TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Nikita Semõnin 223036IAIB
Andrei Mjastsov 223112IAIB
Timofei Beresnjev 222497IAIB

String Diagram Editor Backend Structure Development and Hypergraph Interpretation for Implementing Domain Specific Programming Languages

Bachelor's Thesis

Supervisor: Pawel Maria Sobocinski
PhD

Co-Supervisor: Anton Osvald Kuusk
BSc

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Nikita Semõnin 223036IAIB

Andrei Mjastsov 223112IAIB

Timofei Beresnjev 222497IAIB

Stringdiagrammi redaktori taustastruktuuri arendamine ja hüpergraafide tõlgendamine valdkonnaspetsiifiliste programmeerimiskeelte rakendamiseks

Bakalaureusetöö

Juhendaja: Pawel Maria Sobocinski

PhD

Kaasjuhendaja: Anton Osvald Kuusk

BSc

Tallinn 2025

Author's declaration of originality

We hereby certify that we are the sole authors of this thesis and that this thesis has not been presented for examination or submitted for defense anywhere else. All used materials, references to the literature, and work of others have been cited.

Authors: Nikita Semõnin, Andrei Mjastsov and Timofei Beresnjev

04.06.2025

Abstract

String diagrams[1] represent mathematical operations through drawings. They allow people to visualize complex mathematical processes in a clear visual form, making them easier to understand. Such diagrams are valuable in linear algebra, probability theory, and other domains where visualizing data processing and interrelationships between elements is essential.

This thesis presents the design and implementation of the Ivaldi software system for working with string diagrams, generating diagrams based on Python code, and generating executable Python code based on user-created diagrams. The system allows users to create diagrams of boxes, wires, and spiders representing different processes. Developed initially as a purely visual tool, the system has evolved to include a hypergraph[2]-based structure and server-side functionality supporting JSON-based import/export. The central contribution of this work is integrating a hypergraph structure, which enhances the program theoretically and practically. This thesis also discusses issues related to hypergraph logic, as well as other modifications that contributed to improvements in the program. The final system supports diagram nesting, code and diagram generation, and hypergraph visualization based on the created diagrams, providing a flexible environment for visually designing and testing computational logic.

This work serves as a proof of concept, demonstrating that string diagrams can be applied not only as a theoretical representation but also as a practical tool for working with programming code, addressing real-world problems.

The thesis is in English and contains 50 pages of text, 5 chapters, 49 figures.

Lühikokkuvõte

Stringdiagrammi redaktori taustastruktuuri arendamine ja hüpergraafide tõlgendamine valdkonnaspetsiifiliste programmeerimiskeelte rakendamiseks

String-diagrammid[1] kujutavad matemaatilisi toiminguid joonistuste abil. Need võimaldavad inimestel visualiseerida keerulisi matemaatilisi protsesse selgel visuaalsel kujul, muutes need kergemini mõistetavaks. Sellised diagrammid on kasulikud lineaaralgebras, tõenäosusteoorias ja muudes valdkondades, kus on vaja visualiseerida andmete töötlemist ja nendevahelisi seoseid.

Käesolev lõputöö tutvustab Ivaldi tarkvarasüsteemi loomist ja teostust, mis võimaldab töötada string-diagrammidega, genereerida diagramme Python-koodi põhjal ning luua täidetavat Python-koodi kasutaja loodud diagrammide alusel. Süsteem võimaldab kasutajatel luua "boxes", "wires" ja "spiders" koosnevaid diagramme, mis kujutavad erinevad protsesse. Alguses ainult visuaalse tööriistana loodud süsteem on arenenud hüpergraaf[2] põhise struktuuri ja serveripoolsete funktsioonide, mis toetavad JSON-põhist importi ja eksporti. Lõputöö peamine panus on hüpergraafi struktuuri integreerimine, mis täiustab programmi nii teoreetiliselt kui ka praktiliselt. Samuti käsitletakse töös hüpergraafide loogikaga seotud probleeme ja muid muudatusi, mis viisid programmi täiustamiseni. Lõplik süsteem toetab diagrammide pesitsemist, koodi ja diagrammide genereerimist ning loodud diagrammide põhist hüpergraafi visualiseerimist, pakkudes paindlikku keskkonda arvutusloogika visuaalseks kavandamiseks ja testimiseks.

See töö toimib kontseptsiooni tõestusena, demonstreerides, et stringiskeeme saab rakendada mitte ainult teoreetiliseks esituseks, vaid ka programmeerimiskoodiga töötamise vahendina, esitades mõningaid praktilisi probleeme.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 50 leheküljel, 5 peatükki, 49 joonist.

List of abbreviations and terms

AST	Abstract Syntax Tree
BE	Back End
FE	Front End
GUI	Graphical User Interface
JSON	JavaScript Object Notation
Main execution block	The conditional `if __name__ == "__main__"` in Python, also known as the "main guard" or "main block", is a common programming pattern. It determines whether a Python script is being run directly or imported as a module into another script[3].
PDF	Portable Document Format
PNG	Portable Network Graphics
SQL	Structured Query Language
SVG	Scalable Vector Graphics
UI	User Interface
XML	Extensible Markup Language

Table of contents

List of figures

1 Introduction

1.1 Background

Currently, there is a lack of effective tools for working with string diagrams. Although some tools do exist, they cannot operate beyond their limited scope and are often focused only on small test cases or very narrow fields. The aim of Ivaldi is to be a more general-purpose editor that allows users to draw a diagram and then translate it into their specific sphere or set constraints and rules for the editor based on their needs. Achieving this requires a well-structured and consistent representation of graphical structures.

1.2 Analysis

1.2.1 Cartographer

Carotgrapher[4] is a tool for string diagrammatic reasoning. The program offers a convenient and intuitive interface that can be used to create nodes and connections between them.

Pros:

- Easy to use for creating small string diagrams;
- Visually oriented editor focused on categorical structures[5];
- The overall design should be sufficiently general to be used in other use cases as well.

Cons:

- Does not have a project import/export system: you cannot save and then open a project as a separate file;
- Does not support visualizing hypergraphs or more complex structures that may arise when working with string diagrams;
- No integration with programming languages and the ability to generate code based on the diagram;

- The platform is limited to the functionality of the web interface and does not allow customizing rules or node behavior, which is critical for more general applications.

Cartographer is effective for visualizing simple string diagrams for educational or theoretical purposes, but it does not provide the flexibility needed for deeper interaction with the diagrams, their interpretation, or export.

1.2.2 Diagrams.net

Diagrams.net[6] (draw.io) is a universal editor that enables the creation of various types of diagrams, including string diagrams. It is a powerful visualization tool, but not specialized only for string diagrams.

Pros:

- Support for import/export in various formats (XML, PNG, SVG, etc.);
- Flexible interface, a large number of forms and connections;
- Convenient for the general design of logic and structures, especially when it comes to interaction schemes.

Cons:

- Does not support string diagram semantics;
- Connections do not inherently convey directionality;
- No built-in hypergraph support;
- Working with string diagrams in this program requires a lot of manual configuration and can be inconvenient and time-consuming, especially when complexity increases.

Diagrams.net offers a powerful visualization tool but lacks an understanding of the semantics of string diagrams. This makes it useful for prototyping or presenting diagrams, but it is not the best choice for working with string diagrams.

1.3 Current situation

Initially, Ivaldi provided the ability to work with string diagrams, but only for visual purposes. It was possible to add boxes, wires, and spiders. It has also implemented

logic to save the created diagram in JSON format and import the JSON file to continue working on the project. The initial version of the Ivaldi application backend did not include hypergraphs, but it has logic to communicate the user interface with the backend and store all user-created objects separately on the backend. For more details, see Appendix 2.

1.4 Initial version problems

Current shortcomings of the Ivaldi program:

- The current BE data structure is inefficient and not aligned with the user interface;
- Testing and validation are exhaustive and need better structuring;
- The overall design should be sufficiently general to support other use cases as well;
- The program does not support hypergraphs or all possible functionality that is based on hypergraphs;
- Missing the ability to assign functions to a box;
- Missing code generation and code-based diagram generation.

1.5 Objective

The development project's objectives:

- Improve the backend structure to be more efficient and reliable;
- Ensure backward compatibility to allow for future integration with other components;
- Implement backend structure translation into hypergraphs for diagram rewriting based on existing theory;
- Implement export and import of various theories to simplify diagram usage in specific domains;
- Develop functionality to simplify diagram usage via hypergraphs;
- Add functionality to generate executable Python code from the given diagram;
- Add functionality to generate a diagram from the provided code and show possible errors and problems of this process;
- Enhance the communication between the user interface and backend architecture to improve responsiveness, maintainability, and data consistency;
- Develop a visual representation of the hypergraph-based backend structure.

1.6 Development methodology

To achieve the goals, the following methodology will be used:

1.6.1 Analysis of the current backend structure

- Conduct a comprehensive review of the existing backend structure (BE) to identify inefficiencies in the user interface (UI);
- Analyze the current design's limitations to determine specific areas for improvement;
- Map domain-specific requirements to support compatibility with export/import functionality and hypergraph representation.

1.6.2 Redesign and optimization of the backend structure

- Design a new data structure optimized for hypergraph support and diagram rewriting rule implementation;
- Ensure backward compatibility to enable the program to smoothly operate with existing components and functions;
- Consider the need for efficient processing of large diagrams, including maintaining scalability and performance.

1.6.3 Implementation of the improved structure

- Implement the new BE data structure with mechanisms enabling hypergraph-based diagram export/import;
- Create reliable algorithms to generate code from provided diagrams;
- Identify errors preventing proper code generation;
- Conversely, allow diagram generation from existing code with error-detection mechanisms.

1.6.4 Validation and testing

- Develop comprehensive test cases covering different use cases, including edge conditions and complex diagram manipulations;
- Verify the correctness of hypergraph translations and the feasibility of applying diagram rewriting rules;

- Clearly document all new features and usage instructions to facilitate integration and future development;
- Get feedback and evaluation from Pawel Maria Sobocinski and his research group at the Laboratory for Compositional Systems and Methods.

1.6.5 Semantic enhancement and usability improvement

- Add user-friendly tools that simplify diagram analysis and correction.

2 Technologies

2.1 Python

The main programming language used in the project is Python[7]. All the logic of the initial version of Ivaldi was implemented in Python. Python was selected due to its simplicity of syntax and the wide range of available libraries. It is especially convenient for rapid prototyping and working with data, making it well-suited for implementing functionality related to building and analyzing string diagrams.

2.2 Tkinter

The standard Tkinter library[8] was used to implement the graphical user interface (GUI). It provides tools for creating windows, buttons, canvases, and other interface elements. Thanks to its simplicity and built-in support in Python, Tkinter allows for the quick development of user interfaces.

2.3 ASTor

The ASTor library[9] is used to work with code. It allows interaction with the abstract syntax tree (AST), making it possible to import code to build a diagram and generate code based on a user-created diagram. This approach provides flexibility and reduces the number of errors associated with manual text processing of programs.

2.4 Hypernetx

The Hypernetx library[10] is used for hypergraph visualization. It provides convenient tools for representing and displaying complex relationships between elements, which makes it especially useful when working with diagrams that contain multiple inputs and outputs. Visualization helps users better understand the structure of the hypergraph and the logical relationships between nodes.

3 Overview of the application

In this bachelor's thesis, the authors primarily focused on the backend of the Ivaldi program and developed several features that significantly improved it. Firstly, the authors implemented hypergraph creation, which works in parallel with the user's creation of objects on the diagram. Afterwards, the authors were able to implement code generation, which is based on the logic of the hypergraph and also retrieves code from each diagram box. During the development of code generation, the authors also created the BoxFunction class, which stores the code added by the user to a box. This class facilitated the addition of the code importing feature in Python, generating a diagram in the process. Simultaneously, the authors improved the capabilities for importing and exporting diagrams so that users could save their work in our program. One of the final features added was hypergraph visualization, which significantly assisted the authors during testing of various cases and may also be helpful for some users. Subsequently, the authors will examine each functionality in greater detail.

3.1 Hypergraphs

3.1.1 Essence

A hypergraph is a graph in which edges (also known as hyperedges) can connect an arbitrary number of vertices, rather than just two as in a classical graph. In other words, a hyperedge represents a group, an indivisible connection, a single object. This has its advantages when explaining processes.

For example, consider three events — A, B, and C. Let event C be plant growth, event A be sunlight, and event B be water. Suppose the situation is modelled such that the plant grows only when it has access to both water and sunlight simultaneously. In a classical graph, this process is represented by three vertices and two edges. However, since event C occurs only when both A and B are present, simply showing connections is not enough. A condition must also be introduced, stating that access to C is granted only if both A and B are present.

This may hinder understanding of the process (see Fig. 1), as merely looking at the image or symbolic representation of the connections is insufficient. In contrast, with a hyperedge (see Fig. 2), it becomes clear that A, B, and C form a single unit. And without A, there will be no C, just as without B there will be no C — only together they do work. This allows to explain the process without introducing an additional condition. This approach also works with edge deletion. In the case of a classical graph, only one edge can be deleted, for example, between A and C. This would imply that event B alone would be sufficient for event C, which is incorrect. In a hypergraph, the connection between A and C cannot be deleted separately; only all connections can be deleted at once, which helps structure processes more effectively.

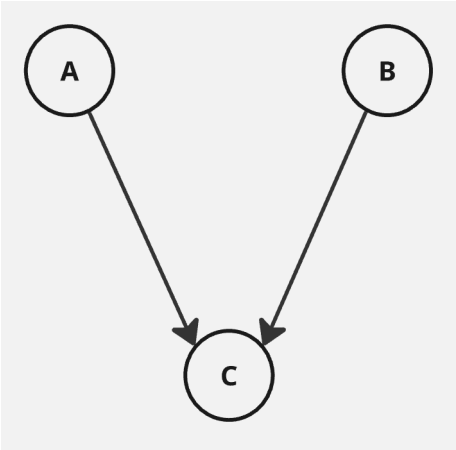

Figure 1. Directed graph with events A, B, C.

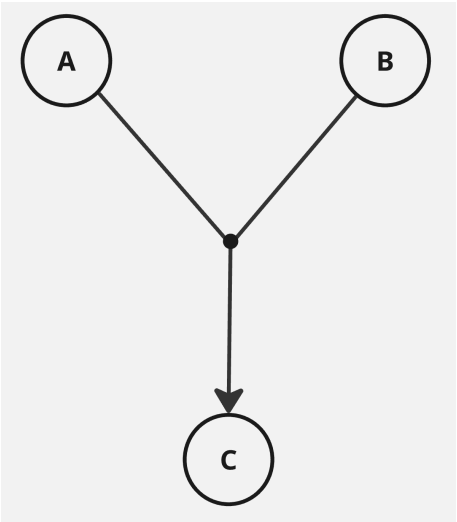

Figure 2. Directed hypergraph with events A, B, C.

