QIR – Specification

Romain Vernoux

January 6, 2017

Contents

1 Notations

Trees In this document, we will use a parenthesis notation to denote trees. This way, the tree leaves are just represented by their name and a tree rooted at a node N with n children $S_1, \ldots, S_n$ is denoted by $N(r_1, \ldots, r_n)$ where r_i is the representation of the subtree rooted at node S_i, recursively. For instance, if a tree T is composed of a node N which has two children P and Q such that Q has one child R, we will write T as $N(P, Q(R))$.

Contexts Contexts are trees in which some subtrees have been removed and replaced by "holes". A hole is denoted by $[]$. For instance a context C can be obtained by removing the subtree rooted at Q in T and will be denoted by $N(P, [])$. A substitution corresponds to filling a hole with a subtree. If a context C contains one hole, $C[S]$ corresponds to the tree C where the hole has been replaced by the tree S. For instance, if $C = N(P, [])$ then $T = C[Q(R)]$. This definition generalizes to contexts with n holes using the $C[S_1, \ldots, S_n]$ notation to denote the n holes subtitutions. Notice that the n holes are ordered using a prefix (depth-first, left-to-right) traversal of the tree, which corresponds to the left-to-right order while reading the tree notation we use in this document. For instance, if $C = N([], Q([]))$ then $T = C[P, R]$.

2 Architecture

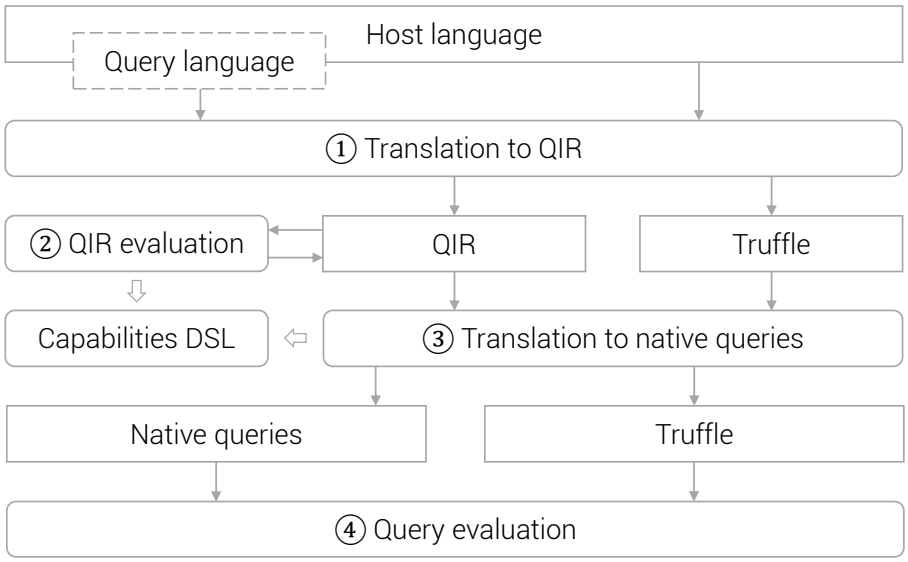

Figure 1: Architecture

The QIR workflow works as follows:

1. Queries specified in the host language syntax (possibly augmented with a special syntax for queries) are mapped to QIR constructs if possible, and kept as Truffle node black boxes otherwise. This translation phase also maps the data model of the host language to the QIR data model. The QIR constructs will be described in Section 3 and the translation mechanism from host language to QIR in Section 4.

2. QIR constructs are rewritten and partially evaluated within the QIR framework, using a description of the capabilities of the target database. The evaluation of QIR expressions will be described in Section 5 and the mechanism that exposes the capabilities of the database in Section 6.

3. The supported QIR operators and expressions are translated into the database native language, whereas expressions that are not supported and black box Truffle nodes are delegated to the Truffle runtime in the database. This translation phase also maps the QIR data model to the database data model. The translation of supported QIR constructs to native queries will be described in Section 7 and the mechanism that distributes Truffle nodes in Section 8.

4. The generated queries and the Truffle nodes are evaluated inside the database and the result is returned to the host language. This backward translation maps database native data to the data model of the host language. The evaluation of queries in the database will be described in Section 9 and the backward translation mechanism in Section 10.

3 QIR data model, expressions and operators

3.1 QIR data model

The QIR data model is independent of the data models of the host language and the database. Data objects are called *values*. The following grammar describes their syntax (for future use in this document) and hierarchy.

$$
\begin{array}{lll}
\langle \textit{Value} \rangle & ::= & \langle \textit{Number} \rangle \\
& | & \langle \textit{String} \rangle \\
& | & \langle \textit{Bool} \rangle \\
\\
\langle \textit{Number} \rangle & ::= & \langle \textit{Java double} \rangle \\
\\
\langle \textit{String} \rangle & ::= & \langle \textit{Java string} \rangle \\
\\
\langle \textit{Bool} \rangle & ::= & \langle \textit{Java boolean} \rangle
\end{array}
$$

3.2 QIR operators

Operators represent computation on database tables. They are designed to be close to operators of the relational algebra, in order to facilitate rewritings and translation to database native languages. Each operator has *children* expressions (in parentheses) and *configuration* expressions (in subscript). Children provide the input tables of the operator whereas parameters describe the computation on each table elements. Again, this is similar to the relational algebra.

$$
\begin{array}{lll}
\langle \textit{Operator} \rangle & ::= & \mathbf{Scan}_{\langle \textit{Expr} \rangle}() \\
& | & \mathbf{Select}_{\langle \textit{Expr} \rangle}(\langle \textit{Expr} \rangle) \\
& | & \mathbf{Project}_{\langle \textit{Expr} \rangle}(\langle \textit{Expr} \rangle) \\
& | & \mathbf{Sort}_{\langle \textit{Expr} \rangle}(\langle \textit{Expr} \rangle) \\
& | & \mathbf{Limit}_{\langle \textit{Expr} \rangle}(\langle \textit{Expr} \rangle) \\
& | & \mathbf{Group}_{\langle \textit{Expr} \rangle, \langle \textit{Expr} \rangle}(\langle \textit{Expr} \rangle) \\
& | & \mathbf{Join}_{\langle \textit{Expr} \rangle}(\langle \textit{Expr} \rangle, \langle \textit{Expr} \rangle)
\end{array}
$$

These operator constructors represent the following database computation:

- $\mathbf{Scan}_{table}()$ outputs the (unordered) list of elements in the target table corresponding to *table*.
- $\mathbf{Select}_{filter}(input)$ outputs the list of elements v in the list corresponding to *input* such that $filter\ v$ reduces to `true`.
- $\mathbf{Project}_{format}(input)$ outputs the list of elements corresponding to $format\ v$, with v ranging in the list corresponding to *input*.

- **Sort**$_{comp}(input)$ outputs the list of elements v in the list corresponding to *input* ordered according to *comp* v ascending.
- **Limit**$_{limit}(input)$ outputs the *limit* first elements of the list corresponding to *input*.
- **Group**$_{eq,agg}(input)$ outputs the list of elements corresponding to *agg* g for each group g in the partition of the elements v in the list corresponding to *input*, according to *eq* v.
- **Join**$_{filter}(input_1, input_2)$ outputs the join of the elements v_1 in the list corresponding to $input_1$ and the elements v_2 in the list corresponding to $input_2$, such that $filter\ v_1\ v_2$ reduces to `true`.

3.3 QIR expressions

QIR expressions form a small, pure language distinct from operators. They perform small computation on values. The following grammar describes their syntax (for future use in this document) and hierarchy.

⟨*Expression*⟩ ::= ⟨*Variable*⟩
| ⟨*Lambda*⟩
| ⟨*Application*⟩
| ⟨*Constant*⟩
| ⟨*TruffleNode*⟩
| ⟨*DataRef*⟩
| ⟨*ValueConstr*⟩
| ⟨*ValueDestr*⟩
| ⟨*ValueFun*⟩
| ⟨*Operator*⟩
| ⟨*BuiltinFun*⟩

⟨*Lambda*⟩ ::= '**λ**' ⟨*Variable*⟩ '`.`' ⟨*Expression*⟩

⟨*Application*⟩ ::= ⟨*Expression*⟩ ⟨*Expression*⟩

⟨*Constant*⟩ ::= ⟨*Value*⟩

⟨*ValueConstr*⟩ ::= ⟨*ListConstr*⟩
| ⟨*TupleConstr*⟩

⟨*ListConstr*⟩ ::= '`nil`'
| '`cons`' ⟨*Expression*⟩ ⟨*Expression*⟩

⟨*TupleConstr*⟩ ::= '`tnil`'
| '`tcons`' ⟨*String*⟩ ⟨*Expression*⟩ ⟨*Expression*⟩

⟨*ValueDestr*⟩ ::= ⟨*ListDestr*⟩
| ⟨*TupleDestr*⟩

⟨*ListDestr*⟩ ::= '`destr`' ⟨*Expression*⟩ ⟨*Expression*⟩ ⟨*Expression*⟩

⟨*TupleDestr*⟩ ::= '`tdestr`' ⟨*Expression*⟩ ⟨*String*⟩

⟨*ValueFun*⟩ ::= '`-`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`and`' ⟨*Expression*⟩
| '`if`' ⟨*Expression*⟩ '`then`' ⟨*Expression*⟩ '`else`' ⟨*Expression*⟩
| ...

⟨*BuiltinFun*⟩ ::= '`avg`' ⟨*Expression*⟩
| '`sum`' ⟨*Expression*⟩
| ...

Lambda-abstractions (resp. applications, constants) represent functions (resp. function applications, constants) of the host language.
The constructor for lists takes an expression for the head and an expression for the tail. Tables are represented by lists, since the output of queries might be ordered.
Tuples are constructed as a list of mappings: the constructor for tuples takes a string for the mapping name, an expression for the mapping value and an expression for the tail of the mapping list. Notice that this definition creates an order between key-value pairs in the tuples and allows duplicate keys (this is an arbitrary design choice).
The list destructor has three arguments: the list to destruct, the term to return when the list is a `nil` and a function with two arguments $\boldsymbol{\lambda}h.\boldsymbol{\lambda}t.M$ to handle lists with a head and a tail.
The tuple destructor has two arguments: the tuple to destruct and the attribute name of the value to return.
Finally, built-in functions represent common database functions (such as aggregation) for easier recognition and translation into native languages.
The formal definition of expression reduction will be given in Section 5.1.

In this document, we will sometimes use the syntactic sugar `let` $x = M$ `in` N for $(\boldsymbol{\lambda}x.\ N)M$ but the `let...in` constructor is not part of the expression language. Similarly, we will sometimes use the notation $t.attr$ for `tdestr` t `"attr"`, `not` x for `if` x `then false else true` and `let rec` $x = M$ `in` N as a shortcut for a fixpoint combinator.

4 Translation to QIR

TODO (Romain)

5 QIR evaluation

5.1 Reduction rules

Definition 1. *Variable substitution in QIR expressions (cf. Section 3.3) is defined similarly as in the pure lambda-calculus. The substitution of a variable x in e_1 by an expression e_2 is denoted $e_1\{e_2/x\}$.*

Definition 2. *The reduction rules for QIR expressions consist in the β-reduction rule augmented with the δ-reduction rules for destructors and ρ-reduction rules for primitive types.*

- $(\boldsymbol{\lambda} x.\ e_1)\ e_2 \rightarrow_\beta e_1\{e_2/x\}$.

