefore start with the pattern in Listing A.2, which we will adapt to the particularities of the different DBMS, as sketched in the following subsections.

```
q₁

  UNION ALL

q₂;
```

```
WITH
  one(a,b) AS (
    q₁
  ),
  two(a,b) AS (
    q₂
  )

SELECT COALESCE(one.a, two.a), COALESCE(one.b, two.b)
FROM (SELECT (SELECT a FROM one), (SELECT b FROM one)) AS one(a,b),
     (SELECT (SELECT a FROM two), (SELECT b FROM two)) AS two(a,b);
```

Figure 3.4.: Original `UNION` query (left) and rewritten, `UNION`-less query (right).
q_1, q_2 are arbitrary queries that yield two columns each, matching in type. Equivalence of this rewrite only holds in a scenario where one of q_1, q_2 yields a single row and the other yields none: ll. 9 and 10 produce a single row for each of q_1, q_2's results, replacing the empty one by a `null` row. Other rewrites can be found for other scenarios.

3.3.1. DuckDB

Upon inspection of DuckDB's documentation, the tagged `UNION` type[8] stands out as a promising way of implementing sum types, whereas `STRUCT`s (the equivalent of PostgreSQL's row type) ideally lend themselves to product types and `LIST`s (that of arrays) can be used for the representation of stacks. Unfortunately, nested composition of types via subsequent `CREATE TYPE` statements shows not to be working, which forces us to write out the entire type definition in a nested expression, as can be seen in Listing 3.9.

```
STRUCT(s UNION(c STRUCT(v text, t integer, e integer), n int)[],
       e integer,
       c UNION(t integer, p primitive)[],
       d STRUCT(s UNION(c STRUCT(v text, t integer, e integer), n int)[],
                e integer,
                c UNION(t integer, p primitive)[])[])
```

Listing 3.9: A DuckDB type for the SECD `machine_state`.

`UNION` types allow us to store information in table `terms` in a single column instead of spreading it over multiple, as we did in PostgreSQL.

Another major hindrance is the missing implementation of the SQL `UNION` operator within recursive CTEs, of which the pattern in Listing A.2 makes heavy use. As illustrated in Figure 3.4, the use of `UNION`s can be avoided, but the resulting queries are of poor readability and, possibly, efficiency.

Environment extension must be done using the *copying* approach, as illustrated in Figure 3.2, since recursive CTEs within recursive CTEs, as needed for variable lookup in the previously used DAG-style environment extension (see Listing 3.7), are not permitted. Missing support of recursive CTEs in correlated subqueries and of recursive UDFs additionally handicap term import and evaluation. We solve the former by letting PostgreSQL create tabular representations of terms and importing them into DuckDB via `.csv` files, and the latter by automatically writing individual evaluation queries for every term to a `.sql` file with the use of the stream editor `sed`.

[8] See https://duckdb.org/docs/sql/data_types/union (visited on 21.3.23).

The resulting DuckDB implementation of the SECD machine can be inspected in DuckDB/SECD/. We will not pursue Krivine machine implementation in view of the SECD machine's poor performance discussed in Chapter 4.

3.3.2. Umbra

Umbra's type system shows to be even less promising than DuckDB's, since `CREATE TYPE` statements are not supported yet, rendering the definition of nested data types impossible. This enforces a "flat" design on the recursive CTE, e.g., for the Krivine machine,

```
WITH RECURSIVE r(finished, ms_t, ms_s, ms_e, e_id, e_c_t, e_c_e, e_p, env_key),
```

in which the original nested design can only be hinted at by column naming. Column `env_key` compensates for the missing implementation of `SEQUENCE`s and holds the next integer to be used as environment ID.

Moreover, no `row` constructor has been implemented yet. This forces us to represent stacks of tuples $(a_1, b_1) : (a_2, b_2) : \ldots$ either as single stacks of values – possibly being of different types, which would require us to wrap them in a common type, like `text` – $a_1 : b_1 : a_2 : b_2 : \ldots$, or as separate stacks $a_1 : a_2 : \ldots$ and $b_1 : b_2 : \ldots$. Since we want our code to stay within the limits of readability, we choose to implement the simpler Krivine machine, whose structure agrees well with the first of the the aforementioned variants.

Apart from this, the implementation is similar to our PostgreSQL one. As with DuckDB, we let PostgreSQL handle term import. The result, a Krivine machine implementation in Umbra, can be inspected in Umbra/Krivine/.