In conclusion, a hyperedge perfectly represents an indivisible connection among any number of objects, which is extremely convenient when describing interconnected processes.

3.1.2 Connection with string diagram

In a string diagram, hypergraphs are used to describe processes. The following establishes a correspondence between the elements of a string diagram and parts of a hypergraph:

1. The input and output of the diagram are the vertices of the hypergraph, labeled as source and target;
2. Wires are also vertices of the hypergraph;
3. Spiders are vertices of the hypergraph;

4. Boxes are hyperedges.

At this stage, a drawing of a simple diagram is presented (see Figure 3). It shows a hypergraph with one source node and three target nodes, oriented from left to right. An important remark — in the figure, the input of the diagram, wires, and the spider are connected. Based on the text above, all these elements are vertices, and since vertices can only be connected by hyperedges (or edges in a classical graph), this would be an invalid hypergraph without additional conditions. However, in the Ivaldi editor, such groups of vertices are treated as a single larger vertex (more on this in section 3.1.3 ”Implementation Approach”).

Hypergraphs are used to mathematically represent a string diagram. The hypergraphs in a string diagram are directed and connected. Since boxes serve as the linking elements in the diagram and can have an arbitrary number of connections, they fit the description of hyperedges. All other objects in the string diagram store values — therefore, they are vertices.

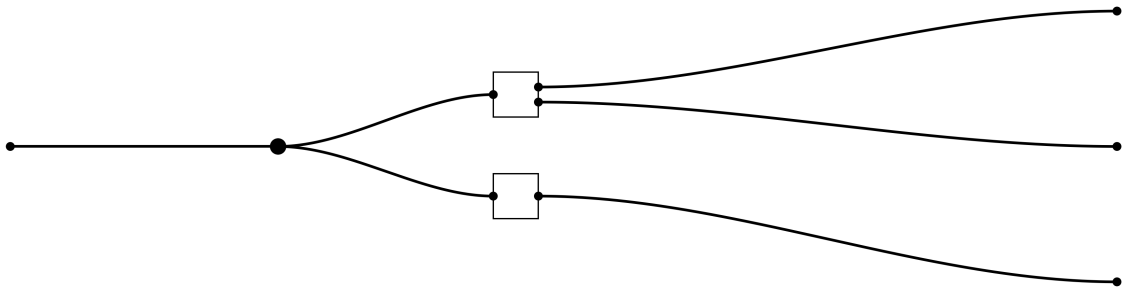

Figure 3. Example of a string diagram.

3.1.3 Implementation approach

In the final version for working with hypergraphs, there are four classes:

1. HyperEdge – which represents a hyperedge;
2. Node – which represents a node(vertex);

3. Hypergraph – which combines vertices and hyperedges;
4. HypergraphManager – which is used to modify the hypergraphs.

Additional details about the implementation: in the authors' approach, the hypergraph is always connected. As a result, when a part of the hypergraph—such as a hyperedge—is deleted from the diagram, two or more separate hypergraphs may be formed. Because of that, the class HypergraphManager is needed since, in the case above, a new hypergraph in the diagram is needed. And execute hypergraph creation in another hypergraph – illogical and violates the SOLID principle, to be more accurate, "S" – Single Responsibility Principle[11]. When deleting a node/hyper edge, the following happens: the HypergraphManager invokes the corresponding method (remove node or remove hyper edge) with the id of the element as an argument. Next comes the search for the hypergraph that contains this element. After a successful search, the hypergraph executes a corresponding method for removing an element (remove node or remove hyper edge), into which the element to remove is passed as an argument, and in this element executes a method that clears all this element‘s connections – this is how element deletion happens. After the hypergraph changes its attributes, for instance, the system removes nodes and hyperedges from the corresponding lists. If the node was the hypergraph source node, then it is deletes from the hypergraph source nodes list. Eventually, after deleting an element, HypergraphManager checks if the hypergraph divides into two or more hypergraphs; if yes, it adds new hypergraphs to the list of hypergraphs. The situation is similar in the operations related to adding a node/hyper edge, except that when an element is needed to be added to a hypergraph, the hypergraph can combine with another hypergraph.

Considering the role of connected nodes, since diagram inputs/outputs, wires, and spiders are nodes, a situation can occur where multiple nodes are connected to each other, but without a hyper edge, that is impossible in graphs. As a result, it was decided to combine those nodes into one big node, and thanks to this, the problem was solved. Class Node has an attribute directly_connected_to of type list, which stores all the nodes connected to this node. An important node stores only the directly connected nodes. For example, in the Fig. 4, the first wire, which is connected directly to the diagram‘s input and to the spider, will have these two elements in its list directly_connected_to.

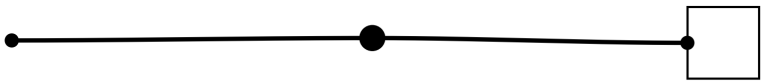

Figure 4. Example of connected Nodes.

But since wires after the spider also belong to the big node, which are often needed to be retrieved in the program, a method “get_united_with()” was created, which returns all nodes that belong to one big node. The method uses the algorithm Breadth First Search[12]. The algorithm collects all nodes from the list directly_connected_to and then collects all nodes from their list directly_connected_to.

In the editor Ivaldi, there is the possibility to select part of the diagram and convert it to a subdiagram. In place of this selected diagram in the parent diagram appears a box that represents a subdiagram. This box has the same number of inputs/outputs as its subdiagram. Since instead of the selected part of the diagram there is now a box, the hypergraph structure also changes, because now it is only one hyper edge instead of combination of nodes and edges. And if count this box as an ordinary hyper edge, loses the information stored in the box‘s subdiagram. In this subdiagram, there can be multiple amounts of hypergraphs. To prevent information loss, a new method was added to hyper edge called “get_hypergrahs_inside()”, which returns all hypergraphs of the box‘s subdiagram. This method is essential as it gives the full information about the hypergraph structure to which this box belongs, with a subdiagram inside. Thanks to this architecture, it is possible to simplify the hypergraph structure when it’s needed, so it can be beneficial when creating big diagrams, because a part of the diagram can be hidden without information loss, and if it is needed, the full structure can be reverted.

3.1.4 Implementation issues

Two issues have already been mentioned in the section implementation approach 3.1.3, namely, about situations when nodes connect to each other, and situation with subdiagrams. But to get the current view, the hypergraph‘s architecture was changed multiple times.

In the first version, the hypergraph was closely connected to the frontend side, namely with objects – Wire, Spider, Box. Since at that moment the hypergraph stored inside itself only the Node and the HyperEdge, without any additional methods for modifying them (due to the fact that all hypergraph changes were made only on the frontend side), and classes Node and HyperEdge referenced the canvas objects (Box, Spider, Wire). Furthermore, every time when the diagram was changed, the hypergraph was calculated completely anew. The problem with this implementation is a significant lack of separation of the frontend (canvas objects) from the backend (hypergraph), which involves the dependence of the hypergraph on the frontend, and every change in the frontend immediately affects the hypergraph, which should not be the case. Also, it makes debugging more complex, because it is difficult to determine the source of the issue – whether this comes from the backend side or the frontend side. Hypergraph creation was also placed on the frontend side, which is illogical and makes debugging more complex. Moreover, hypergraph did not take into account the subdiagram.

In the next version, the hypergraph was separated from the frontend. Created mapping structure – WireAndSpiderToNodeMapping and BoxToHyperEdgeMapping, which were the dictionary, where the key is the id of the canvas object (Wire, Spider, Box) and the value is related to its hypergraph element (Node or HyperEdge). Node and HyperEdge became independent classes, with their own methods and attributes. Now, the issue with dependency on the frontend has been removed. Also, methods were created in the Hypergraph class, required for modifying the hypergraph (adding/removing nodes/hyper edges), but were not considered in some cases (for example, when deleting the node, it does not mean that it is needed to delete all its children, since they can be connected to the other parts of hypergraph), which caused a lot of bugs. Furthermore, at that moment, there was not a definition of united nodes to one big node; nodes were just connected to each other. Not related directly to the hypergraph implementation – there was no intermediary between frontend objects and the hypergraph, i.e., the frontend passed state changes directly to

the hypergraph. Because of this, even though there was a border between frontend and backend, there was still a bit of backend (hypergraph) logic in the frontend code.

In the final version, many issues were solved, subdiagrams were considered, and nodes connected to each other started to become one big node. There is also an intermediary between the hypergraph and the frontend – diagram_callback. Frontend sends events to the diagram_callback with all required information, and from there, the hypergraph applies changes. Also, mapping structures were deleted since a new class – HypergraphManager, that stores all hypergraphs, so it became easy to find the desired node or hyper edge, iterating through hypergraphs. The classes Hypergraph, Node, and HyperEdge were rewritten to enhance code readability and to promote dependency separation.

3.1.5 Future improvements

At this stage, when importing the project, the hypergraph is created in parts, creates some nodes, then connects them to the hyper edges with all checks for hypergraph connectivity, which impacts importing time negatively. Due to this, the project imports not as fast as desired. It can be solved in two ways – when importing, create a hypergraph only when the entire diagram is imported successfully, and just scan all connections, or when exporting the project, export also the hypergraph, thanks to which when importing the diagram, there is no need to calculate everything, just copy the ready structure.

Since the structure of a hypergraph changes (e.g., removing/adding a vertex), the hypergraph can either split into several other hypergraphs or merge with several others - this becomes a performance issue. Because of this, it is needed to do a lot of checks, inspecting for links, connectivity, etc. One of the solutions is to simply change the links of the selected vertex/hypergraph, but not to check the connectivity of the hypergraph, which will immediately speed up the work a lot. And build the hypergraph only when it is needed.

Even though tests for hypergraphs are currently written, still some cases are potentially untested. It is necessary to write more tests for different cases.

3.2 Diagram Callback

3.2.1 Essence

Diagram callback – is the intermediary between the frontend and the backend. Every time objects on the canvas update their state, they send events with all required information to the diagram callback. The diagram callback has its own representation of the diagram, without extra attributes and methods from the frontend. This representation diagram callback updates when it receives events. Also, it sends changes to the hypergraph. And being used in a code generation, when it is needed to determine the exact order of nodes (because in the hypergraph, it is impossible to determine the order of source nodes, but there is the order of nodes that are connected to the hyper edge, since hyper edge connections are ordered). In the future, the diagram callback might be used to communicate with some new structures.

3.2.2 Current state

The diagram callback has simplified versions of canvas objects - Wire, Box, Spider, and Connection. It is important to note that in the diagram callback Box is called Generator, Wire, Spider - Resource, and Connection - ConnectionInfo. The Generator class stores connections and the box function, the Resource class stores connections. In the ConnectionInfo class, the index of the connection connected by Generator(Box) or Resource(Spider and Wire), and on which side this connection is located. Also in diagram callback there is a method def receiver_callback(self, action: ActionType, **kwargs), through which the communication between frontend and backend takes place. The other methods in this class are auxiliary methods that help to recreate a simplified version of the diagram.

3.2.3 Code rewritting

Originally diagram callback was already implemented. But the code was very confusing, undocumented, and untyped (for example, in several places there was a list named "connection" of 4 or sometimes 3 elements that appeared, which had a structure and a value, but it was hard to trace it). Initially, the authors attempted to supplement and document the existing code; however, it became evident that maintaining such code was overly complex.

That is why authors decided to understand the structure of the diagram callback as much as possible and then rewrite it from scratch. The authors managed to eliminate unnecessary methods and confusion. The structure became clearer, the code was documented. Types were added to objects.

3.2.4 Future improvements

At this stage, the diagram callback can be called completed, since all features are implemented, and all known issues have been solved. There is no need to add new features. However, the following steps could have been performed:

1. Document all events and their required arguments in a separate document;
2. Write more tests to check correctness.

3.3 Diagram generation from code

3.3.1 Original feature form

To allow working with the programming code in Ivaldi, the first feature that was added is generating a string diagram representing some code structure. The initial version of the functionality was selecting one file with specific Python programming language content and importing it inside the program. To generate a diagram, a file was obligated to comply with the next rules[13]:

- Contain at least one function declaration;
- Contain the main execution block with at least one function call that was declared previously, and assigned variable with that;
- Every function has at least one argument and returns something;
- Functions has no side effect (do not change external state in addition to returning a result).

Based on the code inside the file, the diagram was generated representing the main execution block structure. The sequence of the code was preserved, rotating from the “top to bottom” position to “left to right.” Every function call had a box representation in the diagram, and wires were representing utilizations of the assigned variables as the arguments passed inside other functions. If a variable had multiple usages, it was represented by the spider

connected with one wire to the box that created the variable, and multiple wires to the boxes, whose functions were using this variable.

For example, the following code (see Fig. 5):

```
def add(a, b):
    return a + b

def multiply_and_return_first(a, b):
    return a * b, a

def subtract(a, b):
    return a - b



if __name__ == "__main__":
    sum_result = add(2, 3)
    product_result, unused = multiply_and_return_first(
        sum_result, sum_result)
    final_result = subtract(sum_result, product_result)
```

Figure 5. Input code

Will be represented as the following diagram (see Fig. 6):

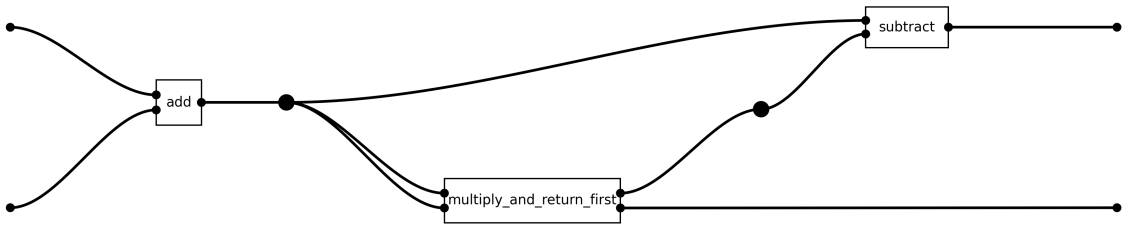

Figure 6. Diagram generated from the code in 5.

The initial version had significant disadvantages. Firstly, it did not allow importing multiple python files to generate a diagram. Typically, complex programs that are used in a production environment consist of many different files that are strongly intertwined, so if this feature wants to qualify as a useful programming tool, it must support importing multiple files. Secondly, it did not consider imports in the file and the fact that some functions can be imported from a python standard library. And lastly, if the main execution block contained any code other than function call, it was not parsed correctly. These important points needed to be improved to widen variants of supported code syntax.