- `destr nil` $e_{nil}\ e_{cons} \rightarrow_\delta e_{nil}$
- `destr (cons` $e_{head}\ e_{tail}$`)` $e_{nil}\ e_{cons} \rightarrow_\delta e_{cons}\ e_{head}\ e_{tail}$
- `tdestr (tcons "name1"` $e_{val1}\ e_{tail}$`) "name1"` $\rightarrow_\delta e_{val1}$
- `tdestr (tcons "name1"` $e_{val1}\ e_{tail}$`) "name2"` $\rightarrow_\delta$ `tdestr` e_{tail} `"name2"`

- `if true then` e_1 `else` $e_2 \rightarrow_\rho e_1$
- `if false then` e_1 `else` $e_2 \rightarrow_\rho e_2$
- `true and` $e_1 \rightarrow_\rho$ `true`
- `false and` $e_1 \rightarrow_\rho e_1$
- ...

In this document, we will use the notation $\rightarrow$ for the relation $\rightarrow_\beta \cup \rightarrow_\delta \cup \rightarrow_\rho$ and talk about redex for β-redexes, δ-redexes and ρ-redexes indifferently.

Reduction strategies will be given in Sections 5.3 and 5.4.

Theorem 1. *QIR with the above reduction rules has the Church-Rosser property (i.e. is confluent).*

Proof. Encoding (i) integer (resp. boolean, string) values and functions in the lambda-calculus (Church encoding), (ii) operators, built-in functions, Truffle nodes, Data references, `nil`, `cons`, `tnil` and `tcons` with constructors and (iii) `destr` and `tdestr` with (linear) pattern-matching, QIR can be transposed in a confluent lambda-calculus with patterns[3] for which the β, δ and ρ reduction rules are compatible. □

As we will see in the following sections, this is an important property, since it allows us to apply any reduction strategy without breaking the semantics of the input expression. Consequently, extending QIR with non-confluent constructs (e.g. side-effects) would require significant work.

Theorem 2. *QIR with the above reduction rules verifies the Standardization theorem, that is, if an expression e has a normal form, it can be obtained from e by reducing successively its left-outermost redex.*

Proof. With the same encoding as in the proof of Theorem 1, QIR can be transposed in a lambda-calculus with patterns[2] in which the Standardization theorem holds and for which the β, δ and ρ reduction rules are compatible. □

5.2 A measure for "good" output plans

Traditional databases use a tree of relational algebra operators (plan) to represent computation. Optimizations consist in commuting operators in the plan or applying local tree rewritings. The plan evaluation usually corresponds to (a variation of) a bottom-up evaluation of each operator, where a particular implementation is chosen for each operator for best performance.

For a same computation, such database engines benefit from working on a large, single plan instead of multiple small plans, since more optimizations become available and more statistics can be gathered to pick the best implementation for each operator. Moreover, there is no need to materialize and transfer intermediate results if all the computation is done in one plan.

Given an input QIR expression, the goal of the QIR evaluation module is to partially evaluate and rewrite it in order to (i) make its translation to the native database easier and (ii) create opportunities for database optimizations by grouping database computation in a small number of large queries, as much as possible.

As we will see later in this section, applying classical reduction strategies (e.g. call-by-value, call-by-name, lazy evaluation, etc.) until a normal form is reached does not satisfy the above requirements, since contracting some redexes might scatter parts of a same query. Instead, we will characterize the shape of the expressions we want to obtain as output of this module with a measure, then investigate reduction strategies that produce "good" expressions according to this measure.

Definition 3. *Supported operators are operators that are supported by the database, according to the mechanism described in Section 6.*

Definition 4. *Supported expressions are expressions that are supported inside operator configurations (cf. Section 3.2) by the database, according to the same mechanism.*

Definition 5. *Compatible operators are supported operators with supported expressions as configuration. Conversely, any other expression is called incompatible.*

Note that the compatibility of an operator is independent from the compatibility of its children.

Definition 6. *Let e be an expression. We say that a context F in e is a* fragment *if* $e = C[t(t_1, \ldots, t_{i-1}, F[e_1, \ldots, e_n], t_{i+1}, \ldots, t_j)]$ *or* $e = F[e_1, \ldots, e_n]$*, where:*

- *C is a one-hole context made of arbitrary expressions*

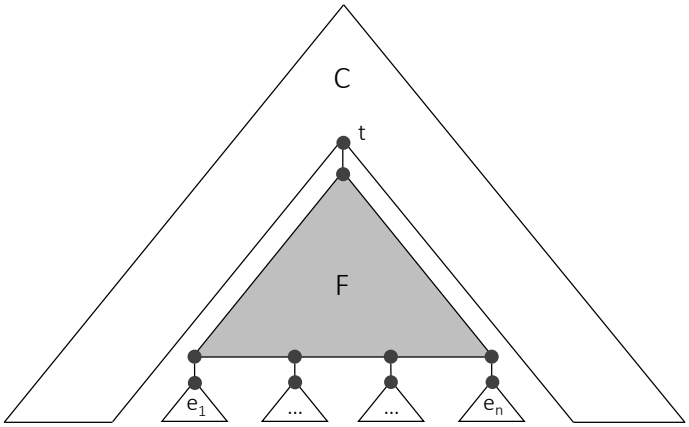

Figure 2: A fragment F

- *t is an incompatible j-ary expression*
- *$t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_j$ and F are the j children of t*
- *F is an n-hole context made of compatible operators only*
- *$e_1, \ldots, e_n$ all have an incompatible head expression*

This definition is illustrated in Figure 2.

Definition 7. *Let e be an expression. We define $\mathsf{Op}(e)$ as the number of operators in e (counting recursively in each subexpression), $\mathsf{Comp}(e)$ as the number of compatible operators in e (counting recursively in each subexpression) and $\mathsf{Frag}(e)$ as the number of fragments in e.*

Definition 8. *Let e be an expression. We define the measure for good output expressions as the pair $M(e) = (\mathsf{Op}(e) - \mathsf{Comp}(e), \mathsf{Frag}(e))$. Moreover, expressions are ordered using the lexicographical order on this measure.*

That is, a reduction $e \to^* e'$ is a good reduction if one of the two following conditions is met:

- $\mathsf{Op}(e') - \mathsf{Comp}(e') < \mathsf{Op}(e) - \mathsf{Comp}(e)$
- $\mathsf{Op}(e') - \mathsf{Comp}(e') = \mathsf{Op}(e) - \mathsf{Comp}(e)$ and $\mathsf{Frag}(e') < \mathsf{Frag}(e)$

The intuition about this measure is as follows:

- When a reduction transforms an incompatible operator into a compatible operator, the measure $\mathsf{Op}(e) - \mathsf{Comp}(e)$ decreases. For instance, in Example 1, $\mathsf{Op}(e) - \mathsf{Comp}(e) = 2 - 1 = 1$ (since f is a variable, it is not supported by the database) and $\mathsf{Op}(e') - \mathsf{Comp}(e') = 2 - 2 = 0$.

- When a reduction inlines operators as children of other operators, $\mathsf{Op}(e) - \mathsf{Comp}(e)$ stays the same (since no operator is created nor modified) and $\mathsf{Frag}(e)$ decreases. For instance, in Examples 2 and 3, $\mathsf{Op}(e) - \mathsf{Comp}(e) = \mathsf{Op}(e') - \mathsf{Comp}(e') = 2 - 2 = 0$, $\mathsf{Frag}(e) = 2$ and $\mathsf{Frag}(e') = 1$.
- In the infinite recursion of Example 4, unfolding one step of recursion does not change the measure, which hints that the plan is not getting better.
- This intuition generalizes well to binary operators, for which a reduction can duplicate operators (e.g. Example 5). If all duplicated operators are compatible, $\mathsf{Op}(e) - \mathsf{Comp}(e)$ stays the same (all the additional operators are compatible) therefore it is a good reduction if the number of fragments decreases. In this situation, the database will get a bigger plan in which it can perform optimizations (such as caching). Conversely, if a duplicated operator is incompatible, $\mathsf{Op}(e) - \mathsf{Comp}(e)$ increases, meaning that the reduction should not happen. In this situation, this operator should in fact be evaluated once in memory and its result cached to be reused multiple times during the expression evaluation. For instance, in Example 5, if C is supported, $\mathsf{Op}(e) - \mathsf{Comp}(e) = 3 - 3 = 0$, $\mathsf{Op}(e') - \mathsf{Comp}(e') = 5 - 5 = 0$, $\mathsf{Frag}(e) = 2$ and $\mathsf{Frag}(e') = 1$, and if C is not supported, $\mathsf{Op}(e) - \mathsf{Comp}(e) = 3 - 2 = 1$, $\mathsf{Op}(e') - \mathsf{Comp}(e') = 5 - 3 = 2$.

Example 1

$$\texttt{let } f = \boldsymbol{\lambda} x.\ x < 2 \texttt{ in } \mathbf{Select}_{\boldsymbol{\lambda} t.\ f(t.id)}(\mathbf{Scan}_X) \quad \to^* \quad \mathbf{Select}_{\boldsymbol{\lambda} t.\ t.id<2}(\mathbf{Scan}_X)$$

Example 2

$$\texttt{let } x = \mathbf{Scan}_X \texttt{ in } \mathbf{Select}_{\boldsymbol{\lambda} t.\ t.id<2}(x) \quad \to^* \quad \mathbf{Select}_{\boldsymbol{\lambda} t.\ t.id<2}(\mathbf{Scan}_X)$$

Example 3

$$\texttt{let } f = \boldsymbol{\lambda} x.\ \mathbf{Select}_{\boldsymbol{\lambda} t.\ t.id<2}(x) \texttt{ in } f(\mathbf{Scan}_X) \quad \to^* \quad \mathbf{Select}_{\boldsymbol{\lambda} t.\ t.id<2}(\mathbf{Scan}_X)$$

Example 4

$$\texttt{let rec } f = \boldsymbol{\lambda} x.\ \mathbf{Select}_{\boldsymbol{\lambda} t.\ t.id<2}(f(x)) \texttt{ in } f(\mathbf{Scan}_X) \quad \to^* \quad \texttt{let rec } f = \boldsymbol{\lambda} x.\ \mathbf{Select}_{\boldsymbol{\lambda} t.\ t.id<2}(f(x)) \texttt{ in } \mathbf{Select}_{\boldsymbol{\lambda} t.\ t.id<2}(f(\mathbf{Scan}_X))$$

Example 5

$$\texttt{let } x = \mathbf{Select}_C(\mathbf{Scan}_X) \texttt{ in } \mathbf{Join}_{\boldsymbol{\lambda} t1.\boldsymbol{\lambda} t2.\ t1.id=t2.name}(x, x) \quad \to^* \quad \mathbf{Join}_{\boldsymbol{\lambda} t1.\boldsymbol{\lambda} t2.\ t1.id=t2.name}(\mathbf{Select}_C(\mathbf{Scan}_X), \mathbf{Select}_C(\mathbf{Scan}_X))$$

Lemma 1. *The measure M defined above induces a well-founded order on expressions.*

Proof. The order induced by M is a lexicographical order on the natural order of positive integers, which is well-founded, and the lexicographical order preserves well-foundness. □

5.3 An exhaustive reduction strategy

Given the measure M described in Section 5.2 and an expression e, the question is now to find a reduced term e' such that $e \to^* e'$ and $M(e')$ is minimal. We will first consider an exhaustive strategy.

