Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

GraphRedex: Look at Your Research

Robbert Gurdeep Singh* | Christophe Scholliers

[1] Department of Applied Mathematics, Computer Science and Statistics, Universiteit Gent, Belgium

Correspondence
*Robbert Gurdeep Singh
Krijgslaan 281, Ghent, Belgium
Email: Robbert.GurdeepSingh@UGent.be

Present Address
Robbert Gurdeep Singh
Krijgslaan 281, Ghent, Belgium
Email: Robbert.GurdeepSingh@UGent.be

Summary

A significant aspect of designing new programming languages is to define their operational semantics. Working with a pen and paper version of such a semantics is notoriously difficult. For this reason, tools for computer aided semantics engineering were created. Many of these tools allow programmers to execute their language's operational semantics. An executable semantics makes it easier to verify whether the execution of a program leads to the desired result. When a program exhibits unexpected behavior, the programmer can consult the reduction graph to see what went wrong. Unfortunately, visualization of these graphs is currently not well-supported by most tools. Consequently, the comprehension of errors remains challenging.

In this article we present GraphRedex an open-source tool that empowers language designers to interactively explore their reduction graphs, offering three main benefits. First, a global exploration mode allows users to obtain a bird's-eye overview of the reduction graph and learn its high level workings. Second, a local exploration mode lets the programmer closely interact with the individual reduction rules. Third, our query interface allows the programmer to filter out and highlight specific regions of the reduction graph.

We evaluated our tool by carrying out a user study showing that participants comprehend programs on average twice as fast while being able to answer questions more accurately. Finally, we demonstrate how GraphRedex helps to understand the semantics of two published works. Exploration of the semantics with GraphRedex unveiled an error in one of the implementations of these works, which the author confirmed.

KEYWORDS:
Semantics Engineering, Visualization, State Explosion, PLT Redex, Tooling, Operational Semantics

1 | INTRODUCTION

Inventing new programming language constructs and developing their formal semantics is a difficult process. In 2012, it was discovered that many published papers contain small mistakes in their formalization [1]. This hampers understanding and limits the bandwidth of communication. Many programming language researchers develop the semantics of their language constructs with pen and paper and then rigorously prove desirable properties. A classic example is to prove soundness of a typed language by proving progress and preservation [2]. Unfortunately, even with a soundness proof, the language designer cannot be sure whether the defined semantics exactly captures the intent of the language.

To truly explore newly invented constructs it is necessary to run programs. In order to be able to test newly invented programming language constructs, researchers implement a proof of concept interpreter in a language like Haskell or OCaml. This

has two main disadvantages. First, the language designer needs to make a costly implementation while unsure about the final language design. Second, by developing both a formal system and an implementation, non-trivial differences between the two may arise.

In the last decade, the community developed many tools to help language designers define executable semantics. Tools such as Ott[3], Maude[4], PLT Redex[5] and the $\mathbb{K}$ framework[6] permit language designers to mechanize their semantics and execute programs written in their language. These tools greatly mitigate the aforementioned problems. Many of them also incorporate forms of unit testing and typesetting. There are also tools for PLT Redex, the $\mathbb{K}$ framework, and Maude that enable the programmer to visualize the evaluation of a program.

Visualizing the transition steps through which a program evaluates can help uncover errors during semantics development. This works well when only one reduction rule applies for each state of the program. However, with the rise of concurrency and non-deterministic semantics in general, visualization becomes significantly harder. This is caused by an undesired side effect of non-determinism: state explosion[7]. The amount of states that need to be shown when working with non-deterministic semantics grows exponentially with the number of steps taken. The visualizations of PLT Redex, the $\mathbb{K}$ framework, and Maude do not handle this explosion very well. These visualizations become cluttered quickly or do not support non-determinism at all. As a consequence, these tools cannot be used to see the effects of non-deterministic language constructs.

In this paper we present *GraphRedex*, an open-source user-friendly web application that empowers semanticists to interactively explore the reduction of a term within their semantics. Our tool supports reduction graph exploration in three ways. (a) A local exploration mode helps following reduction steps at the level of the individual reduction rules. (b) Our global exploration mode uncovers the higher level workings of the submitted language. Lastly, (c) a querying interface helps language designers to bring important states and reductions to the foreground. GraphRedex keeps the view manageable by enabling interactive expansion the reduction graph. By not expanding the whole graph at once, we restrain state explosion. Thanks to this, language designers can explore of the limitations and affordances of a mechanized language effectively.

This article starts out with a simple example (Section 2) to give a high level overview of our tool. Then, in Section 3, we describe how GraphRedex handles reduction graphs with thousands of nodes and how we restrain the state explosion problem. The querying functionality is laid out in Section 4. Next, we showcase the benefits of GraphRedex with two usage scenarios (Section 5.1) and carry out a user study (Section 5.2) to evaluate our work. Finally, we compare our contributions to related work (Section 6) and conclude this paper (Section 7).

GraphRedex is an open source tool, its code is available on GitHub (TOPLLab/GraphRedex). A demo site is available at https://redex.ugent.be/. To secure the server a login is required (UserName: `SPE` Password: `SPE`). A video demonstrating the tool can be found at https://redex.ugent.be/demo-video.

2 | THE NEED FOR VISUAL EXPLORATION

To show the need for interactive reduction graph exploration we use a variation on the running example of the PLT Redex tutorial: the $\lambda_{\texttt{amb}}$ language of ambiguous computation[8]. This language is simple enough to have its full definition in a paper while still containing interesting non-determinism. We lay out the definition of the language in Section 2.1, readers familiar with the language may choose to jump ahead to Section 2.2, where we show how to explore programs written in $\lambda_{\texttt{amb}}$ in GraphRedex.

2.1 | Syntax and Semantics of the $\lambda_{\texttt{amb}}$ language in PLT Redex

The $\lambda_{\texttt{amb}}$ language is a variation on a simplified lambda calculus extended with a variation on McCarthy's `amb` operator of ambiguous choice[8]. An `(amb ...)` construct reduces non-deterministically to one of its arguments. For example, `(amb 1 2 3)` evaluates to `1`, `2` or `3`. An `(amb ...)` construct may show up anywhere in an expression. The term `(+ 2 (amb 3 4))` can thus evaluate non-deterministically to `(+ 2 3)` or `(+ 2 4)` and then deterministically to `5` or `6` respectively.

The operational semantics and an implementation of this semantics in PLT Redex can be found in Figure 1 and Listing 1 respectively. Both representations first describe the shape of terms and then the reduction rules of the evaluator.

In PLT Redex, `define-language` specifies the shape of the terms in a language (lines 5-7). This function takes the name of the language and a specification of its non-terminals as arguments. Our implementation specifies two non-terminals: `e` and `E` (lines 6 and 7). These correspond to the e and E in Figure 1 respectively. The only notable difference between the formal notation and the implementation in Listing 1 is that stars (*) have been replaced with ellipsis (...) and that the hole ($\diamond$) is replaced with "`hole`".

$$
\begin{array}{lll}
e & ::= & (+\ e^*) \mid (amb\ e^*) \mid number & \text{Expressions} \\
E & ::= & \diamond \mid (+\ number^*\ E\ e^*) & \text{Context}
\end{array}
$$

$$
\frac{v = v_1 + \ldots + v_i}{E[(+\ v_1\ \ldots\ v_i)] \rightarrow E[v]}\ (\text{ADD}) \qquad \frac{e_x \in \{e_1, e_2, \ldots, e_n\}}{E[(amb\ e_1\ e_2\ \ldots\ e_n)] \rightarrow E[e_x]}\ (\text{AMB})
$$

FIGURE 1 The reduction relation of λ_{amb} language in mathematical notation.

```
1  #lang racket
2  (require redex/reduction-semantics)
3  (provide reductions)
4
5  (define-language Amb
6    (e (+ e ...) (amb e ...) number)    ; e ::= (+ e*) | (amb e*) | number
7    (E hole (+ number ... E e ...)))    ; E ::= (+ number* E number*)
8
9  (define reductions
10   (reduction-relation Amb #:domain e
11     (--> (in-hole E (+ number ...))
12          (in-hole E ,(apply + (term (number ...)))) "add")
13     (--> (in-hole E (amb e_x ... e_2  e_y ...))
14          (in-hole E e_2)                           "amb")))
```

LISTING 1 An implementation of the λ_{amb} language in PLT Redex.

The reduction rules also correspond one-to-one with their formal counterparts. We build a set of PLT Redex reduction rules using the `reduction-relation` function (lines 9-14). This function takes various rules of the form `(--> from to)` where `from` represents the pattern to match and `to` specifies what this pattern should be replaced with. Variables matched in `from` can be used in `to`. If a comma is placed before an expression in `to`, it is evaluated as a Racket expression. The `(in-hole E term)` construct represents the formal plugging operation "E[`term`]".

In a deterministic evaluator an arbitrary term is either a value or has exactly one term it reduces to. Here, the "amb" rule allows an `(amb ...)` term to evaluate to one of n possible terms. The language is thus non-deterministic.

2.2 | λ_{amb} in GraphRedex

With GraphRedex language designers interact with their operational semantics in three ways: global exploration, local exploration, and querying. Global exploration provides a bird's-eye overview of a reduction graph, explaining how terms reduce by taking multiple steps. Local exploration focuses on individual reduction rules, and explains how a term t reduces to a term t' in one step. Querying filters or highlights important terms and reduction rules.

In this section we look at the execution of the example term below.

```
(+ (amb 100 1) (amb 1 (amb (+ 1 1) (+ (amb 4 5) 4)) (amb 1 2)))
```

When we submit this term it is repeatedly reduced using the `reductions` reduction relation from Listing 1. The resulting reduction graph is a graph where nodes represent terms and edges represent applications of reduction rules.

Note: you can look at this example by selecting the "amb" example in the sidebar on redex.ugent.be.

2.2.1 | Global Exploration

To get an overview of what the program above does, we will look at its reduction graph. Figure 2 shows the reduction graph in global exploration mode on the left. Each node in this graph represents a term. Blue nodes represent terms that can be reduced further. Red nodes are normalized states, i.e., no rule applies to the term. The green, leftmost, node represents our input term. We immediately see that the program has 8 possible outcomes (red nodes). By placing our mouse over these nodes we see their

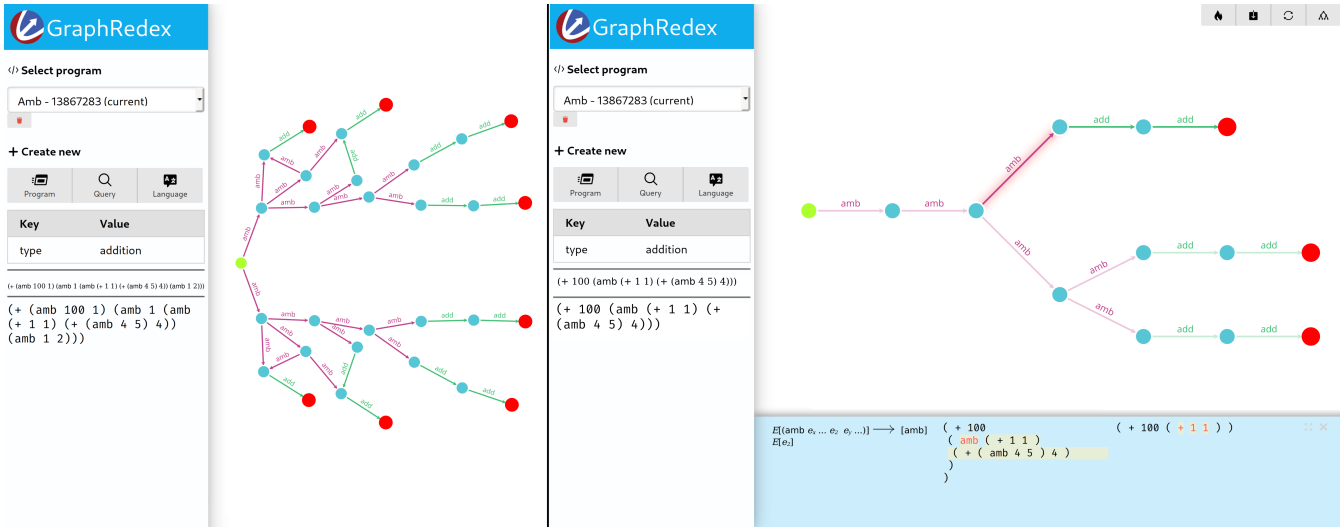

FIGURE 2 Exploration of a program in the λ_{amb} language. Left: Global exploration, users hover over nodes to get information, nodes can be moved by dragging them. Right: Local exploration, users select arrows to follow with the keyboard. We cropped both screenshots and increased the size of the labels to improve readability.

value in the sidebar. Each reduction rule is associated with a different arrow color. Thanks to this we quickly see that purple arrows are executed before green ones, that is "amb" executes before "add".

Our global exploration mode follows the Visual Information Seeking mantra of "Overview first, zoom and filter, then details on demand"[9]. The user first gets an overview of the reduction graph and can then choose to zoom in to inspect certain regions more closely. With our query functionality, the view can be filtered. Hovering over a node of the graph shows details on demand in the sidebar.

2.2.2 | Local Exploration

To get a better understanding of the one-step reduction relation, we use local exploration. Contrary to global exploration, local exploration aims to enable the inspection of individual reduction steps, focusing on single steps, not global effects: the arrows are the central focus, not the nodes. In local exploration, the user can follow a reduction path without being distracted by unrelated nodes. The right side of Figure 2 shows the interactive reduction graph that GraphRedex builds for our example in local exploration mode. Users navigate through the reduction graph by selecting reductions to apply. At the bottom of the screen, a panel shows the selected reduction rule and the term that results from applying it. To help the programmer identify how the term reduces, we highlight the differences between the term before and after reduction. To find these differences, we recursively compare the S-expression representations of the terms. The sidebar on the left contains information about the currently selected term. By default, we show the term, but language designers can choose to add extra information to the sidebar (Section 3.2). In the screenshot (Figure 2), the language designer opted to add a field named "type" to the sidebar.