4

Evaluation

In the implementation variants described in the previous chapter, performance considerations played a key role. Therefore, it remains to be verified whether indeed the use of the hash table extension brings about the expected speedup, and how DuckDB or Umbra compare to PostgreSQL.

Before actual evaluation runtimes can be measured and discussed, we first need to determine which terms to test our machines on and develop an appropriate testing set-up.

4.1. Setup

4.1.1. Term generation

Since the few hand-picked example terms used for development so far cannot suffice to assess the performance of our implementations and no corpus of λ-terms is to be found online, we have no choice but to generate term sets ourselves. Bendkowski's *Lambda sampler*[1] [30], which offers random, Boltzmann-sampling-based [31] generation of λ-terms of arbitrary depth, can be used to this end. Implemented in Haskell, the sampler provides an option for generating only closed terms, as well as an interface for rejection-based filters. We use both to ensure that the generated terms are *closed* and *reducible*:[2] the former is needed because the implemented machines are not designed to deal with free variable occurrences, the latter because at least a few reduction steps need to be performed if performance is to be evaluated.

Bendkowski's sampler outputs terms in De-Bruijn notation represented as an Algebraic Data Type (ADT) in Haskell. Since the SECD machine operates on standard, variable-based notation, we first need a mean of conversion between the different notations. Therefore, we implement the function `toLambdaVar :: Lambda → LambdaVar`, which converts a given term of type `Lambda` – the provided, index-based representation – to one of type `LambdaVar` – our own ADT for variable-based terms, whose definition corresponds to the one sketched in Section 3.1.1. Furthermore, we need functions `lambdaToJSON :: Lambda → String` and `lambdaVarToJSON :: LambdaVar → String`, that convert the ADT representations to JSON. These Haskell functions can be found in `Generation/Generator.hs`.

Early testing showed that our machine implementations didn't halt on some of the generated terms within several minutes of computation. This is hardly surprising, as the machines are not

[1] See https://github.com/maciej-bendkowski/lambda-sampler (visited on 21.3.23).
[2] For instance, reducibility can be ensured by checking whether a term is of form $(\lambda x.t_1)\ t_2$.

Input: number of terms to generate n, boundaries on # of evaluation steps min and max
Output: set T of n terms

1 $ctr \leftarrow 0$, $T \leftarrow \varnothing$
2 **while** $ctr < n$ **do**
3 | generate term set T_{new} of size $n - ctr$
4 | $T_{new} \leftarrow \{t \in T_{new} \mid min \leq steps_M(t) \leq max\}$
5 | $T \leftarrow T \cup T_{new}$
6 | $ctr \leftarrow ctr + |T_{new}|$
7 **end**
8 **return** T

Algorithm 1: Sieving algorithm for generation of terms with specific reduction behavior. The function $steps_M$ in l. 4 maps any λ-term t to the number of steps a particular machine M needs to reduce it to normal form.

guaranteed to halt on terms which do not possess a normal form, like Ω. A λ-term t's normalization behavior under a particular evaluation strategy s is a *non-trivial* – certain terms reach normal form under s, while others don't – and *semantic* – it pertains to the behavior of a Turing machine M, which can be constructed to simulate the evaluation of t under s – property. Hence, it is generally undecidable by Rice's theorem [32, p. 398]. We therefore cannot, by means of however sophisticated filters, configure the sampler to discard terms whose evaluation doesn't terminate. This calls for the number of evaluation steps to be limited upwards.

Closer examination further reveals that this has to be done separately for the SECD and the Krivine machine, because evaluation strategy affects termination behavior. Consider the example of $t = ((\lambda x.(\lambda y.x))\ \Omega)$, which reduces to $t' = (\lambda y.\Omega)$ by weak head reduction. While this normal form t' is found by the lazy Krivine machine (that leaves Ω unevaluated), the SECD machine's call-by-value strategy evaluates Ω and subsequently never halts. In order to realize this upper boundary on evaluation steps, we implement some variants (`Generation/secd_interrupt.sql` and `Generation/krivine_interrupt.sql`) of our hash-table-based implementations, which abort evaluation after a specified number of steps.

Conversely, the number of evaluation steps also has to be limited downwards if sufficient term complexity is to be guaranteed.[3] This results in the selection, or *sieving*, procedure shown in Algorithm 1, that generates an arbitrary number of terms, given an upper and lower bound on the number of evaluation steps. This algorithm can be implemented using a combination of SQL and shell scripting, as can be seen in `Generation/secd_generation.sh` and `Generation/krivine_generation.sh`. Specifically, the actual sieving in l. 4 relies on the variants `krivine_interrupt` and `secd_interrupt` mentioned above.

