SDQL Compiler in Lean

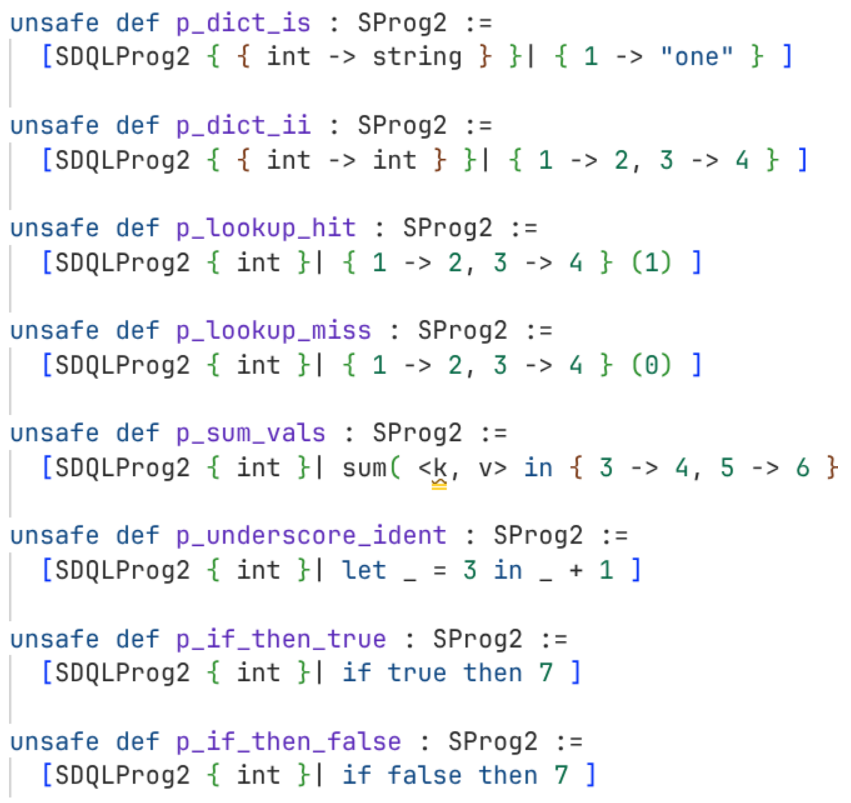
Supervised by Neel Krishnaswami

Attics Kuhn (ak2518)