Navigation can go both forward and backward. The former option permits following reductions in the natural reducing direction. With the latter option, users can traverse the reductions backwards, i.e., go back to all the states it could have reduced from (backwards reduction).

2.2.3 | Querying

As we can see on the left side of Figure 2, some values can be obtained in multiple ways. The semanticist may be interested in finding out all the paths leading to a specific value. Using a predefined "paths to selected" query, the graph reduces to a subgraph that only contains nodes that may reduce to the focused node. The result can be seen on the left side of Figure 3.

Finding the path to all stuck nodes is another valuable query. Figure 3 shows the query's result on the right side. This query reduces the amount of nodes the developer sees when trying to figure out how the different end states may be reached. Querying works in both global and local exploration mode.

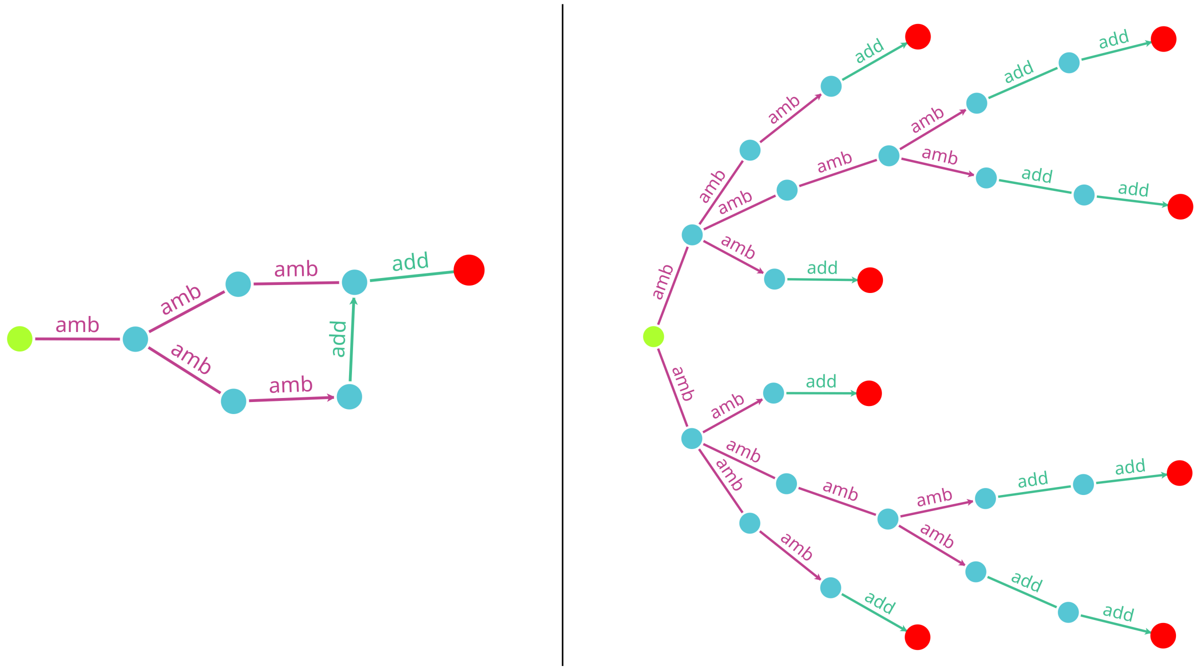

FIGURE 3 Queried versions of the reduction graph of a program in the λ_{amb} language. On the left all paths to the selected node. On the right the shortest path to each normalized node.

3 | GRAPHREDEX AT SCALE

A significant problem when visualizing a reduction graph is that non-determinism can lead to a large amount of states. This problem is known in literature as the state explosion problem[7]. It makes it difficult to see the wood for the trees when looking at a reduction graph. The goal of GraphRedex is to make these visualizations scale despite state explosion. To this end, we enforce a limit on the amount of reductions we initially show. The semanticist can then select where to continue expansion from. This reduces the perceived state explosion and allows interactive exploration of non-deterministic semantics.

Submitting a term to GraphRedex causes it to be reduced repeatedly with the exported reduction relation. If the semantics features non-determinism, terms can reduce in multiple ways. In this case, our tool traverses these possibilities breadth-first. We continue reducing until no reductions remain, we carried out 1000 reductions[*] or the execution times-out. The visited nodes and edges of the reduction graph are stored in a graph database. Using a database specialized for graphs speeds up visualization and enables queries with graph primitives (e.g., finding the shortest path). Once we have our initial set of states, users can start exploring and querying the reduction graph.

In the previous section we showed the basic functionality of GraphRedex. In this section we will show that it scales to larger graphs. We will focus our discussion on global exploration, as this mode is most impacted by the addition of nodes. Our local exploration mode scales well with the amount of nodes because we only render the reduction graph to a depth of five steps[†]. After global exploration we shift our attention to customizability, an important quality of tools that explore large graphs[10,11].

[*]This value was determined through experimentation and can be adjusted by the user when setting up the server. There is a significant cost to starting PLT Redex, so once it is started, we carry out at least as many reductions as the time it takes to startup PLT Redex. The time per reduction varies greatly between languages, for this reason we added a timeout.

[†]Rendering in local exploration mode has a time and space complexity of $\mathcal{O}(nf^4)$ where n is the total number of loaded nodes and f is the maximal outdegree of the language (branching factor).

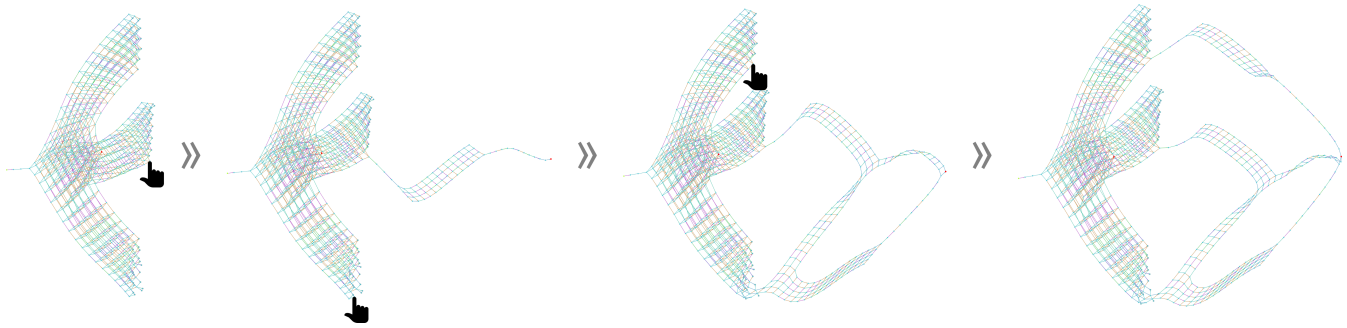

FIGURE 4 Interactively exploring a reduction graph in GraphRedex. Clicking on an unexpaned node in the graph expands it with the nodes that reduce from it.

3.1 | Global Exploration at Scale

In global exploration, the user wants to get an overview of a program's execution. With state explosion the myriad of states can make it difficult to see the wood for the trees. GraphRedex limits the amount of states by only showing the graph up to a limited depth. This enables programmers to see the start of the reduction graph and then select where they want to continue exploring.

The reduction graphs we visualize are typically directed acyclic graphs. Our goal is to present them such that the reduction relation is placed in the foreground. We want to make two properties apparent. First, if t reduces to t', these two terms are closely related, so we place them in close proximity. Secondly, we want to make it clear that t comes before t'. To convey the direction of the reduction relation, we aim to place t to the left of t', making the horizontal axis an approximation for the number of applied reductions. GraphRedex aims to place nodes with the same distance to the start node on a vertical line.

A tweaked force directed layout implemented in D3 [12] positions the nodes. We tweak D3's default spring embedder (all nodes repel, adjacent nodes attract) by adding an extra force. This extra force pulls nodes to the correct depth. By doing this the horizontal axis represents the number of reductions, and the vertical axis represents options.

We opted for a force directed layout because this permits the user to move nodes in the view around. This can help to move things out of the way or move terms that are similar according to the user closer together.

Users can interactively choose the parts of the reduction graph they want to see. To restrain state explosion, not all nodes are expanded. Nodes that can be expanded are highlighted. They expand with a double click. There are also provisions to expand all descendants of a selected node at once. Figure 4 shows how a graph evolves by selecting what to expand. To keep the graph concise we merge identical states that come from different expansions.

The sidebar plays a critical role in guiding users towards interesting states by showing the contents of nodes. With this information users can make an informed choice about which node they wish to expand.

Looking at the reduction graph from a distance can give interesting insights into the higher level workings of a language. Figure 5 shows the entire reduction graph of a dining philosophers program in a fork-join language with three philosophers in global exploration mode. In this program, each of the philosophers needs to take turn in performing a task. When all philosophers have completed their task, the program ends. We use the dining philosophers as it is a widely understood problem in the computer science community. We choose three philosophers as a minimal interesting example. Interpreting the graph from left to right, shows how terms evolve with repeated reductions. The start of the graph splits up in three groups of nodes, one for each philosopher. Each group splits up again in two groups, one for each remaining philosopher. We see that all these groups join up with a group that branched off from another of the initial three groups. As we approach the right side of the graph, we see fewer edges, this is indicative of the lowered amount of non-determinism in the system at that point. At the right, we see a red (normalized) node, the final state.

Upon close inspection we see a second red dot, on the left side. This is a stuck state in which none of the philosophers have finished their work, an unexpected deadlock. GraphRedex helps language designers by making these kinds of unpredicted errors visible.

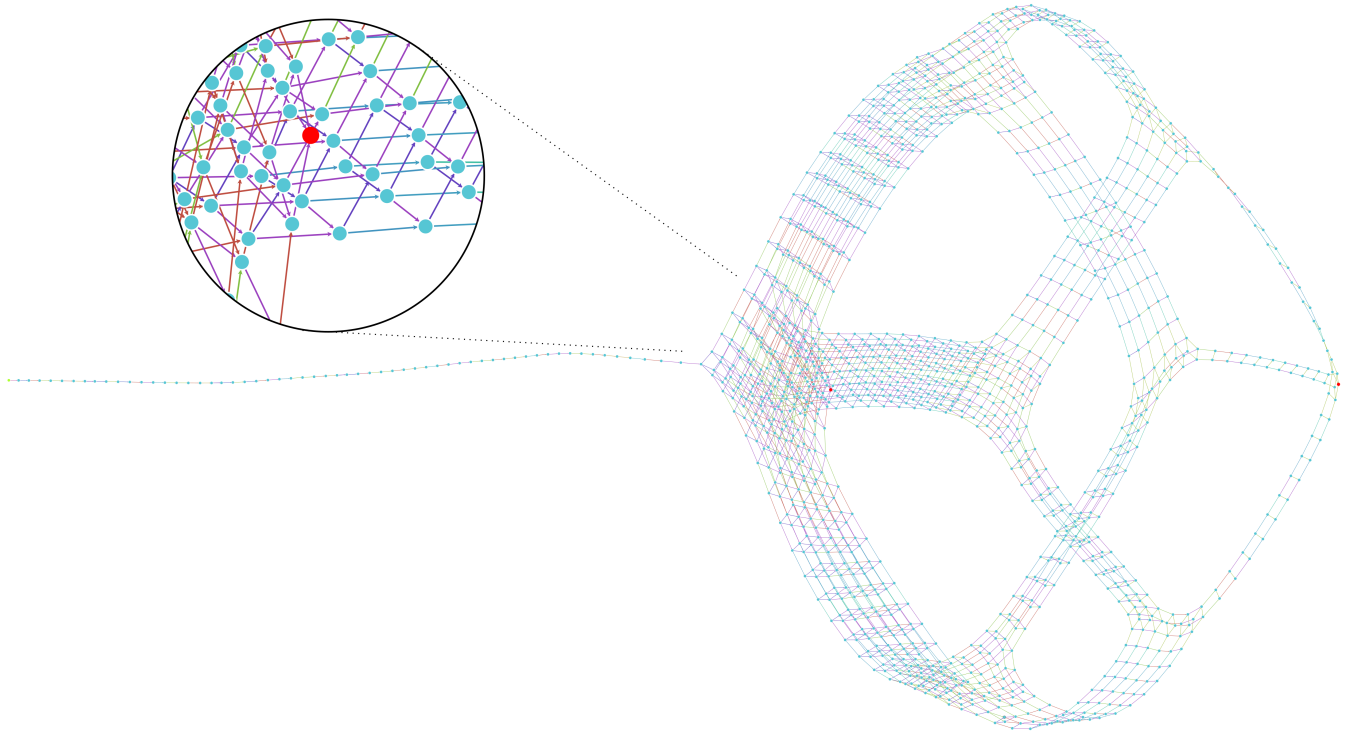

FIGURE 5 The reduction graph of a program implementing the dining philosophers program in a language with fork-join and locks. The zoomed in section focuses on an unwanted deadlock.

3.2 | Customizability

In order to work for large graphs, visualization tools need to be customizable [10,11]. In this section, we show the ways in which GraphRedex can be adapted to fit the users needs: custom data extraction, custom syntax and custom visualizations. We end the section with an example.

Our tool works with existing semantics implemented in PLT Redex. The bare minimum it needs to function is a reference to the reduction relation of the language at hand. Language designers can expose extra values to customize GraphRedex to their needs.

One of the central features of GraphRedex is custom data extraction. It permits language designers to extract certain pieces of data to show in the sidebar and to use in queries. Users only need to define one function: `term->kv`. This function takes a term of the language and should return extra domain-specific information about the term as key-value pairs. One may, for example, wish to extract the return value of an execution, which the defined semantics wraps in some way. Each returned key-value pair shows up in the sidebar as a row of the "Node information" table. Additionally, these fields can be queried. If `term->kv` returns the key-value pair `("numThreads", 5)` for a term, all nodes with that value for `numThreads` can be found with the query `FOR n in @@nodes FILTER n.numThreads == 5 RETURN n` (See Section 4: Querying).