4.1.2. Test sets

Equipped with the generation tools presented in the previous subsection, we can now proceed with the creation of test sets. We decide to create four sets of 100 terms each and varying execution difficulty, each covering a range of evaluation step numbers. Minimum and maximum

[3] Without the selection procedure described hereafter, most generated terms took no more than six steps to reach normal form.

	min	*max*	∅ # of eval. steps		∅ env. size		`terms` size	
			SECD	Krivine	SECD	Krivine	SECD	Krivine
I	1	25	11.3	6.46	2.06	2.55	26497	23894
II	25	200	65.5	58.76	12.9	20.47	25545	26519
III	200	1000	376.1	380.22	75.02	122.59	26731	27685
IV	1000	2000	1339.55	1384.64	267.71	447.91	34557	30703

Table 4.1.: Selected attributes of test sets I-IV:
the arguments *min* and *max* used for their generation with Algorithm 1, the average number of evaluation steps per term, the average number of `env_entry` rows per term and the size of table `terms`.

term depth used for the Lambda sampler is 100 and 1000, respectively. Generating terms with significantly more than 1000 evaluation steps is time-consuming, since such terms are rare in the Lambda sampler's output. Although their frequency could be increased by raising term depth, this would also lead to a far greater number of subterms and corresponding rows in table `terms`, which would slow down evaluation and obstruct performance comparison. For this reason, we choose not to generate any set with $min \geq 2000$.

Table 4.1 lists the generated sets along with some characterizing attributes.

As can be seen in the rightmost column, table `terms` holds a comparable number of rows for each set (with the exception of set IV resulting in slightly more rows, since terms with many evaluation steps tend to be deeper). Hence, we can assume its size and the scans on it (which need to be performed at most once per iteration, as we recall from Subsection 3.1.2) not to affect the following section's performance comparison to a noticeable extent. Column "∅ env. size" further reveals that the Krivine machine tends to create more environment entries than the SECD machine, which can be attributed to the former's call-by-name strategy.

4.2. Results and discussion

In the following, Umbra results can unfortunately not be presented nor discussed, since testing is made impossible by the inexplicable presence of a `Floating point exception`. It appears to only be raised when evaluating terms featuring De-Bruijn indexes of one or above, which suggests a connection with the nested recursive CTE in rules (4) / (5). Indeed, a minimal working example of recursive CTEs nested within each other yields incorrect results.

As for PostgreSQL and DuckDB, Table 4.2 shows *evaluation times* for term sets I-IV (all term import having been done prior to timing). The given measurements correspond to the median of five runs, performed on PostgreSQL v14.6 and DuckDB v0.6.1. The database systems were hosted on a 64-bit Linux machine with 96 AMD EPYC™ 7402 CPUs[4] at 2.8GHz and 2TB of RAM.

At first sight, the extreme slowness of `Duck:SECD` is salient. Several factors can account for this. First, it is the only implementation that extends environments by copying, which early

[4] Running `top` while testing reveals that only 100% of CPU, i.e., the equivalent of only one of 96 cores, is used, though.

	PG:SECD	PG:Kri	PG:HT:SECD	PG:HT:Kri	Duck:SECD
I	31.83	21.51	15.40	9.91	69773
II	166.39	181.42	53.18	43.08	257350
III	2958.47	4416.29	274.21	242.19	>10^6
IV	30206.25	47386.21	1034.79	887.14	>10^7

Table 4.2.: Evaluation runtimes in *ms*. `PG:SECD`, `PG:Kri` designate Vanilla PostgreSQL implementations, and `PG:HT:SECD`, `PG:HT:Kri` those relying on the hash table extension.

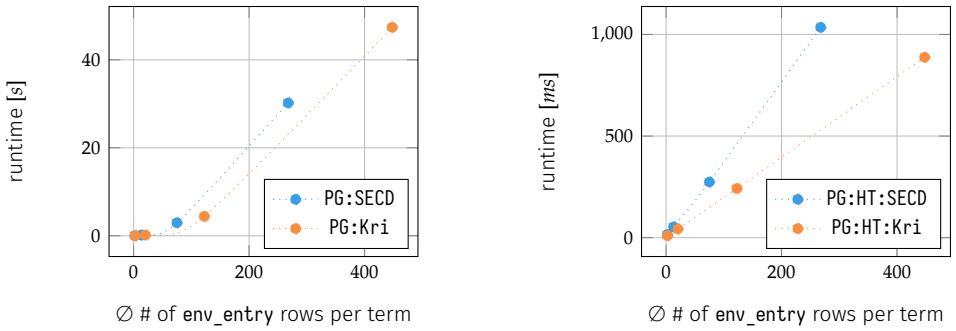


Figure 4.1.: Evaluation runtimes of the four PostgreSQL implementations plotted against the average environment sizes.