3.3.2 Supporting a greater variety of code

First of all, logic was implemented to support importing multiple files with Python code for diagram generation based on them. In order to be sure that the diagram can be created, validation was added for the selected files. One of the files was obligated to contain the main execution block, because without it the program was not able to produce the main diagram. Also, in case when more than one file contained the main execution block, the error was thrown. The main diagram could use declared functions from any file. Moreover, function declarations could use other declared functions in their body[14].

After that was added the ability to have imports in the Python code and use imported functions in declared functions. This was a significant improvement because it allowed users to use methods from the Python standard library in the imported code.

Handling plain python code in the main execution block was a big step in making this feature a useful and effective programming tool. For this purpose the logic was changed which shows how the program parses the imported code and what data structures are used for that. The new approach made it possible to treat both function call and plain code expression as a base line of code and correctly generate diagram parts for essentially any possible variant of code except more complex blocks as conditions and loops. Moreover, this fundamental change of the underlying logic, aimed at increasing its level of abstraction and ensuring future extensibility, enabled the introduction of a new import mode.

3.3.3 Deep diagram generation

All previously described functionality can be classified as the shallow diagram generation. The program creates only one main diagram that does not contain any subdiagram. In opposition to that can be classified another mode which is named deep diagram generation. The main difference and strong part of this variant was the fact that it allowed reflecting the declared function structure as the subdiagrams of the boxes in the main diagram. To switch between different diagram generation modes, the user is prompted to choose whether he wants to generate a deep diagram or a shallow one. For more details, see Appendix 3.

For example, the following function declaration (see Fig. 7):

```
def calculate_paint_cost(liters_needed, price_per_liter):
    base_cost = liters_needed * price_per_liter
    tax = base_cost * 0.2
    service_fee = 150
    total_cost = base_cost + tax + service_fee
    return round(total_cost, 2)
```

Figure 7. Input code.

Will be represented as the following diagram (see Fig. 8):

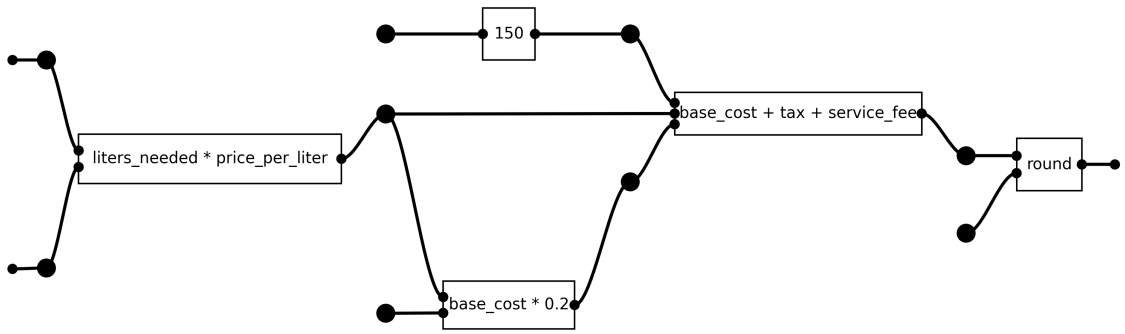

Figure 8. Diagram generated from the function declaration in 7.

Deep diagram generation can be useful in multiple cases. The most obvious example is that user can clearly see the flow of the variables in code, what resources are used and how many use cases they have. This can help in identifying potential unused variables or vice versa unintended usages of the resource. If some variables are incorrectly changed or reassigned, a created diagram helps notice these bugs or errors. In addition, by using the generated diagram, the user can clearly see places where the code is weakly linked together. If a subdiagram can be visually separated to the big chunks of the boxes, this indicates that the function has multiple separate flows of the data, and it can be split into more functions, which is considered as a good code design[15]. Finally, creating a deep diagram based on a code can bring a fresh perspective on it, enhancing the understanding of how it works and what are the key aspects of that. Such code visualization can be particularly useful when analyzing unfamiliar code for the first time or revisiting one's own code after a significant period of time.

3.3.4 Further improvements

Current state of the generation diagram from code is not as effective as it could be. There are many different aspects that can be added to improve efficiency and widen this feature scope of the application.

Many different programming structures, such as conditions, loops, and exceptions, are not treated correctly. Complex parts of the code must be examined carefully in order to be converted into diagrams because the string diagram theory is specific. Not all of the programming concepts can be easily translated to the domain of the string diagrams.

Another part of the feature that can be improved is making this feature more convenient to use with real production code. For example, automatically selecting files from the user computer, based on imports in the initially selected files, so that the only file that needs to be imported into the program is the starting point of the program. Any other files will be fetched automatically according to their location in python imports.

Also, this feature could use support for another programming language, such as Java, C#, JavaScript, or PHP. This can benefit more people because they can import their programs written differently and use this feature without needing to change their source code. A strong base of the parsing algorithm and sufficiently general data structures open a window

of opportunity for every modern programming language. Even an SQL query can be represented as a string diagram using this feature if correct support for this language is added.

3.4 Code generation from the diagram

3.4.1 Essence

When the users has created their string diagram, they can add functions to the boxes. Currently, the program supports only Python code. Once the user has added all functions to all boxes, he can generate a Python code file containing all the functions they have added and a main function (see Appendix 4). This main function takes as many arguments as inputs in the diagram, logically calls all tasks based on the diagram's structure, and returns as many outputs as there are in the diagram.

3.4.2 Implementation approach

To implement this feature, it was necessary to work directly with code. Two main approaches are using regex or the ASTor library. The autopep8[16] module is also beneficial, making the code more readable and clear. Regex is a rather primitive tool, and using it requires handling many edge cases, as Python code can vary significantly. That is why the ASTor library is more suitable for this task[17]. ASTor can automatically determine where variables or function names are, which significantly simplifies the task. To simplify the implementation, the authors decided to break the task into several stages: adding imports, adding all functions that are added to boxes, and creating the main function. Creating the main function was also divided into steps: defining the function signature, generating the function content, and generating the return statement.

3.4.3 Implementation issues

Changes in hypergraph logic

Since the first implemented hypergraph logic was incorrect, the authors had to rewrite it, so code generation stopped working, and the authors had to adapt it to the new logic. However, this led to a better implementation and fixing some bugs from the previous logic.

Function and global variable names

Since users add functions separately to each diagram box, function and global variable names may be repeated. This may cause errors when adding them to the final file, so the authors had to rename them. The authors solved this by adding different indexes to all functions to ensure that names are always unique.

Correct order of diagram inputs and function arguments

Another issue was adding the input arguments to the main function in the correct order. If not done correctly, the logic fails. For example, if the user expects the first argument to be a string and the second one a number, the whole logic breaks if the variables are misordered.

3.4.4 BoxFunction

The BoxFunction class plays an important role in code generation. It is created when code is added to a diagram box. Initially, the authors simply stored code as a string inside BoxFunction, making it a storage object. However, as development continued, the authors realized this approach lacked flexibility, which made using it in "Diagram generation from code" difficult. So the authors improved it: when created, the class splits the code into imports, helper functions, global variables, and the main function that takes as many arguments as the number of box inputs and returns as many as the number of box outputs[18]. After improving this class, working with code generation became easier, as there was no longer a need to search separately for imports and other code parts.

3.4.5 How Code Generation Works

Code generation begins by scanning all hypergraphs on the canvas. For each connected hypergraph, a separate main function is created. This function receives input values, calls the corresponding functions from each box in the correct logical order, and produces output values. The process starts by mapping the diagram's input nodes to the main function's input arguments. Then, using the hypergraph's structure, the generator determines the order of function execution and connects them with variable assignments, mirroring the connections in the diagram. Each function from the boxes is inserted into the file before the main function. The main function acts as a coordinator, calling all other functions and wiring data between them, as shown in the diagram. If a diagram contains subdiagrams (boxes that contain subdiagrams), the code generator will descend recursively into them,

generating code for the subprocesses and integrating them into the higher-level function. However, it will not ascend to parent diagrams — only the visible scope is processed. The final result is a well-structured Python file with one or more main functions, depending on the number of hypergraphs. Each main function is self-contained and executable independently.

3.4.6 Features

Code generation creates as many main functions as connected hypergraphs on the canvas. So, if two hypergraphs exist in the diagram (see Fig. 9), two main functions are generated, as two separate processes.

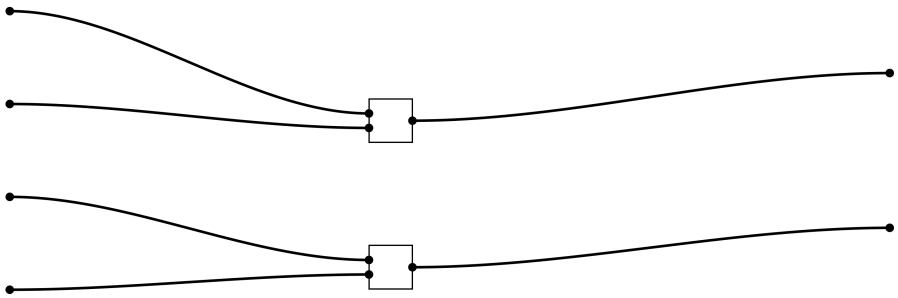

Figure 9. Example of two separate connected hypergraphs.

The following is an example of how the code will look (see Fig. 10):

```
def invoke_0(*numbers: list) -> int:
    if len(numbers) < 2:
        raise ValueError(
            'Numbers amount should be at least 2')
    return sum(numbers)


def invoke_1(*numbers: list) -> int:
    if len(numbers) < 2:
        raise ValueError(
            'Numbers amount should be at least 2')
    return numbers[0] - sum(numbers[1:])


def main_0(input_0, input_1):
    res_0 = invoke_1(input_0, input_1)
    return res_0


def main_1(input_0, input_1):
    res_0 = invoke_0(input_0, input_1)
    return res_0
```

Figure 10. Generated code.

However, there is also a way to see how combining these two processes would look. To do this, users must create a subdiagram from all elements on the current canvas (see Fig. 11).

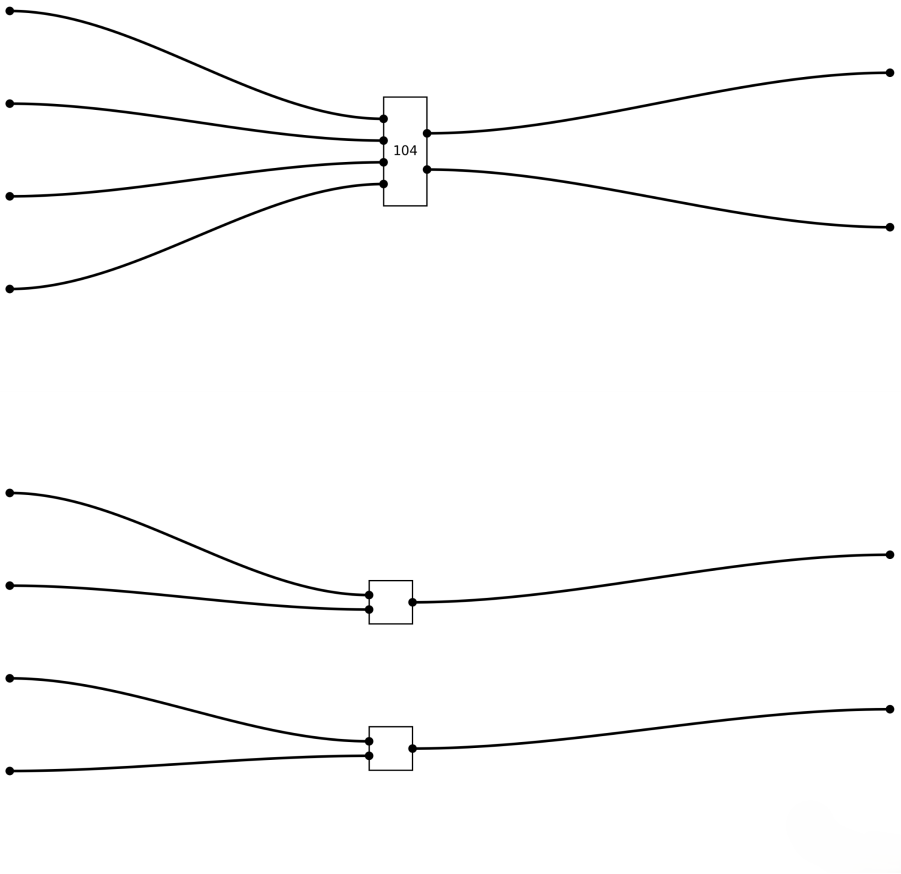

Figure 11. Example of two separate connected hypergraphs combined into a subdiagram.

The following is an example of how the code will look (see Fig. 12):

```
def invoke_0(*numbers: list) -> int:
    if len(numbers) < 2:
        raise ValueError(
            'Numbers amount should be at least 2')
    return sum(numbers)


def invoke_1(*numbers: list) -> int:
    if len(numbers) < 2:
        raise ValueError(
            'Numbers amount should be at least 2')
    return numbers[0] - sum(numbers[1:])


def main_0(input_0, input_1, input_2, input_3):
    res_0 = invoke_0(input_0, input_1)
    res_1 = invoke_1(input_2, input_3)
    return res_0, res_1
```

Figure 12. Generated code.

This approach adds greater flexibility and allows the user to determine the preferred format for viewing the final code. Since the authors addressed the concept of subdiagrams, code generation operates recursively. If the canvas opened in the program contains a box with a diagram, the generator will process that diagram too, and so on, deeper into the structure. However, the code generator does not look upward in the hierarchy. Specifically, if the current canvas is a subdiagram of another one, it will not be processed. The authors adopted this design because it is more intuitive and convenient for the user. If the user wants to see the bigger picture, they should go to a higher-level canvas. If they are interested in individual processes, they can create a subdiagram and generate code only for that.

3.4.7 Limitations

Currently, several important limitations might prevent the user from generating code from the diagram:

1. All diagram boxes must contain code;
2. The number of box inputs/outputs must match the number of function inputs/outputs;
3. A box cannot be connected by a wire that is not linked to one of the diagram's inputs.

   The following case is not valid for code generation (see Fig. 13):

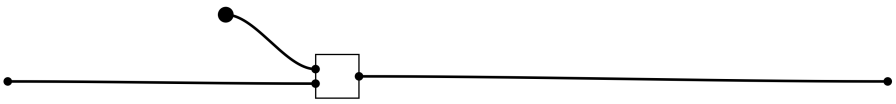

Figure 13. Example of the diagram, that is not valid for code generation.