Programming languages have a concrete syntax. The syntax describes how terms of the language appear to the end user. It is the interface between the programmer and the programming language. GraphRedex enables language designers to use their own syntax to input programs. To do this the language implementer should export a function that acts as a parser. This feature was requested by many researchers to whom we showed intermediate versions of GraphRedex. The custom syntax provisions allow language users rather than only the language designers to use GraphRedex.

By default, GraphRedex's sidebar shows the term the user has selected as an S-expression. Instead of showing S-expressions, the language designer can configure alternative representations. Our tool is capable of handling: Racket `picts`, formatted plain text and HTML.

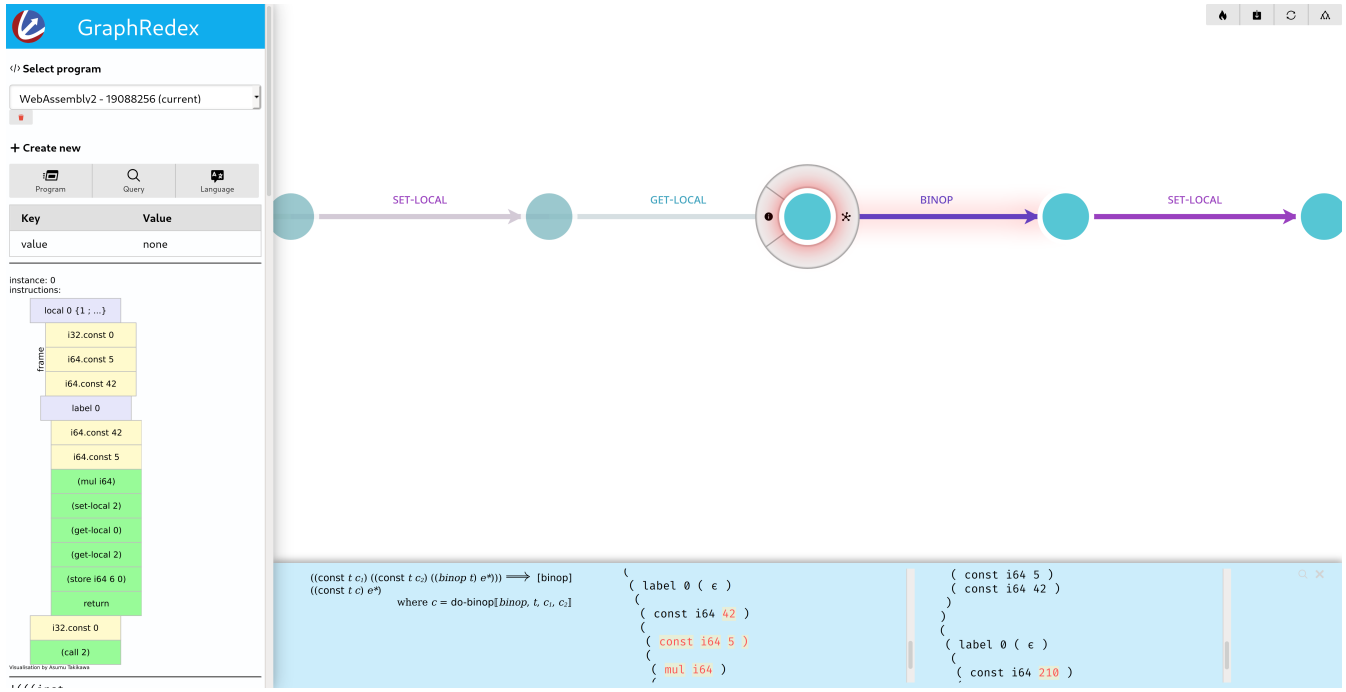

FIGURE 6 The multiplication step in a WebAssembly program that calculates `42*5`. The stack [42, 5, *mul*] becomes [210].

Rackets `pict-lib`* provides a means to generate "functional pictures" called `picts`. Support for this has also been added to PLT Redex. The library provides a means to functionally define how one draws a picture representing a term. We chose to add `pict` support because they are commonly used by language designers. Language designers who used the `pict-lib` in the past can reuse their existing rendering code in GraphRedex.

Some semanticists may wish to use an alternative plain text representation instead of the S-expression format terms come in by default. To this end, language designers may export a string value for the key "`_formatted`". This string will then render instead of the S-expression. This feature is especially useful when the language has custom syntax. For even more control, HTML is also supported.

As an example Figure 6 shows GraphRedex in local exploration mode with a WebAssembly program† loaded. WebAssembly (WA) is a stack based virtual machine designed as a low-level code platform aimed at the web [13]. Because WA is deterministic, the graph is a straight line. The `pict` shown in the sidebar is the output work we found online in a blogpost§ of Asumu Takikawa. He implemented the WA formalism in Redex and published the code on GitHub**. An interesting feature of this Redex implementation of WA is that it contains a function to draw a picture of the state of the running WA machine. We have added the eight lines shown in Listing 2 to export the `pict` and the current value of the WebAssembly VM to GraphRedex. Our tool can then show these in the sidebar. The former functionality is implemented on line 2 of Listing 2 by simply calling the original `pict` rendering function. Lines 3-5 obtain the value of the VM by using `match` to look for the pattern of a final result and extract it. Finally, on line 6 we create a `_formatted` field that contains a formatted version of the term using rackets built in `pretty-format`.

4 | QUERYING REDUCTION GRAPHS

Reduction graphs in GraphRedex can be queried. To enable this, our tool exposes a graph query interface through which users can carry out queries over the reduction graph. In the following sections we give a quick overview of the query language and its

*https://docs.racket-lang.org/pict/

†The program executes $\mathtt{f}_0(0,42); \mathtt{f}_1(0,5); \mathtt{f}_2(0)$, where $\mathtt{f}_0(\mathtt{i},\mathtt{b})$ stores $\mathtt{b}$ in `mem[i]`, $\mathtt{f}_1(\mathtt{i},\mathtt{m})$ multiplies the value in `mem[i]` with $\mathtt{m}$ and $\mathtt{f}_2(\mathtt{i})$ retrieves `mem[i]`. Listing 12 in Appendix C shows the exact code submitted to our tool.

§https://www.asumu.xyz/blog/2019/04/29/webassembly-in-redex/

**https://github.com/takikawa/wasm-redex

```
(define (term->kv exp)
   ((’_pict . (make-pict-of exp))
    (’value . (match exp
                [ ‘(,s ,F ((const ,type ,value) nil) ,i) value]
                [else "none"])))
    (’_formatted . (pretty-format exp 30)))

(provide term->kv (rename-out [wasm-> reductions]))
```

LISTING 2 Extra lines of code required to add full GraphRedex support to Takikawa's PLT Redex implementation of WebAssembly.

```
FOR n IN @@nodes
  FILTER n.numThreads > 1
  RETURN n
```

LISTING 3 An AQL query to select all nodes in the graph with more than one thread.

features. Then, we look at two predefined queries that are available in our tool. Finally, we show how users can combine queries with their own plug-ins.

4.1 | ArangoDB Query Language

The graph database GraphRedex uses under the hood is ArangoDB. ArangoDB[14] is a schema free NoSQL graph database of which we use the open-source edition. Internally, this database system represents each graph by two collections. The first collection contains the nodes of the graph. The second collection relates the nodes from the first collection with directed arrows stored as pairs.

GraphRedex creates a new graph in the database for each language. In plain AQL, the names of the underlying collections need to be referred to when making queries. These names are not of interest to the user. Therefore, we added *bind parameters* which can be used instead of the real names of the collections. The required collections can be referred to by a name starting with `@`. We support the following list of custom bind parameters:

- `@graph` contains the name of the ArangoDB graph representing the active reduction graph,
- `@@nodes` is a collection of nodes representing terms in the language,
- `@@edges` is a collection of edges representing the reductions in the language between the terms in `@@nodes`.

The unique identifiers (`_id`) of certain nodes, in the graph also have a shortcut name:

- `@start` is the name of the currently selected node,
- `@focus` contains the unique identifier of the focused term.

These abbreviations eliminate the need to know the internal name of the nodes in the graph, reducing the mental burden on the user. Additionally, they enable the programmer to write generic queries that can be used to query reduction graphs generated by different programs.

Listing 3 shows a minimal interesting query that selects all nodes in a graph with an attribute "`numThreads`" set to a value greater than one. To select nodes from the graph we use the `FOR ... IN` construct. This collects all the nodes that are projected outward using `RETURN`. The `FILTER` modifier prevents selected items that do not meet a certain condition from being collected. A `FOR ... IN` query always returns an array. Thanks to the *bind parameters*, the query in Listing 3 works for any language.

In the next section we look at more elaborate queries in detail and explain the AQL language further. A comprehensive overview of the query language can be found in Appendix A.

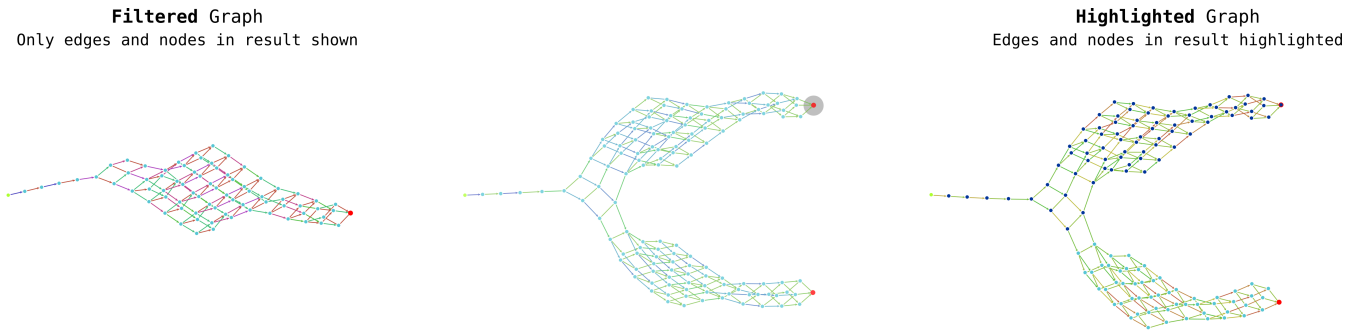

FIGURE 7 The filtered (left) and highlighted (right) versions of query that selects all paths to the selected node in the original reduction graph (center)

```
1  // find all stuck nodes
2  LET stuckNodes = (FOR n in @@nodes FILTER n._stuck RETURN n)
3
4  // find path to each stuck node and join the results
5  LET path = FLATTEN(
6      FOR target IN stuckNodes
7          FOR n,e IN OUTBOUND SHORTEST_PATH
8              @start TO target._id GRAPH @graph
9                  RETURN {n,e})
10
11 // convert to needed format
12 RETURN {
13     edges: (FOR d in path FILTER d.e != null RETURN DISTINCT d.e),
14     nodes: (FOR d in path FILTER d.n != null RETURN DISTINCT d.n)
15 }
```

LISTING 4 AQL query to find the shortest path to all normalized nodes in a reduction graph. We first select all normalized nodes, then we calculate the shortest path to all of them. Finally, we flatten the results to one graph.

4.2 | Highlighting, Filtering and Plugins

Queries can be used for filtering and highlighting. Figure 7 sketches the difference between the two for a query that selects all paths to a selected node (Listing 5). Using filter, only the nodes that are in the result of the query are shown. With highlight, the nodes in the result are highlighted, the rest of the nodes remain. The user can select the color of highlights. For this example, we chose a dark blue color. Multiple highlights can be active at the same time. Next to highlighting and filtering, GraphRedex allows users to have even more control over the graph with plugins.

GraphRedex has two preloaded queries that help reduce the complexity of reduction graphs: “Shortest path to all normalized nodes” and “All paths to selected node”. They apply to most reduction graphs. Users can easily add other queries. In this section we will look at the implementation of the preloaded queries, and we touch on how one can use plugins.

The first generally applicable query finds the shortest path to all normalized nodes. The result is the union of the shortest paths to each normalized node in the reduction graph. Listing 4 shows the full query. Applying it can considerably reduce the amount of states a developer sees while still maintaining all end states of a program.

The query first collects all normalized nodes (line 2) using a filtered `FOR` which returns a list of nodes in the graph that have their `_stuck` attribute set to `true`. This attribute is a predefined attribute set by GraphRedex. We use ArangoDB’s `LET` keyword to store the intermediate result. The next step is to find the shortest path to each of those nodes. For this, we map over all the stuck nodes (with `FOR`), and we use the `SHORTEST_PATH` keyword to get the shortest path. This built-in returns a list of nodes (`n`) and edges (`e`) in a shortest path to the node. The result is a list of paths (i.e., a list of lists of objects with `e` and `n` attributes) which we flatten. Finally, to render the graph, GraphRedex expects an object with a list of edges and a list of nodes, lines 12 to 15 construct this object.

The second predefined query finds all paths to a certain node. To do this we traverse the reduction graph breadth-first two times. Listing 5 shows the query. During the first traversal, we collect the nodes we visit by following the arrows backwards (`INBOUND`)

```
LET depth = 1000
LET backReachable = FLATTEN(
      FOR v IN 0..depth INBOUND @focus GRAPH @graph
        OPTIONS {bfs:true,uniqueVertices: 'global'}
        RETURN DISTINCT v._key)

LET nodes = FLATTEN(
      FOR v IN 0..depth OUTBOUND @start GRAPH @graph
        OPTIONS {bfs:true,uniqueVertices: 'global'}
        FILTER (v._key IN backReachable)
        RETURN DISTINCT v)

LET nodesKeys = nodes[*]._id
LET edges = (FOR e IN @@edges
               FILTER e._from IN nodesKeys
                   OR e._to   IN nodesKeys
               RETURN DISTINCT e)
RETURN {nodes,edges}
```

LISTING 5 Query to efficiently find all paths between two nodes. The first block selects nodes that may lead to the selected node, the second block intersects this with the set of nodes reachable from the start node, the final block finds the edges between the node and constructs the graph object.

```
define(["exports"], function (exports) {
    "use strict";
    Object.defineProperty(exports, "__esModule", { value: true });
    exports.default = (graphredex, result)=>{
        // Expand nodes in result
        for(let n of result){
            if(n.someValue > 0){
                graphredex.expand(n);
            }
        }
        return false;
    }
});
```

LISTING 6 The source code of a plugin that expands all nodes in the result of a query.

from the selected node (lines 2-5). The result is a collection of all nodes that may reduce to the selected node (`backReachable`). The second traversal collects the nodes that were in the first collection and are reachable from the startnode (`OUTBOUND`, lines 7-11). This gives us the desired result: all nodes on a path to the selected node. Finally, the third block of code selects all the edges between nodes in our collection (lines 13-17).