Definition 9. *Let e be an expression. We denote by $\mathsf{Rdxs}(e)$ the set of redexes in e, and for $r \in \mathsf{Rdxs}(e)$, we write $e \to_r e'$ to state that e' can be obtained from e in one reduction step by contracting r.*

Definition 10. *Let e be an expression. We recursively define $\mathsf{Red}_0(e)$ as the singleton $\{e\}$ and $\forall n > 0, \mathsf{Red}_n(e) = \{e'' \mid e' \in \mathsf{Red}_{n-1}(e), r \in \mathsf{Rdxs}(e'), e' \to_r e''\}$. We write $\mathsf{Reds}(e)$ to denote $\bigcup_{n\in\mathbb{N}} \mathsf{Red}_n(e)$. Finally we define $\mathsf{MinReds}(e)$ as the set $\operatorname{argmin}_{e'\in\mathsf{Reds}(e)} M(e')$.*

Notice that for some expressions e (e.g. expressions containing recursions), $\mathsf{Reds}(e)$ is infinite, $\mathsf{MinReds}(e)$ can be infinite and an algorithm that iteratively constructs the $\mathsf{Red}_n(e)$ will not terminate. Conversely, if e is strongly normalizing, $\mathsf{Reds}(e)$ and $\mathsf{MinReds}(e)$ are finite and such an algorithm terminates.

Theorem 3. *This reduction strategy is exhaustive, i.e. $\forall e, \forall e', e \to^* e' \Leftrightarrow e' \in \mathsf{Reds}(e)$.*

Proof. ($\Leftarrow$) If $e' \in \mathsf{Reds}(e)$ then $\exists n \in \mathbb{N}, e' \in \mathsf{Red}_n(e)$. The proof follows an induction on n. If $n = 0$ then $e = e'$ therefore $e \to^* e'$. For $n > 0$, we know that $\exists e'' \in \mathsf{Red}_{n-1}(e), \exists r \in \mathsf{Rdxs}(e''), e'' \to_r e'$. Thus, $e'' \to e'$ and by induction hypothesis $e \to^* e''$ therefore $e \to^* e'$.
($\Rightarrow$) If $e \to^* e'$ then $\exists n \in \mathbb{N}, e \to^n e'$. The proof follows an induction on n. If $n = 0$ then $e = e'$ therefore $e' \in \mathsf{Red}_0(e) \subseteq \mathsf{Reds}(e)$. For $n > 0$, we know that $\exists e'', e \to^{n-1} e'' \to e'$. By induction hypothesis, $e'' \in \mathsf{Reds}(e)$ therefore $\exists n' \in \mathbb{N}, e'' \in \mathsf{Red}_{n'}(e)$. It follows that $e' \in \mathsf{Red}_{n'+1}(e) \subseteq \mathsf{Reds}(e)$. □

Lemma 2. *$\forall e, \mathsf{MinReds}(e) \neq \emptyset$.*

Proof. Since $e \in \mathsf{Red}_0(e) \subseteq \mathsf{Reds}(e)$, we know that $\mathsf{Reds}(e) \neq \emptyset$. Moreover, the order induced by M is well-founded (Lemma 1) therefore M has a minimum on $\mathsf{Reds}(e)$ and $\mathsf{MinReds}(e)$ is non-empty. □

Theorem 4. *This reduction strategy is optimal, i.e. $\forall e, \forall e', e \to^* e' \Rightarrow e' \in \mathsf{MinReds}(e) \vee \exists e'' \in \mathsf{MinReds}(e), M(e'') < M(e')$.*

Proof. Suppose that $e \to^* e'$. Using Theorem 3 we know that $e' \in \mathsf{Reds}(e)$. Using Lemma 2, we know that $\mathsf{MinReds}(e) \neq \emptyset$ and we denote by M_{min} the measure M of the elements of $\mathsf{MinReds}(e)$. Then, either $M(e') = M_{min}$ and $e' \in \mathsf{MinReds}(e)$ by definition, or $M(e') > M_{min}$ and using again Lemma 2 we can find $e'' \in \mathsf{MinReds}(e), M(e'') < M(e')$. □

5.4 A heuristic-based reduction strategy

The reduction strategy described in Section 5.3 is not realistic in terms of complexity for a direct implementation, and might not even terminate in some cases. In this section, we will describe an efficient heuristic corresponding to a partial exploration of the possible reductions that guarantees termination.

The heuristic-based reduction strategy supposes the existence of an integer constant F representing the "fuel" that can be used by the reduction. It consists of two main passes. The first pass tries to reduce redexes that could make operators compatible by assuming that (i) operators with free variables in their configuration have few chances to be compatible and (ii) reducing redexes inside operator configurations increases the odds of making an operator compatible. The second pass tries to decrease the number of fragments by reducing redexes inside the children expressions of the operators. Both passes guarantee that the measure of the global expression always decreases after a number of reduction steps bounded by F.

For readability purposes, we will first describe the search space tree explored by the heuristic then define the result of the algorithm as a particular expression in this search space, but keep in mind that the actual implementation corresponds to a depth-first exploration of the search space, with decision points and back-tracking.

Definition 11. *Similarly to sets and the $\{x \mid P\}$ notation for set comprehensions, we use lists (the mathematical object) and the $[x \mid P]$ notation for list comprehensions. List comprehensions are order-preserving, that is, a list $[f(x) \mid x \in L]$ respects the order of L if L is a list.*

In the following definitions, we will use this notation to build the list of children of a node in a tree. For instance, $0([x+1 \mid x \in [1,2,3]])$ will stand for the tree $0(2,3,4)$. A node with an empty children list will be considered to be a leaf.

Definition 12. *Let e be an expression. We define its operators contexts $\mathsf{OpContexts}(e)$ as the set of contexts $\{C[] \mid e = C[op],\ op \text{ is an operator}\}$.*

Definition 13. *Let e be an expression and $C[]$ an operator context in $\mathsf{OpContexts}(e)$. We define the configuration free variables $\mathsf{ConfigFVars}(e, C[])$ as the list of variables $[v \mid e = C[op_{C'[v]}(\ldots)],\ v \text{ is a free variable in } C'[v]]$ sorted using a depth-first left-to-right traversal of $C'[v]$.*

Definition 14. *Let e be an expression and $C[]$ an operator context in $\mathsf{OpContexts}(e)$. We define the configuration redexes $\mathsf{ConfigRdxs}(e, C[])$ as the list of redexes $[r \mid e = C[op_{C'[r]}(\ldots)],\ r \text{ is a redex}]$ sorted using a depth-first left-to-right traversal of $C'[v]$.*

Definition 15. *Let e be an expression and $C[]$ an operator context in $\mathsf{OpContexts}(e)$. We define the children expressions $\mathsf{Children}(e, C[])$ as the list of expressions $[c_i \mid e = C[op_{\ldots}(c_1, \ldots, c_n)]]$.*

Definition 16. *The following three definitions are mutually recursive and therefore presented together in this document.*
Let e be an expression and e' a subexpression of e. We define $\mathsf{HMakeRedex}(e, e')$ as

- *e'', such that $e \to_r e''$, if e' is already a redex r*

- *HMakeRedex* (e, e'') *if* $e' = (e''\ e_1)$, $e' = \texttt{destr}(e'', e_1, e_2)$, $e' = \texttt{tdestr}(e'', s)$, $e' = \texttt{if}\ e''\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, $e' = e''\ \texttt{and}\ (\texttt{true/false})$ *or* $e' = (\texttt{true/false})\ \texttt{and}\ e''$ *(and similarily for other* ρ*-redexes).*
- *HInlineVar* (e, v) *if* e' *is a variable* v
- *None otherwise*

Let e *be an expression and* e' *a subexpression of* e*. We define* *HContractLam* (e, e') *as*

- *HMakeRedex* (e, e'')*, such that* $e = C[e'']$*, if* $e = C[e'\ e_1]$, $e = C[e_1\ e']$, $e = C[\texttt{destr}(e', e_1, e_2)]$, $e = C[\texttt{destr}(e_1, e', e_2)]$, $e = C[\texttt{destr}(e_1, e_2, e')]$, $e = C[\texttt{tdestr}(e', s)]$, $e = C[\texttt{if}\ e_1\ \texttt{then}\ e'\ \texttt{else}\ e_2]$ *or* $e = C[\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e']$
- *HContractLam* (e, e'')*, such that* $e = C[e'']$*, if* $e = C[\boldsymbol{\lambda} v.\ e']$, $e = C[\texttt{cons}(e', e_1)]$, $e = C[\texttt{cons}(e_1, e')]$, $e = C[\texttt{tcons}(s, e', e_1)]$ *or* $e = C[\texttt{tcons}(s, e_1, e')]$
- *None otherwise*

Let e *be an expression and* v *a variable in* e*. We define the inlining* *HInlineVar* (e, v) *as*

- *None if* v *is free in* e
- *HContractLam* (e, l)*, where* l *is the* $\boldsymbol{\lambda}$ *binding* v *in* e*, otherwise*

HMakeRedex (e, e') corresponds to (i) contracting e' if e' is already a redex and (ii) contracting a necessary redex in order for e' to become a redex, otherwise. HContractLam (e, e') corresponds to (i) contracting the redex containing e' if e' is already part of a redex and (ii) contracting a necessary redex in order for e' to become part of a redex, otherwise. HInlineVar (e, v) corresponds to contracting a necessary redex in order to eventually substitute v by its definition.

Consider for instance the expression e given in Example 6. To go towards the inlining of the variable tl in the configuration of the **Scan** operator, the heuristic calls HInlineVar (e, tl), which tries to contract the binding lambda by calling HContractLam $(e, \boldsymbol{\lambda} tl.\ \mathbf{Scan}_{f\ tl}())$. This lambda cannot be contracted before the lambda above it, and therefore HContractLam $(e, \boldsymbol{\lambda} hd.\ \boldsymbol{\lambda} tl.\ \ldots)$ is called recursively. Again, this lambda cannot be contracted before the head `destr`, but this requires its first child expression to be reduced to a `cons`. That's why the heuristic recursively calls HMakeRedex $(e, (\boldsymbol{\lambda} x.\ x)\ (\texttt{cons}\ \ldots))$, which contracts the argument redex.