testing had already shown to be low-performing, even in PostgreSQL (but still 10× faster than DuckDB). Second, the cumbersome rewrites compensating for the missing `UNION` operator in recursive CTEs (see Section 3.3.1) might result in inefficient query plans. Third, as mentioned in Subsection 1.1.2, DuckDB's engine is specialized on processing rows in batches. Yet, the missing support of recursive CTEs within correlated subqueries limits queries to evaluating no more than a single term each, which might prevent the engine from exploiting its full potential.

Comparing `PG:SECD` to `PG:HT:SECD`, or `PG:Kri` to `PG:HT:Kri`, reveals that the use of the hash table extension indeed gives rise to a substantial speedup, reaching up to 53× for the Krivine machine on set `IV`. This benefit even seems to compensate for the Krivine machine's larger number of environment entries, as `PG:HT:Kri` performs better than `PG:HT:SECD` despite of the inverse holding for the Vanilla Postgres variants. As can be furthermore seen in Figure 4.1, the `PG:HT` implementations' runtimes increase approximately linearly with the average environment size, whereas the Vanilla Postgres one's evolve rather exponentially.

This discrepancy conforms to our expectations. We attribute it to the costly copying of an increasing number of `env_entry` rows from iteration to iteration within the recursive CTE, as well as to internal lookups in them, whose efficiency fades in comparison to the power of external hash table lookups.

Lastly, it should be emphasized that the clocked executions solely consist of term evaluation, consuming and producing tabular representations. Any conversion from or to nested representations (as JSON) requires additional work to be done.

5 Conclusion

We have successfully implemented two types of abstract machines in PostgreSQL: the SECD and the Krivine machine. For this, we have developed a tabular representation of terms, as well as a general approach to realizing such machines in SQL. In doing so, we have found that the representation of environments plays a crucial role and is neither easily nor efficiently achieved in Vanilla PostgreSQL. Hence, we have proposed an alternative solution relying on an ACID-breaking extension and seen that, indeed, outsourcing environment rows from recursive CTEs into an external hash table results in sizeable speedup. We have attempted to prove feasibility of our implementations in newer, potentially better-performing DBMS, but this is hindered by the missing support of certain syntactic constructs and the systems generally being unpolished, for still in the making.

5.1. Future work

Some aspects of our implementations offer potential for improvement and further development in future work:

- First, it has to be noted that our machines return references to volatile environments, that stop existing once query execution is completed. This deficit could be remedied by returning all environment rows in addition to the closure and, optionally, assembling these to a nested, final JSON value for post-processing, with the help of table `terms`.
- We designed environment entries to be added, but not to be deleted, resulting in a continuous increase in their number during term evaluation. Yet, this could be avoided by strategically garbage-collecting them (e.g., any environment not referenced in any of the machine's component can be safely deleted; but checking for this might be computationally expensive and thus counter the benefits of garbage-collection).
- The poor performance of the Vanilla PostgreSQL abstract machines is in parts due to the costly copying of environment rows from iteration to iteration, which is inevitable when using SQL's standard `WITH RECURSIVE`. Some elegant variations on the construct's semantics have been proposed and implemented in Postgres, some of which, like `WITH ITERATIVE` with upsert semantics [8, p. 2], suggest themselves for our purposes.
- DuckDB's missing support of `UNION`s within recursive CTEs constitutes a major limitation. Filling this gap,[1] as well as some others we encountered in our implementation attempt, will allow to re-tackle abstract machines in DuckDB.

[1] During the final stages of writing this thesis, a fix for this had already been developed within the Database Systems Research Group, but not yet included in the official DuckDB `cli` release, which was used for this work.

- The same holds for Umbra, which should be less burdensome to work with once development will be further advanced.

Finally, it shall be emphasized that, in effect, we have built SQL-based interpreters of λ-calculus, which evaluate λ-terms input in JSON notation. These interpreters could be extended and modified to support more features and richer languages, where a possible starting point could be Felleisen's CEK machine [26]. In any case, this thesis has provided insights into the SQL implementation of interpreting machines and the data structures they are built upon, which might help inform future work on source-to-source translations into SQL.