This query helps developers to limit their view to configurations that lead to some, perhaps faulty, state. After applying it, they have less to inspect when trying to find out what went wrong. By using the highlighting function, this query can be used to pinpoint where correct and faulty computations diverge.

We already showed how one can use AQL queries to alter the shown graph. To facilitate even more expressive alterations, GraphRedex permits extending the query functionality. Users can define an arbitrary JavaScript function to be executed on query results before they are rendered. This JavaScript code can modify the result in any way and may choose to present the result however it likes. Apart from modification of the result, the JavaScript function can also dynamically choose nodes to expand.

A language designer may, for example, wish to automatically expand all nodes with a strictly positive value. Listing 6 shows the full implementation of a plug-in that expands all nodes in a query's result if its `someValue` field is strictly positive. As this use-case is not useful for all languages, we did not include it in the default interface. When a plug-in is added, a button to use it appears.

FIGURE 8 Unbounded resource consumption

5 | EVALUATION

Our evaluation is conducted according to the methodology outlined in a systematic literature review of software visualization evaluation [11]. Consequently, our evaluation is not only focused on speed and correctness but also usability, adoptability, scalability, customizability, integration and query support.

The customization options of GraphRedex are described in Section 3.2. In this section we will first look at two usage scenarios to demonstrate our query support. Then, we validate our tool in terms of speed, correctness, usability, adoptability and perceived scalability in Section 5.2 with a user study.

5.1 | Usage Scenarios

We added the nine languages of the "Run your research" paper [1] by Klein et al. to GraphRedex. Of those nine, two interesting examples will be highlighted here: "A concurrent ML library in Concurrent Haskell" [15] and "A Theory of Typed Coercions and Its Applications" [16]. Modifying the existing semantics to be GraphRedex compatible took fewer than five minutes per language once we understood the original code.

5.1.1 | A concurrent ML library in Concurrent Haskell

The first semantics we look at is a PLT Redex implementation of "A concurrent ML library in Concurrent Haskell" [15]. In his paper Chaudhuri shows that one can implement Concurrent ML (CML) events in a π-calculus based language such as concurrent Haskell.

Concurrent ML is a high-level programming language that supports the creation of first-class synchronization abstractions [17]. One such synchronization abstraction is the non-deterministic `select` operator, which takes a list of communication operations and returns the result of the first communication operation that successfully completes and aborts the others.

The work shows how one can encode the `select` operator in the π-calculus such that it is non-deterministic. His encoding introduces so-called synchronizers that replicate the behavior of `select`. A distributed state machine then manages the channels and synchronizers in accordance to the CML semantics.

Comprehending Previously Uncovered Issues

It was previously uncovered that this semantics allows programs to consume unbounded resources [1]. A minimal example of a program that exhibits this behavior is shown below:

```
1 select(in c, out c)
```

The program simultaneously tries to receive and send on a freshly created channel. Due to the way the semantics is defined, the process will be retried indefinitely. While this is the expected semantics, the term gains extra synchronization data at each iteration leading to unbounded resource consumption.

Figure 8 shows the reduction graph of the encoded version of the minimal example above. The exact input is shown in Listing 9 of Appendix C. We see a repeating pattern: first either "`in c`" or "`out c`" gets selected for matching, then the matching fails, and the process repeats. By investigating the nodes we see that the terms behind the nodes grow in size, a fact also pointed out by Klein et al.

Discovering new Issues with the PLT Redex Model

The original PLT Redex model tries to accurately represent the semantics in the work of Chaudhuri [15]. Indeed, the model was sufficiently correct to uncover a small issue with the semantics. Interestingly, the model itself also contains a small issue, terms that are semantically equal are duplicated. While this does not lead to any issues regarding semantics, the duplication of the terms makes execution of tests and visualization of the model slower.

We were led to the error by inspecting the reduction graph of the compiled version of following code:

`select(out a, out b)`	`select(in a, in b)`	Outcome
`out a`	`in a`	channel `a` used
`out a`	`in b`	retry
`out b`	`in a`	retry
`out b`	`in b`	channel `b` used

TABLE 1 Possible outcomes of the `select(out a, out b) | select(in a, in b)`. The first two columns indicate the selected action by the threads, the last column shows the consequence of the selection

```
select(out a, out b) | select(in a, in b)
```

Where "`out ch`" indicates sending on the channel `ch`, "`in ch`" represents receiving on the channel `ch`, and the pipe symbol ("`|`") represents parallel composition. Listing 10 in Appendix C shows the output of this compilation step. Table 1 presents the four possible outcomes of this program of which three are unique (use channel `a`, use channel `b` or retry).

When `select` chooses to send and receive on the same channel, the program succeeds, if different channels are chosen, the `send` and `receive` never complete and `select` must make new choices. When running the program we thus expect three different outcomes, two resulting in a normalized node and one restarting the non-deterministic selection process.

Figure 9A shows the reduction graph of the original code in GraphRedex. We see six different branches coming from the start node, while we expected just three (use channel `a`, use channel `b` and retry). To reduce the number of nodes in the graph we issue a query to only show the shortest path to each normalized (red) node. With the reduced view, in the bottom right of Figure 9A, we see that four of the six branches end in two normalized nodes rather than one. Using GraphRedex we quickly find that the two normalized nodes at the end of the end of each lobe are semantically equivalent. We notice that the second argument of the structure differs only in the order of its elements. Since this argument of the term semantically represents a set, the order of its elements bears no meaning. We change the semantics to ensure that identical sets are represented by identical lists. This way the actual reduction graph agrees with the intuition behind the system. Doing this drastically reduces the amount of states in the reduction graph. Figure 9B shows the new reduction graph. We can now identify three branches coming from the start node, as expected. Two of them end in a single normalized node. If we place the mouse over those nodes, we see that they indeed represent the usage of channels `a` and `b` respectively.

Now that we have improved the semantics and obtained a reduction graph in it, we can learn from the visualization. In the original semantics it was not clear at first sight that there were three options for each level. The semantics of the language was thus edited to match the semanticists intent.

With queries, we can get an even more stripped down version of the reduction graph. The shortest path to all normalized nodes query (Listing 4) gives us the result shown in Figure 9C. Because we are now looking at the essence of the reduction graph, the overall structure is now clear.

We have shown how to use GraphRedex to detect problems with semantics by visually inspecting the reduction graph. We found that discrepancies between the language designers intuition, and the actual execution can quickly be found and corrected with GraphRedex.

GraphRedex Specific Adjustments

To make the existing PLT Redex implementation work in GraphRedex, we only needed to make a small adjustment. The existing implementation was split over multiple files, one file (`target.rkt`) contained the reduction relation describing Chaudhuri's state machine under the name `t->`. To make the implementation GraphRedex compatible, we created a new file named "`PLTGraphRedex.rkt`" shown in Listing 7. This file imports the reduction relation from `target.rkt` (line 2) and exports it under the name "`reductions`" (line 3) using the `rename-out` modifier from Racket. We pack the existing files and `PLTGraphRedex.rkt` in a zip and submit it to GraphRedex.

5.1.2 | A Theory of Typed Coercions and Its Applications

The central theme of the work we discuss in this usage scenario is the automatic insertion of type casts, also called coercions[18]. An example coercion is the conversion of a number to a string in languages like JavaScript where `"string" + 5` becomes `"string" + "5"`. Swamy et al.[16] illustrate that one can express several important rewriting scenarios as type-directed coercion insertions. The rewriting scenarios they have in mind include the automatic insertion of type casts in gradually typed languages,

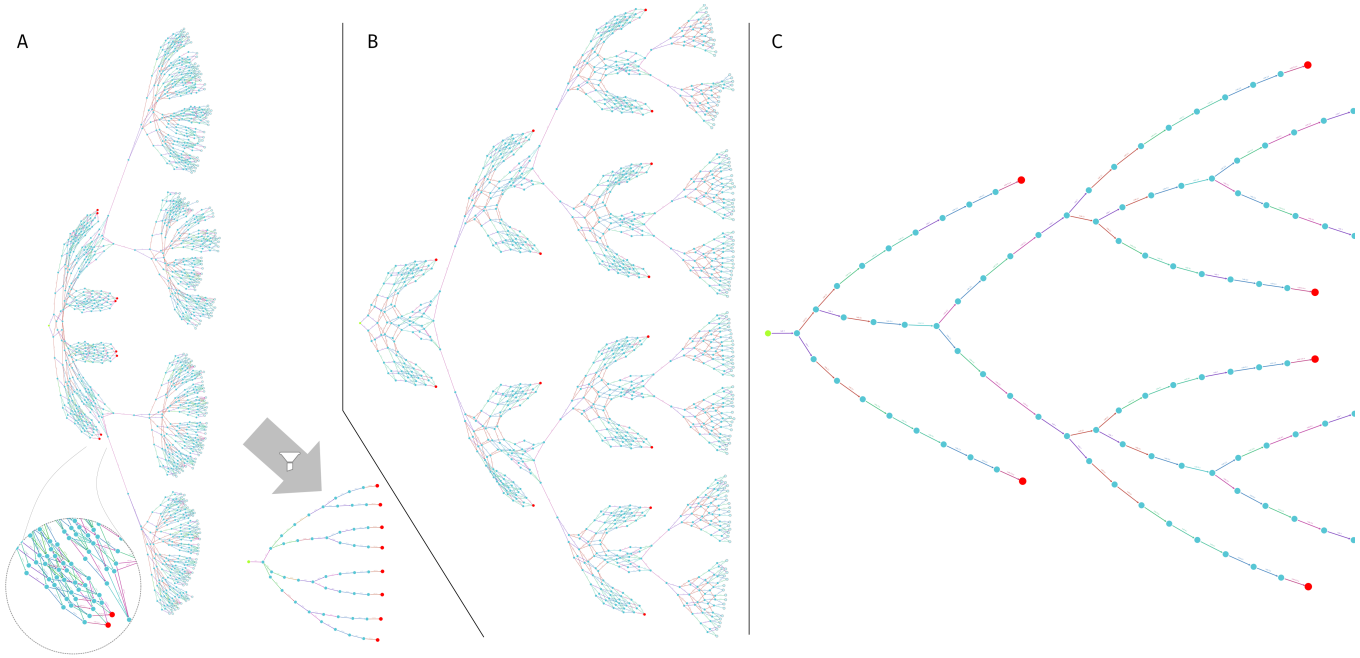

FIGURE 9 Result of the code of Listing 10 in the original semantics (left), and the adapted semantics (middle). Result of a filter query to show the path to all stuck nodes (right).

```
#lang racket
(require "target.rkt")
(provide (rename-out (t-> reductions)))
```

LISTING 7 PLTGraphRedex.rkt file added to the concurrent haskel implemention to make the semantics availible to GraphRedex.

automated tracking of the origin of values, automatically forcing the evaluation of lazy values and transparent access rights validation for function calls. They show that all these rewrites can be expressed as coercions. The work illustrates how to use type information to automatically determine the place to insert the calls to conversion functions for their rewrite scenarios. In this section we will show how GraphRedex's queries can used to perform more complicated tasks.

A PLT Redex implementation of a language that describes the judgments of the system was implemented[1]. A reduction in that language corresponds to an application of a judgment in the original system. As such, the reduction graph is a search tree.

Comprehending an Elaborate Example

We run the papers "Dynamic proxies example" (shown in Listing 11 of Appendix C) as implemented by Klein et al. in GraphRedex. The yielded reduction graph is shown in Figure 10. Leaves in the graph represent terms to which no further judgment applies. We can inspect a term to determine whether it represents a successful coercion insertion or not. A successful insertion distinguishes itself by its shape. Lines 7 to 11 of Listing 8 show a function called `is-ok?` that carries out this inspection and returns true when given a term representing a successful coercion insertion.

To make the value of `is-ok?` available in the graph, we add it to our `term->kv` function under the `solved` key. Each node in the graph will now have an attribute boolean `solved` in the sidebar indicating whether the term represents a successful coercion insertion. Queries can also access the attribute. This allows us to highlight all nodes that are successful with the query below.

```
FOR n IN @@nodes
  FILTER n.solved == true
  RETURN n
```

We can slightly alter the "shortest path to normalized nodes" query (Listing 4) by replacing the subquery on line 2 by the query above. The new query can highlight the path to all nodes with `solved == true`, as shown in lime green in Figure 10.

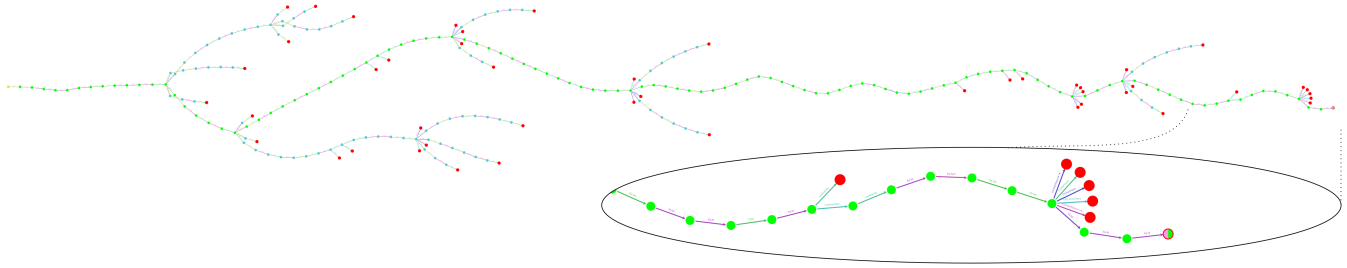

FIGURE 10 The reduction graph for coercion insertion. The lime green path is the path to a successful coercion. The pick node represents a successful coercion, because it is both in the path to a successful coercion insertion and the successful coercion, it is highlighted twice.

```
#lang racket
(require "atotcaia-search.rkt")
(provide (rename-out (all-rules reductions))
         term->kv)

; A coercion has succeeded if the term is an env with one argument
(define (is-ok? exp)
   (match exp
     [`(env ([,x_id ,any_val] ...)) #t]
     [_     #f])
  )

(define (term->kv exp) `((solved . ,(is-ok? exp))))
```

LISTING 8 PLTGraphRedex.rkt file added to the implementation of A Theory of Typed Coercions and Its Applications to make it GraphRedex compatibe.