Example 6

$e = \texttt{destr}\ ((\boldsymbol{\lambda} x.\ x)\ (\texttt{cons}\ 1\ (\texttt{cons}\ 2\ \texttt{nil})))$
$\texttt{false}\ (\boldsymbol{\lambda} hd.\ \boldsymbol{\lambda} tl.\ \mathbf{Scan}_{f\ tl}())$
$\rightarrow$
$e' = \texttt{destr}\ (\texttt{cons}\ 1\ (\texttt{cons}\ 2\ \texttt{nil}))$
$\texttt{false}\ (\boldsymbol{\lambda} hd.\ \boldsymbol{\lambda} tl.\ \mathbf{Scan}_{f\ tl}())$

Definition 17. *Let* e *be an expression and* $C[]$ *an operator context in* *OpContexts* (e)*. We define the one-step reduction of the operator configuration* *HOneRedConfig* $(e, C[])$ *as*

- *HInlineVar* (e, v) *if there is a first (in the list) variable* $v \in$ *ConfigFVars* $(e, C[])$ *such that* *HInlineVar* $(e, v) \neq$ *None*

- e', *such that* $e \to_r e'$, *if there is a first (in the list) redex* $r \in \mathsf{ConfigRdxs}(e, C[])$
- None *otherwise*

This means that for a given operator, the heuristic first tries to find a free variable that can be inlined, then if there is no such variable, it reduces the configuration using a left-outermost (call-by-name) reduction strategy, and finally if there is no redex to reduce, it returns None.

Definition 18. *Let* e *be an expression and* $C[]$ *an operator context in* $\mathsf{OpContexts}(e)$*. Using the notation* e' *for* $\mathsf{HOneRedConfig}(e, C[])$*, we define* $\mathsf{HRedConfig}_\phi(e, C[])$ *as*

- $e()$ *if* $\phi = 0$ *or if* $e' = \mathsf{None}$
- $e'([\mathsf{HRedConfig}_F(e', op) \mid op \in \mathsf{OpContexts}(e')])$ *if* $M(e') < M(e)$
- $e([\mathsf{HRedConfig}_{\phi-1}(e', op) \mid op \in \mathsf{OpContexts}(e')])$ *otherwise*

$\mathsf{HRedConfig}_\phi(e, C[])$ corresponds to the recursive exploration of the search space tree for reductions in configurations, with the guarantee that search space subtrees in which the measure does not decrease have a depth bounded by F.

Definition 19. *Let* e *be an expression. The search space* $\mathsf{HConfigSS}_F(e)$ *after the first pass is defined as* $e([\mathsf{HRedConfig}_F(e, op) \mid op \in \mathsf{OpContexts}(e)])$.

Definition 20. *Let* e *be an expression. The result expression* $\mathsf{HConfig}_F(e)$ *of the first pass is defined as follows. First, the rewrite rule* $x(s_1, \ldots, s_i, x(), s_{i+2}, s_n) \to x(s_1, \ldots, s_i, s_{i+2}, s_n)$ *is applied on the search space* $\mathsf{HConfigSS}_F(e)$ *as much as possible. Then, the left-most leaf is chosen.*

The rewrite rule corresponds to pruning search space subtrees in $\mathsf{HConfigSS}_F(e)$ that failed to reach an expression with smaller measure, whereas taking the left-most leaf corresponds to a heuristic decision.

Definition 21. *Let* e *be an expression and* $C[]$ *an operator context in* $\mathsf{OpContexts}(e)$*. We define the one-step reduction of the operator children* $\mathsf{HOneRedChild}(e, C[])$ *as*

- $\mathsf{HMakeRedex}(e, c)$ *if there is a first (in the list) child* $c \in \mathsf{Children}(e, C[])$ *such that* $\mathsf{HMakeRedex}(e, c) \neq \mathsf{None}$
- None *otherwise*

This means that for a given operator, the heuristic tries to find a children expression that can be reduced (using a call-by-name reduction strategy) in order to regroup fragments, and if there is no such child, it returns None.

Definition 22. *Let* e *be an expression and* $C[]$ *an operator context in* $\mathsf{OpContexts}(e)$*. Using the notation* e' *for* $\mathsf{HOneRedChild}(e, C[])$*, we define* $\mathsf{HRedChild}_\phi(e, C[])$ *as*

- $e()$ *if* $\phi = 0$ *or if* $e' = \mathsf{None}$
- $e'([\mathsf{HRedChild}_F(e', op) \mid op \in \mathsf{OpContexts}(e')])$ *if* $M(e') < M(e)$
- $e([\mathsf{HRedChild}_{\phi-1}(e', op) \mid op \in \mathsf{OpContexts}(e')])$ *otherwise*

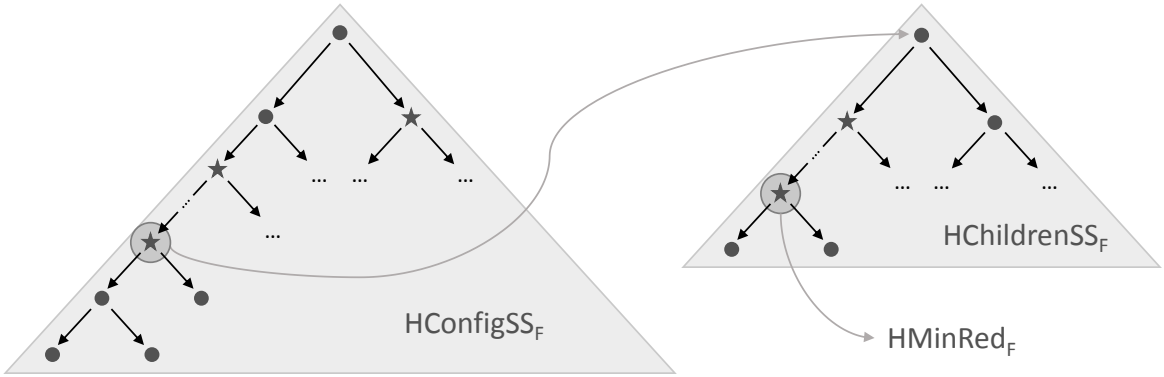

Figure 3: Relation between $\mathsf{HConfigSS}_F(e)$, $\mathsf{HChildrenSS}_F(\mathsf{HConfig}_F(e))$ and $\mathsf{HMinRed}_F(e)$

$\mathsf{HRedChild}_\phi(e, C[])$ corresponds to the recursive exploration of the search space tree for reduction in children, with the guarantee that search space subtrees in which the measure does not decrease have a depth bounded by F.

Definition 23. *Let e be an expression. The search space $\mathsf{HChildrenSS}_F(e)$ after the second pass is defined as $e([\mathsf{HRedChild}_F(e, op) \mid op \in \mathsf{OpContexts}(e)])$.*

Definition 24. *Let e be an expression. The result expression $\mathsf{HChildren}_F(e)$ of the second pass is defined as follows. First, the rewrite rule $x(s_1, \ldots, s_i, x(), s_{i+2}, s_n) \to x(s_1, \ldots, s_i, s_{i+2}, s_n)$ is applied on the search space $\mathsf{HChildrenSS}_F(e)$ as much as possible. Then, the left-most leaf is chosen.*

Similarly to Definition 20, the rewrite rule corresponds to pruning search space subtrees in $\mathsf{HChildrenSS}_F(e)$ that failed to reach an expression with smaller measure, whereas taking the left-most leaf corresponds to a heuristic decision.

Definition 25. *Let e be an expression. The result expression $\mathsf{HMinRed}_F(e)$ of the heuristic reduction is defined as $\mathsf{HChildren}_F(\mathsf{HConfig}_F(e))$.*

Figure 3 illustrates the construction of the two search spaces (cf. Definitions 19 and 23) leading to the computation of $\mathsf{HMinRed}_F(e)$. Dots and stars represent expressions considered by the heuristic as a possible reduction of their parent expression, but stars correspond to the special case where the measure of the expression is smaller than the measure of its parent, i.e. when the heuristic made progress.

We will now continue towards a proof of termination of this heuristic-based reduction strategy (Theorem 5).

Definition 26. *Considering an expression with subexpressions as a tree with subtrees, we define a traversal of an expression as follows, starting from the head expression:*

- *for a nullary expression, visit this expression.*
- *for $\boldsymbol{\lambda} x.\ e_1$, $\texttt{cons}\ e_1\ e_2$ and $\texttt{tcons}\ s\ e_1\ e_2$, visit this expression then traverse the children from left to right (e_1 then e_2 ... then e_n).*

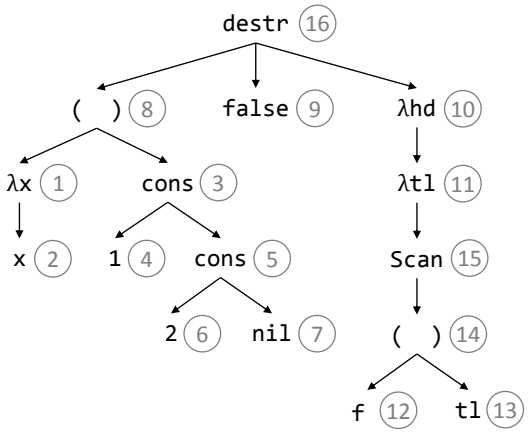

Figure 4: Example of expression traversal

- *for other n-ary expressions (e.g.* $(e_1\ e_2)$*,* $\texttt{destr}\ e_1\ e_2\ e_3$*, etc.), traverse the children from left to right (*e_1 *then* e_2 *... then* e_n*), then visit the expression.*

We denote by $\mathsf{TraversalPos}\,(e, e')$ *the position of a subexpression* e' *in this traversal of* e*.*

$\mathsf{TraversalPos}\,(e, .)$ actually induces a well-founded order on the locations of the subexpressions that are arguments of the recursive calls to $\mathsf{HInlineVar}\,(.,.)$, $\mathsf{HMakeRedex}\,(.,.)$ and $\mathsf{HContractLam}\,(.,.)$ made by the heuristic. It will be used in inductions in some of the following proofs.

For example, consider again the expression e from Example 6 and its traversal given in Figure 4. Remember the chain of recursive calls made by the heuristic ($\mathsf{HInlineVar}\,(e, tl)$, which calls $\mathsf{HContractLam}\,(e, \boldsymbol{\lambda} tl.\ \mathbf{Scan}_{f\ tl}())$, which calls $\mathsf{HContractLam}\,(e, \boldsymbol{\lambda} hd.\ \boldsymbol{\lambda} tl.\ \ldots)$, which indirectly calls $\mathsf{HMakeRedex}\,(e, (\boldsymbol{\lambda} x.\ x)\ (\texttt{cons}\ \ldots))$, which finally contracts this redex) and notice that $\mathsf{TraversalPos}\,(e, tl) > \mathsf{TraversalPos}\,(e, \boldsymbol{\lambda} tl.\ \mathbf{Scan}_{f\ tl}()) > \mathsf{TraversalPos}\,(e, \boldsymbol{\lambda} hd.\ \boldsymbol{\lambda} tl.\ \ldots) > \mathsf{TraversalPos}\,(e, (\boldsymbol{\lambda} x.\ x)\ (\texttt{cons}\ \ldots))$.

Lemma 3. *For all expression* e *the following three properties hold.*

- *For a variable* e' *in* e*,* $\mathsf{HInlineVar}\,(e, e')$ *returns* None *or contracts a redex.*
- *For a subexpression* e' *in* e*,* $\mathsf{HMakeRedex}\,(e, e')$ *returns* None *or contracts a redex.*
- *For a subexpression* e' *in* e*,* $\mathsf{HContractLam}\,(e, e')$ *returns* None *or contracts a redex.*

Proof. The proof follows an induction on $\mathsf{TraversalPos}\,(e, e')$.