Bibliography

[1] Stephan Diehl et al. “Abstract machines for programming language implementation”. In: *Future Generation Computer Systems* 16.7 (2000), pp. 739–751. URL: https://www.sciencedirect.com/science/article/pii/S0167739X99000886.

[2] Donald D. Chamberlin and Raymond F. Boyce. “SEQUEL: A Structured English Query Language”. In: *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET ’74. Ann Arbor, Michigan: Association for Computing Machinery, 1974, pp. 249–264. URL: https://doi.org/10.1145/800296.811515.

[3] Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. 1st. Undergraduate Topics in Computer Science. London: Springer, 2010.

[4] Lawrence A. Rowe and Michael Stonebraker. “The POSTGRES Data Model”. In: *Proceedings of the 13th International Conference on Very Large Data Bases*. VLDB ’87. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, pp. 83–96.

[5] Denis Hirn and Torsten Grust. “One `WITH RECURSIVE` is Worth Many `GOTO`s”. In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD ’21. Virtual Event, China: Association for Computing Machinery, 2021, pp. 723–735. URL: https://doi.org/10.1145/3448016.3457272.

[6] Tim Fischer et al. “Snakes on a Plan: Compiling Python Functions into Plain SQL Queries”. In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD ’22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 2389–2392. URL: https://doi.org/10.1145/3514221.3520175.

[7] Tobias Burghardt et al. “Functional Programming on Top of SQL Engines”. In: *Practical Aspects of Declarative Languages: 24th International Symposium, PADL 2022, Philadelphia, PA, USA, January 17–18, 2022, Proceedings*. Philadelphia, PA, USA: Springer-Verlag, 2022, pp. 59–78. URL: https://doi.org/10.1007/978-3-030-94479-7_5.

[8] Denis Hirn and Torsten Grust. “A Fix for the Fixation on Fixpoints”. In: *Proceedings of the 13th Conference on Innovative Data Systems Research*. CIDR ’23. Amsterdam, Netherlands, 2023. URL: https://www.cidrdb.org/cidr2023/papers/p14-hirn.pdf.

[9] *SQL:1999 Standard. Database Languages–SQL–Part 2: Foundation*. ISO/IEC 9075-2:1999.

[10] Edgar F. Codd. “A relational model of data for large shared data banks”. In: *Communications of the ACM* 13.6 (1970), pp. 377–387. URL: https://dl.acm.org/doi/10.1145/362384.362685.

[11] Thomas M. Connolly and Carolyn E. Begg. *Database Systems – A Practical Approach to Design Implementation and Management*. 6th. Essex, England: Pearson, 2014.

[12] Mark Raasveldt and Hannes Mühleisen. “DuckDB: an Embeddable Analytical Database”. In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD ’19. Amsterdam, Netherlands: ACM, 2019, pp. 1981–1984. URL: https://duckdb.org/pdf/SIGMOD2019-demo-duckdb.pdf.

[13] Timo Kersten et al. “Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask”. In: *Proc. VLDB Endow.* 11.13 (2018), pp. 2209–2222. URL: https://www.vldb.org/pvldb/vol11/p2209-kersten.pdf.

[14] Thomas Neumann and Michael J. Freitag. “Umbra: A Disk-Based System with In-Memory Performance”. In: *Proceedings of the 10th Conference on Innovative Data Systems Research*. CIDR ’20. Amsterdam, Netherlands, 2020. URL: http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf.

[15] Alonzo Church. “A Set of Postulates for the Foundation of Logic”. In: *Annals of Mathematics* 33.2 (1932), pp. 346–366. URL: http://www.jstor.org/stable/1968337.

[16] Alan M. Turing. “Computability and λ-Definability”. In: *The Journal of Symbolic Logic* 2.4 (1937), pp. 153–163. URL: http://www.jstor.org/stable/2268280.

[17] Henk Barendregt and Erik Barendsen. *Introduction to Lambda Calculus*. 2000. URL: https://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf (visited on 12.3.2023).

[18] Małgorzata Biernacka et al. “The Zoo of Lambda-Calculus Reduction Strategies, And Coq”. In: *13th International Conference on Interactive Theorem Proving (ITP 2022)*. Vol. 237. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 7:1–7:19. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16716.

[19] Nicolas G. de Bruijn. “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem”. In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. URL: https://www.sciencedirect.com/science/article/pii/1385725872900340.

[20] Peter J. Landin. “The Mechanical Evaluation of Expressions”. In: *The Computer Journal* 6.4 (1964), pp. 308–320. URL: https://doi.org/10.1093/comjnl/6.4.308.

