**Frege IDE**

Master Thesis

*(Please note, this is a work in progress, the text and the graphical layout will be updated)*

**Outline**

* Chapter 1: Introduction
  + Motivation / Introduction *(what was already done, why this work even exists, how this relates to other IDEs, what is Frege)*
  + Goals *(What I strived to achieve in this work, what subset of features should make / made it here, why exactly this, how this relates to projectional IDEs – MPS, why choose a projectional IDE, what is MPS – very shortly)*
  + Outline *(what other chapters are here, what can be found there)*
* Chapter 2: Meta-programming system / JetBrains MPS
  + - *what it is, what it can do, links & references*
    - *difference between projectional and classic text-based IDEs, how this relates to MPS, what makes MPS different*
    - *examples (images), what it would be like to use MPS, so the reader has a clearer picture*
    - *working with MPS, structure, editor, constraints, behaviour, checking rules, typesystem, textgen – generic explanation + exemplar language (not necessarily tutorial, but the explanation should be clear and how it would be used)*
* Chapter 3: Frege
  + *what is Frege*
  + *Frege vs. Haskell*
  + *Frege language, examples, and what we decided to support in 'Frege-IDE'*
* Chapter 4: Frege in MPS
  + *Subset related to what we decided to support in “Frege-IDE”*
  + *Grammar transformation for MPS structure aspect, design of the Frege structure, how and why (analysis + design decisions)*
  + *Editor*
    - *“Visuals” (FunctionName FunctionParameters = Expression)*
    - *Editor transformation and substitution actions*
    - *Lacking documentation on the MPS part.*
  + *References, scopes*
    - *(Constraints + behaviour)*
  + *Typesystem*
    - *Expression evaluation*
    - *TBA*
* *Chapter 5: Evaluation*
  + *What we strived to achieve, again? Recapitulation?*
  + *User-friendliness of the “Frege-IDE”, against classical text-based IDEs, advantages and disadvantages*
    - *(Should be probably mentioned user's time investment into learning to use the IDE)*
  + *Limitations*
    - *Something couldn't be done easily, e.g. Enter key-press does not always create a new line, ambiguity*
    - *Not all transformations are possible, e.g. rewriting f x y = x + y to f , x, y :: Int -> Int -> Int (by removing = and adding , , ::) is basically impossible to cover – it is not a text editor*
    - *Intentions – for the example above intention is an option to cover such transformation*
    - *(Mention built-in JAVA-like language, that also has these limitations)*
  + *Experience*
    - *What I found MPS lacking about, flexibility, not much detailed documentation*
* *Conclusion*
  + *Are projectional IDEs good for functional languages?*
  + *Is “Frege-IDE” usable?Future work, possible extensions.*
* *References*
* *Appendix*
  + *Frege formal grammar*
  + *What can be found on the attached CD (source code, examples, MPS setup, …)*
  + *Examples*
  + *User manual for “Frege-IDE”*
* Chapter 1: Introduction
* Motivation / Introduction *(what was already done, why this work even exists, how this relates to other IDEs, what is Frege)*

(Classic intro,a lot of wiki references to say the obvious and well known.)

(what are IDES) (<https://en.wikipedia.org/wiki/Integrated_development_environment>) Integrated development environments (abbreviated IDEs) are a set of software applications that provide tools and facilities to computer programmers. They greatly ease the process of software development, providing features like intelligent code completion, syntax highlighting, build automation tools, debugger, etc. (…add anything??)

(what is Frege) (<https://en.wikipedia.org/wiki/Frege_(programming_language)> Frege, named after the German mathematician, ~~logician and philosopher~~ [Gottlob Frege](https://en.wikipedia.org/wiki/Gottlob_Frege), is a functional language heavily based on Haskell (add ref. link) for the Java virtual machine. It is considered a Haskell dialect, sometimes called “a Haskell for the JVM”. (…todo)

known IDEs

There are several IDEs for Haskell, not many specifically built to support Frege. Examples can be found at <https://wiki.haskell.org/IDEs>, most of them providing mainly syntax highlighting, macros and project management features, while some also provide more advanced features like code completion or type evaluation and inspection.