- If e' is free in e then $\mathsf{HInlineVar}\,(e, e')$ returns None. Otherwise, denoting by l the lambda binding e', $\mathsf{HInlineVar}\,(e, e')$ corresponds to $\mathsf{HContractLam}\,(e, l)$, and since $\mathsf{TraversalPos}\,(e, l) < \mathsf{TraversalPos}\,(e, e')$, the induction hypothesis yields the result.
- If e' is a redex, $\mathsf{HMakeRedex}\,(e, e')$ contracts this redex. If $e' = (e''\ e_1)$, $e' = \texttt{destr}\ e''\ e_1\ e_2$, $e' = \texttt{tdestr}\ e''\ s$ or $e' = \texttt{if}\ e''\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ (and similarly for other ρ-redexes),

$\mathsf{HMakeRedex}\,(e, e')$ corresponds to $\mathsf{HMakeRedex}\,(e, e'')$ and since $\mathsf{TraversalPos}\,(e, e'') < \mathsf{TraversalPos}\,(e, e')$ the induction hypothesis yields the result. If e' is a variable, $\mathsf{HMakeRedex}\,(e, e')$ corresponds to $\mathsf{HInlineVar}\,(e, e')$ and we proved the result above. Otherwise, $\mathsf{HMakeRedex}\,(e, e')$ returns None.

- We denote by e'' the parent expression of e'. If $e'' = (e'\ e_1)$, $e'' = \texttt{destr}\ e'\ e_1\ e_2$, $e'' = \texttt{tdestr}\ e'\ s$ or $e'' = \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ (and similarly for other ρ-redexes), $\mathsf{HContractLam}\,(e, e')$ corresponds to $\mathsf{HMakeRedex}\,(e, e'')$ which either contracts e'' and we are done, or corresponds to $\mathsf{HMakeRedex}\,(e, e')$ and we proved the result above. If $e'' = (e_1\ e')$, $e'' = \texttt{destr}\ e_1\ e'\ e_2$, $e'' = \texttt{destr}\ e_1\ e_2\ e'$, $e'' = \texttt{if}\ e_1\ \texttt{then}\ e'\ \texttt{else}\ e_2$ or $e'' = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e'$ (and similarly for other ρ-redexes), $\mathsf{HContractLam}\,(e, e')$ corresponds to $\mathsf{HMakeRedex}\,(e, e_1)$ and since $\mathsf{TraversalPos}\,(e, e_1) < \mathsf{TraversalPos}\,(e, e')$, the induction hypothesis yields the result. If $e'' = \boldsymbol{\lambda} x.\ e'$, $e'' = \texttt{cons}\ e'\ e_1$, $e'' = \texttt{cons}\ e_1\ e'$, $e'' = \texttt{tcons}\ s\ e'\ e_1$, $e'' = \texttt{tcons}\ s\ e_1\ e'$, $\mathsf{HContractLam}\,(e, e')$ corresponds to $\mathsf{HContractLam}\,(e, e'')$ and since $\mathsf{TraversalPos}\,(e, e'') < \mathsf{TraversalPos}\,(e, e')$, the induction hypothesis yields the result. Otherwise, $\mathsf{HContractLam}\,(e, e')$ returns None.

□

These three properties not only show that the objects used to describe the heuristic are well defined, but also mean that correct implementations of $\mathsf{HInlineVar}\,(.,.)$, $\mathsf{HMakeRedex}\,(.,.)$ and $\mathsf{HContractLam}\,(.,.)$ terminate.

Lemma 4. *For all expression e, the search space $\mathsf{HConfigSS}_F\,(e)$ (resp. $\mathsf{HChildrenSS}_F\,(e)$) has bounded size.*

Proof. Each recursive call to $\mathsf{HRedConfig}_\phi\,(e, C[])$ (resp. $\mathsf{HRedChild}_\phi\,(e, C[])$) is well defined (corollary of Lemma 3) and is such that the pair $(M(e), \phi)$ decreases. Using lemma 1 it is easy to prove that the lexicographical order induced by this pair is also well-founded, therefore the search space has bounded depth. Moreover, any expression has a bounded number of operators and operator contexts therefore each node in the search space has a bounded number of children. □

Theorem 5. *This reduction strategy always terminates, that is, for an input expression e, it always considers a finite number of expressions e' such that $e \to^* e'$ in order to find $\mathsf{HMinRed}_F\,(e)$.*

Proof. Using Lemma 4, this reduction strategy only considers a finite number of expressions e_1 such that $e \to^* e_1$ in order to find $\mathsf{HConfig}_F\,(e)$, then using Lemma 4 again, it only considers a finite number of expressions e_2 such that $\mathsf{HConfig}_F\,(e) \to^* e_2$ in order to find $\mathsf{HChildren}_F\,(\mathsf{HConfig}_F\,(e)) = \mathsf{HMinRed}_F\,(e)$. □

We will now continue towards a proof that under some hypotheses this reduction strategy is complete, that is, it returns an optimal result for a big enough fuel value (Theorem 6).

In the following proofs, we will need a way to observe how subexpressions of an original expression are affected by a chain of reductions. Borrowing an idea used in [1] to define residuals, we will attach unique identifiers to all subexpressions of the original expression, and

reason on identifiers in the final expression. Definitions 27 and 28 correspond to a formalization of this concept.

Definition 27. *We define the language QIR* as the language of QIR constructs (cf. Section 3) in which integer identifiers can be attached to each subexpression and reductions are extended accordingly. Non-exhaustively, this means that*

- *QIR* operators contain* $\mathbf{Scan}_{e_1}()$, $\mathbf{Scan}^{id}_{e_1}()$, $\mathbf{Select}_{e_1}(e_2)$, $\mathbf{Select}^{id}_{e_1}(e_2)$, *etc.*
- *QIR* expressions contain* x, x^{id}, $\boldsymbol{\lambda} x.\ e_1$, $\boldsymbol{\lambda}^{id} x.\ e_1$, $(e_1\ e_2)$, $(e_1\ e_2)^{id}$, $\texttt{true}$, $\texttt{true}^{id}$ *etc.*
- $x^{id}\{e_1/x\} = e_1$, $x\{e_1/x\} = e_1$
- $(\boldsymbol{\lambda}^{id} x.\ e_1)\ e_2 \to e_1\{e_2/x\}$, $\texttt{destr}^{id}\ (\texttt{cons}^{id'}\ e_1\ e_2)\ e_3\ e_4 \to e_4\ e_1\ e_2$, $\texttt{true}^{id'}\ \texttt{and}^{id}\ e_1 \to \texttt{true}$, *etc.*

where id, id' *are integer identifiers and the* e_i *are QIR* expressions.*
Let e *be a QIR expression and* e_1 *a subexpression of* e. *We denote by* $\star(e)$ *the QIR* expression in which a unique identifier is attached to each subexpression of* e. *There is a one-to-one correspondance between the subexpressions of* e *and the subexpressions of* $\star(e)$, *thus we will also denote by* $\star_{\text{sub}}(e_1)$ *the subexpression of* $\star(e)$ *corresponding to* e_1.
Let e' *be a QIR* expression and* e'_1 *a subexpression of* e'. *We denote by* $\star^{-1}(e')$ *the QIR expression in which all identifers are removed from* e'. *There is a one-to-one correspondance between the subexpressions of* e' *and the subexpressions of* $\star^{-1}(e')$, *thus we will also denote by* $\star^{-1}_{\text{sub}}(e'_1)$ *the subexpression of* $\star^{-1}(e')$ *corresponding to* e'_1.
Let e' *be a QIR* expression and* e'_1 *a subexpression of* e'. *We denote by* $\mathsf{ExprId}(e', e'_1)$ *the identifier of* e'_1 *in* e', *if it has one (it is not defined otherwise).*

Lemma 5. *Let* e_0 *be a QIR expression and* e'_0 *a QIR* expression such that* $\star^{-1}(e'_0) = e_0$. *Any QIR reduction chain* $e_0 \to_{r_1} \ldots \to_{r_n} e_n$ *can be simulated by the QIR* reduction chain* $e'_0 \to_{r'_1} \ldots \to_{r'_n} e'_n$ *such that, for all* $1 \leq i \leq n$, $\star^{-1}(e'_i) = e_i$ *and* $\star^{-1}_{\text{sub}}(r'_i) = r_i$.

Proof. The proof follows an induction on n. For $n = 0$, the result is trivial. For $n > 0$, contracting the redex r'_1 such that $\star^{-1}_{\text{sub}}(r'_1) = r_1$ yields a QIR* expression e'_1. It is easy to prove by case analysis on r_1 that $\star^{-1}(e'_1) = e_1$. By induction hypothesis, the remaining of the reduction chain $e_1 \to^* e_n$ can be simulated by $e'_1 \to^* e'_n$, which means that the entire chain $e_0 \to^* e_n$ can be simulated by $e'_0 \to^* e'_n$. □

Definition 28. *Let* e_0 *be a QIR expression,* e *a subexpression of* e_0 *and* e_n *a QIR expression such that* $e_0 \to^* e_n$. *Using Lemma 5 and the fact that* $\star^{-1}(\star(e_0)) = e_0$, *we know that this QIR reduction chain can be simulated by a QIR* reduction chain* $e'_0 \to^* e'_n$ *starting from* $e'_0 = \star(e_0)$.
This way, we define $\mathsf{Occurences}(e_n, e)$ *as the set* $\{\star^{-1}_{\text{sub}}(e') \mid e' \text{ is a subexpression of } e'_n, \mathsf{ExprId}(e'_n, e') = \mathsf{ExprId}(\star(e_0), \star_{\text{sub}}(e))\}$.

For instance, the QIR reduction
$(\boldsymbol{\lambda} x.\ \boldsymbol{\lambda} y.\ f\ x\ x)\ z\ z \to (\boldsymbol{\lambda} y.\ f\ z\ z)\ z \to f\ z\ z$
can be simulated by the QIR* reduction
$(\boldsymbol{\lambda}^1 x.\ \boldsymbol{\lambda}^2 y.\ f^3\ x^4\ x^5)\ z^6\ z^7 \to (\boldsymbol{\lambda}^2 y.\ f^3\ z^6\ z^6)\ z^7 \to f^3\ z^6\ z^6$.

One can see, just by looking at identifiers, that the first z variable produced both z variables of the final expression : it has two occurences in the final expression. The other one

disappeared : it has no occurence in the final expression.

The following definitions and proofs will not use QIR* anymore, but only the $\mathsf{Occurences}(.,.)$ notation.