3.4.8 Future possible improvements

There are many ways to improve the current code generation: First, the authors did not have enough time to add support for other programming languages besides Python, such as Java, JavaScript, C++, C, etc. Another idea that came up during development was generating a function from code strings if a separate function is visualized using a string diagram. However, this must be done carefully to ensure theoretical consistency. The authors could also add more settings for code generation, such as generating code only for a selected part of the diagram.

3.5 Importing/Exporting

3.5.1 Original state

Initially Ivaldi had functionality to save the state of the project as a JSON file and load a previously saved file to continue working on the diagram. There were no other options to save the project. This process had multiple minor inconsistencies that could lead to the problem in some cases. Overall code state was problematic due to being too specific. This led to the conclusion that in order to implement more variants of exporting and importing,

the code structure needed to be refactored. The result was the generalized code composition that could support the easy process of implementing new import and export variants.

3.5.2 Hypergraph exporting

After hypergraph logic implementation was added, the opportunity to export hypergraph structure as a JSON file with all data about graph structure preserved. This feature allows the user to extract diagram backend underlying logic to examine it or use as input in another program. Every exported hypergraph contains his ID, hyperedges, source nodes and connected nodes groups. This allows users to save all data about the graph structure and not lose any necessary part of it.

3.5.3 Support for new diagram data

Additionally, saving and loading the project from a file required modifications to accommodate the updated diagram data. After adding features which allowed to work on the programming code in the Ivaldi project, the saving process lacked the preservation of the assigned functions to the box. Also, to allow the user to load a Python file as the code for the box in a diagram, the program needed a way to import code for the box. The authors reviewed two approaches. Importing a Python file itself, or importing a JSON file with Python code representation. After consideration, the first approach was chosen. The main problem with the second variant was the fact that to use the JSON format, some type of algorithm to parse a Python file and convert it to JSON is needed, which is an extra step[19]. In addition, the JSON file with embedded Python code is harder to review, and its length is longer.

Furthermore, numerous other things were lost after reloading the project. To fix it, the new enhanced project JSON structure was developed and implemented in the project.

3.6 Hypergraph visualization

3.6.1 Essence

Hypergraph visualization is helpful for users to understand the structure and relationships between diagram elements. It allows users to see the connections between nodes and hyperedges. The system supports the display of all connected hypergraphs located on a

single canvas, each visualized in a separate area of the overall graphical window.

3.6.2 Implementation Approach

The main visualization library used is Hypernetx, which specializes in working with hypergraphs. It provides correct rendering of nodes and edges and their automatic layout. The use of matplotlib, in turn, enables flexible control over the graphical design and placement of elements. The authors avoided excessive custom drawing by choosing a combination of proven libraries, ensuring both reliability and clarity of the result[20]. To improve the visualization's readability, a grid of visualized hypergraphs was added: Depending on the number of connected hypergraphs on the canvas, the program creates the corresponding number of visualizations. Each hypergraph is visualized in a separate area with a frame and a title, making perception intuitive. Here is the example of the diagram (see Fig. 14) and its hypergraph visualization (see Fig. 15):

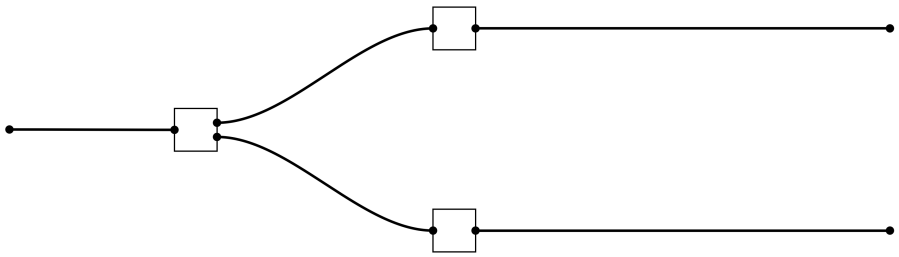

Figure 14. Example of the diagram.

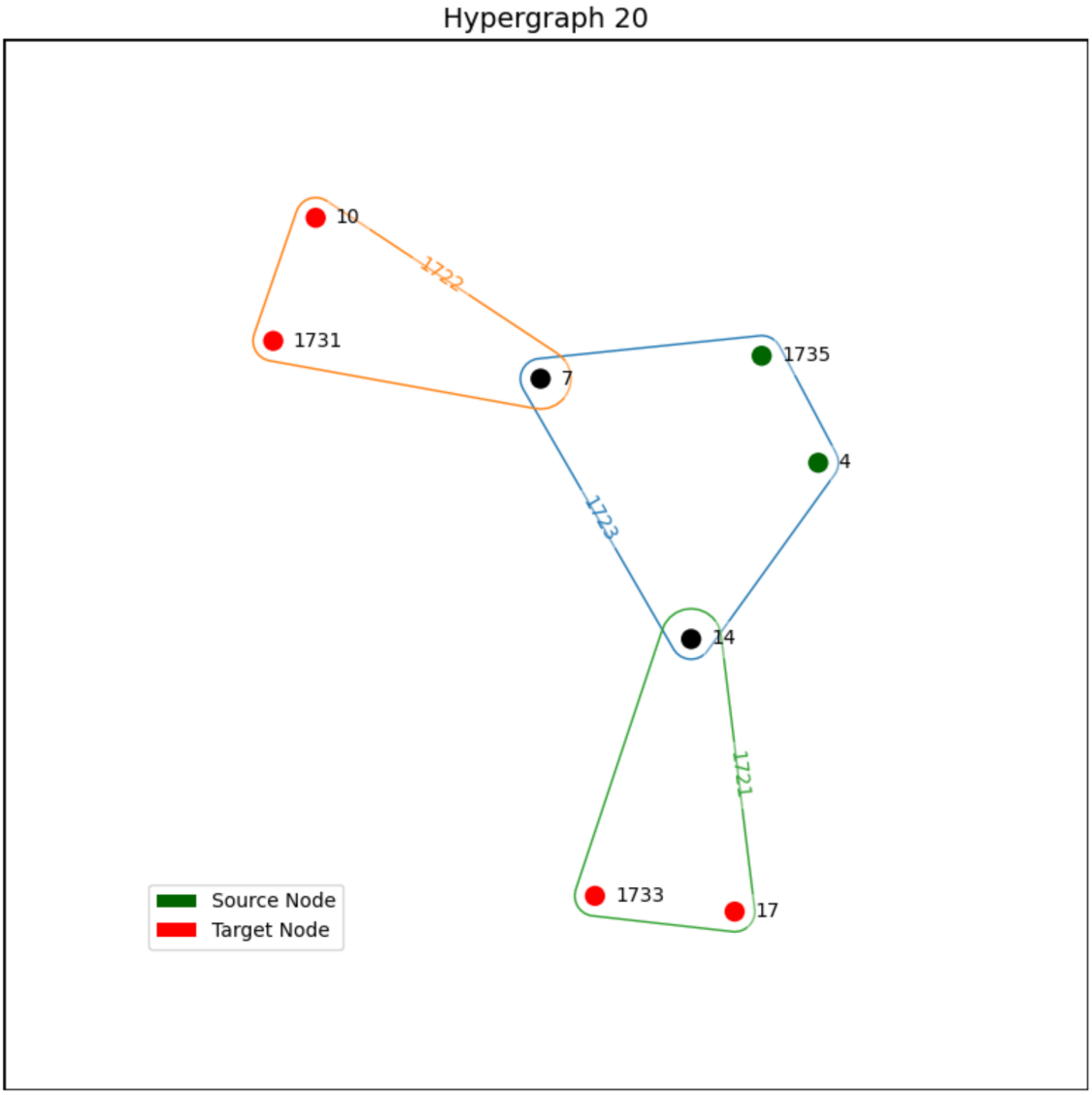

Figure 15. Hypergraph vizualization for the diagram above.

3.6.3 Implementation Features

Special attention was given to visualizing node roles. Nodes that serve as inputs to the hypergraph (source) are highlighted in green, while outputs (target) are shown in red. It allows users to determine the structure of the data flow quickly. If a node combines both roles or belongs to neither, it is displayed in a neutral color.

3.6.4 Limitations

Currently, the visualization is focused on displaying structure rather than the semantics of computations. This means that users can visually see how nodes and hyperedges are connected, but elements such as function names or data values are not displayed.

Interactivity is also lacking: the user cannot click on elements or modify them using the graphical interface—the visualization is purely for overview purposes.

3.6.5 Possible Improvements

The following areas of development are possible in the future:

- Adding interactivity (highlighting elements, tooltips, and the ability to move nodes manually);
- Displaying additional information about nodes and hyperedges (e.g., data types, number of inputs/outputs);
- Exporting visualizations as images or PDFs for reporting purposes.

4 Testing, results, and further development opportunities

4.1 Validation of results

In order to validate accomplished work, two approaches were used.

Firstly, a big number of unit tests was developed and implemented to ensure that the underlying logic is correct and operates according to the theory. Unit tests were used for the functionality that is hard to test using the program interface, mainly hypergraphs logic and functions related to them. All main aspects of the features were tested and checked to meet the theory.

Secondly, were developed three test scenarios that display the possible use cases for added features. Each scenario validates multiple features on different levels.

4.1.1 Scenario 1 - average user

Scenario 1 demonstrates how an average user might use Ivaldi.

Josh does not have good skills in creating string diagrams. His aim is to try creating a diagram and understand how it works. As his objective he decides to choose a morning routine of coffee preparation. This procedure consists of the following steps:

1. take a cup;
2. add milk;
3. add coffee;
4. mix the drink;
5. serve the coffee to his girlfriend.

Note that the order of the second and the third steps can be changed because it does not affect the final result.

Josh starts the program, chooses to create a new diagram and sees the empty canvas. At

the beginning, he creates a box for every previously examined step and labels them. Also, he puts them in the correct order for convenience (see Fig. 16).

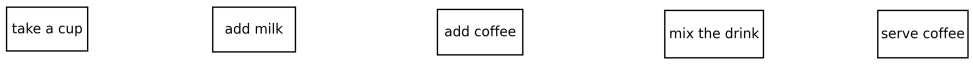

Figure 16. Main diagram with boxes for every step in the coffee preparation proccess.

Next, Josh decides to add inputs and outputs for the diagram. His process has only one starting point (take a cup) and one exit point (serve the coffee to his girlfriend), so one input and one output are added to the diagram. Also, Josh adds input and output for every box (see Fig. 17).

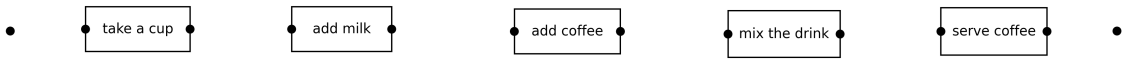

Figure 17. Main diagram with diagram input and output and connections for every box.

As the final step, Josh connects every element of the diagram with the wires in the correct order (see Fig. 18).

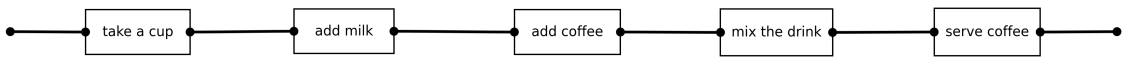

Figure 18. Main diagram of the coffee preparation process connected linearly.

After careful examination of the result, Josh decides to change the connection of the second (add milk) and third (add coffee) steps with the rest of the diagram. There is no difference in which order these steps will be done, so to demonstrate this fact, he connects them in

parallel with the previous and next steps. Also, adding supplementary input for the next step was needed. The result is a connected and valid string diagram that demonstrates the process of the coffee preparation (see Fig. 19).

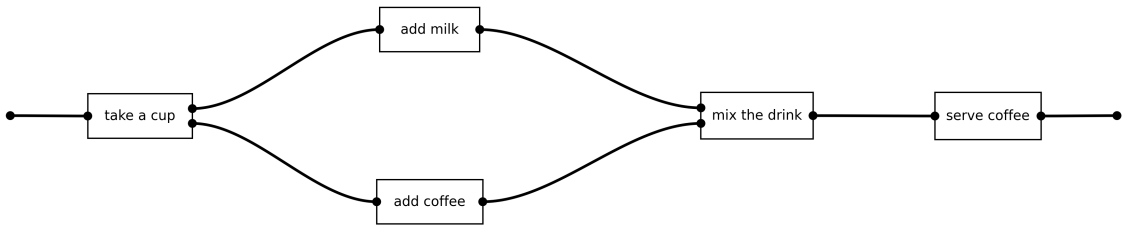

Figure 19. Main diagram of the coffee preparation process connected in parallel.

Josh saves the new project to the file in his computer and closes the Ivaldi. After a few days, he wants to see the diagram again, so he opens the redactor and loads the diagram from the previously saved file. As a result, he can see exactly the same diagram that preserves every detail about it.

In conclusion, this scenario tested the following features:

- Diagram callback;
- Importing/Exporting.

The diagram callback is tested with every manipulation of the diagram. If during the diagram creation process no errors or inconsistencies appeared, it means that this feature worked correctly. Importing and exporting features were tested when the user saved the project to a file and then loaded it. The identity of the diagrams shows that this logic worked as intended.

4.1.2 Scenario 2 - programmer

Scenario 2 demonstrates how a software engineer might use Ivaldi.

Emma studies informatics at university. She has good skills in programming and understands the main concepts of string diagrams. While working on her personal project, Emma wants to review previously written algorithms, however, she does not remember how they work.

That is why she decided to use Ivaldi to help her revise the code logic.

Her algorithm consists of two Python files: “cost.py” and “calculate.py.” The first contains two functions that calculate the total cost of the fuel and round it. The second file calculates monthly distance and fuel consumption based on some inputs. Also, the second file contains the main execution block with all cross-function calculations. The inputs of the algorithm are set in variables in the main execution block.

File “cost.py” (see Fig. 20):

```
def calculate_total_cost(fuel_liters, price_per_liter):
    total = fuel_liters * price_per_liter
    return total

def round_cost(cost):
    divided = cost / 10
    rounded = round(divided)
    result = rounded * 10
    return result
```

Figure 20. First input file code.

File “calculate.py” (see Fig. 21):

```
from example_python_code.petrol.cost import (
    calculate_total_cost, round_cost)

def calculate_monthly_distance(daily_distance):
    days_in_month = 30
    monthly_distance = daily_distance * days_in_month
    return monthly_distance * 1.01

def calculate_fuel_needed(distance, consumption_per_100km):
    units = distance / 100
    fuel_needed = units * consumption_per_100km
    return fuel_needed

if __name__ == "__main__":
    distance_per_day = 30
    fuel_consumption = 7.5
    fuel_price = 50

    monthly_distance = calculate_monthly_distance(
        distance_per_day)
    fuel_needed = calculate_fuel_needed(monthly_distance,
                                        fuel_consumption)
    total_cost = calculate_total_cost(fuel_needed,
                                      fuel_price)
    rounded_cost = round_cost(total_cost)
    print(rounded_cost)
```

Figure 21. Second file input code.