https://github.com/AtticusKuhn/SDQL-Compiler

Contents

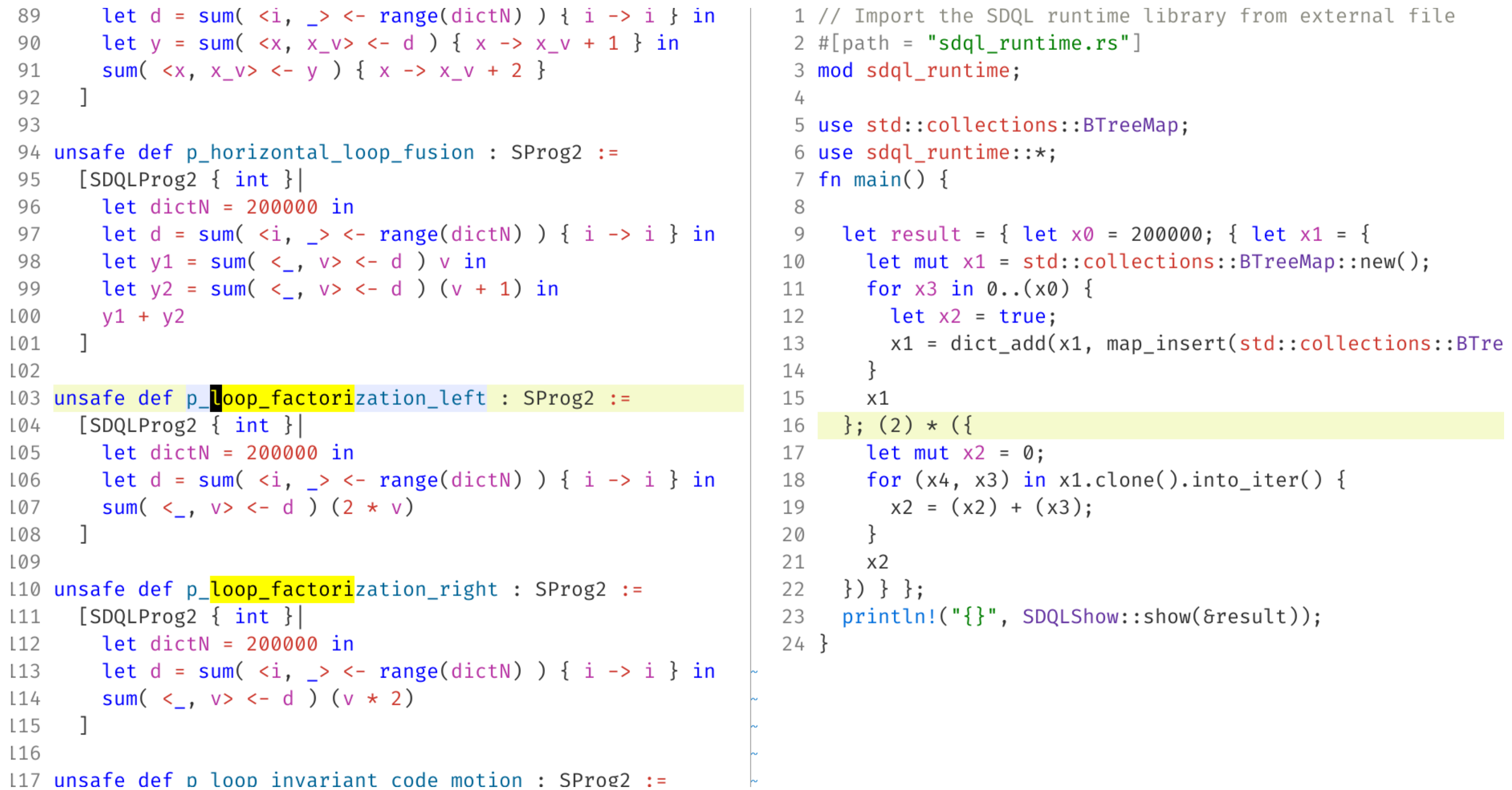
Work Completed



```
unsafe def p_dict_is : SProg2 :=
  [SDQLProg2 { { int -> string } }| { 1 -> "one" } ]

unsafe def p_dict_ii : SProg2 :=
  [SDQLProg2 { { int -> int } }| { 1 -> 2, 3 -> 4 } ]

unsafe def p_lookup_hit : SProg2 :=
  [SDQLProg2 { int }| { 1 -> 2, 3 -> 4 } (1) ]

unsafe def p_lookup_miss : SProg2 :=
  [SDQLProg2 { int }| { 1 -> 2, 3 -> 4 } (0) ]

unsafe def p_sum_vals : SProg2 :=
  [SDQLProg2 { int }| sum( <k, v> in { 3 -> 4, 5 -> 6 }

unsafe def p_underscore_ident : SProg2 :=
  [SDQLProg2 { int }| let _ = 3 in _ + 1 ]

unsafe def p_if_then_true : SProg2 :=
  [SDQLProg2 { int }| if true then 7 ]

unsafe def p_if_then_false : SProg2 :=
  [SDQLProg2 { int }| if false then 7 ]
```

Figure 1: A DSL for SDQL in Lean

Work Completed



```
    let d = sum( <i, _> <- range(dictN) ) { i -> i } in
    let y = sum( <x, x_v> <- d ) { x -> x_v + 1 } in
    sum( <x, x_v> <- y ) { x -> x_v + 2 }
  ]

unsafe def p_horizontal_loop_fusion : SProg2 :=
  [SDQLProg2 { int }|
    let dictN = 200000 in
    let d = sum( <i, _> <- range(dictN) ) { i -> i } in
    let y1 = sum( <_, v> <- d ) v in
    let y2 = sum( <_, v> <- d ) (v + 1) in
    y1 + y2
  ]

unsafe def p_loop_factorization_left : SProg2 :=
  [SDQLProg2 { int }|
    let dictN = 200000 in
    let d = sum( <i, _> <- range(dictN) ) { i -> i } in
    sum( <_, v> <- d ) (2 * v)
  ]

unsafe def p_loop_factorization_right : SProg2 :=
  [SDQLProg2 { int }|
    let dictN = 200000 in
    let d = sum( <i, _> <- range(dictN) ) { i -> i } in
    sum( <_, v> <- d ) (v * 2)
  ]

unsafe def p_loop_invariant_code_motion : SProg2 :=
```