Definition 29. *Let e be an expression, e' a subexpression of e and r a redex such that $e \to_r e''$ and $\mathsf{Occurences}(e'', e') = \emptyset$. Borrowing a concept from [1], we say that r* erases *e' if e' is not part of the redex r, that is if e' is a subexpression of an expression e'' such that one of the following conditions holds.*

- $r = (\boldsymbol{\lambda} x.\ e_1)\ e''$ *and x is not free in e_1*
- $r = \texttt{destr nil}\ e_1\ e''$
- $r = \texttt{destr (cons ...)}\ e''\ e_1$
- $r = \texttt{tdestr (tcons "name1"}\ e_1\ e''\texttt{) "name1"}$
- $r = \texttt{tdestr (tcons "name1"}\ e''\ e_1\texttt{) "name2"}$
- $r = \texttt{if true then}\ e_1\ \texttt{else}\ e''$
- $r = \texttt{if false then}\ e''\ \texttt{else}\ e_1$
- $r = true\ \texttt{and}\ e''$ *(and similarly for other ρ-redexes)*

Definition 30. *Let e be an expression, e' a subexpression of e and r a redex such that $e \to_r e''$ and $\mathsf{Occurences}(e'', e') = \emptyset$. We say that r* consumes *e' if e' is part of the redex r, that is if one of the following conditions holds.*

- $r = e'$
- $r = (e'\ e_1)$
- $r = (\boldsymbol{\lambda} v.\ e_1)\ e_2$ *and e' is the variable v free in e_1*
- $r = \texttt{destr}\ e'\ e_1\ e_2$
- $r = \texttt{tdestr}\ e'\ s$
- $r = \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2$
- $r = e'\ \texttt{and}\ e_1$ *(and similarly for other ρ-redexes)*

Lemma 6. *Let e be an expression, e' a subexpression of e and r a redex in e. If $e \to_r e''$ and $\mathsf{Occurences}(e'', e') = \emptyset$, then r either consumes or erases e'.*

Proof. By case analysis on r. All cases are similar therefore we will only write the proof for $r = (\boldsymbol{\lambda} x.e_1)\ e_2$. By hypothesis, there is a context $C[]$ such that $e = C[r]$. Then we do a case analysis on the position of e' in e. We know that e' cannot be a subexpression of C, otherwise $\mathsf{Occurences}(e'', e') \neq \emptyset$. For the same reason, we also know that e' is not a subexpression of e_1 other than the free variable x. If $e' = r$, if $e' = \boldsymbol{\lambda} x.e_1$ or if e' is the variable x free in e_1, then e is consumed. Otherwise, e' is a subexpression of e_2 and x is not free in e_1 (since $\mathsf{Occurences}(e'', e') = \emptyset$), and in this case e' is erased. □

Definition 31. *Let e be an expression, e' a subexpression of e and r a redex in e such that $e \to_r e''$, we say that r* duplicates *e' if $|\mathsf{Occurences}\,(e'', e')| > 1$.*

Definition 32. *Let e_0 be an expression and e' a subexpression of e_0. We say that e'* cannot be erased *(resp.* consumed, duplicated*) if there is no reduction chain $e_0 \to_{r_1} \ldots \to_{r_n} e_n$ such that r_n erases (resp. consumes, duplicates) an element of $\mathsf{Occurences}\,(e_{n-1}, e')$.*

Lemma 7. *Let e be an expression and e' a subexpression of e. The following three properties hold.*

- *If e' is a variable v and if $\mathsf{HInlineVar}\,(e, v)$ returns None then e' cannot be consumed.*
- *If $\mathsf{HMakeRedex}\,(e, e')$ returns None then e' cannot be consumed.*
- *If $\mathsf{HContractLam}\,(e, e')$ returns None then e' cannot be consumed.*

Proof. The proof follows an induction on $\mathsf{TraversalPos}\,(e, e')$.

- Suppose that $\mathsf{HInlineVar}\,(e, v)$ returns None. If v is free in e then by definition it cannot be consumed. Otherwise, denoting by l the lambda binding v, this means that $\mathsf{HContractLam}\,(e, l)$ returns None. Since $\mathsf{TraversalPos}\,(e, l) < \mathsf{TraversalPos}\,(e, v)$, the induction hypothesis tells that l cannot be consumed and (by definition of the consumption of a variable) neither can e'.
- Suppose that $\mathsf{HMakeRedex}\,(e, e')$ returns None. If e' is a variable, this means that $\mathsf{HInlineVar}\,(e, e')$ returns None and we proved the result above. If e' is a redex, this is absurd ($\mathsf{HMakeRedex}\,(e, e')$ cannot return None). If $e' = (e''\ e_1)$, $e' = \texttt{destr}\ e''\ e_1\ e_2$, $e' = \texttt{tdestr}\ e''\ s$ or $e' = \texttt{if}\ e''\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ (and similarly for other ρ-redexes), this means that $\mathsf{HMakeRedex}\,(e, e'')$ returns None and since $\mathsf{TraversalPos}\,(e, e'') < \mathsf{TraversalPos}\,(e, e')$ the induction hypothesis tells that e'' cannot be consumed and (by definition of the consumption of the considered expressions for e') neither can e'. If $e' = \boldsymbol{\lambda} x.\ e_1$, $e' = \texttt{cons}\ e_1\ e_2$, $e' = \texttt{tcons}\ s\ e_1\ e_2$, $e' = \texttt{nil}$, $e' = \texttt{tnil}$ or if e' is a constant (integer, boolean, etc.), by looking at all the call sites of $\mathsf{HMakeRedex}\,(.,.)$ we know that the parent of e' is such that e' is in an ill-formed term (e.g. $\texttt{destr tnil}\ e_1\ e_2$) and therefore cannot be consumed. The remaining possible expressions for e' (e.g. Truffle nodes, Data references, operators, etc.) cannot be consumed in any expression.
- Suppose that $\mathsf{HContractLam}\,(e, e')$ returns None. We denote by e'' the parent expression of e'. If $e'' = (e'\ e_1)$, $e'' = \texttt{destr}\ e'\ e_1\ e_2$, $e'' = \texttt{tdestr}\ e'\ s$ or $e'' = \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ (and similarly for other ρ-redexes), this means that $\mathsf{HMakeRedex}\,(e, e'')$ returns None, which in turn means that $\mathsf{HMakeRedex}\,(e, e')$ returns None and we proved the result above. If $e'' = (e_1\ e')$, $e'' = \texttt{destr}\ e_1\ e'\ e_2$, $e'' = \texttt{destr}\ e_1\ e_2\ e'$, $e'' = \texttt{if}\ e_1\ \texttt{then}\ e'\ \texttt{else}\ e_2$ or $e'' = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e'$ (and similarly for other ρ-redexes), this means that $\mathsf{HMakeRedex}\,(e, e'')$ returns None, which in turn means that $\mathsf{HMakeRedex}\,(e, e_1)$ returns None. Since $\mathsf{TraversalPos}\,(e, e_1) < \mathsf{TraversalPos}\,(e, e')$, the induction hypothesis tells that e_1 cannot be consumed and (by definition of the consumption of the considered expressions for e'') neither can e'', which implies that e' cannot be consumed either. If $e'' = \boldsymbol{\lambda} x.\ e'$, $e'' = \texttt{cons}\ e'\ e_1$, $e'' = \texttt{cons}\ e_1\ e'$, $e'' = \texttt{tcons}\ s\ e'\ e_1$, $e'' = \texttt{tcons}\ s\ e_1\ e'$, this means that $\mathsf{HContractLam}\,(e, e'')$ returns None and since $\mathsf{TraversalPos}\,(e, e'') < \mathsf{TraversalPos}\,(e, e')$, the induction hypothesis tells that e'' cannot be consumed and (by definition of the consumption of the considered expressions for e') neither can e'. Similarly to above, the

remaining expressions to consider for e'' either correspond to ill-formed expressions (that cannot be consumed) or expressions that can never be consumed.

□

Definition 33. *We say that an expression e has the fixed operators property if no operator op in e can be erased or duplicated.*

Remember that the definition of compatible operators (Definitions 3, 4 and 5) depends on an (arbitrary) external module describing the capabilities of the target database (cf. Section 2). The following definition allows to

Definition 34. *We say that a database verifies the stable compatibility property if and only if, given an expression e, an operator op in $\{op \mid C[] \in \mathsf{OpContexts}(e),\ e = C[op]\}$ such that op is compatible and an expression e' such that $e \to^* e'$, each operator $op' \in \mathsf{Occurences}(op, e')$ is also compatible.*

This last definition should hold for a realistic database and an accurate description of its capabilities. Indeed, it basically says that if an operator is compatible, any reduction either does not affect the operator or helps the database by simplifying its configuration.

Lemma 8. *Let e be an expression with fixed operators and r a redex in e. For a database with stable compatibility, if $e \to_r e'$ then $M(e') \leq M(e)$.*

Proof. By case analysis on r. All cases are similar therefore we will only write the proof for $r = (\boldsymbol{\lambda} x.e_1)\ e_2$. By hypothesis, there is a context $C[]$ such that $e = C[r]$. Since e has fixed operators, there is a one-to-one correspondance between the operators of e and the operators of e'. For each operator op in e, denoting by op' the corresponding operator in e', the stable compatibility hypothesis tells us that if op is compatible, then op' is also compatible. Since no redex can create operators, this implies that $\mathsf{Op}(e') - \mathsf{Comp}(e') \leq \mathsf{Op}(e) - \mathsf{Comp}(e)$. The only case to treat is when $\mathsf{Op}(e') - \mathsf{Comp}(e') = \mathsf{Op}(e) - \mathsf{Comp}(e)$. Looking at the definition of fragments (cf. Definition 6), we see that there is only three ways to increase the number of fragments in e.

- Duplicate an existing fragment. This cannot happen under the fixed operator hypothesis, since a fragment contains at least one operator.
- Create a new fragment by making an incompatible operator compatible. This cannot happen either. Indeed, if r turns an incompatible operator into a compatible one, using the stable compatibility hypothesis, we know that all other compatible operators in e are still compatible in e' which contradicts $\mathsf{Op}(e') - \mathsf{Comp}(e') = \mathsf{Op}(e) - \mathsf{Comp}(e)$.
- Split an existing fragment into at least two fragments. This again, cannot happen. Indeed, let F be a fragment in $e = C[(\boldsymbol{\lambda} x.e_1)\ e_2]$. By definition of a fragment, we only have to consider the following cases:
  - if F is a subexpression of e_2 or $C[]$, it is intact in e'.
  - if F is a subexpression of e_1, either x is not free in F and F is not affected by the reduction or x is in the configuration of some operators of F and (stable compatibility) these operators stay compatible after reduction and r cannot split F.

– if r is in the fragment, it is necessarily in a configuration of an operator of the fragment which (stable compatibility) stays compatible after reduction, therefore r cannot split F .