To remind herself what is the flow of this algorithm, Emma opens the Ivaldi and selects to import a diagram from these two files. When the redactor inquires if she wants to make a deep diagram generation, she chooses this option. The result is the main diagram with subdiagrams for every declared function (see Fig. 22, Fig. 23, Fig. 24, Fig. 25, Fig. 26).

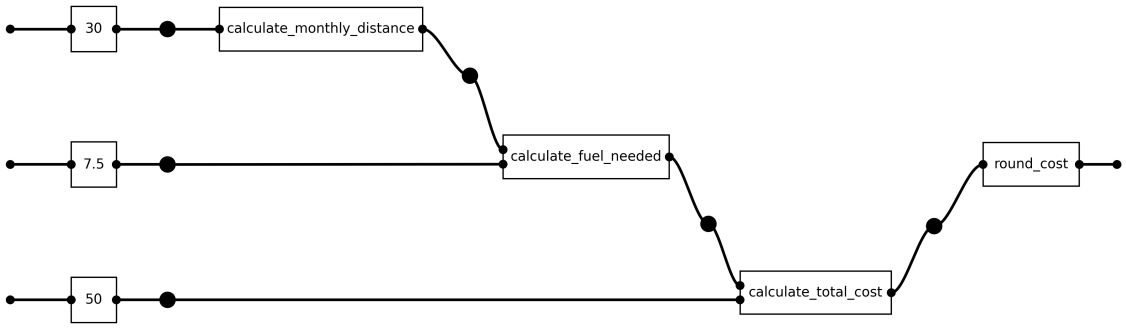

Figure 22. Main diagram generated from the fuel consumption algorithm.

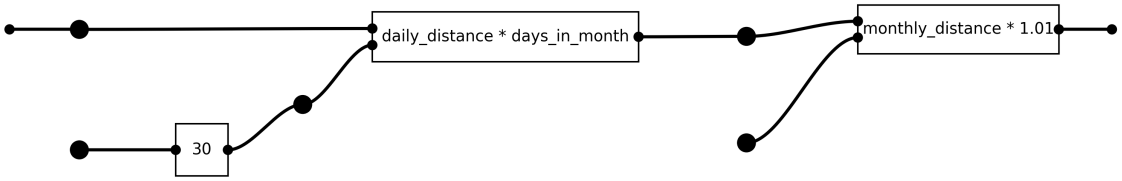

Figure 23. Subdiagram generated from the "calculate_monthly_distance" function.

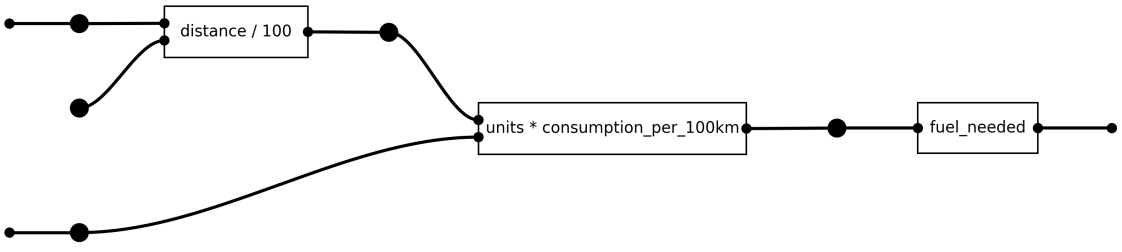

Figure 24. Subdiagram generated from the "calculate_fuel_needed" function.

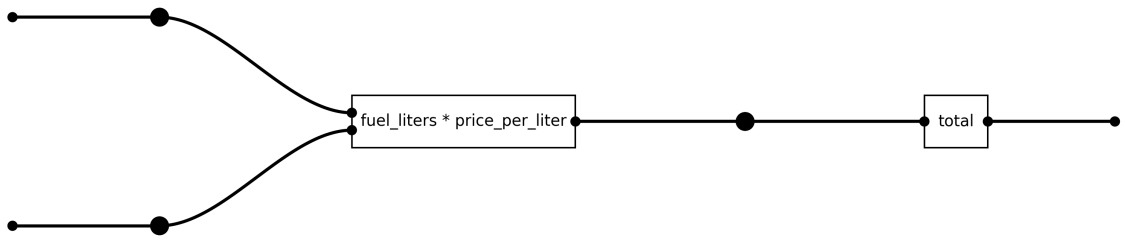

Figure 25. Subdiagram generated from the "calculate_total_cost" function.

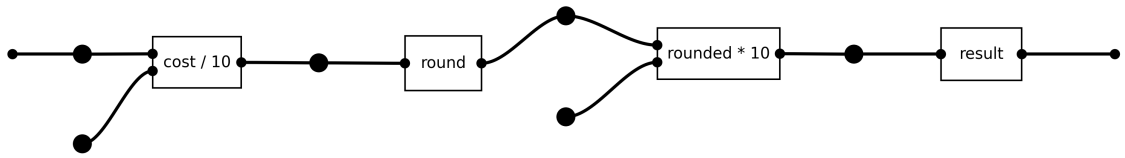

Figure 26. Subdiagram generated from the "round_cost" function.

After examining the diagrams, Emma gained a clearer understanding of the underlying structure and functionality of the original code.

After a while, Emma needs to make some changes to the code. Instead of opening the text editor, she decides to work with the string diagram algorithm representation in the Ivaldi. She opens the editor one more time and imports the same files into the program. This time, when she was prompted to select diagram generation mode, she chose shallow diagram generation, which does not create any subdiagrams for the declared functions. As a result, she sees the created diagram, which is exactly the same as the previous result (Figure 22). The only difference is that no subdiagrams were generated.

Emma reviews the code (see Fig. 27) contained in the box with the label “calculate_-monthly_distance.”

```
def calculate_monthly_distance(daily_distance):
    days_in_month = 30
    monthly_distance = daily_distance * days_in_month
    return monthly_distance * 1.01
```

Figure 27. Code in the box.

This time she needs to calculate fuel consumption for March, which includes 31 days. However, the previous algorithm only considered 30 days in a month, which is not applicable to her request. She changes the constant to a new value, and saves the updated code (see Fig. 28).

```
def calculate_monthly_distance(daily_distance):
    days_in_month = 31
    monthly_distance = daily_distance * days_in_month
    return monthly_distance * 1.01
```

Figure 28. Updated code.

Eventually, Emma generates the final code from the diagram using the corrected algorithm (see Fig. 29).

```
def _fun_7_5_MphKCmNp3R_6(var_7uOuccXxKE):
    return var_7uOuccXxKE

def fun_50_brmmVRZIu0_4(var_iLllpp0oAq):
    return var_iLllpp0oAq

def calculate_fuel_needed_0(distance, consumption_per_100km):
    units = distance / 100
    fuel_needed = units * consumption_per_100km
    return fuel_needed

def calculate_monthly_distance_2(daily_distance):
    days_in_month = 30
    monthly_distance = daily_distance * days_in_month
    return monthly_distance * 1.01

def fun_30_ZuyoPUbzJY_1(var_A3Bfi68aUH):
    return var_A3Bfi68aUH

def round_cost_3(cost):
    divided = cost / 10
    rounded = round(divided)
    result = rounded * 10
    return result

def calculate_total_cost_5(fuel_liters, price_per_liter):
    total = fuel_liters * price_per_liter
    return total

def main_0(input_0, input_1, input_2):
    res_0 = fun_50_brmmVRZIu0_4(input_2)
    res_1 = _fun_7_5_MphKCmNp3R_6(input_1)
    res_2 = fun_30_ZuyoPUbzJY_1(input_0)
    res_3 = calculate_monthly_distance_2(res_2)
    res_4 = calculate_fuel_needed_0(res_3, res_1)
    res_5 = calculate_total_cost_5(res_4, res_0)
    res_6 = round_cost_3(res_5)
    return res_6
```

Figure 29. Generated code.

Emma uses the generated code to produce the number which shows the fuel consumption in March. However, Emma can see that the new code does not look the same as the initial algorithm. So she ensures to make the necessary changes to the initial code and executes it. The results of both algorithms are exactly the same.

In conclusion, this scenario tested the following features:

- Diagram generation from code;
- Code generation from the diagram.

The diagram generation from code was tested two times for accuracy when the user imported the source code and generated the diagram from it. The first time the diagram was generated shallowly (without any subdiagrams), and the second time it was generated deeply (with subdiagrams). The accuracy of the generated diagram was checked by logically comparing the final diagram and the initial code. The code generation from the diagram was tested when the user generated the code based on the diagram. To ensure that the generated algorithm gives the same result and the source code, both algorithms were executed with the same input data. As a result, the output number of both algorithms was exactly the same, which means that diagram generation from code and code generation from the diagram operate correctly.

4.1.3 Scenario 3 - mathematician

Scenario 3 demonstrates how a mathematician might use Ivaldi.

Andrew, who is a mathematician, has a problem to find the altitude BH of the right triangle ABC (see Fig. 30), provided that the sides AB = 3 and BC = 4 are known. In order to solve the problem efficiently, Andrew decided to use the Ivaldi editor so that he would not make a mistake, and also later would be able to share the solution with his friend Nikita.

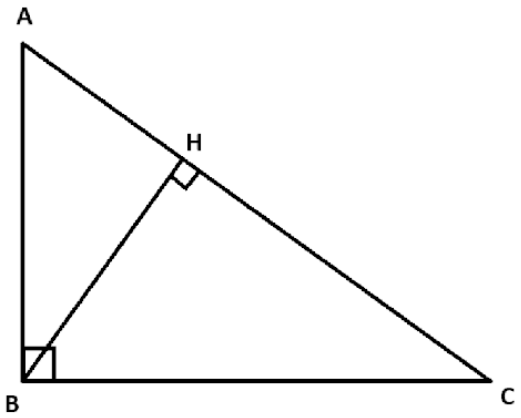

Figure 30. Triangle ABC.

Andrew approached the problem from the opposite direction, so he started by expressing the altitude, for this purpose he constructed a diagram with finding the area of a triangle using the altitude (see Fig. 31), where the first input of the diagram is the length of the base AC and the second input is the altitude BH.

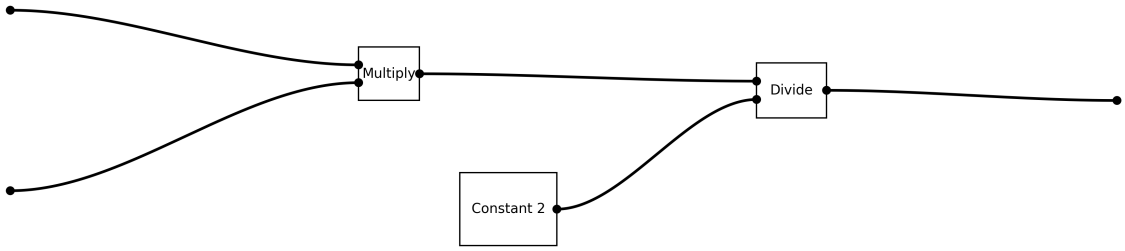

Figure 31. Diagram for triangle area.

It is quite easy to derive the altitude - it is only necessary to build an inverse diagram, where the input will be the output of the diagram in Fig. 31 i.e. the area of the triangle, and another input will be the known base AC, also since we do the inverse operation, the functions will also be replaced, division by multiplication, and multiplication by division. The result is the diagram in Fig. 32.

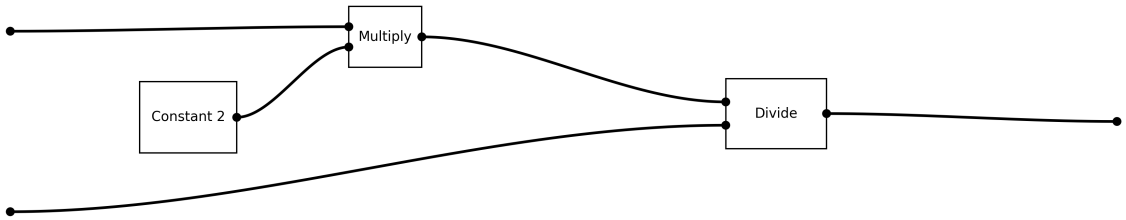

Figure 32. Diagram for triangle height.

Thanks to editor Ivaldi, Andrew was able to visually express the height of a triangle using the general triangle formula for finding the area. He also knows the formula for finding the area of a right triangle by multiplying the cathetes AB and BC, and dividing the result by 2. Andrew decided to replace the first input of the diagram in Fig. 32 (which is the area of the triangle) with this formula in Fig. 33.

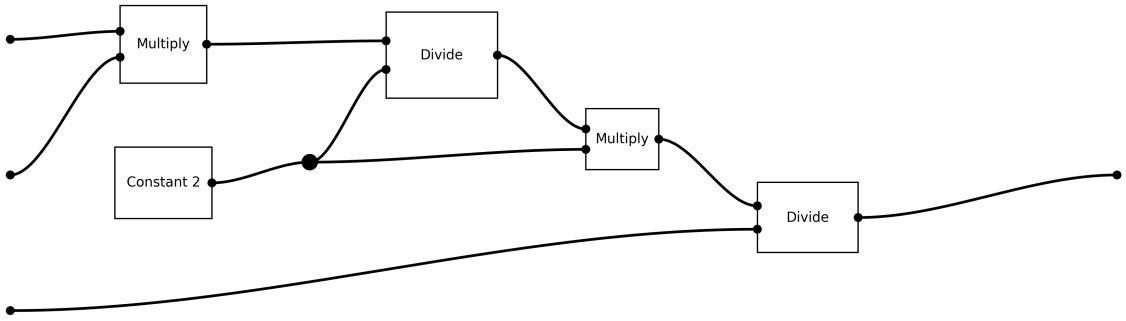

Figure 33. Extended diagram for triangle height.

As a result, the diagram has three inputs - cathetus AB, cathetus BC and base (aka hypotenuse) AC, the output of the diagram is still height BH. The diagram also clearly

shows that the constant 2 no longer applicable, because algorithm first divide the result of multiplying the cathetes by the constant 2, and then multiply by 2 again. Noticing it, Andrew immediately simplified the diagram, removed the unnecessary constant and the operation of multiplication and division. Therefore, he has got the following diagram (see Fig. 34).