The implementation in the artifacts of the Run your research paper fixed the problem in the semantics. Therefore, we are unable to reproduce the error found there here.

5.2 | User Study

To evaluate the effectiveness of GraphRedex as a tool for reduction graph exploration we carry out a randomized controlled user study. In our study, users complete a comprehension task and a debugging task. Depending on their group, they use either GraphRedex (experimental group), or the built-in visualization tools of PLT Redex (control group). We then compare the performance differences between the groups in terms of speed and accuracy. After carrying out the tasks, participants also fill in a user experience questionnaire of which we compare the results.

5.2.1 | Design

Participants of a survey about reduction graph exploration should have a good understanding of reduction relations and programming language design. To recruit fit participants, we e-mail researchers in the field and students that have at least passed a class in which reduction relations played a central role. Our mailing list is composed of participants of a Programming Languages Summer School, students who passed the "Fundamentals of Programming Languages" course at Ghent University, and other eligible participants to whom our invitation was forwarded. In total, we contact more than one hundred people of which sixteen respond and complete the survey.

Potential participants are split up randomly in two groups: a GraphRedex group, and a PLT Redex group. The former group only uses GraphRedex to carry out the given tasks. The latter group is limited to the built-in `traces` and `stepper` tools of PLT Redex in the Dr Racket environment[19]. Before learning what the task was, users in the PLT Redex group can opt to become a member of the GraphRedex group if they do not have Racket installed and do not want to install it. This may skew the number of participants per group and may induce confounding, but it increases the total number of participants.

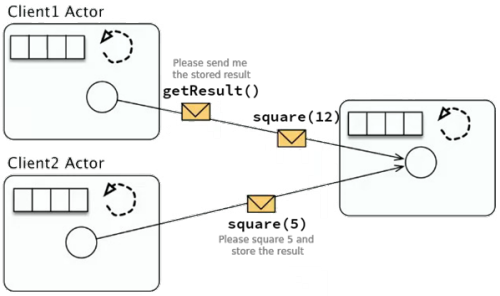

FIGURE 11 A schematic representation of the program analyzed in the comprehension part of the study. Two actors interact with a third math actor. The two actors ask the math actor to square a certain value and to store the result locally. When the math actor receives a getResult message, it returns its currently stored value.

We carry out the survey asynchronously in a remote setting with a Microsoft Forms online questionnaire. Both groups have their own questionnaire containing tool-specific instructions on how to start their tool with the semantics of each task. The full questionnaire can be found in Appendix B. The results of the survey are exported as a Microsoft Excel Open XML Spreadsheet with one row per participant. A Python program summarizes the results.

Before starting, we lay out the structure of the survey (two tasks followed by a questionnaire) and explain how required software can be obtained. Our survey starts with an eight-minute video introducing the participants' tool with a language and an example that has non-determinism. After the video, the tasks are executed.

We build our survey to cover all types of search tasks: *lookup* (known target, known location), *locate* (known target, unknown location), *browse* (unknown target, known location) and *explore* (unknown target, unknown location)[20,11]. We compare the participants' performance in terms of speed and accuracy for all types of tasks. At the end of the survey we also ask users to grade the usability and adoptability of the tool they worked with.

Comprehension Task

The first task in the study is a comprehension task. Participants investigate the reduction graph of a program in an unseen mid-sized language (AmbientTalk[21]). The core calculus of this language consists of 30 evaluation rules (excluding helper functions). In comparison, Featherweight Java[22], a minimal core calculus for Java and GJ, only has 10 evaluation rules.

When starting this task participants do not know what the underlying reduction rules are, except that it is an actor-based language. We do inform the participants of the intention of the program they are about to interact with. We show a schematic representation (Figure 11) of the program and give a brief explanation. The program has three actors. Two of them interact with a third "math" actor. The two actors each ask the "math" actor to `square` a certain value and to store the result locally. When the "math" actor receives a `getResult` message, it returns its currently stored value. The exact program used can be found in Listing 13 in Appendix C.

Users measure how long it takes to fill in six questions about the program using their tool. The full list of tasks can be found in Appendix B, in short, we ask the participants to:

- *lookup* the number of unique final states and detect if they represent the same value.
- *locate* and determine what a specific rule does,
- *browse* the graph to find a rule that ends in a finalized state,
- *explore* the graph to find a rule that introduces non-determinism and to find deadlocks.

To ensure that we only measure the time it takes to inspect the visualization and answer the questions we ask to install all required software before timing. To further improve the accuracy of the self-measured times we explicitly state when the timer should be started and when it should be stopped. We also make sure exploration can be started immediately by preparing the commands users need to execute to start exploration. For PLT Redex, we stored the input term in a variable, such that users only needed to copy-paste a small command to start exploring. For GraphRedex, we added the program to the sidebar such that it can be quickly selected.

$$
\begin{array}{rl}
p & ::= ((\text{store } v\ (x\ v)\ ...)\ (\text{threads}\ \ se ...)) \\
se & ::= (\text{start } e) \\
e & ::= (\text{getlock } x\ v\ e) \mid (\text{releaselock } x\ v\ e) \mid (+\ e\ e) \mid (\text{set } e\ e) \mid (\text{get}) \mid x \mid v \\
v & ::= number \\
x & ::= variable
\end{array}
$$

$$
\begin{array}{rl}
tc & ::= (\text{threads } se\ ...\ (\text{start } ec)\ se\ ...) \\
ec & ::= \diamond \mid (+\ ec\ e) \mid (+\ number\ ec) \mid (\text{set } ec\ e)
\end{array}
$$

$$
\begin{array}{l}
((\text{store } v_0\ (x_1\ v_1)\ ...)\ tc_1[(\text{get})]) \longrightarrow \\
((\text{store } v_0\ (x_1\ v_1)\ ...)\ tc_1[v_0])
\end{array}
\qquad [\text{get}]
$$

$$
\begin{array}{l}
((\text{store } v_0\ (x_1\ v_1)\ ...)\ tc_1[(\text{set } v_2\ e_1)]) \longrightarrow \\
((\text{store } v_2\ (x_1\ v_1)\ ...)\ tc_1[e_1])
\end{array}
\qquad [\text{set}]
$$

$$
\begin{array}{l}
((\text{store } v_0\ (x_1\ v_1)\ ...\ (x_i\ 0)\quad (x_2\ v_2)\ ...)\ tc_1[(\text{getlock } x_i\ v_{new}\ e_1)]) \longrightarrow \\
((\text{store } v_0\ (x_1\ v_1)\ ...\ (x_i\ v_{new})\ (x_2\ v_2)\ ...)\ tc_1[e_1])
\end{array}
\qquad [\text{getlock}]
$$

$$
\begin{array}{l}
((\text{store } v_0\ (x_1\ v_1)\ ...\ (x_i\ v_{lock})\ (x_2\ v_2)\ ...)\ tc_1[(\text{releaselock } x_i\ v_{lock}\ e_1)]) \longrightarrow \\
((\text{store } v_0\ (x_1\ v_1)\ ...\ (x_i\ 0)\quad (x_2\ v_2)\ ...)\ tc_1[e_1])
\end{array}
\qquad [\text{releaselock}]
$$

$$
\begin{array}{lll}
(+\ 0\ \ v_4) \Longrightarrow & v_4 & [\text{add0}] \\
(+\ v_3\ \ v_4) \Longrightarrow & (+\ \texttt{(-}\ v_3\ \ \texttt{1)}\ \texttt{(+ 1}\ v_4\texttt{)}) & [\text{add1}]
\end{array}
$$

FIGURE 12 The ThreadsLock language used in the debugging section of the user study. $A \implies B$ is short for $tc[A] \rightarrow tc[B]$. The add1 rule does not have a guard condition.

Debugging Task

One of the goals of creating an exploration tool is to help detect errors in a program's execution caused by a bug in the semantics. Finding an unknown bug is a typical *exploration* task: the target nor its location are known when searching. In the second task we measure how long it takes to find a bug in a small semantics using GraphRedex. Finding a bug in a system requires a good understanding of the expected behavior. To ensure that all participants have a good intuition for the semantics of the language, we take two measures. First, we opt to work with a small language, this has the benefit that it is easier for us to explain, but has the downside that it might be easier to find the bug. Second, we make a four-minute video explaining the intuition behind the language without showing the reduction rules explicitly.

There was a small bug in the language definition that participants needed to find by exploring the reduction graph. We ask the users to describe the bug in their own words in a free-form text field. By doing this we do not give suggestions about the bug. In a post-processing step, we manually analyze the answers to determine whether the participant found the bug.

We use a small threads lock language with addition for this task. The full reduction rules are shown in Figure **??**. The error resides in the add1 rule: no check verifies whether the first integer is strictly postitive. Therefore, the add1 rule could be used forevermore, leading to an infinite execution path (`(+ 0 5)` → `(+ -1 6)` → ...).

User Experience Evaluation

After the tasks, we evaluate the user experience and adoptability using the System Usability Scale questionnaire (SUS)[23]. We append questions about the perceived scalability to the SUS questionnaire. PLT Redex users are also shown GraphRedex if they wanted to* and are asked if they feel that GraphRedex would be a better fit for executing the tasks. We did not pose the opposite question to the GraphRedex group because using PLT Redex requires using software that not all users have installed.

*This question is placed at the end of the survey and is marked as optional. We do this to ensure that participants would finish the survey without the last question rather than not submitting entirely if they felt they had already done enough.

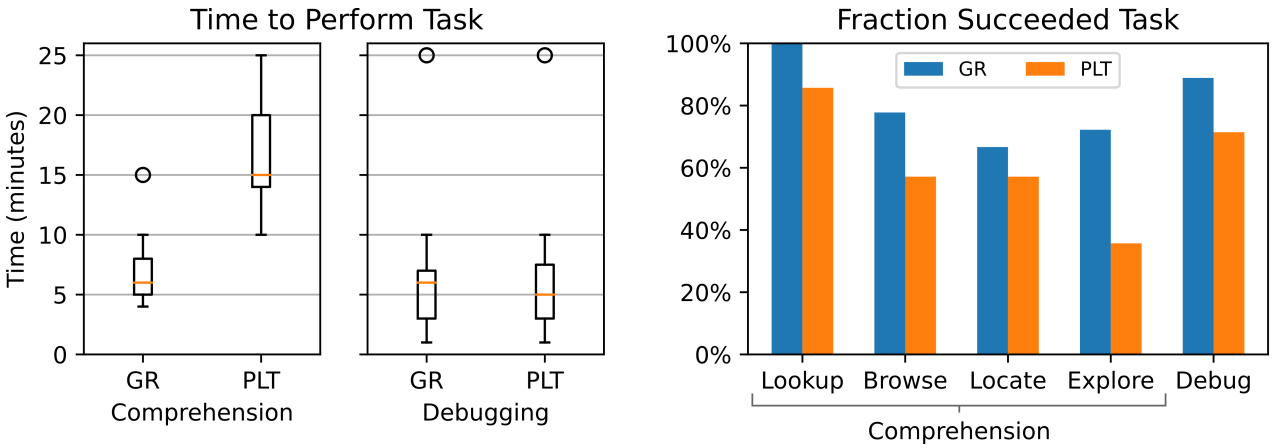

FIGURE 13 *Left*: Box plot of the time in minutes taken by participants to complete the comprehension and debugging tasks in GraphRedex (GR) and PLT Redex (PLT) including the time to reach incorrect results. *Right*: Fraction of participants that correctly fulfilled tasks. Debugging could also be classified as an "Explore" task.

5.2.2 | Results

Sixteen participants (7 PLT Redex, 9 GraphRedex) complete our survey. Although the sample size is small, we see that on average GraphRedex users answer questions about a semantics twice as fast and more accurate than PLT Redex users. Almost all PLT Redex participants (6/7) stated that GraphRedex would make completing the tasks easier.

Time and Accuracy

The box plots on the left of Figure 13 show the time taken by the participants to perform the tasks. The right side shows the participants' accuracy. We see that, on average for the comprehension task, GraphRedex users complete the task twice as fast and with a higher accuracy than the PLT Redex users (GraphRedex: $7' \pm 3'$, PLT Redex: $17' \pm 5'$).

For the debugging task, GraphRedex users have a higher success rate for finding the bug that PLT Redex users (GraphRedex: 8/9, PLT Redex: 5/7). The recorded times for the debugging task are similar between groups: excluding outliers but including participants with incorrect final results[†], GraphRedex and PLT Redex users both need $5' \pm 3'$ to find the bug. The high standard deviation on this value is caused by the fact that some users find the bug quickly while others need some time. The right box plot in Figure 13 also teaches us that for both GraphRedex and PLT Redex one participant took ± 25 minutes to find the bug. Closely inspecting the data reveals that the GraphRedex outlier reported that they "spend most of the time playing with the UI". The PLT Redex outlier reported that their approach was to check each of the languages operations with `traces` rather than looking for unexpected global effects, this may explain their increased debugging time.

User Experience

One of our goals with GraphRedex is to make it at least as user-friendly as PLT Redex. Our survey concludes that we reached this goal in our experiment. The System Usability Scores for GraphRedex (70/100) and PLT Redex (68/100) are comparable. These scores are considered "good"[24]. A graph with the results of the individual SUS questions can be found in Figure B1 of Appendix B.