That being said, all of the notable IDEs are text-based, (add explanation?

See <https://en.wikipedia.org/wiki/Structure_editor>

…most source code editors are instead text editors with additional features such as syntax highlighting and code folding, rather than structure editors. The editors in some integrated development environments parse the source code and generate a parse tree, allowing the same analysis as by a structure editor, but the actual editing of the source code is generally done as raw text.

). In this work, however, we try to approach the topic from a different view and try to design a projectional editor (/IDE) specifically for the Frege programming language.

(what is projectional editor) quoting <https://en.wikipedia.org/wiki/Structure_editor>

A projectional editor (aka structure, structured editor), is a document editor that is cognizant of the document’s underlying structure. They are usually used to edit hierarchical or marked up text, computer programs, diagrams, and any other type of content with clear and well-defined structure. While for the most computer programs a conventional text-based IDE may be more suitable, for specific programming languages, especially DSL (<https://en.wikipedia.org/wiki/Domain-specific_language>) a projectional editor might prove to be a more effective tool.

<https://martinfowler.com/bliki/ProjectionalEditing.html> *- additional info sources*

In this work we want to design a projectional editor supporting a certain subset of the (features of) Frege programming language and examine what are the advantages and disadvantages of such approach over creating a conventional text-based IDE.

* Goals (What I strived to achieve in this work, what subset of features should make / made it here, why exactly this, how this relates to projectional IDEs – MPS, why choose a projectional IDE, what is MPS – very shortly)

As was already mentioned in this paper, in this work we design a projectional editor for a subset of (features of) Frege programming language.

We have chosen JetBrains® *(legality, trademarks, etc.? check)* MPS platform as an underlying tool for designing our IDE. (briefly - why:) MPS (standing for “Meta-programming system”) is an open source software solution allowing developers and language designers to create different kinds of projectional IDEs. It is a so called language workbench (A language workbench is a software development tool designed to define, reuse and compose domain-specific languages together with their integrated development environment. - <https://en.wikipedia.org/wiki/Language_workbench>)

It allows for creating both simple and complex languages, especially DSLs, while also allowing extending existing ones, when the languages available do not meet the needs of a developer. *(quoting MPSLW I.)* MPS has a large set of features allowing for designing editors which closely resemble those from conventional, text-based, IDEs.

Frege, based on Haskell language, has rather many syntactic (and semantic) constructs for this work to be able to include them all. We have therefore focused our attention only on the most important features worth examining, such as function declaration and definition, operators and (custom) datatypes. Our ideal IDE will have a user-friendly editor that should emulate normal text editing and writing code in the way that most Frege and Haskell developers are used to. This should be accompanied by a context help, sometimes referred to as ‘intellisense’, which would allow for referencing already defined functions, operators, variables, etc., in the corrects spots in the code. Last, but not least, we will strive for a type checker, which would be able to find small mistakes in the code, such as calling a function with illegal arguments, or evaluate type of an expression.

* *Outline (what other chapters are here, what can be found there)*

More about MPS tool is written in chapter 2, which examines what MPS is, what it can do and what are its limitations. The chapter examines a project structure in MPS, how to define an editor for a simple language and how to tackle certain problems.

Frege is described more in chapter 3, where we also take a look into what features we actually wanted to implement in our IDE.

Chapter 4 examines Frege grammar and how it relates to the features we want to support. We describe how the grammar should be transformed into concepts in MPS tool on the high-level. We provide a brief analysis and try to explain our decisions.

Chapter 5 is dedicated to the concrete work implementation. We show how exactly the grammar from chapter 4 was transformed into MPS concepts, how the editor was designed, how we tackled problems with references and context help. We conclude the chapter with type system, where we describe some of the more interesting algorithms used in the work, such as the one to evaluate type of expressions or inference types of function arguments, when provided with the function’s annotation.

We evaluate our decisions in chapter 6 where we also examine the advantages and disadvantages of the implemented projectional editor over standard text-based IDEs.