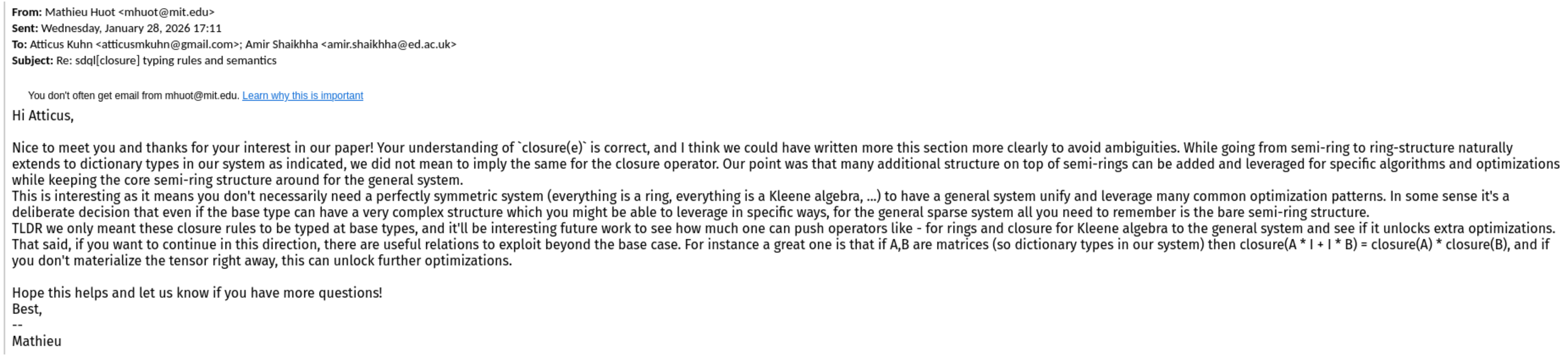
```
// Import the SDQL runtime library from external file
#[path = "sdql_runtime.rs"]
mod sdql_runtime;

use std::collections::BTreeMap;
use sdql_runtime::*;
fn main() {

  let result = { let x0 = 200000; { let x1 = {
    let mut x1 = std::collections::BTreeMap::new();
    for x3 in 0..(x0) {
      let x2 = true;
      x1 = dict_add(x1, map_insert(std::collections::BTre
    }
    x1
  }; (2) * ({
    let mut x2 = 0;
    for (x4, x3) in x1.clone().into_iter() {
      x2 = (x2) + (x3);
    }
    x2
  }) } };
  println!("{}", SDQLShow::show(&result));
}
```

Figure 2: Rust Code Generation

Work Schedule

Original Task	**Progress**
project core (SDQL compiler)	☑
algebraic rewrite optimisation	☑
graph path fixpoint	☒

Unexpected Difficulties



From: Mathieu Huot <mhuot@mit.edu>
Sent: Wednesday, January 28, 2026 17:11
To: Atticus Kuhn <atticusmkuhn@gmail.com>; Amir Shaikhha <amir.shaikhha@ed.ac.uk>
Subject: Re: sdql[closure] typing rules and semantics

You don't often get email from mhuot@mit.edu. Learn why this is important

Hi Atticus,

Nice to meet you and thanks for your interest in our paper! Your understanding of `closure(e)` is correct, and I think we could have written more this section more clearly to avoid ambiguities. While going from semi-ring to ring-structure naturally extends to dictionary types in our system as indicated, we did not mean to imply the same for the closure operator. Our point was that many additional structure on top of semi-rings can be added and leveraged for specific algorithms and optimizations while keeping the core semi-ring structure around for the general system.
This is interesting as it means you don't necessarily need a perfectly symmetric system (everything is a ring, everything is a Kleene algebra, ...) to have a general system unify and leverage many common optimization patterns. In some sense it's a deliberate decision that even if the base type can have a very complex structure which you might be able to leverage in specific ways, for the general sparse system all you need to remember is the bare semi-ring structure.
TLDR we only meant these closure rules to be typed at base types, and it'll be interesting future work to see how much one can push operators like - for rings and closure for Kleene algebra to the general system and see if it unlocks extra optimizations.
That said, if you want to continue in this direction, there are useful relations to exploit beyond the base case. For instance a great one is that if A,B are matrices (so dictionary types in our system) then closure(A * I + I * B) = closure(A) * closure(B), and if you don't materialize the tensor right away, this can unlock further optimizations.

Hope this helps and let us know if you have more questions!
Best,
--
Mathieu

Figure 3: Writing to the original paper authors