□

Lemma 9. *Let e be a weakly-normalizing expression with fixed operators. For a database with stable compatibility, the normal form of e has minimal measure.*

Proof. Since e is weakly-normalizing, it has a normal form e_N. Suppose, to the contrary, that there exists e' such that $e \to^* e'$ and $M(e') < M(e_N)$. Using Theorem 1 (confluence) and the fact that e_N is normal, we know there is a finite reduction chain $e' \to^* e_N$ and applying Lemma 8 on each reduction of the chain leads to a contradiction. □

Lemma 10. *Let e be a weakly-normalizing expression with fixed operators, e_{min} an expression in $\mathsf{MinReds}(e)$ and e' an expression such that $e \to^* e'$ and $\mathsf{Op}(e') - \mathsf{Comp}(e') = \mathsf{Op}(e_{min}) - \mathsf{Comp}(e_{min})$. For a database with stable compatibility, an operator is compatible in e_{min} if and only if it is compatible in e'.*

Proof. Using the fixed operator hypothesis, we know that there is a one-to-one correspondance between the operators of e and the operators in any reduced form of e. Therefore, $\mathsf{Comp}(e') = \mathsf{Comp}(e_{min})$.
Suppose now, to the contrary, that there exists an operator op in e such that $\mathsf{Occurences}(e', op) = \{op'\}$, $\mathsf{Occurences}(e_{min}, op) = \{op_{min}\}$, op' compatible and op_{min} not compatible. Using Theorem 1 (confluence), we know there is an expression e'' such that $e' \to^* e''$ and $e_{min} \to^* e''$. Using the stable compatibility hypothesis, op is compatible in e'' and all operators compatible in e_{min} stay compatible in e'', which contradicts the minimality of the measure of e_{min}.
Similarly, if there is an operator op in e such that $\mathsf{Occurences}(e', op) = \{op'\}$, $\mathsf{Occurences}(e_{min}, op) = \{op_{min}\}$, op' not compatible and op_{min} compatible, the minimality of e_{min} tells that all operators compatible in e'' are also compatible in e_{min} and the stable compatibility hypothesis tells that each operator compatible in e' are still compatible in e'', which contradicts the fact that $\mathsf{Comp}(e') = \mathsf{Comp}(e_{min})$. □

Theorem 6. *For databases with stable compatibility, this reduction strategy is complete on strongly-normalizing expressions with fixed operators. That is, for a database with stable compatibility, given a strongly-normalizing input expression e with fixed operators, there exists a fuel value F such that $\mathsf{HMinRed}_F(e) \in \mathsf{MinReds}(e)$.*

Proof. Remember from Definition 10 that all expressions in $\mathsf{MinReds}(e)$ have same (minimal) measure. Using Lemma 9, we know that the normal form e_N of e is in $\mathsf{MinReds}(e)$. Let M_{min} be its measure. Consider now $e_h = \mathsf{HMinRed}_F(e)$. Using Theorem 4, we know that $e_h \in \mathsf{MinReds}(e)$ or $M(e_h) > M_{min}$ and we want to prove that the latter cannot happen. Suppose to the contrary that $M(e_h) > M_{min}$. Using the definition of M (cf. Definition 8), this means that one of the two following statements holds.

- $\mathsf{Op}(e_h) - \mathsf{Comp}(e_h)$ is greater than the first component of M_{min}
- $\mathsf{Op}(e_h) - \mathsf{Comp}(e_h)$ is equal to the first component of M_{min} and $\mathsf{Frag}(e_h)$ is greater than the second component of M_{min}

We will prove that none of these cases can happen.

- Suppose that $\mathsf{Op}(e_h) - \mathsf{Comp}(e_h)$ is greater than the first component of M_{min}. Since e has the fixed operators property, there is a one-to-one correspondance between the operators of e_N and e_h. Therefore, we know that $\mathsf{Comp}(e_h) < \mathsf{Comp}(e_N)$ and there exists an operator op in e such that $\mathsf{Occurences}(e_h, op) = \{op_h\}$, $\mathsf{Occurences}(e_N, op) = \{op_N\}$, op_N is compatible and op_h is not compatible. Let c_h (resp. c_N) be the configuration of op_h (resp. op_N). The question now is to understand how the first pass of the heuristic-based algorithm (cf. Definition 20) could fail to make op compatible. Remember Lemma 3 telling that $\mathsf{HInlineVar}(.,.)$ and $\mathsf{HMakeRedex}(.,.)$ either contract a redex or return None, and keep in mind that such reductions maintain a single instance of c_h in the reduced forms of e_h (fixed operator hypothesis). Since e is strongly-normalizing, this means that there is a fuel value F allowing the heuristic to make enough calls to $\mathsf{HInlineVar}(.,.)$ on the free variables of c_h in order to get to an expression $e'_h = C'[op_{c'_h}(\ldots)]$ such that (i) there is no free variable in c'_h or (ii) calls to $\mathsf{HInlineVar}(e'_h, .)$ return None for every free variable of c'_h. Continuing from this point, since e is strongly-normalizing, e'_h and c'_h are also strongly normalizing. Thus, Theorem 2 tells that there is a fuel value F allowing the heuristic to reduce all redexes of c'_h and reach a normal form c''_h and an expression $e''_h = C'[op_{c''_h}(\ldots)]$. Since we supposed that the heuristic failed to make op compatible, this means that c''_h is different from c_N. Using Theorem 1 (confluence), we know there is reduction $e''_h \to^* e_N$. Since the redexes contracted in this chain cannot erase nor duplicate operators (fixed operator hypothesis), the reduction chain can only affect c''_h in the following ways.

  - Substitute free variables in c''_h. This cannot happen: by hypothesis, either (i) there is no free variable in c'_h and therefore in c''_h or (ii) calls to $\mathsf{HInlineVar}(e'_h, .)$ return None for every free variable of c'_h and using Lemma 7, such a variable cannot be consumed.
  - Reduce redexes located inside c''_h. This cannot happen since c''_h is in normal form.
  - Leave c''_h untouched. This leads to a contradiction: c''_h is equal to c_N.

  Therefore, there is a fuel value such that the heuristic makes op compatible. Now, taking the maximum of the required values of F to make each operator compatible, there exists a value of F such that $\mathsf{Op}(e_h) - \mathsf{Comp}(e_h)$ is equal to the first component of M_{min}.

- Suppose now that $\mathsf{Op}(e_h) - \mathsf{Comp}(e_h)$ is equal to the first component of M_{min} and $\mathsf{Frag}(e_h)$ is greater than the second component of M_{min}. Since e has the fixed operators property, there is a one-to-one correspondance between the operators of e_N and e_h. Using Lemma 10, we know that there exists an operator op in e such that $\mathsf{Occurences}(e_h, op) = \{op_h\}$, $\mathsf{Occurences}(e_N, op) = \{op_N\}$, op_N and op_h are both compatible, op_N has a compatible child operator c_N and the child expression c_h of op_h is incompatible (i.e. not a compatible operator). The question now is to understand how the second pass of the heuristic-based algorithm (cf. Definition 24) could fail to reduce c_h to a compatible operator. Remember Lemma 3 telling that $\mathsf{HMakeRedex}(.,.)$ either contracts a redex or returns None, and keep in mind that such reductions maintain a single instance of op_h in the reduced forms of e_h (fixed operator hypothesis). Since e is strongly-normalizing, this means that there is a fuel value F allowing the heuristic to make enough calls to $\mathsf{HMakeRedex}(.,.)$ on c_h in order to get to an expression

$e'_h = C'[op_{...}(\ldots, c'_h, \ldots)]$ such that calls to $\mathsf{HMakeRedex}\,(c'_h, .)$ returns None. Since we supposed that the heuristic failed to reduce c_h to a compatible operator, this means that the head of c'_h is different from the head of c_N (which is a compatible operator). Using Lemma 7, c'_h cannot be consumed, and as the child expression of an operator that cannot be erased, c'_h cannot be erased either. According to Lemma 6 this contradicts the confluence theorem telling that there is a reduction $e'_h \to^* e_N$. Therefore, there is a fuel value such that the heuristic reduces c_h to a compatible operator. Now, taking the maximum of the required values of F to reduce the children of all operators, there exists a value of F such that $\mathsf{Frag}\,(e_h)$ is equal to the second component of M_{min}.

□

We also conjecture that the result of Theorem 6 still holds for weakly-normalizing expressions.

Conjecture 1. *For databases with stable compatibility, this reduction strategy is complete on weakly-normalizing expressions with fixed operators. That is, for a database with stable compatibility, given a weakly-normalizing input expression e with fixed operators, there exists a fuel value F such that $\mathsf{HMinRed}_F\,(e) \in \mathsf{MinReds}\,(e)$.*

None of the remaining hypotheses can be removed.

- Stable compatiblity: consider a database for which the **Scan** operator is compatible if its configuration has more than two free variables. Obviously, it would not have the stable compatibility property, since inlining the definition of these variables could reduce the number of free variables in the configuration. Take now expression e from Example 7. Since the heuristic tries to inline variables before reducing redexes in the configurations, it will never consider expression e', which is the only element of $\mathsf{MinReds}\,(e)$.

  For the next counterexamples, we will suppose a simplistic database capabilities description, for which all operators are compatible as long as there is no free variable in their configuration (such a database would have the stable compatibility property).

- Normalizing: consider expression e from Example 8. Obviously, it is non-normalizing because of the $\Omega = (\boldsymbol{\lambda} x.\ x\ x)\ (\boldsymbol{\lambda} x.\ x\ x)$ in the configuration. Since the heuristic applies a call-by-name reduction strategy on the configurations once all free variables are inlined, it will consume all the fuel reducing Ω and never consider e', which is the only element of $\mathsf{MinReds}\,(e)$.

- Operators cannot be erased: consider expression e from Example 9. Obviously, the $\mathbf{Scan}_x()$ operator can be erased. Since the heuristic only tries to inline variables in the configurations and reduce them, it will never consider e', which is the only element of $\mathsf{MinReds}\,(e)$.

- Operators cannot be duplicated: consider expression e from Example 10. Obviously, the $\mathbf{Scan}_z$ operator can be duplicated. The heuristic will try to inline y in the configuration of the two **Join** operators, which requires to inline x first. Since this two-step reduction decreases the measure and because the heuristic chooses the left-most leaf of the configuration search space, e' will never be considered although it is the only element of $\mathsf{MinReds}\,(e)$.

Example 7

$e = (\boldsymbol{\lambda} t.\ \mathbf{Scan}_{(\boldsymbol{\lambda} x.\ x=x)\ t}())\ 1 \quad \rightarrow^* \quad e' = (\boldsymbol{\lambda} t.\ \mathbf{Scan}_{\boldsymbol{\lambda} x.\ t=t}())\ 1$

Example 8

$e = \mathbf{Scan}_{((\boldsymbol{\lambda} x.\ x\ x)\ (\boldsymbol{\lambda} x.\ x\ x))\ ((\boldsymbol{\lambda} x.\ 1)\ y)}() \quad \rightarrow^* \quad e' = \mathbf{Scan}_{((\boldsymbol{\lambda} x.\ x\ x)\ (\boldsymbol{\lambda} x.\ x\ x))\ 1}()$

Example 9

$e =$ `if false then` $\mathbf{Scan}_{x}()$
`else` $\mathbf{Scan}_{db.table}()$ $\quad \rightarrow^* \quad e' = \mathbf{Scan}_{db.table}()$

Example 10

$e = (\boldsymbol{\lambda} x.\ \boldsymbol{\lambda} y.\ \mathbf{Join}_{\texttt{if false then } y \texttt{ else true}}($
$\mathbf{Scan}_{db.table}(),$
$\mathbf{Join}_{\texttt{if false then } y \texttt{ else true}}(x, x))$
$)\ \mathbf{Scan}_{z}()\ \texttt{false}$
$\quad \rightarrow^* \quad$
$e' = \boldsymbol{\lambda} x.\ \boldsymbol{\lambda} y.\ \mathbf{Join}_{\texttt{true}}($
$\mathbf{Scan}_{db.table}(),$
$\mathbf{Join}_{\texttt{true}}(x, x))$
$)\ \mathbf{Scan}_{z}()\ \texttt{false}$

Figure 5.4 sums up the situation. In this Venn diagram, E stands for the set of all QIR expressions, SN for the set of strongly-normalizing QIR expressions, FO for the set of QIR expression with fixed operators, H for the set of QIR expressions on which the heuristic returns an optimal result and P for the set of QIR expressions for which we proved that the heuristic returns such a result. As we will discuss in Section 5.5, H is much bigger than P: in fact the heuristic returns optimal results on all our real-world use cases.