A brief summary of the whole work may be found in the conclusion, where we strive to answer the final question, whether projectional IDEs are good for functional languages.

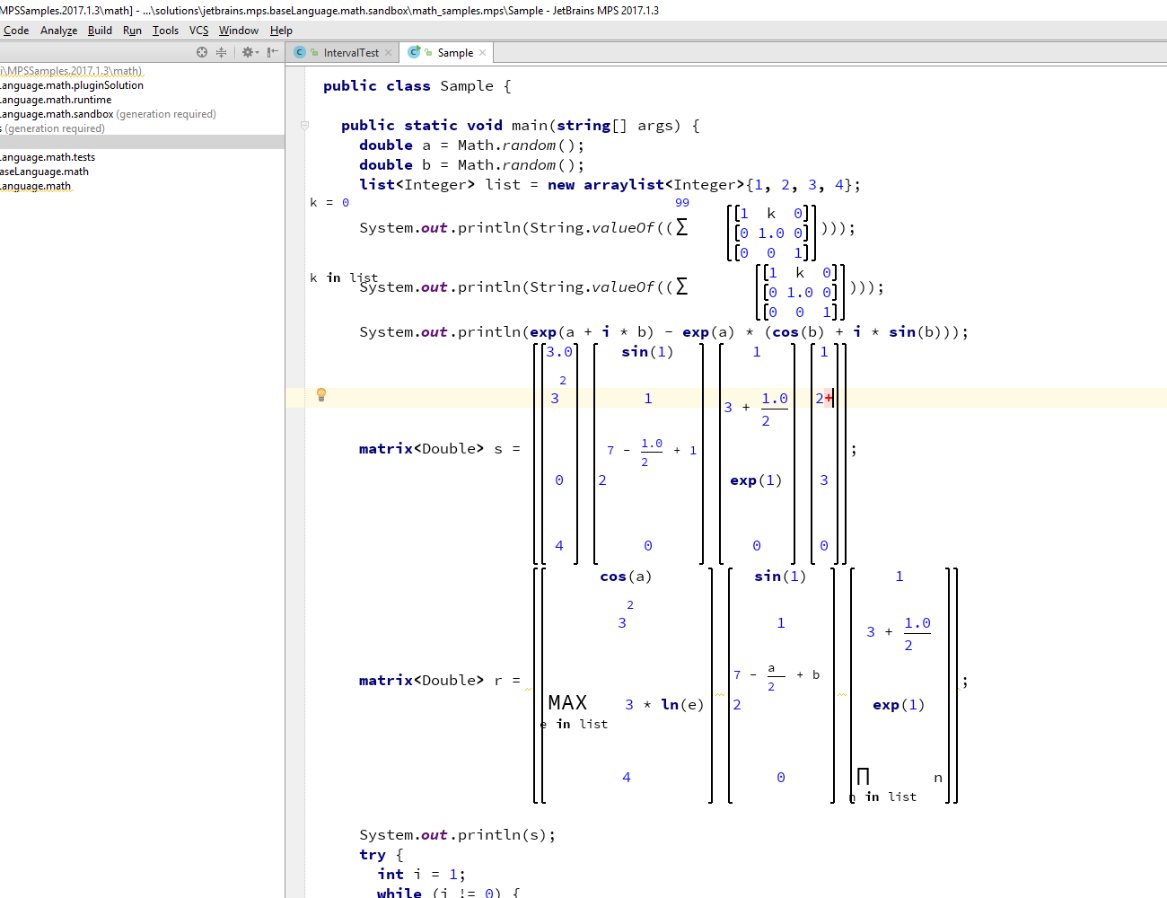
After that, references of the used literature may be found and appendix, which provides some of the more interesting source materials.