[21] Michael Sperber and Herbert Klaeren. “Exkurs: Die SECD-Maschine”. In: *Schreibe Dein Programm! Einführung in die Programmierung*. Periodically updated online book. 2023, pp. 479–522. URL: https://www.deinprogramm.de/sdp/ (visited on 12.3.2023).

[22] Olivier Danvy. “A Rational Deconstruction of Landin’s SECD Machine”. In: *BRICS Report Series* 10.33 (2003). URL: https://tidsskrift.dk/brics/article/view/21801.

[23] Werner E. Kluge. “Abstract λ-Calculus Machines”. In: *Central European Functional Programming School: Second Summer School, CEFP 2007, Cluj-Napoca, Romania, June 23-30, 2007, Revised Selected Lectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 112–157. URL: https://www.informatik.uni-kiel.de/~wk/springer.pdf.

[24] Jean-Louis Krivine. “A Call-by-Name Lambda-Calculus Machine”. In: *Higher Order Symbol. Comput.* 20.3 (2007), pp. 199–207. URL: https://doi.org/10.1007/s10990-007-9018-9.

[25] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. 2nd. Boston, MA: Birkhäuser Boston, 1993.

[26] Mattias Felleisen and Daniel P. Friedman. “A Calculus for Assignments in Higher-Order Languages”. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’87. Munich, Germany: Association for Computing Machinery, 1987, p. 314. URL: https://doi.org/10.1145/41625.41654.

[27] Theo Haerder and Andreas Reuter. “Principles of Transaction-Oriented Database Recovery”. In: *ACM Comput. Surv.* 15.4 (1983), pp. 287–317. URL: https://doi.org/10.1145/289.291.

[28] Madeleine Mauz. “Using Postgres Hash Table Extension in Compiled Functional-Style SQL UDFs”. MA thesis. Tübingen, Germany: Eberhard Karls Universität, 2022. URL: https://db.cs.uni-tuebingen.de/theses/2022/madeleine-mauz/Masterarbeit_Madeleine_Mauz.pdf.

[29] Mark S. Miller et al. “Uncanny Valleys in Declarative Language Design”. In: *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Vol. 71. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 9:1–9:12. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7129.

[30] Maciej Bendkowski et al. *Combinatorics of λ-terms: a natural approach*. 2016. URL: https://arxiv.org/abs/1609.07593.

[31] Philippe Duchon et al. “Boltzmann Samplers for the Random Generation of Combinatorial Structures”. In: *Combinatorics, Probability and Computing* 13.4-5 (2004), pp. 577–625. URL: https://algo.inria.fr/flajolet/Publications/DuFlLoSc04.pdf.

[32] John E. Hopcroft et al. *Introduction to Automata Theory, Languages, and Computation*. 3rd. Bposton, MA: Addison-Wesley, 2006.

Appendices

A. Listings

```
1  WITH RECURSIVE r(finished, mc₁, ..., mcᵣ) AS (
2    SELECT false, inj_expr₁, ..., inj_exprᵣ   -- inject term into machine
3      UNION ALL (
4    WITH
5      machine(mc₁, ..., mcᵣ) AS (
6        SELECT r.mc₁, ..., r.mcᵣ
7        FROM r
8        WHERE NOT r.finished  -- stop once terminating rule has been applied
9      ),
10     -- fetch details of the term stored in term_component from table terms
11     term(lit, var, lam, app) AS (
12         SELECT t.lit, t.var, t.lam, t.app
13         FROM machine AS ms, terms AS t
14         WHERE ms.term_component = t.id
15       ),
16     -- compute the next machine state
17     step(finished, mc₁, ..., mcᵣ) AS (
18       -- rule 1 (terminates computation)
19       SELECT true, rule₁_expr₁, ..., rule₁_exprᵣ
20       FROM machine
21       WHERE rule₁_cond
22         UNION ALL
23           .
24           .
25           .
26         UNION ALL
27       -- rule k
28       SELECT false, ruleₖ_expr₁, ..., ruleₖ_exprᵣ
29       FROM machine
30       WHERE ruleₖ_cond
31     )
32     SELECT *
33     FROM step
34   )
35 )
36 SELECT return_expr
37 FROM r
38 WHERE r.finished;
```