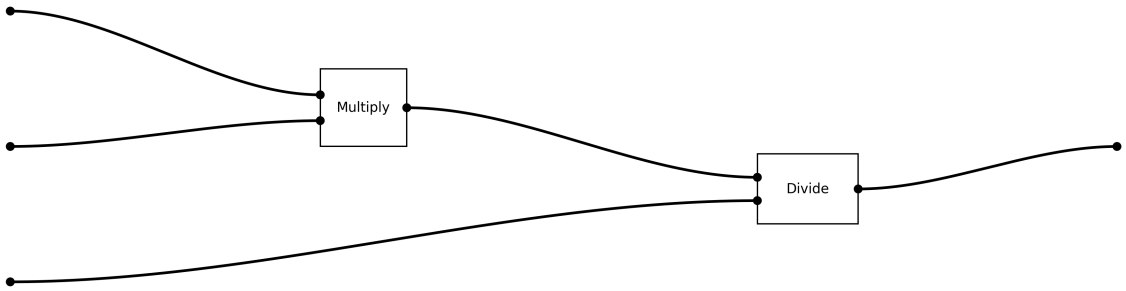

Figure 34. Simplified diagram for triangle height.

Andrew realized that to find the height, he needs to multiply the cathetes AB and BC and divide them by the base (hypotenuse) AC. Only hypotenuse AC is unknown, and then Andrew remembered the Pythagorean theorem and immediately drew it on the diagram (see Fig. 35).

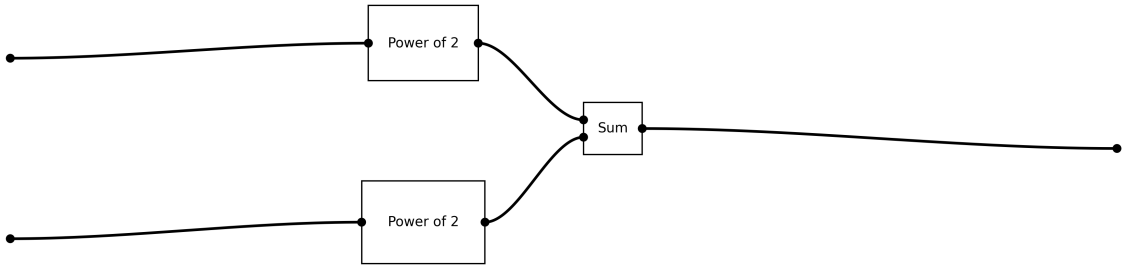

Figure 35. Diagram for Pyrhagorean theorem.

Input - cathetus AB and cathe BC - output hypotenuse squared. Andrew decided to save this diagram and import it into the main diagram as a sub-diagram. That is, the Pythagoras theorem will be conveniently reduced to just one box, and if needed, it is possible to view how it works. But, since the formula needs a base (hypotenuse) in the first degree - Andrew took the square root of the result of the Pythagoras theorem formula. Moreover, Andrew removed the third input of the diagram, because now the base (hypotenuse) can be calculated using only the first two inputs (see Fig. 36).

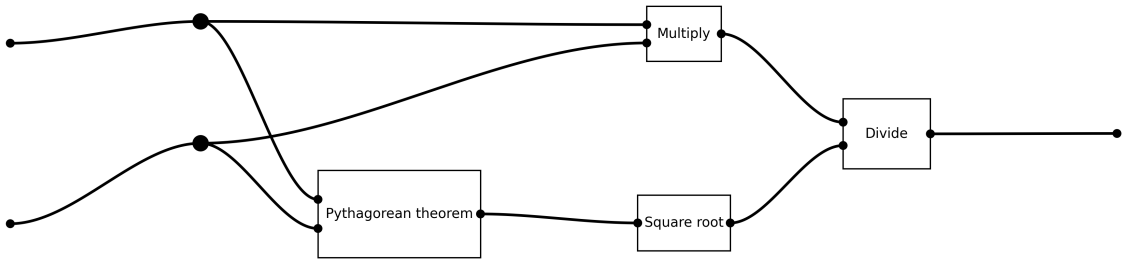

Figure 36. Final diagram for triangle height.

As a result, Andrew simply and efficiently solved the problem without writing it down in his notebook. Now he can also share the solution with Nikita. In addition, Andrew decided to check how his diagram is represented as a hypergraph (see Fig. 37).

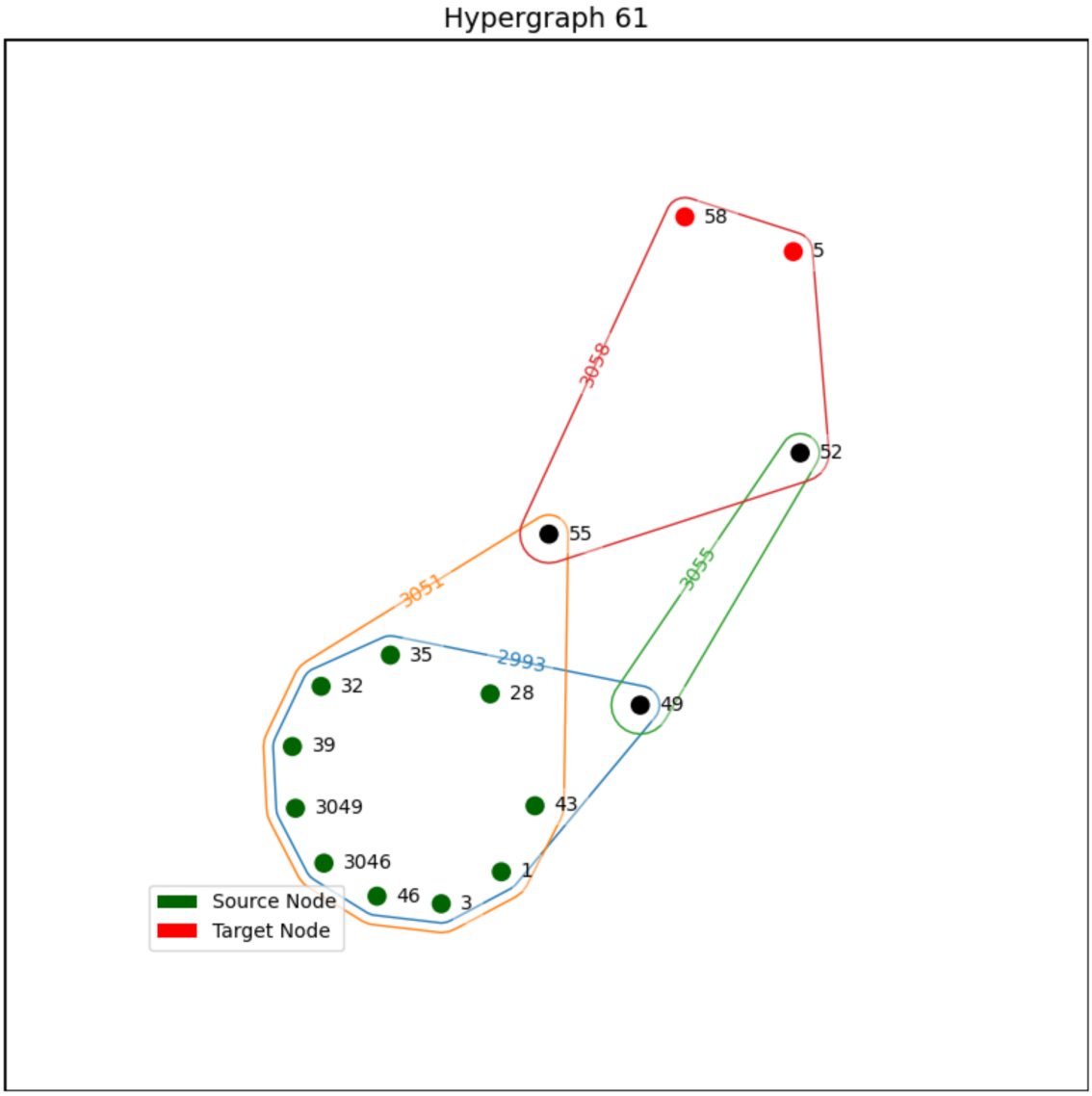

Figure 37. Main diagram‘s hypergraph visualization.

While the hypergraph accurately replicates the diagram, this visualization alone is insufficient; it is necessary to examine the structure of the hypergraph corresponding to the subdiagram of the Pythagorean Theorem (see Fig. 38).

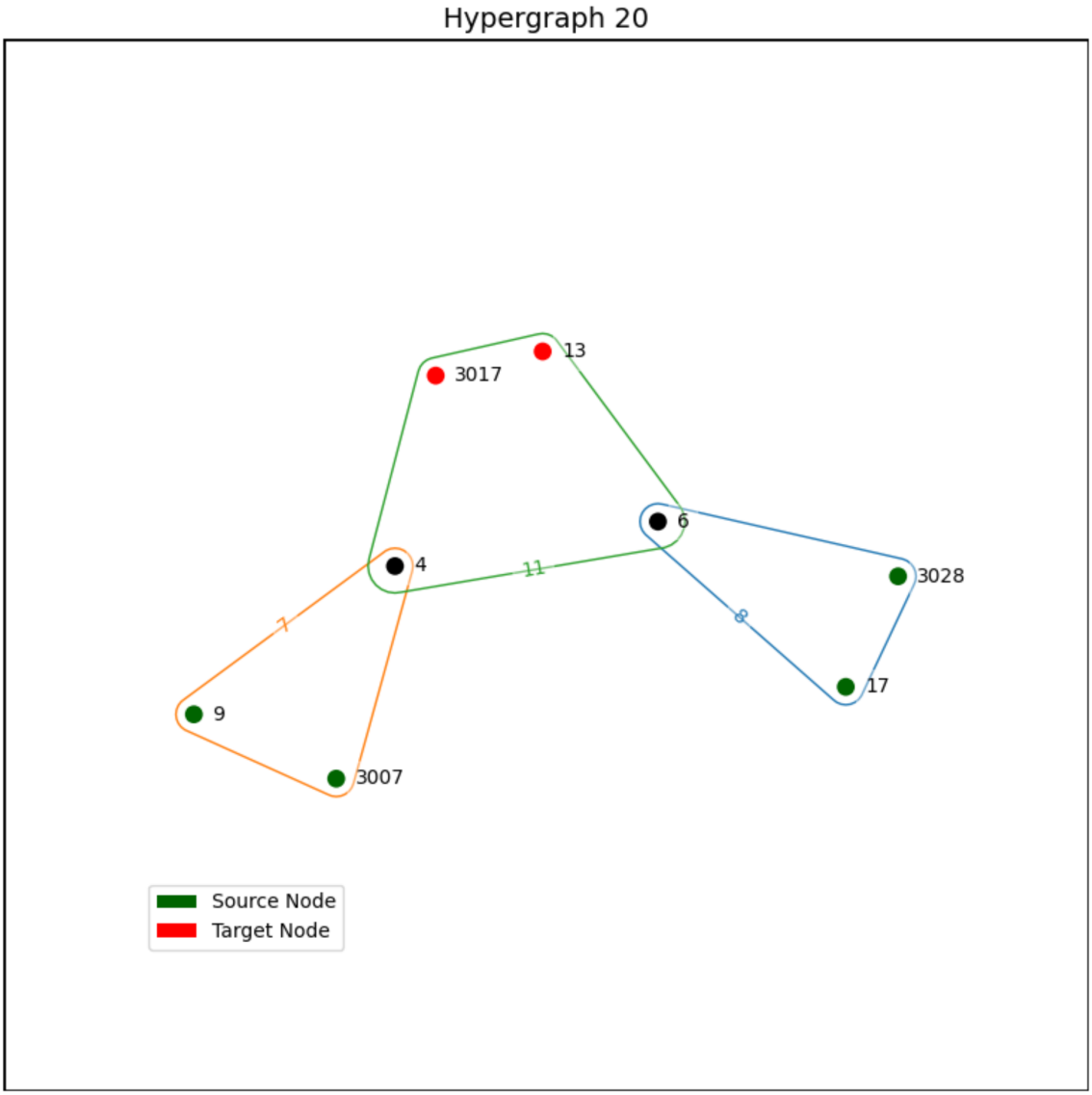

Figure 38. Phytagorean diagram‘s hypergraph visualization.

No issues have been identified here either. Since the hypergraphs are correctly constructed, Andrew decided to generate a code to solve his problem, for this purpose he assigned a mathematical function to each box and pressed “generate code”. His code produced accurate and expected results (see Fig. 39), therefore, Andrew can use this code to solve similar problems.

```
def Divide_1(a, b):
    return a / b


def Sum_2(a, b):
    return a + b


def Multiply_4(a, b):
    return a * b


def Square_root_3(a):
    return a ** 0.5


def Power_of_2_0(a):
    return a ** 2


def main_0(input_0, input_1):
    res_0 = Power_of_2_0(input_0)
    res_1 = Power_of_2_0(input_1)
    res_2 = Sum_2(res_1, res_0)
    res_3 = Multiply_4(input_0, input_1)
    res_4 = Square_root_3(res_2)
    res_5 = Divide_1(res_4, res_3)
    return res_5
```

Figure 39. Generated code for math problem.

In conclusion, this scenario tested the following features:

- Hypergraph structure;
- Code generation from the diagram;
- Hypergraph visualization.

In the final diagram, the hypergraph structure was tested using visualisation. Also, due to

the fact that code generation works using a hypergraph and the generated code is correct, it also confirms the right implementation of the hypergraph.

4.2 Authors' assessment of the project and development work

Reviewing the project, the authors assess accomplished work differently. From the beginning, project development was not a trivial challenge. The initial code base was rather large and difficult to understand. There was no documentation or any substantial comments for the code. That is the reason why the first month of development was extra challenging, not so productive for new features. However, after overcoming this obstacle, work process started to accelerate. In the end, the authors were well-versed in the project codebase.

Another challenge was that fact that Python programming language was used for the project. Python is a dynamically typed language. That is why in some cases, during the project development process, it was hard to comprehend the type of variables in code. This problem was minimized by using Python type hints functionality in the newly written code[21].

Additionally, learning a new theory about string diagrams was a big task. In order to implement any feature, the authors spent a reasonable time understanding the underlying concepts of the theory. Subsequently, a significant amount of time was devoted to applying the acquired knowledge to the program.

Most of the developed features are currently in a good state. These features can be used in the production environment with some limitations. They provide valuable opportunities to develop an understanding of string diagram concepts, to learn how to work with them, and to visualize programs within the string diagram canvas. Naturally, there is a place to extend the functionality of the features, make them more accessible for users, and work on documentation for future development. However, within the available timeframe, the accomplished work is at a good level.

If the authors started development again from the beginning, they would do multiple things differently. Firstly, they would focus more on the theoretical aspects of the string diagrams. The lack of knowledge has resulted in developing some features incorrectly and resolving the errors in their logic afterwards. Secondly, certain inconsistencies in team communication resulted in a lack of coordination between some feature aspects, which

subsequently required additional effort to resolve. Finally, the assessment and validation of the final result could have been broader. For instance, giving the program to professionals from different fields for practical use and receiving real feedback about the flows and strengths of the redactor.