* Chapter 2: Meta-programming system / JetBrains MPS
  + - *what it is, what it can do, links & references, difference between projectional and classic text-based IDEs, how this relates to MPS, what makes MPS different*

<https://www.jetbrains.com/mps/>, <https://www.youtube.com/watch?v=XGm_khXZl44&feature=youtu.be>

Meta-programming system, or MPS, is an open-source language workbench that focuses on DSL (domain-specific languages). [JB-YT] It is a software engineering tool which helps its users to create a new language and then to write other programs in that language. [FC] MPS has a wide range of users, with areas currently being applied in including electrical engineering, data mining, insurance industry and other. The tool can be used to create new languages as well as extending existing ones. (It should be noted that only the languages created in MPS tool may be further extended.) Programs written in the defined languages may then be conveniently transformed into pure text in a specific, usually generic-purpose language. [JB-YT]

MPS is a projectional editor, which means it does not treat the document as a text, but rather as structured concepts. This allows its users to create languages which involve non-parsable notations, such as decision tables, diagrams, and other controls. [JB-YT] Additionally, several editors may be specified for a single language, thus allowing users to switch between different visual representations of a document.



2.1 View of ‘math’ demo language in MPS which includes non-text elements to extend the existing Java language

Traditional process of compiling written code involves lexers and parsers to read programs, which are expressed as text files. The two combined then transform the code into abstract syntax tree (AST) data structures [FC], from which in the process of semantic analysis and code generation an executable program is created. A similar technique is used in the text-based IDEs. In contrast, in MPS, the user works with AST directly, therefore completely omitting the process of lexical analysis and subsequent parsing. [FC] This brings certain advantages:

* It may be easier to extend an existing language.

Extending lexers and parsers to accommodate for the changes in a language requires a certain set of skills and a deeper knowledge of the language’s grammar. The process is complicated, since it requires a programmer to keep track of the possible ambiguities that may arise when defining new grammar rules for a parser.[FC] (See, so called, ‘Dangling else’)

On the other hand, in MPS, the process usually only requires defining new concepts that can act as AST nodes and specifying places in the corresponding AST, where the new nodes can be created.

This also means that in MPS we can combine syntax of several different languages and introduce no syntax ambiguities whatsoever. (This however still may look ambiguous to the user, if there are several different concepts with the same textual representation.)

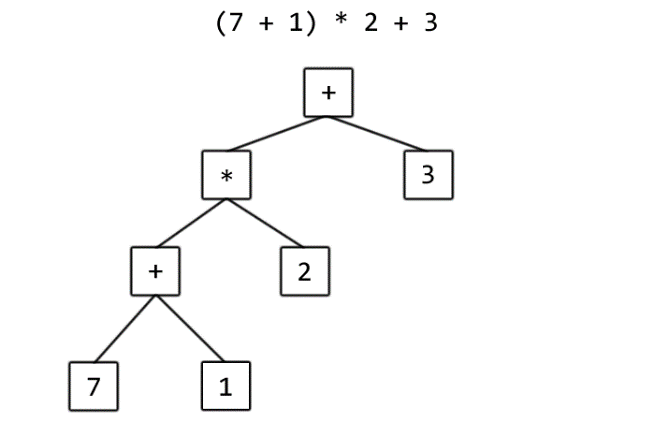
* We can check for type errors and other mistakes in the code at almost any time.

When writing a code in a traditional environment, one has to define a specific set of rules to deal with the incorrect syntax. Code being currently written means, it almost certainly cannot be correctly evaluated by the defined parser, so in an example case such as below, we will not be able to tell the user right away that the integral and string types are incomparable between themselves:

f = if 1 = "hello" then |

(syntax error on '=')

This cannot happen in MPS, since the code is already ‘parsed’. Even though ‘then’ and ‘else’ branches are not set yet, we can already check for errors in ‘condition’ node (1 = "hello"). This is also useful when designing a smart code completion feature, which requires a certain knowledge of the surrounding code context. This is impossible if we cannot parse the code beforehand, unless certain ‘tricks’ are used.