Listing A.1: Pattern for abstract machine implementation with external environment handling in PostgreSQL.
The machine state has `r` components $\texttt{mc}_1, \ldots, \texttt{mc}_r$ and `k` transition rules are specified, where rule `1` terminates computation, rule `i` applies on $\texttt{rule}_i\texttt{\_cond}$ and writes $\texttt{rule}_i\texttt{\_expr}_j$ into $\texttt{mc}_j$. `term_component` is the machine's component which holds the term on which is pattern-matched. Evaluation is started by injecting $\texttt{inj\_expr}_1, \ldots, \texttt{inj\_expr}_r$ into the machine and termination ensured by the `WHERE` clause in l.8, yielding `return_expr`. Each CTE `machine`, `term` and `step` holds at most a single row only.
Within the CTE `step`, expressions need to affect *external* objects representing environments.

```
WITH RECURSIVE r(finished, ms, e) AS (
  -- inject term into machine
  SELECT false,
         row(inj_expr₁, ..., inj_exprᵣ)::machine_state,
         null::env_entry

    UNION ALL (

  WITH
    r AS (TABLE r)      -- non-linear recursion hack
    machine(mc₁, ..., mcᵣ) AS (
      SELECT (r.ms).*
      FROM r
      WHERE r.ms IS NOT NULL AND NOT r.finished
    ),
    environments(id, name, val, parent) AS (
      SELECT (r.e).*
      FROM r
      WHERE r.e IS NOT NULL AND NOT r.finished
    ),
    -- fetch details of the term stored in term_component from table
       terms
    term(lit, var, lam, app) AS (
        SELECT t.lit, t.var, t.lam, t.app
        FROM machine AS ms, terms AS t
        WHERE ms.term_component = t.id
      ),
    -- compute the next machine state
    step(finished, mc₁, ..., mcᵣ, new_env_entry) AS (
      -- rule 1 (terminates computation)
      SELECT true, rule₁_expr₁, ..., rule₁_exprᵣ, null
      FROM machine
      WHERE rule₁_cond

        UNION ALL
           .
           .
           .
        UNION ALL
      -- rule k (extends environment)
      SELECT false, ruleₖ_expr₁, ..., ruleₖ_exprᵣ,
             row(new_env_id, new_name, new_val, extended_env)
      FROM machine
      WHERE ruleₖ_cond
    ),
    -- update the environments according to the rule applied in step
    new_envs(id, name, val, parent) AS (
      -- copy old environments
      SELECT e.*
      FROM step AS s, environments AS e
```

```
47          UNION ALL
48        -- extend environment if specified by s.new_env_entry
49        SELECT (s.new_env_entry).*
50        FROM step AS s
51        WHERE s.new_env_entry IS NOT NULL
52      )
53      SELECT s.finished,
54             row(s.mc_1, ..., s.mc_r)::machine_state,
55             null::env_entry
56      FROM step AS s

57          UNION ALL

58      SELECT s.finished,
59             null::machine_state,
60             ne::env_entry
61      FROM step AS s, new_envs AS ne
62    )
63  )
64  SELECT return_expr
65  FROM r
66  WHERE r.finished AND r.ms IS NOT NULL;
```

Listing A.2: Pattern for abstract machine implementation with CTE-internal environment handling in PostgreSQL.
Notation conventions are the same as in Listing A.1. Rule k is an environment-modifying rule, whereas rule 1 is not.

terms

id	i	lam	app	
1	1	□	□	1
2	□	1	□	$\lambda\, 1$
3	□	2	□	$\lambda(\lambda\, 1)$
4	1	□	□	1
5	□	4	□	$\lambda\, 1$
6	□	5	□	$\lambda(\lambda\, 1)$
7	□	□	(3,6)	$(\lambda(\lambda\, 1))\ (\lambda(\lambda\, 1))$
8	0	□	□	0
9	□	8	□	$\lambda\, 0$
10	□	□	(7,9)	$((\lambda(\lambda\, 1))\ (\lambda(\lambda\, 1)))\ (\lambda\, 0)$

Table B.1.: Tabular, index-based representation of $((\lambda(\lambda\, 1))\ (\lambda(\lambda\, 1)))\ (\lambda\, 0)$. On the right, each (sub-)term is shown in its written-out form. The root term is highlighted.

r

finished	ms			e			
	t	s	e	id	c	parent	
f	10	[]	1		□		
f	7	[(9,1)]	1		□		(2)
f	3	[(6,1),(9,1)]	1		□		(2)
f	2	[(9,1)]	2		□		(3)
f		□		2	(6,1)	1	
f	1	[]	3		□		(3)
f		□		2	(6,1)	1	
f		□		3	(9,1)	2	
f	6	[]	1		□		(4)/(5)
f		□		2	(6,1)	1	
f		□		3	(9,1)	2	
t	6	[]	1		□		(1)
t		□		2	(6,1)	1	
t		□		3	(9,1)	2	