4.3 Further development opportunities

Despite the fact that all the goals of the Bachelor's thesis were achieved, there are plenty of development opportunities.

Firstly, more unit tests can be written for the application. Although the majority of features are tested to a certain degree, the code coverage is not comprehensive. Some of the edge cases are not validated, and the overall tests number could have been bigger.

Secondly, the documentation of the project can be expanded. The patterns used in the program and other technical solutions can also be added to the documentation. For future developers, this information might be essential to ease the learning curve of the project structure.

Thirdly, the efficiency of the hypergraph operations can be improved further, especially when reacting to the frontend operations. Currently, changes in the hypergraph's structure are triggered after every change in the diagram. This may be a problem if the program performs a large number of operations on the diagram in a short time, for example, importing a diagram from a file.

Finally, the opportunity to work with programming code in Ivaldi can be expanded. Support for the different programming languages can be implemented to make this functionality broader and more general. Also, more programming structures such as "classes" can be converted to a string diagram theoretical representation and implemented into the program. Additionally, this feature can be expanded to validate correspondence between the diagram and the inserted code to show warnings and errors if the code structure does not align with the diagram.

5 Summary

The main purpose of the work was to improve the efficiency and functionality of the Ivaldi backend by implementing the hypergraph structure and adding new functions, such as generating code from string diagrams and generating string diagrams from code. In addition, it was planned to expand the functionality of importing and exporting projects created in the Ivaldi.

The tasks were completed and tested. Tests were written to enhance the verification of the program's functionality. The documentation was created to support future development for new participants. A function for visualizing hypergraphs, developed based on string diagrams, was also implemented to facilitate the understanding and testing of hypergraphs.

In addition, most of the functions were analyzed by Pawel Maria Sobocinski and his research group at the Laboratory for Compositional Systems and Methods, which improved the accuracy of the implementation from a theoretical point of view.

The results of the work can be accessed on the GitHub repository. For more details, see Appendix 5 – Ivaldi GitHub Link.

References

[1] Pawel Sobociński. *Why string diagrams?* WWW. URL: https://graphicallinearalgebra.net/2017/04/24/why-string-diagrams/.

[2] Katie Howgate. *Hypergraphs – not just a cool name!* WWW. URL: https://www.lancaster.ac.uk/stor-i-student-sites/katie-howgate/2021/04/29/hypergraphs-not-just-a-cool-name/.

[3] Abdela Umer. *What is the main block in Python? `if __name__ == "__main__"`.* WWW. 2023. URL: https://medium.com/@adheremo65/what-is-the-main-block-in-python-if-name-main-d9f7410ef2f2.

[4] Fabio Zanasi Paweł Sobociński Paul Wilson. *Cartographer - a tool for string diagrammatic reasoning*. WWW. URL: https://cartographer.id/.

[5] Ross Street. *Categorical Structures*. WWW. URL: https://www.sciencedirect.com/science/article/abs/pii/S1570795496800192.

[6] JGraph Ltd. *The official blog of the draw.io project*. WWW. URL: https://www.drawio.com/blog.

[7] Python Software Foundation. *Python*. WWW. URL: https://www.python.org/.

[8] Python Software Foundation. *tkinter — Python interface to Tcl/Tk*. WWW. URL: https://docs.python.org/3/library/tkinter.html.

[9] Patrick Maupin. *astor 0.8.1*. WWW. URL: https://pypi.org/project/astor/.

[10] Battelle Memorial Institute. *HyperNetX (HNX)*. WWW. URL: https://hypernetx.readthedocs.io/en/latest/.

[11] *SOLID principle*. WWW. URL: https://coderspace.io/en/blog/what-is-solid-examples-of-solid-principles/.

[12] *Breadth First Search for a graph*. WWW. URL: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/.

[13] Dusko Pavlovic. *Programs as Diagrams - From Categorical Computability to Computable Categories*. WWW. 2023. URL: https://arxiv.org/pdf/2208.03817.

[14] Fabio Zanasi Robin Piedeleu. *An Introduction to String Diagrams for Computer Scientists*. WWW. 2023. URL: https://arxiv.org/pdf/2305.08768.

[15] BrowserStack. *Coding Standards and Best Practices to Follow*. WWW. 2024. URL: https://www.browserstack.com/guide/coding-standards-best-practices.

[16] Hideo Hattori. *autopep8*. WWW. URL: https://pypi.org/project/autopep8/.

[17] Shanshan. *Intro to Python ast Module*. WWW. URL: https://medium.com/@wshanshan/intro-to-python-ast-module-bbd22cd505f7.

[18] Kabeer Sahib. *Components of a Python Program*. WWW. URL: https://qissba.com/python-program-components/.

[19] Abhishek Jaiswal. *JSON: Introduction, Benefits, Applications, and Drawbacks*. WWW. 2022. URL: https://www.turing.com/kb/what-is-json.

[20] Taku A Tokuyasu Jesse Paquette. *Hypergraph visualization and enrichment statistics: how the EGAN paradigm facilitates organic discovery from Big Data*. WWW. URL: https://www.researchgate.net/publication/228792268_Hypergraph_visualization_and_enrichment_statistics_how_the_EGAN_paradigm_facilitates_organic_discovery_from_Big_Data.

[21] Jukka Lehtosalo. *Type hints cheat sheet*. WWW. 2025. URL: https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

We Nikita Semõnin, Andrei Mjastsov and Timofei Beresnjev

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for our thesis "String Diagram Editor Backend Structure Development and Hypergraph Interpretation for Implementing Domain Specific Programming Languages", supervised by Pawel Maria Sobocinski and Anton Osvald Kuusk
    1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
    1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright
2. We are aware that the authors also retain the rights specified in clause 1 of the nonexclusive licence.
3. We confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

04.06.2025

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive licence shall not be valid for the period.

Appendix 2 – Ivaldi UI. Diagram with subdiagram example

Example of the diagram in Ivaldi (see Fig. 40) with a box named "sub1" that contains a subdiagram (see Fig. 41).

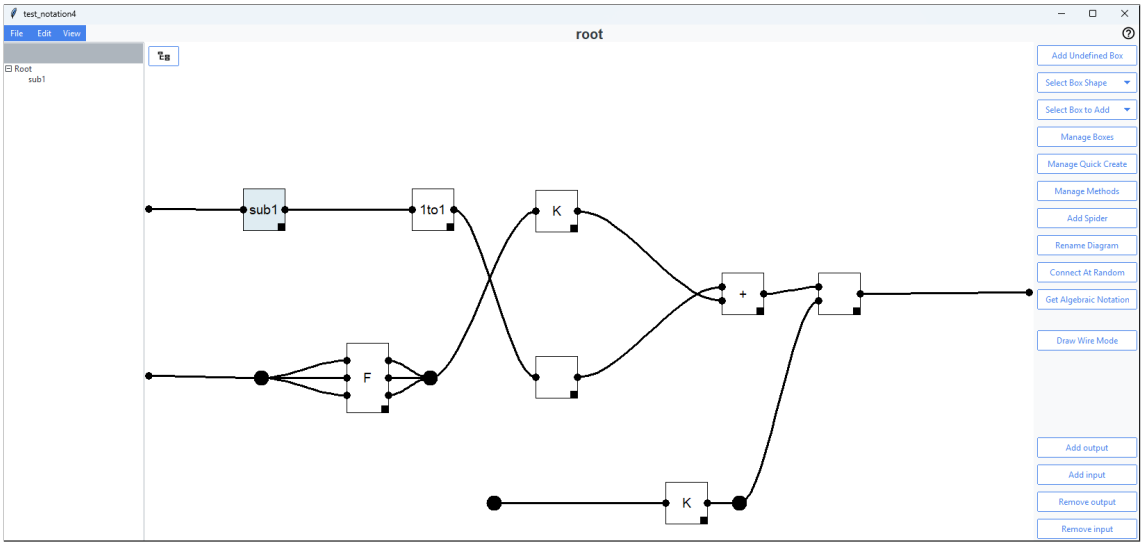

Figure 40. Diagram with subdiagram in Ivaldi.

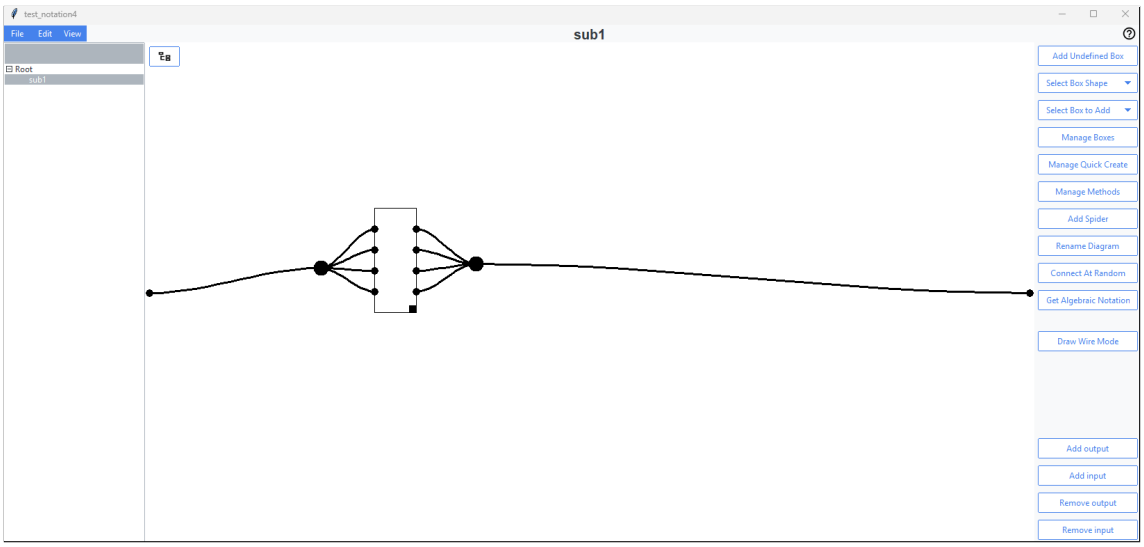

Figure 41. Subdiagram.

Appendix 3 – Ivaldi UI. Diagram generation from code

Example of importing a Python file into the diagram. In figure 42, the message box that appears when importing a Python file. In the next figure 43, the imported Python file can be seen, and the ‘deep’ mode was used. There are three functions (see Fig. 44, 45, 46) on the main diagram, and on the left side, three sub-diagrams can be seen, which indicate how the functions from the main diagram are implemented.

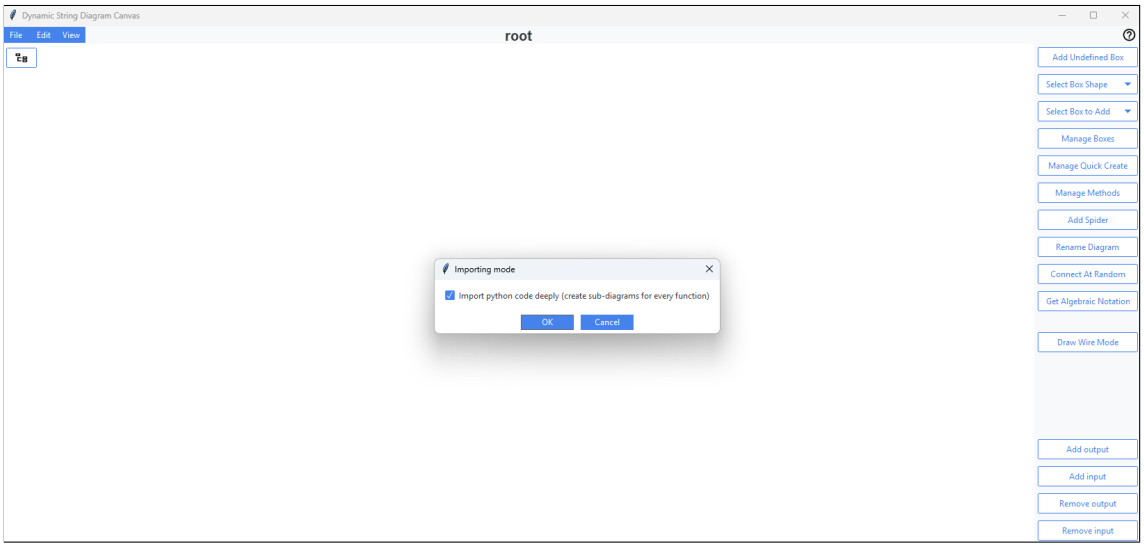

Figure 42. Message box with question, choose import mode(deep or shallow).

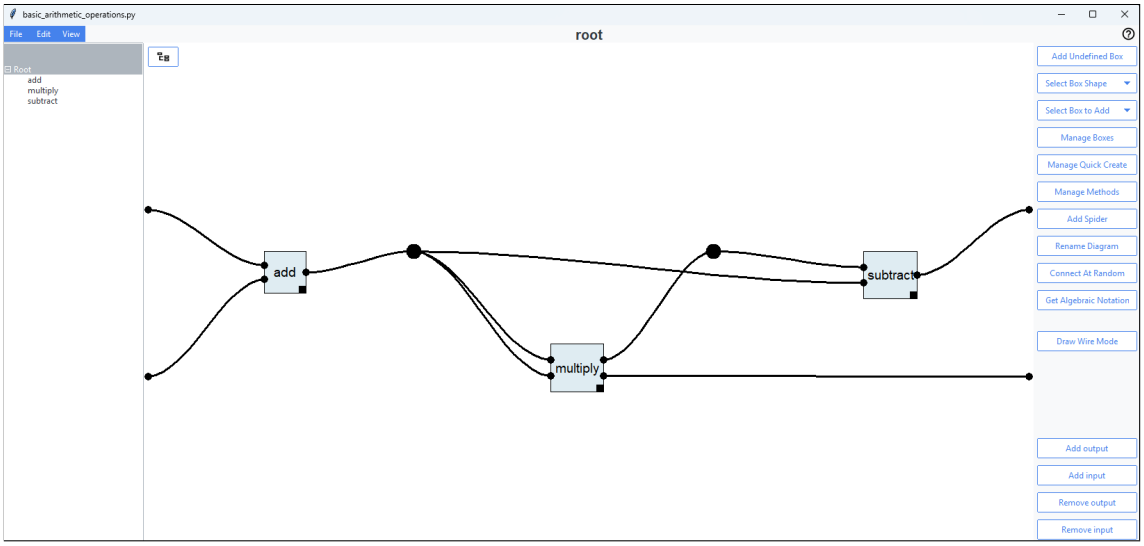

Figure 43. Generated diagram using deep importing.