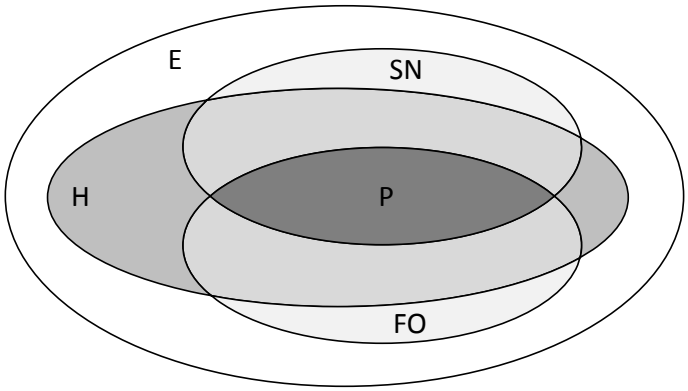

Figure 5: Architecture

5.5 Concrete examples

5.5.1 Example of code factorization: analytics queries

Consider the following analytics query written in SQL and freely inspired by the *TPC-H* benchmark.

```
SELECT
  l_returnflag AS return_flag,
  l_linestatus AS line_status,
  SUM(l_extended_price) AS sum_base_price,
  SUM(l_extended_price * (1 - l_discount)) AS sum_disc_price,
  SUM(l_extended_price * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
  SUM(l_extended_price) * 0.75 AS sum_real_cost,
  SUM(l_extended_price) * 0.25 AS sum_margin,
  AVG(l_extended_price) AS avg_base_price,
  AVG(l_extended_price * (1 - l_discount)) AS avg_disc_price,
  AVG(l_extended_price * (1 - l_discount) * (1 + l_tax)) AS avg_charge,
  AVG(l_extended_price) * 0.75 AS avg_real_cost,
  AVG(l_extended_price) * 0.25 AS avg_margin
FROM
  db.lineitem
GROUP BY
  l_return_flag, l_linestatus
ORDER BY
  l_return_flag, l_linestatus
```

Notice in particular the many common expressions that are used to compose the aggregation functions of the query. To factorize code and increase the maintainability of the code, one would like to define these expressions only once and reuse them throughout the query. This can be achieved in QIR as follows (note that in QIR, as opposed to SQL's strange syntax, aggregations functions belong to the configuration of the **Group** operator).

```
/* Constructs the list of projected attributes */
let project_list = λtup.
   let real_cost = 0.75 in
   let margin = 0.25 in

   let return_flag = tdestr tup "l_returnflag" in
   let line_status = tdestr tup "l_linestatus" in
   let sum_base_price = tdestr tup "sum_base_price" in
   let sum_disc_price = tdestr tup "sum_disc_price" in
   let sum_charge = tdestr tup "sum_charge" in
   let avg_base_price = tdestr tup "avg_base_price" in
   let avg_disc_price = tdestr tup "avg_disc_price" in
   let avg_charge = tdestr tup "avg_charge" in

   tcons "return_flag" return_flag (
   tcons "line_status" line_status (
```

```
   tcons "sum_base_price" sum_base_price (
   tcons "sum_disc_price" sum_disc_price (
   tcons "sum_charge" sum_charge (
   tcons "sum_real_cost" (sum_base_price * real_cost) (
   tcons "sum_margin" (sum_base_price * margin) (
   tcons "avg_base_price" avg_base_price (
   tcons "avg_disc_price" avg_disc_price (
   tcons "avg_charge" avg_charge (
   tcons "avg_real_cost" (avg_base_price * real_cost) (
   tcons "avg_margin" (avg_base_price * margin) (
   tnil))))))))))))
in

/* Constructs the list of grouping/sorting attributes */
let group_sort_attr = λtup.
   cons (tdestr tup "l_returnflag") (
   cons (tdestr tup "l_linestatus") (
   tnil))
in

/* Constructs the aggregate functions */
let group_agg = λtup.

   let extended_price = tdestr tup "l_extended_price" in
   let discount = tdestr tup "l_discount" in
   let tax = tdestr tup "l_tax" in
   let disc_price = extended_price * (1 - discount) in
   let charge = disc_price * (1 + tax) in

   tcons "sum_base_price" (sum extended_price) (
   tcons "sum_disc_price" (sum disc_price) (
   tcons "sum_charge" (sum charge) (
   tcons "avg_base_price" (avg extended_price) (
   tcons "avg_disc_price" (avg disc_price) (
   tcons "avg_charge" (avg charge) (
   tnil))))))
in

/* Main query */
Project_project_list(
   Sort_group_sort_attr(
      Group_group_sort_attr,group_agg(
         Scan_db.lineitem())))
```

On this example, the implementation of the exhaustive reduction strategy described in Section 5.3 does not terminate in reasonable time due to a combinatorial explosion. Indeed, there are many redexes in the original expression, and the algorithm will, in parallel, reduce

each of them then try again all the remaining redexes, etc. Nevertheless, it eventually finds the reduced expression with minimal measure, i.e the original expression with all definitions and common expressions inlined. On the other hand, the implementation of the heuristic-based strategy described in Section 5.4 quickly finds the result for a fuel value $F > 10$.

5.5.2 Example of fragment grouping: dynamic queries

Consider now a simple website on which users can read small ads from people selling furnitures, cars, etc. and in particular a page to browse through the offers. This page would consist mainly in a form with (i) a dropdown menu to optionnally sort the result by date or price, (ii) a set of checkboxes to filter by category and (iii) an integer field with a default value to specify the number of results to display on the page.

The corresponding query, fetching the results, would have to be built dynamically depending on the presence of filters and ordering. The following QIR code is a possible implementation of this logic, in which we assume that variables `is_sorted`, `sort_attr`, `cat_list` and `limit` are provided by the application context and encode the presence of a sorting attribute, the list of selected categories and the number of results to display.

```
/* Recursively constructs a list of OR of the selected categories */
let make_cat_filter = λcat_list. λtup.
   let rec aux = λfilter. λcat_list.
      destr cat_list filter (λhd. λtl. aux ((tdestr tup "category" = hd) or filter) tl)
   in
   let aux2 = λcat_list.
      destr cat_list nil (λhd. λtl. aux (tdestr tup "category" = hd) tl)
   in
   aux2 cat_list
in

/* Constructs the ordering attributes */
let make_ordering = λattr. λtup.
   if attr = "price" then
      cons (tdestr tup "price") nil
   else if attr = "date" then
      cons (tdestr tup "timestamp") nil
   else
      nil
in

/* Constructs the list of projected attributes */
let project_list = λtup.
   tcons "title" (tdestr tup "title") (
   tcons "description" (tdestr tup "description") (
   tnil))
in
```

```
/* Base table */
let ads =
   Scan_{db.ads}()
in

/* After category filters */
let ads_filtered =
   destr cat_list ads (λhd. λtl. Select_{make_cat_filter cat_list}(ads))
in

/* After (optional) ordering */
let ads_ordered =
   if is_sorted then Sort_{make_ordering sort_attr}(ads_filtered) else ads_filtered
in

/* Main query */
Project_{project_list}(Limit_{limit}(ads_ordered))
```

On this example, the implementation of the exhaustive reduction strategy described in Section 5.3 does not terminate, since one can obtain infinitely many distinct reductions of the original expression by unfolding the recursive call of `aux` in `make_cat_filter`.

Conversely, the implementation of the heuristic-based strategy described in Section 5.4 quickly finds the result expression with minimal measure for a fuel value $F > 25$. For instance, with `cat_list` set to `nil`, `is_sorted` set to `false` and `limit` set to `20` the result expression is

```
Project_{λt. tcons "title" (tdestr t "title") (tcons "description" (tdestr t "description") tnil)}(
   Limit_{20}(
      Scan_{db.ads}()))
```

And with `cat_list` set to `cons "housing" (cons "cars" nil)`, `is_sorted` set to `true`, `sort_attr` set to `date` and `limit` set to `30` the result expression is

```
Project_{λt. tcons "title" (tdestr t "title") (tcons "description" (tdestr t "description") tnil)}(
   Limit_{30}(
      Sort_{λt. cons (tdestr t "timestamp") nil}(
         Select_{λt. tdestr t "category"="cars" or tdestr t "category"="housing"}(
            Scan_{db.ads}()))))
```

5.5.3 Example of incompatibility: caching

Following on the previous example, consider a page on which an admin can detect if an unexperienced user has published the same ad twice. Assume that the application context provides a function `unexperienced(user_id)` telling if a user is unexperienced and that this function is too complicated to be inlined inside the configuration of an operator. The following QIR

code corresponds to the query used by the page.

```
/* Constructs the list of projected attributes */
let project_list = λtup.
   tcons "user_id" (tdestr tup "user_id") tnil
in

/* Constructs the join condition */
let join_cond = λtup1. λtup2.
   (tdestr tup1 "title" = tdestr tup2 "title") and
   not (tdestr tup1 "ad_id" = tdestr tup2 "ad_id")
in

/* Unexperienced users */
let ads_from_unex_users =
   Select_{λtup. unexperienced (tdestr tup "user_id")}(
      Scan_{db.ads}())
in

/* Main query */
Project_{project_list}(
   Join_{join_cond}(
      ads_from_unex_users,
      ads_from_unex_users))
```

This is one example of situation where reducing a redex (e.g. inlining `ads_from_unex_users`) is not beneficial for the database. In this case, both the implementation of the exhaustive strategy described in Section 5.3 and the heuristic-based strategy described in Section 5.4 return the correct answer, given below.

```
let ads_from_unex_users =
   Select_{λtup. unexperienced (tdestr tup "user_id")}(
      Scan_{db.ads}())
in

Project_{tcons "user_id" (tdestr tup "user_id") tnil}(
   Join_{(tdestr tup1 "title"=tdestr tup2 "title") and not (tdestr tup1 "ad_id"=tdestr tup2 "ad_id")}(
      ads_from_unex_users,
      ads_from_unex_users))
```

This corresponds to a computation where the Truffle runtime (cf. Section 2) evaluates once `ads_from_unex_users`, stores the result on the database storage then passes the main query to the database engine.

6 Source capabilities

TODO (Julien)

7 Translation to native queries

TODO (Julien)

8 Evaluation of Truffle nodes in the database

TODO (Julien)

9 Query evaluation in the database

TODO (Julien)

10 Returning result to the host language

TODO (Julien)

References

[1] Henk P Barendregt. *The Lambda Calculus: Its Syntax and Semantics, revised ed., vol. 103 of Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1984.

[2] Delia Kesner, Carlos Lombardi, and Alejandro Ríos. A standardisation proof for algebraic pattern calculi. *arXiv preprint arXiv:1102.3734*, 2011.

[3] Jan Willem Klop, Vincent van Oostrom, and Roel C. de Vrijer. Lambda calculus with patterns. *Theor. Comput. Sci.*, 398(1-3):16–31, 2008.