Table B.2.: Evaluation of $((\lambda(\lambda\, 1))\ (\lambda(\lambda\, 1)))\ (\lambda\, 0) \rightsquigarrow (\lambda(\lambda\, 1), \varnothing)$ with the Vanilla PostgreSQL Krivine machine: all rows emitted by recursive CTE `r`.
For each `machine_state` row, the rule which has been applied is shown on the right. The empty environment $\varnothing$ is represented by `id 1`. Table B.1 shows the corresponding `terms` table and Listing 3.5 the relevant type definitions.

terms

id	lit	var	lam	app	
1	□	x	□	□	x
2	□	y	□	□	y
3	□	□	□	(1,2)	$(x\ y)$
4	□	□	(y,3)	□	$(\lambda y.(x\ y))$
5	□	□	(x,4)	□	$(\lambda x.(\lambda y.(x\ y)))$
6	□	x	□	□	x
7	□	□	(x,6)	□	$(\lambda x.x)$
8	□	□	□	(5,7)	$(\lambda x.(\lambda y.(x\ y)))\ (\lambda x.x)$
9	42	□	□	□	42
10	□	□	□	(8,9)	$(((\lambda x.(\lambda y.(x\ y)))\ (\lambda x.x))\ 42)$

Table B.3.: Tabular, variable-based representation of $(((\lambda x.(\lambda y.(x\ y)))\ (\lambda x.x))\ 42)$. Notation conventions are the same as in Table B.1.

r

finished	ms				e				
	s	e	c	d	id	name	val	parent	
f	[]	1	[10]	[]			□		
f	[]	1	[9,8,apply]	[]			□		(6)
f	[42]	1	[8,apply]	[]			□		(3)
f	[42]	1	[7,5,apply,apply]	[]			□		(6)
f	[(x,6,1),42]	1	[5,apply,apply]	[]			□		(5)
f	[(x,4,1),(x,6,1),42)]	1	[apply,apply]	[]			□		(5)
f	[]	2	[4]	[([42],1,[apply])]			□		(7)
f			□		2	x	(x,6,1)	1	
f	[(y,3,2)]	2	[]	[([42],1,[apply])]			□		(5)
f			□		2	x	(x,6,1)	1	
f	[(y,3,2),42]	1	[apply]	[]			□		(2)
f			□		2	x	(x,6,1)	1	
f	[]	3	[3]	[([],1,[])]			□		(7)
f			□		2	x	(x,6,1)	1	
f			□		3	y	42	2	
f	[]	3	[2,1,apply]	[([],1,[])]			□		(6)
f			□		2	x	(x,6,1)	1	
f			□		3	y	42	2	
f	[42]	3	[1,apply]	[([],1,[])]			□		(3)
f			□		2	x	(x,6,1)	1	
f			□		3	y	42	2	
f	[(x,6,1),42]	3	[apply]	[([],1,[])]			□		(5)
f			□		2	x	(x,6,1)	1	
f			□		3	y	42	2	
f	[]	4	[6]	[([],3,[]),([],1,[])]			□		(7)
f			□		2	x	(x,6,1)	1	
f			□		3	y	42	2	
f			□		4	x	42	1	
f	[42]	4	[]	[([],3,[]),([],1,[])]			□		(3)
f			□		2	x	(x,6,1)	1	
f			□		3	y	42	2	
f			□		4	x	42	1	
f	[42]	3	[]	[([],1,[])]			□		(2)
f			□		2	x	(x,6,1)	1	
f			□		3	y	42	2	
f			□		4	x	42	1	
f	[42]	1	[]	[]			□		(2)
f			□		2	x	(x,6,1)	1	
f			□		3	y	42	2	
f			□		4	x	42	1	
t	[42]	1	[]	[]			□		(1)
t			□		2	x	(x,6,1)	1	
t			□		3	y	42	2	
t			□		4	x	42	1	

Table B.4.: Evaluation of $(((\lambda x.(\lambda y.(x\ y)))\ (\lambda x.x))\ 42) \rightsquigarrow 42$ with the Vanilla PostgreSQL SECD machine: all rows emitted by recursive CTE `r`.
Notation conventions are the same as in Table B.2. Table B.3 shows the corresponding `terms` table. Additionally, any value of a sum type `T = T1 | T2`, which in our implementation is represented as `(a,□)` or `(□,b)`, is simply denoted by `a` or `b` for the sake of readability. This notation is unambiguous, as can be verified with the help of the type definitions shown in Listing 3.3.