Scalability

Because we do not expand the full reduction graph at once GraphRedex does not need to render as many nodes as a naive tool that expands the full reduction graph. Therefore, GraphRedex can handle large graphs better. The left graph of Figure 14 shows the scalability perceived by the participants during the survey. In this figure we compare our tool to the best-rated visualizer in PLT Redex (`traces`). We see that a larger fraction of participants indicates that GraphRedex is fit for large reduction graphs.

Some participants report performance problems with PLT Redex in the free-form feedback field at the end of the survey. One participant reports that the PLT Redex visualization became "sluggish", another participant states that "DrRacket froze 3 times

[†]Results excluding participants that did not find the bug and excluding the outliers the results are similar: $5' \pm 3'$ for both groups.

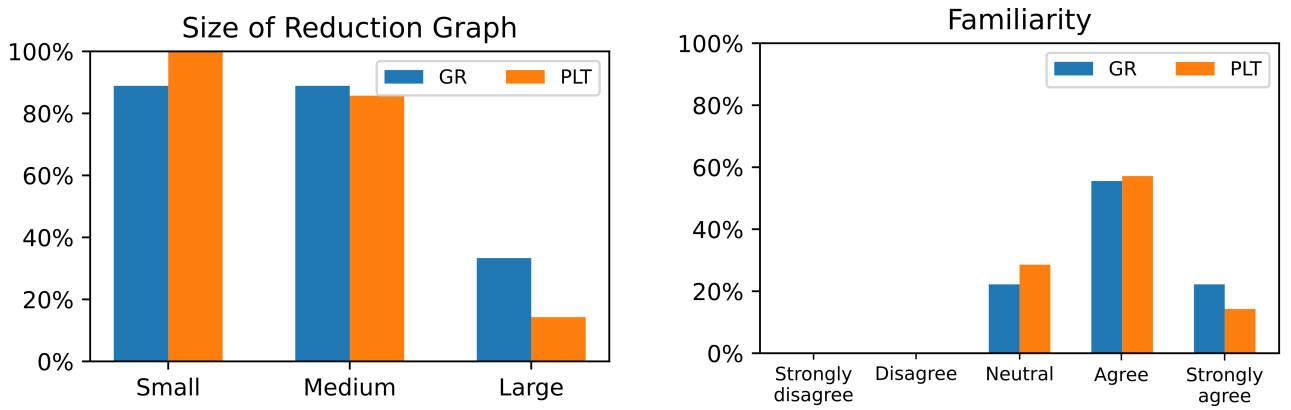

FIGURE 14 *Left*: Response to the question "I think *tool* is effective for exploring … sized reduction graphs". For PLT Redex only the result for the traces tool was used as it was superior to the results of the stepper. *Right*: Self reported expertise of the participants in both groups. The answers to "I am familiar with the theory of programming languages." are shown.

during this experiment". None of the GraphRedex participants reported similar problems although our tool renders more nodes by default than PLT Redex and there were more GraphRedex participants.

Analysis of Expertise

The self-reported level of expertise within both groups is similar (Figure 14 right). It must be noted in this context that some participants already had prior experience with PLT Redex and its tools before the survey, none of the participants had ever used GraphRedex before.

Comparison with PLT Redex

The last question in the PLT Redex survey briefly showed GraphRedex to the participants. Almost all PLT Redex participants (6/7) stated that they felt that GraphRedex would make it easier to answer the questions. Combining this result with our previous results suggests that PLT Redex users will likely adopt GraphRedex as it is seen as easier and while having a comparable user experience score.

6 | RELATED WORK

GraphRedex's contributions lie at the intersection of semantics engineering and graph visualization. In this section we first give an overview of existing semantics engineering tools, their visualization options and how they compare to our work. Next, we look at related work in graph visualization from the information visualization standpoint.

6.1 | Semantics Engineering Tools

Specifying a programming language using formal semantics has many benefits. By writing down the operational semantics[25] (small- or big-step), one can start making proofs about the system. To aid language creators in writing down and carrying out proofs about structural semantics many tools exists[26,27,28]. Coq[27], for example, is an interactive proof assistant. The tool allows language designers to write a mathematically rigorous definition of their language. They can then use a semi-interactive workflow to prove theorems about the defined mathematical object. Coq has been successfully used in various program certification efforts such as the Compcert compiler verification project[29]. Unfortunately, defining the semantics in Coq is an elaborate process. To reduce the burden on the language designer in the design phase of the language, more lightweight tools have been created[3,6,4,5]. These tools do not require the construction of a rigorous proof for every modification of the mathematical model, reducing the time between making a modification and testing it. GraphRedex targets semanticists in this design phase. We continue the related work by comparing GraphRedex with existing lightweight tools for semantics engineering.

Ott[3] defines a metalanguage to express languages in a syntax reminiscent to how reduction rules are usually written on paper (in Gentzen-style[2]). Once the rules are written down, the Ott tool validates the specification. It then continues by generating

FIGURE 15 An illustration of the ANIMA Maude stepper for their preconfigured DAM example (example 1 in their ÁTAME paper[33])

alternative representations of the rules. Supported representations include: definitions for use in various proof assistants, OCaml boilerplate code and Latex code for typesetting. The automated validation can catch simple errors in the language. Ott saves developers time when trying to carry out formal proofs with the proof assistant definitions. Unfortunately, Ott does not produce visualizations of how a program evaluates. Although it enables the working semanticist to start making proofs quickly, making proofs requires much effort if you are still unsure about the semantics. GraphRedex empowers developers to explore their language to find out what properties may hold by providing visual exploration.

The $\mathbb{K}$ framework[6,30] provides a way to write and efficiently interpret structural semantics. When writing down the reduction rules, language designers only need to write the parts of the evaluation context that change, rather than the whole configuration. The framework also permits language designers to use reachability logic to verify global properties about programs. In their paper[30], the designers of the framework note however that finding the right properties to verify is typically hard. They state:

> "However, we have found that in practice one spends a lot more time on coming up with the conceptually right properties to verify than on writing them in a particular notation."[30]

One approach the $\mathbb{K}$ framework has written in their future work* to alleviate this, is to create a visualizer. GraphRedex is such a visualizer that can help to find the "conceptually right properties". The PLT Redex component in GraphRedex could be replaced by a $\mathbb{K}$ framework component to bring exploration to the $\mathbb{K}$ framework. An existing visualizer for this framework is Theia[31], a tool for educators to visualize deterministic single-threaded reductions where the reduction graph is a line. GraphRedex focuses on exploration of non-deterministic reduction relations with complex reduction graphs and enables the user to query this graph. Theia focuses on a particular way of defining languages by making use of an abstract machine framework, GraphRedex is more general but requires user input to define custom visualizations.

Maude[4] resembles the $\mathbb{K}$ framework. It focuses on being a performant, simple and expressive programming language to define rewrite systems for operational semantics. Grounded in its logical foundations, Maude can leverage the agreement between mathematical and operational semantics to provide formal verification support to language creators. It also provides a "free" interpreter for the defined language. Contrary to the $\mathbb{K}$ framework, Maude does not work with evaluation contexts, it is a tool for rewrite systems written down as rewrite rules. The ANIMA Maude stepper[32,33] permits interactive exploration of the rewrite system defined in Maude. This stepper allows users to repeatedly choose where they want the program to take a step. Figure 15 shows the result of stepping through example 1 of their ÁTAME paper[33] and expanding all nodes up to depth five. We see that the view gets unmanageably wide very quickly. GraphRedex provides a similar exploration mechanisms and gives researchers a graph-aware query interface.

PLT Redex[5] is a domain-specific language designed for specifying and debugging operational semantics. It supports language creators in expressing the syntax and semantics of their language through its reduction rules. After writing down the reduction rules, PLT Redex can immediately interpret the language at hand. GraphRedex is built on this interpreting functionality. Even without GraphRedex semanticists can use PLT Redex to execute programs in their semantics. The system also has built-in unit testing functionality[1]. This works great when all desired properties are known beforehand. Unfortunately, some of these properties only become apparent once unexpected results crop up during the execution of larger programs. Even when automated testing finds a counter example, it can be hard for the developer to figure out what went wrong exactly. For small counterexamples, PLT Redex's `traces` and `stepper` functions can visualize the reduction of a faulty term. For highly non-deterministic programs such visualization quickly becomes complex and difficult to navigate. Because PLT Redex is embedded in the Racket programming language, semanticists can define arbitrary programs over the PLT Redex reductions. Unfortunately, the provided

*http://www.kframework.org/index.php?title=ProjectIdeas&oldid=947#Visual_stepper

interface is quite low-level and requires a good understanding of the tool to use it as an effective querying tool. GraphRedex fills the querying and visualization needs of PLT Redex users wishing to explore the execution of larger programs in their semantics. It also provides users with a graph-aware query language to filter and highlight the graph.

GraphRedex can visualize many PLT Redex semantics, but not all of them produce useful reduction graphs. Languages build primarily with meta-functions do not work well with GraphRedex. Meta-functions are transformations performed on terms, that are not reductions. They are meant to make small bookkeeping like adjustments to terms. Languages implemented entirely in meta-functions therefore mostly have a reduction graph of two nodes: the input and the result of applying the meta-functions. Although GraphRedex renders these graphs correctly, their usefulness may be limited. This is also the case for the visualizations in PLT Redex. Luckily, most meta-functions can be transformed to actual reduction rules.

Other related work lies in the realm of visualizing the execution of concurrent programs. Torres Lopez et al. showed how one can derive the semantics for a debugger based on the reduction semantics of the language to debug[34]. As a proof of concept, they applied their "debugging recipe" to a distributed programming language called AmbientTalk[21]. To make a working prototype out of their result, they created a tool named *Voyager* which visualizes the reductions of their debugging semantics. This tool was based on an early version of GraphRedex. We used the feedback from Torres Lopez et al. to improve GraphRedex. Since then, we added support for alternative syntax, visualization with `pict` and HTML, highlighting changes introduced by reductions rules, and many bug fixes.

CallFlow[35] is another visualizer for concurrent programs. Instead of working with the semantics of the language like Voyager, they use profiling information. With this information they create multiple interlinked interactive visualizations of the call flow. Their main visualization is a Sankey diagram that aggregates the profiling data. Users can interactively choose the level of detail in the diagram to locate bottlenecks and hotspots in their code. The goals of GraphRedex and CallFlow are complementary, while CallFlow focuses on specific hierarchical data from concurrent programs, GraphRedex focuses on the visualization of the reduction graph of arbitrary programming language semantics. GraphRedex enables general reduction graph exploration, which makes it hard or impossible to define a meaningful hierarchical structure that fits all arbitrary programming language semantics. Nevertheless, combining our query infrastructure with Sankey diagrams to extract specific hierarchical information from the reduction graph would be an interesting avenue of future work.

6.2 | Graph Visualization

There is a lot of work that aims to visualize graphs in an informative and aesthetically pleasing way. A descriptive 2011 survey[36] identified that one of the key issues in graph visualization is the size of the graph to display. In their survey they give an overview of various graph layout techniques that may help overcome this issue. Because of state-explosion, reduction graphs may become so large they cannot be handled in full. Therefore, GraphRedex uses an incremental exploration technique as suggested by this survey. The survey also proposes using clustering techniques to reduce the amount of visible elements. This improves both the clarity and performance of the rendering. The querying functionality in GraphRedex already supports for clustering, but with complex queries. As part of our future work, we would like to automate the generation of these queries. To this end, we will need to research what the important parts of reduction graphs are and how these should be aggregated with minimal user input.

To actually draw the graph GraphRedex uses a force directed layout (spring embedders). This class of algorithms place nodes in a graph only using the structural properties of the graph itself, rather than using domain specific information about a graph. One of the first algorithms in this class is Tuttes barycentric method[37,38]. Tuttes 1963 algorithm repeatedly places each node of the graph at the center of its neighboring nodes until a fixed point is reached. The algorithm of Eades[39,38] uses spring forces between adjacent nodes and repulsive forces between all nodes. The spring forces keep adjacent nodes at a certain distance from each other. The repulsive forces ensure that unrelated nodes are pushed away. Later, Frucherman and Reingold added simulated annealing to this algorithm[40]. Both the algorithms of Eades, and Frucherman and Reingold work iteratively. Each iteration $\mathcal{O}(|E| + |V|^2)$ forces need to be calculated. Where $|E|$ is the amount of edges (reductions) in the graph, and $|V|$ is the amount of nodes. These algorithms do not scale well when the amount of nodes increases.

GraphRedex's force-directed layout is similar to Frucherman and Reingold algorithm. We use spring like forces for the edges and a repulsive force between all nodes and simulated annealing. To improve performance the repulsive force is implemented using a Barnes-Hut simulation[41,42,38]. This technique builds a quadtree for the nodes in the graph and summarizes distant forces in the quad tree. This gives a $\mathcal{O}(|E| + |V| \log(|V|))$ time complexity per iteration of the simulation. Our approach resembles the GEM algorithm[43], the main difference is that GraphRedex computes the movement of all nodes for each iteration while the GEM algorithm only updates one node.

7 | CONCLUSION

GraphRedex is a user-friendly web application that empowers semanticists to explore the execution of programs in their programming language. This can be done without needing to put in an enormous amount of effort into visualization. What sets GraphRedex apart from other tools is its scalability and ease of use.

GraphRedex works for larger programs because it does not expand the whole reduction graph at once. Instead, it only expands to show the first 1000 nodes of the graph. Then it allows the user to interactively expand the reduction graph further. This way, the explosion of the amount of possibilities is limited to the cases the user deems interesting.

A second functionality that enables the scalability of GraphRedex is querying. Querying permits users to remove states form the graph that are not of interest to them. Using the predefined queries, graphs can be reduced to only show, for example, the shortest path to all normalized states. This reduction in the number of shown states keeps the view manageable. Users can also opt to highlight certain nodes in the graph to get better insights.

GraphRedex has two modes of exploration: Global and Local. In the former mode, high level semantics become apparent. In Local Exploration mode, the individual one-step reduction rules can be inspected, GraphRedex highlights the changes each reduction rule introduces.