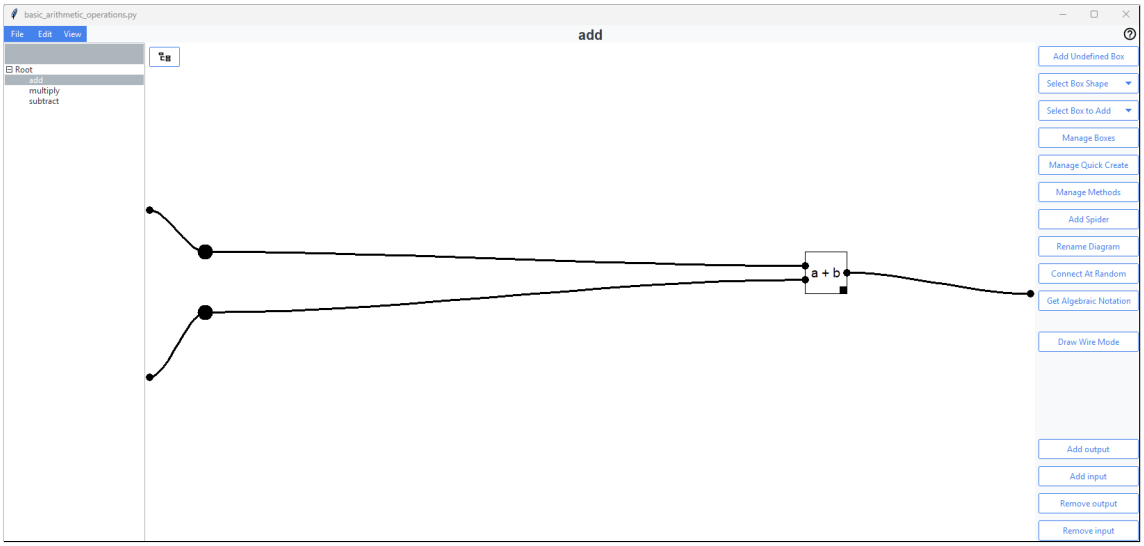

Figure 44. Subdiagram "add".

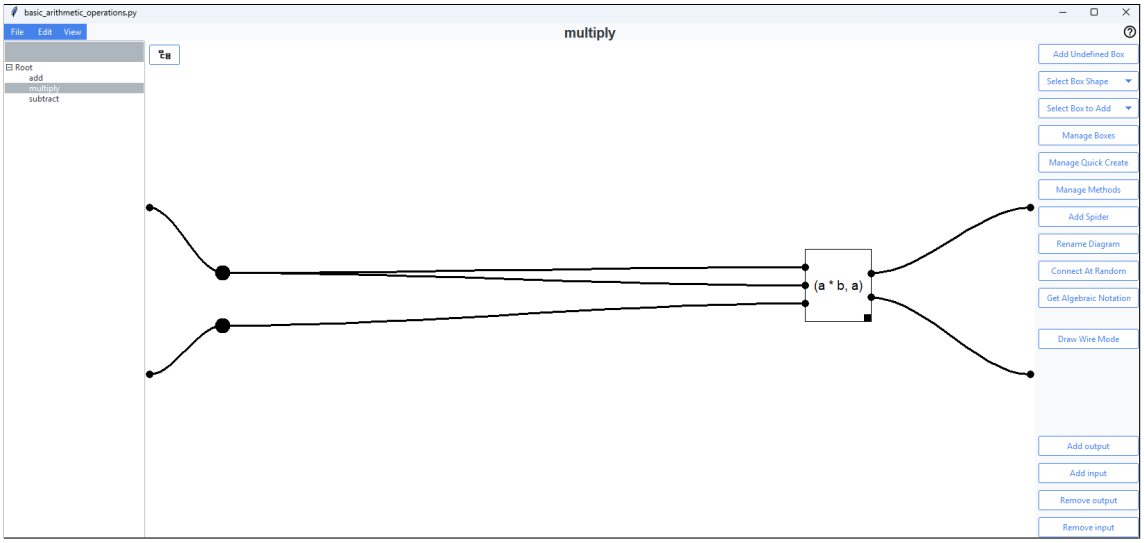

Figure 45. Subdiagram "multiply".

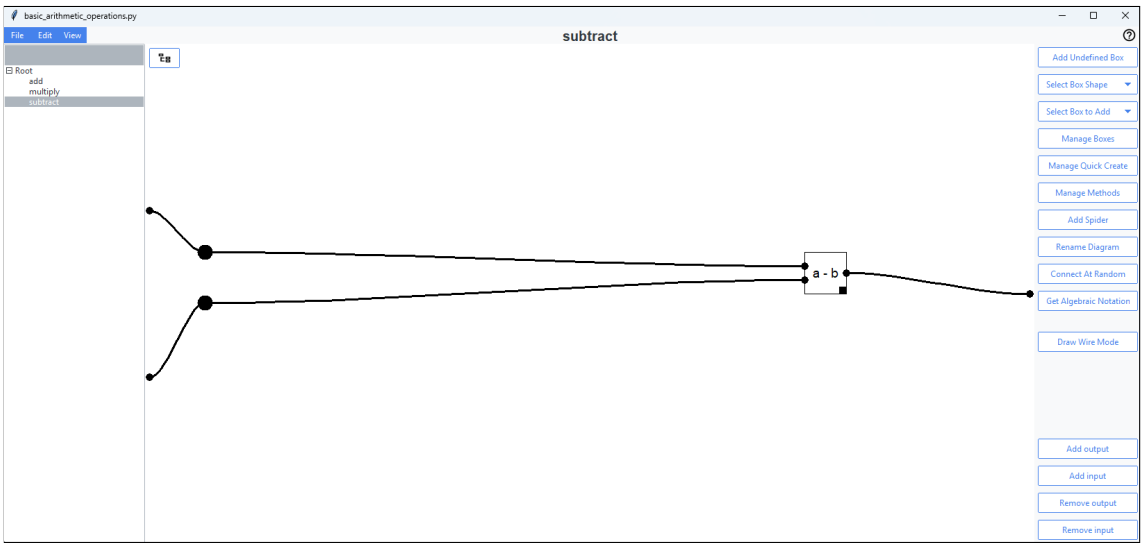

Figure 46. Subdiagram "subtract".

Appendix 4 – Ivaldi UI. Code generation from diagram

Figure 47 shows how to generate code from the diagram.

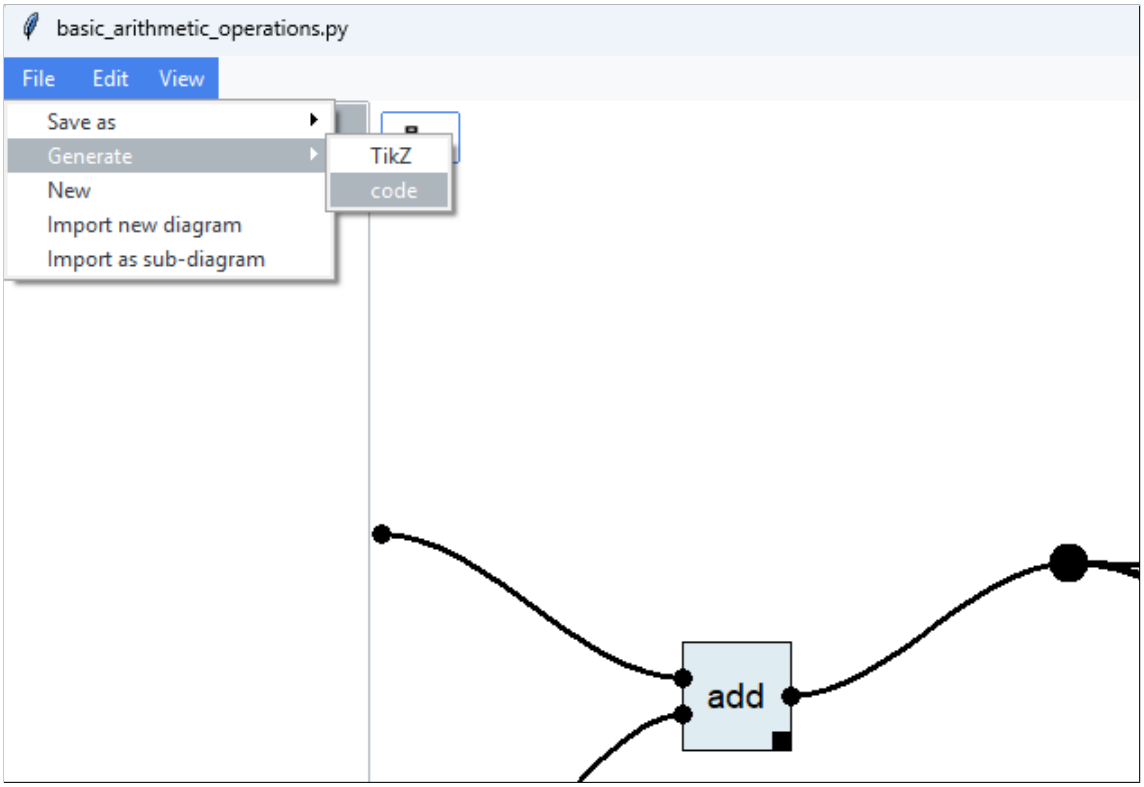

Figure 47. How to generate code from diagram.

Figure 48 displays the code editor interface in Ivaldi.

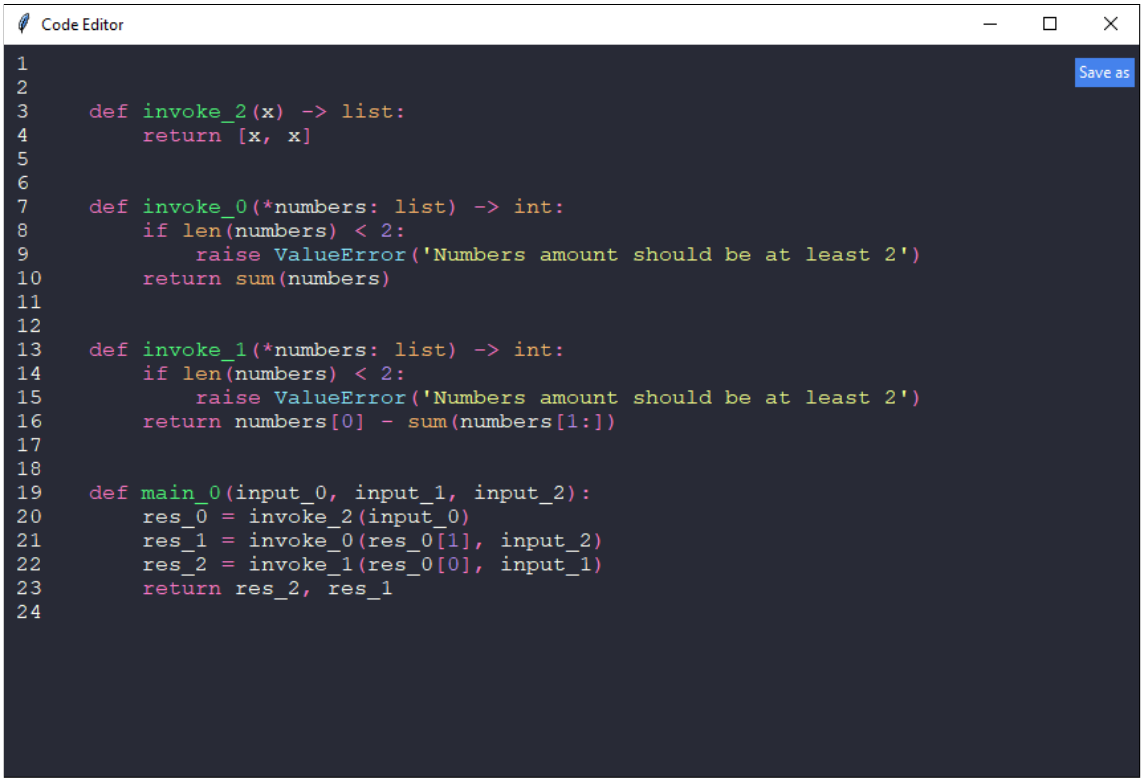

Figure 48. Ivaldi code editor UI.

Appendix 5 – Ivaldi GitHub Link

Link to the GitHub (see Fig. 49) of the application:

https://github.com/Taltech-bachelor-thesis/Ivaldi

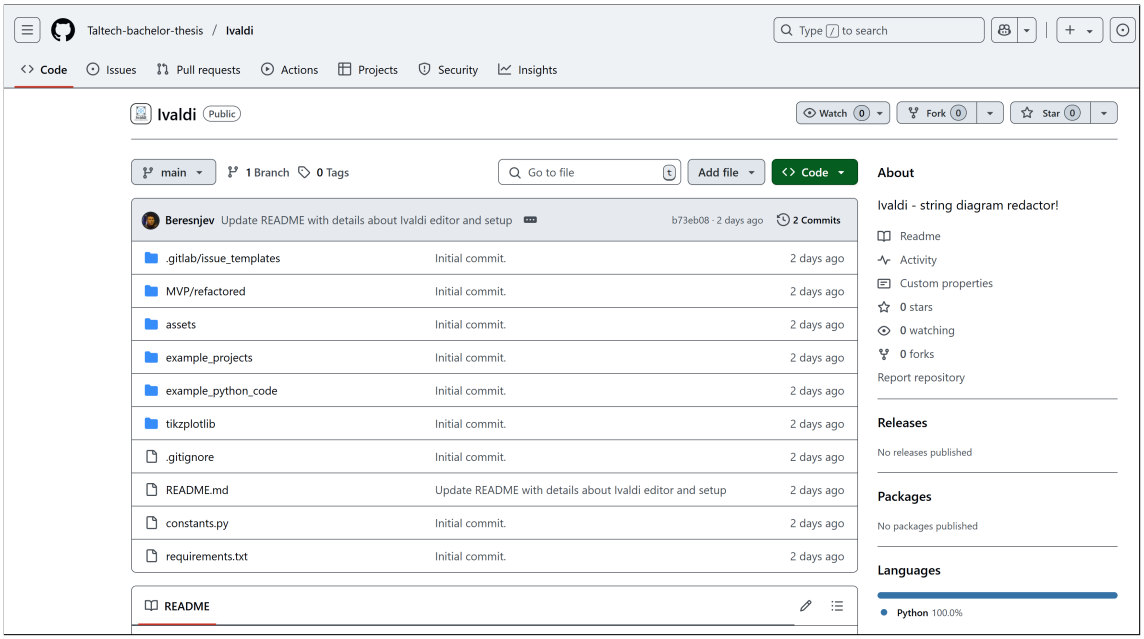

Figure 49. Editor‘s Ivaldi GitHub repository.