We loaded all nine languages of the run your research paper[1]. With GraphRedex we found an error in one of the implementations, which one of the authors confirmed. We also conducted a randomized controlled user study that concluded that GraphRedex facilitates the comprehension of languages and eases debugging them while having good usability. GraphRedex users were able to complete a comprehension task in half the time it took PLT Redex users while maintaining a higher accuracy. All except one participant in the (PLT Redex) control group indicated that GraphRedex would make completing the survey easier.

ACKNOWLEDGMENTS

We would like to thank Robby Findler and Éric Tanter for their valuable feedback and discussions on an earlier version of GraphRedex. In particular, we want to thank Robby Findler for his help with the usage scenarios based on a paper[1] from his group. Finally, we would also like to thank the reviewers for their invaluable feedback.

A first draft version of GraphRedex was developed by Thomas Dupriez (ENS Paris-Saclay - RMoD, Inria, Lille- Nord Europe).

Robbert Gurdeep Singh received funding from the Special Research Fund (BOF) of Ghent University under grant number BOF18/DOC/327.

References

1. Klein C, Clements J, Dimoulas C, et al. Run Your Research: On the Effectiveness of Lightweight Mechanization. *SIGPLAN Not.* 2012; 47(1): 285–296. doi: 10.1145/2103621.2103691

2. Pierce BC. *Types and programming languages*. MIT press . 2002.

3. Sewell P, Nardelli FZ, Owens S, et al. Ott: Effective Tool Support for the Working Semanticist. In: ICFP '07. ACM; 2007; New York, NY, USA: 1–12

4. Clavel M, Durán F, Eker S, et al. *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Berlin, Heidelberg: Springer-Verlag . 2007.

5. Felleisen M, Findler RB, Flatt M. *Semantics engineering with PLT Redex*. Mit Press . 2009.

6. Roşu G, Şerbănuţă TF. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming* 2010; 79(6): 397–434. doi: 10.1016/j.jlap.2010.03.012

7. Valmari A. The state explosion problem. In: Reisig W, Rozenberg G., eds. *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*Berlin, Heidelberg: Springer Berlin Heidelberg. 1998 (pp. 429–528)

8. McCarthy J. A Basis for a Mathematical Theory of Computation. In: Braffort P, Hirschberg D., eds. *Computer Programming and Formal Systems*. 26 of *Studies in Logic and the Foundations of Mathematics*. Elsevier. 1959 (pp. 33 - 70)

9. Shneiderman B. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In: BEDERSON BB, SHNEIDERMAN B., eds. *The Craft of Information Visualization*Interactive Technologies. San Francisco: Morgan Kaufmann. 2003 (pp. 364 - 371)

10. Kienle HM, Müller HA. The Tools Perspective on Software Reverse Engineering: Requirements, Construction, and Evaluation. In: Zelkowitz MV., ed. *Advances in Computers*. 79. Elsevier. 2010 (pp. 189 - 290)

11. Merino L, Ghafari M, Anslow C, Nierstrasz O. A systematic literature review of software visualization evaluation. *Journal of Systems and Software* 2018; 144: 165 - 180. doi: 10.1016/j.jss.2018.06.027

12. Bostock M, Ogievetsky V, Heer J. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 2011; 17(12): 2301-2309. doi: 10.1109/TVCG.2011.185

13. Haas A, Rossberg A, Schuff DL, et al. Bringing the Web up to Speed with WebAssembly. In: PLDI 2017. Association for Computing Machinery; 2017; New York, NY, USA: 185–200

14. ArangoDB contributors . ArangoDB Query Language (AQL) Introduction. Online; . [Online; https://www.arangodb.com/docs/3.5/aql/; accessed 2019-21-05].

15. Chaudhuri A. A Concurrent ML Library in Concurrent Haskell. In: ICFP '09. Association for Computing Machinery; 2009; New York, NY, USA: 269–280

16. Swamy N, Hicks M, Bierman GM. A Theory of Typed Coercions and Its Applications. In: ICFP '09. Association for Computing Machinery; 2009; New York, NY, USA: 329–340

17. Reppy J, Russo CV, Xiao Y. Parallel Concurrent ML. In: ICFP '09. Association for Computing Machinery; 2009; New York, NY, USA: 257–268

18. Breazu-Tannen V, Coquand T, Gunter CA, Scedrov A. Inheritance as implicit coercion. *Information and Computation* 1991; 93(1): 172 - 221. Selections from 1989 IEEE Symposium on Logic in Computer Sciencedoi: 10.1016/0890-5401(91)90055-7

19. Findler RB, Clements J, Flanagan C, et al. DrScheme: A Programming Environment for Scheme. *J. Funct. Program.* 2002; 12(2): 159–182. doi: 10.1017/S0956796801004208

20. Munzner T. *Visualization analysis and design*. CRC press . 2014.

21. Van Cutsem T, Gonzalez Boix E, Scholliers C, et al. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures* 2014; 40(3-4): 112–136. doi: 10.1016/j.cl.2014.05.002

22. Igarashi A, Pierce BC, Wadler P. Featherweight Java: a minimal core calculus for Java and GJ.. *ACM Trans. Program. Lang. Syst.* 2001; 23(3): 396-450. doi: 10.1145/503502.503505

23. Brooke J. SUS: A "quick and dirty" usability scale. *Usability evaluation in industry* 1996: 189.

24. Bangor A, Kortum P, Miller J. Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale. *J. Usability Studies* 2009; 4(3): 114–123.

25. Plotkin G. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming* 2004; 60-61: 17 - 139. Structural Operational Semanticsdoi: 10.1016/j.jlap.2004.05.001

26. Nipkow T, Paulson LC, Wenzel M. *Isabelle/HOL: a proof assistant for higher-order logic*. 2283. Springer Science & Business Media . 2002.

27. Castéran P, Bertot Y. *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* Texts in Theoretical Computer ScienceSpringer Verlag . 2004. Traduction en chinois parue en 2010. Tsinghua University Press. ISBN 9787302208136.

28. Martínez G, Ahman D, Dumitrescu V, et al. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In: Springer; 2019: 30–59

29. Boldo S, Jourdan JH, Leroy X, Melquiond G. Verified Compilation of Floating-Point Computations. *Journal of Automated Reasoning* 2015; 54(2): 135–163. doi: 10.1007/s10817-014-9317-x

30. Roșu G. From Rewriting Logic, to Programming Language Semantics, to Program Verification. In: Martí-Oliet N, Ölveczky PC, Talcott C., eds. *Logic, Rewriting, and Concurrency: Essays Dedicated to José Meseguer on the Occasion of His 65th Birthday*. 9200. Springer International Publishing. 2015 (pp. 598–616)

31. Pollock J, Roesch J, Woos D, Tatlock Z. Theia: Automatically Generating Correct Program State Visualizations. In: SPLASH-E 2019. SPLASH-E. Association for Computing Machinery; 2019; New York, NY, USA: 46–56

32. Alpuente M, Ballis D, Frechina F, Sapiña J. Exploring conditional rewriting logic computations. *Journal of Symbolic Computation* 2015; 69: 3 - 39. Symbolic Computation in Software Sciencedoi: 10.1016/j.jsc.2014.09.028

33. Alpuente M, Ballis D, Sapiña J. Inferring Safe Maude Programs with ÁTAME. In: Davenport JH, Kauers M, Labahn G, Urban J., eds. *Mathematical Software – ICMS 2018*Springer International Publishing; 2018; Cham: 1–10.

34. Lopez CT, Singh RG, Marr S, Boix EG, Scholliers C. Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Brave New Idea Paper). In: Donaldson AF., ed. *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik; 2019; Dagstuhl, Germany: 27:1–27:30

35. Nguyen HTP, Bhatele A, Jain N, et al. Visualizing Hierarchical Performance Profiles of Parallel Codes using CallFlow. *IEEE Transactions on Visualization and Computer Graphics* 2019: 1-1. doi: 10.1109/TVCG.2019.2953746

36. Landesberger vT, Kuijper A, Schreck T, et al. Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges. *Computer Graphics Forum* 2011; 30(6): 1719-1749. doi: https://doi.org/10.1111/j.1467-8659.2011.01898.x

37. Tutte WT. How to Draw a Graph. *Proceedings of the London Mathematical Society* 1963; s3-13(1): 743-767. doi: 10.1112/plms/s3-13.1.743

38. Tamassia R. *Handbook of Graph Drawing and Visualization*. Chapman and Hall/CRC. 1st ed. 2016.

39. Eades P. A heuristic for graph drawing. *Congressus numerantium* 1984; 42: 149–160.

40. Fruchterman TMJ, Reingold EM. Graph drawing by force-directed placement. *Software: Practice and Experience* 1991; 21(11): 1129-1164. doi: 10.1002/spe.4380211102

41. Barnes J, Hut P. A hierarchical O(N log N) force-calculation algorithm. *Nature* 1986; 324(6096): 446–449. doi: 10.1038/324446a0

42. Brinkmann GG, Rietveld KF, Takes FW. Exploiting gpus for fast force-directed visualization of large-scale networks. In: IEEE. ; 2017: 382–391

43. Frick A, Ludwig A, Mehldau H. A fast adaptive layout algorithm for undirected graphs (extended abstract and system demonstration). In: Tamassia R, Tollis IG., eds. *Graph Drawing*Springer Berlin Heidelberg; 1995; Berlin, Heidelberg: 388-403.

How to cite this article: Gurdeep Singh, R, Scholliers, C (2020), GraphRedex: Look at Your Research, *Softw Pract Exper*, *2021; 51: 1322-1351*. https://doi.org/10.1002/spe.2959

APPENDIX

A ARANGODB

The graph database system that stores the reduction graphs is ArangoDB. We chose this database because it provides three things we needed:

- A good and extensible query language that treats graphs as a first class objects
- Support for storing data without Schema
- Support for multiple databases

Other database systems like Neo4j and MongoDB did not fulfill all our needs. In this appendix we will give an overview of the query language of ArangoDB: the ArangoDB Query Language (AQL).

A.1 Fundamentals

Reduction graphs in GraphRedex can be queried with the ArangoDB Query Language (AQL). Users can issue any read-only AQL query to their reduction graph. One fundamental read operation exists in ArangoDB: `FOR`. The `FOR` keyword allows iterating over collections. As stated before, `@@nodes` is such a collection, it contains all terms generated so far in the language. With the query below, we iterate over that collection.

```
FOR node in @@nodes
  RETURN node
```

The `RETURN` keyword specifies the value to project outwards. Here, the result of the query is an array of nodes. GraphRedex nodes have some predefined fields. For a node `n` in the reduction graph:

- `n.term` contains a string representation of the term represented by the node,
- `n._stuck` contains a boolean indicating if the term is normalized (i.e., no reductions apply to the term),
- `n._expanded` contains a boolean indicating if the term has been expanded already,
- `n._id` holds a unique identifier of the node in the graph.

We can use the `RETURN` keyword to project the set of nodes into an array of string representations of terms.

```
FOR node in @@nodes
  RETURN node.term
```

Apart from `term`, `_stuck`, `_expanded` and `_id`, nodes may have an arbitrary amount of other fields chosen by the language designer (see Section 3.2). A multi-threaded language may, for example, provide a field "`numThreads`" containing the amount of active threads. To find all terms with concurrency (more than one thread) the `FILTER` keyword can be used with `FOR`:

```
FOR n in @@nodes
  FILTER n.numThreads > 1
  RETURN n
```

With the above query we can highlight all nodes with concurrency in a reduction graph, we discuss this further in Section 4.2.

A.2 Graph Traversals

`FOR` can also traverse graphs. To this end AQL provides the `OUTBOUND` and `GRAPH` keywords. The former keyword announces that we wish to traverse a graph by following arrows (in the direction they point). The `GRAPH` keyword lets us specify the graph to traverse. To return all nodes reachable from the start node (`@start`) in the current reduction graph (`@graph`) developers can use the following query.

```
FOR n IN OUTBOUND @start GRAPH @graph
  RETURN n
```

Note that this query is different from querying `@nodes`, as `@nodes` contains all nodes in the language, including unreachable ones. The query above only selects reachable nodes. By default, AQL's traversals are depth-first, to use breadth-first search one can add "`OPTION {bfs: true}`". To limit the depth of search we can add a depth range after the `IN`. Following query returns all nodes within 5 reduction steps from the start node.

```
FOR n IN 0..5 OUTBOUND @start GRAPH @graph
  RETURN n
```

We can also capture the applied reductions (edges) with `FOR`, to do this we add an extra variable to the list of variables `FOR` binds. In the next query we changed "`n`" to "`n,e`", `e` will now be bound to the edge last followed to get to the node `n`. In GraphRedex edges have a `name` attribute holding the name of the reduction the arrow represents. The following query returns a list all used reduction rules within 5 steps from the start node. The `DISTINCT` modifier ensures that each name is only returned once.

```
FOR n,e IN 0..5 OUTBOUND @start GRAPH @graph
  RETURN DISTINCT e.name
```

The full path ArangoDB follows from the start node to `n` can be captured by adding a third variable. This variable iterates over the entire path taken to get to `n` (the first variable). ArangoDB represents a path as an object with two fields: `vertices` and `edges` containing respectively an array of traversed nodes and edges. The following query gives all paths from the start node to any node that has more than one thread.

```
FOR n, e, p IN 0..5 OUTBOUND @start GRAPH @graph
  FILTER n.numThreads > 1
  RETURN p
```

The query above also illustrates that `FILTER` can be used in the same way as it was used for collections. During a graph traversal, pruning certain branches may improve performance significantly. The `PRUNE` keyword enables this in AQL. As soon as the expression given it evaluates `true`, the traversal does not grow the path beyond the pruned node. We repeat the previous query and forbid ArangoDB to follow reductions with the name `par_step`.

```
FOR n,e,p IN 0..5 OUTBOUND @start GRAPH @graph
  FILTER n.numThreads > 1 && e.name != "par_step"
  PRUNE e.name == "par_step"
  RETURN p
```

Note that we also altered the `FILTER` to ignore paths that end with a `par_step` reductions. This is needed as the first node of a pruned branch is still returned. The result of the above query is a list of paths.

A.3 Graph Algorithms

When talking about graphs, the shortest path between two nodes is commonly requested. GraphRedex therefore has built-in methods for finding these paths. The `SHORTEST_PATH` and the `K_SHORTEST_PATHS` keywords provide this. The former modifier finds a shortest path from a starting point to an endpoint and iterates over the edges and nodes in it. The latter returns some number of paths to a node in increasing order of length. The syntax for using these functions is similar to normal traversal, the names of the functions are modifiers after the word `OUTBOUND`. The following query iterates over pairs of nodes and edges on the shortest path from the start node (`@start`) to a node with "`targetId`" as `_id`, the result will be a list of followed edges (`e` is returned).

```
FOR n,e IN OUTBOUND SHORTEST_PATH @start TO "targetId" GRAPH @graph
  RETURN e
```

To find multiple paths to a node sorted by length, one can use the `K_SHORTEST_PATHS` primitive. Following code captures the nodes of the 7 shortest paths from the startnode to the node with unique identifier "`targerId`".

```
FOR p IN OUTBOUND KSHORTEST_PATHS @start TO "targetId" GRAPH @graph
  LIMIT 7
  RETURN p.vertices
```

The `KSHORTEST_PATHS` function iterates over paths `p`, it does not provide nodes and edges to be bound by `FOR`. Each `p`, however has a `edges` and `vertices` field.

B SURVEY

B.1 Questionnaire

The Survey we used is shown below. Some alterations have been made to present it here. We prepended "SUS Q…" to certain questions to indicate that they correspond to the System Usability Scale questions[23].

S0: Intro
Watch this brief introduction to *the tool*.
S1: Exploration

1.Open *the tool* with example 1.

2.Look at this schematic overview of a program in an actors language….

3.Start a timer

Please answer the questions below about example 1. If you do not know the answer state "I don't know".
In the execution of the example, …

- How many possible end states are there? (One / Multiple / Infinitely many / I don't know)
- Do all endstates represent the same value? (Yes/No/I don't know) (Only shown if previous answer was not "One" or "I don't know")
- Can you find a rule that introduces non-determinism?
- What does the rule named $*$ do?
- What rule finalizes the result?
- Are there "deadlock"/"stuck" states? (States that are endstates, but should not) (Yes/No/I don't know)
- How long did it take you to fill in these questions? (excluding the time to watch the video)

S2: Finding errors

1.Watch this brief introduction to the semantics of the language we will be exploring.

2.Open *the tool* with example 2.

3.Start a timer

Example 2 contains a program in a language with a bug.

- Describe the bug in your own words.
- How long did it take you to find the bug?
- How did you find the bug?
- *The tool* helped to find the bug. (Yes/No)

S3: Experience
The following questions are about your experience with *the tool*. (There are no wrong answers).

- I think *the tool* is effective to explore (tick all that apply)
  - small reduction graphs
  - medium-sized reduction graphs
  - large reduction graphs
  - None of the above
- I think *the tool* is effective to explore (tick all that apply)
  - tiny/toy languages
  - medium complexity languages
  - complex languages
  - None of the above

For each of the following statements, please mark one box that best describes your reactions to *the tool* today.
(5 point rating from "Strongly Disagree" to "Strongly Agree")

- SUS Q1: I think that I would like to use *the tool* frequently.
- SUS Q2: I found *the tool* unnecessarily complex.
- SUS Q3: I thought *the tool* was easy to use.
- SUS Q4: I think that I would need the support of a technical person to be able to use *the tool*.
- SUS Q5: I found the various functions in *the tool* were well integrated.
- SUS Q6: I thought there was too much inconsistency in *the tool*.
- SUS Q7: I would imagine that most people would learn to use *the tool* very quickly.
- SUS Q8: I found *the tool* very cumbersome (awkward) to use.
- SUS Q9: I felt very confident using *the tool*.
- SUS Q10: I needed to learn a lot of things before I could get going with *the tool*.
- I found the querying functionality in *the tool* is useful.
- *The tool* is a useful tool for language designers.
- Language users can benefit from *the tool*.
- I am familiar with the theory of programming languages (Reduction relations, …).

S4: Optional comparison to GraphRedex (Only for the PLT Redex group)
Open example 1 and example 2 in GraphRedex

- Using GraphRedex would make answering the questions
  - Example 1 (much harder/harder/equally hard/easier/much easier)
  - Example 2 (much harder/harder/equally hard/easier/much easier)

S5: Other

- A free-form field for your thoughts (text area)

The score for the comprehension task is constructed as follows:

- *Lookup*: fraction of participants who answered both questions below correctly:
  - How many possible end states are there? (Multiple)
  - Do all endstates represent the same value? (No)
- *Browse*: fraction of participants who answered the question below correctly:
  - What rule finalizes the result? (`update` or `let`)
- *Locate*: fraction of participants who answered the question below correctly:
  - What does the rule named `*` do? (answers that mention multiplication or product)
- *Explore*: average of the fractions of participants who answered the questions below correctly:
  - Can you find a rule that introduces non-determinism? (`asynchronous-send-d`)
  - Are there "deadlock"/"stuck" states? (No)

B.2 SUS Results

The results of the SUS questions can be found in Figure B1. The list of questions can be found in Appendix B.1.

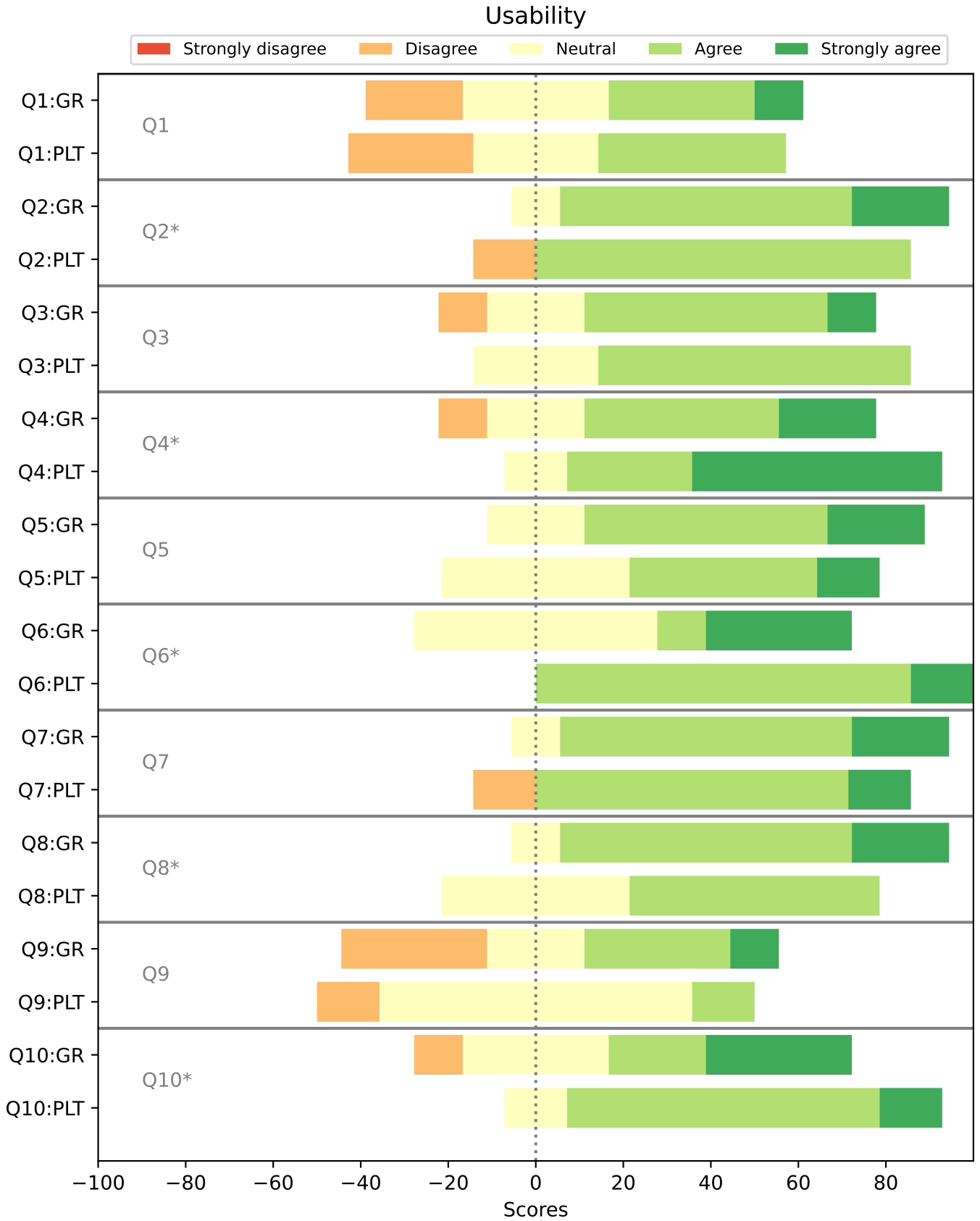

FIGURE B1 Summarized results of the System Usability Score questionnaire. GraphRedex (GR) has a total score of 70/100, PLT Redex has a total score of 68/100. Questions marked with a star have been inverted such that "Strongly agree" corresponds to "Very good usability"

C INPUT PROGRAMS

C.1 A Concurrent ML library in Concurrent Haskell

The code for Figure 8 is obtained by executing `(term (compile ((select (out ch1) (in ch1)))))` in the context of `compile.rkt` in example 4.4 from artifacts of the run your research paper.

```
(nu (v v1)
  ((active v1 (in ch1))
   (active v (out ch1))
   (free ch1)
   (open ((v (out ch1)) (v1 (in ch1))))))
```

LISTING 9: A program that does not stop.

The input that generated the graphs in Figure 9 is derived by executing the following:

```
(term (compile ((select (in ch1) (in ch2)) (select (out ch1) (out ch2)))))
```

The result is shown in Listing 10.

```
(nu
 (a b c d)
 ((active a (in ch1))
  (active b (in ch2))
  (active c (out ch1))
  (active d (out ch2))
  (free ch1)
  (free ch2)
  (open ((a (in ch1)) (b (in ch2))))
  (open ((c (out ch1)) (d (out ch2))))))
```

LISTING 10: A program that consumes infinite resources.

C.2 A Theory of Typed Coercions and Its Applications

The used example is from modified form page 337 of the "A Theory of Typed Coercions and Its Applications" paper[16], the original code is shown below, it translates to the code in Listing 11. The code in this listing was taken from the run your research paper.

$$\Sigma;\cdot \vdash (\lambda y : Lazy\ Int.y + 1)(\lambda : Unit.e) \rightsquigarrow (\lambda y : Lazy\ Int.(force[Int]\ y) + 1)(lazy\ \lambda x : Unit.e)$$

```
(env
  ()
  (coerce ((
      (· lazy : (forall (a) ((Unit → a) → (Lazy a))))
        force : (forall (a) ((Lazy a) → a))
    )
    ((· + : (Int → (Int → Int))) one : Int)
    ⊢ q q (λ y : (Lazy Int) ((+ y) one))
    ~~>
    (λ y : (Lazy Int) ((+ (((id Int) ∘ (inst force (Int))) y)) one))
    :
    ((Lazy Int) → Int))))
```

LISTING 11: Input for the dynamic proxy example.

C.3 WebAssembly

The code in Listing 12 was used to create the WebAssembly examples.

```
(((inst
    ((func
      ((inst 0)
       (code
        (func
         ()
```

```
          (-> (i32 i64) ())
          local
          ()
          ((get-local 0)
          ((get-local 1)
          ((store i64 6 0)
           (return ϵ)))))))
       ((inst 1)
        (code
         (func
          ()
          (-> (i32 i64) ())
          local
          (i64)
          ((get-local 0)
           ((load i64 6 0)
           ((tee-local 2)
           ((get-local 1)
           ((mul i64)
           ((set-local 2)
           ((get-local 0)
           ((get-local 2)
           ((store i64 6 0)
            (return ϵ)))))))))))))
       ((inst 0)
        (code
         (func
          ()
          (-> (i32) (i64))
          local
          ()
          ((get-local 0)
          ((load i64 6 0)
           (return ϵ)))))))
      (glob)
      (mem 0))
     ((func
       ((inst 1)
        (code
         (func
          ()
          (-> (i32 i64) ())
          local
          (i64)
          ((get-local 0)
          ((load i64 6 0)
          ((tee-local 2)
          ((get-local 1)
          ((mul i64)
          ((set-local 2)
          ((get-local 0)
          ((get-local 2)
          ((store i64 6 0)
           (return ϵ)))))))))))))
      (glob)
      (mem 0)))
   (tab () ())
   (mem (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)))
  ()
   ((const i32 0)
   ((const i64 42)
   ((call 0)
   ((const i32 0)
   ((const i64 5)
   ((call 1)
   ((const i32 0)
   ((call 2) ϵ)))))))))
  0)
```

LISTING 12: The input program for the WebAssembly example.

C.4 Survey: comprehension task

The code used in the comprehension task is shown in Listing 13.

```
(
  (actor client () ()
    (let (math (actor
                  (field result 0)
                  (method square x (set! (this $ result) (* x x)))
                  (method get-result y (this $ result))))
     in
     (let (client1 (actor
                  (method start math (let (a
                        (send math square (12) id-c1-math-square)) in
                        (when (sendf (ref id_new id_o)
                               get-result 0 id-c1-math-result)
                               result result)))))
     in (let (a
        (send client1 start (math) id-c-to-c1)) in
        (let (b (send math square (5) id-c-math-square)) in
        (when (sendf math get-result 0 id-c-math-result)
               result result))))))
  )
```

LISTING 13: The input program for the comprehension test of the survey.

C.5 Survey: debugging task

The code used in the comprehension task is shown in Listing 14.

```
((store 5 (x 0) (y 0))
    (threads
        (start
            (getlock x 1
            (getlock y 1
            (releaselock y 1
            (releaselock x 1
            1)))))
        (start
            (getlock y 2
            (getlock x 2
            (set (+ 2 (get))
            (releaselock x 2
            (releaselock y 2
            (get)))))))))
```

LISTING 14: The input program for the comprehension test of the survey.