****

2.2 Exemplar AST for a simple arithmetic expression

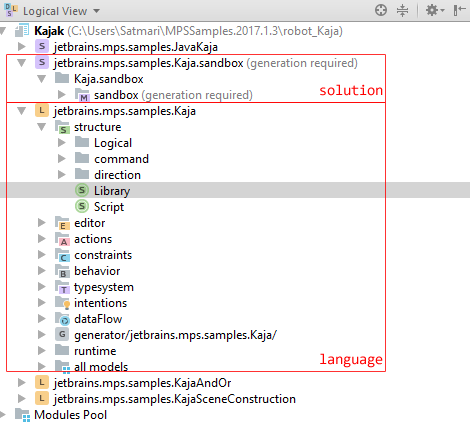
Since a user may in MPS work only with AST directly, several features are supported to allow easy transformations of ASTs and allow creating IDEs that may even closely resemble the traditional, text-based, ones. What this means is that we can define actions how the AST should transform when, for instance, a certain node (or a whole subtree) is deleted, a certain specific text is written at the end or a beginning of a visual representation of a node, etc. The MPS actions are described further in this chapter.

* *working with MPS, structure, editor, constraints, behaviour, checking rules, typesystem, textgen – generic explanation + exemplar language (not necessarily tutorial, but the explanation should be clear and how it would be used)*

A project in MPS is divided into two main categories: solutions and languages.

A ‘language’ is the user defined language. It may represent a completely new language or an extension of an existing one. There may be several different languages defined in a single MPS project. They can act as an extension of each other or be completely independent languages.

A ‘solution’, on the other hand, is a part of the project that represents documents, or a code, written in (usually) one or more of the defined languages. Sometimes the solution only acts as a runtime support for one (or more) of the defined languages to be used, for example, in a code generation process.

******

2.3 Logical view of a project in MPS

Solution is a set of models. They act as packaging units that make it possible to reference the corresponding set of models from other solutions of languages. [FC]

Model is simply a set of ASTs. In the languages used by a solution, a certain AST nodes may be set to act as a root node. Model then basically consists of such AST root nodes. Each AST is basically analogous to a single source file in traditional programming paradigm. [FC]

Language describes what types of ASTs can be created with the language. [FC] Furthermore, it includes visual representation of each node, AST transformation actions, syntax and semantic rules and many other ‘settings’. It is separated into several categories, named aspects. We will describe only the most important ones that were also used in this work.

**Structure**

<https://confluence.jetbrains.com/display/MPSD20173/Basic+notions>

Structure aspect of a language allows to define structure of possible ASTs that can be expressed with the corresponding language. [FC] It defines what kind of nodes may be used in user models, what properties, children and references they may have. [JB-D]

Key notion in this aspect is a ‘concept’. Concept represents a sort of a class of AST nodes. It specifies what children and properties all nodes of the ‘class’ have, or can have. It closely resembles working with classes and instances in many popular object-oriented programming languages. In this analogy, a concept is basically a class, which extends another class. (All concepts have to extend at least BaseConcept, similar to Java language, where each class extends Object class.) It may be defined as abstract, in which case no AST nodes may be created directly for such concept. We may define interfaces, which concepts may implement, concept properties, which are analogous to fields and attributes of primitive types (this is not exactly true as property may be of type string or enumeration), children, which represent public fields of object types (i.e. referencing an AST node of a certain type with ownership, not necessarily exclusive) and references, which are similar to children but without the ‘ownership’ part.

Each concept may specify visual appearance of the node (more in ‘editor aspect’), implement methods (behavior aspect) and specify additional syntax constraints on where they can actually be placed in AST (constraints aspect).

******

2.4 Definition of a new concept in MPS

We start by naming a concept. This is similar to naming a class in languages like Java and must follow a similar set of naming rules.

The ‘extends’ clause provides a reference to the super-concept. By default, concepts are created with BaseConcept as their super concept, but this can be changed to a more specific super-concept. This basically encodes ‘is-a’ relationship in UML. [FC] If concept A extends concept B, it indicates that concept A has all of properties, children, references, methods, and definitions from all other language aspects, as B.

Concept may implement interface(s) by using ‘implements’ clause. More about interfaces is discussed later in ‘structure’ chapter.

A concept may be set for its instances to act as root AST nodes. As it was mentioned before, a solution is a set of models all of which consist of root AST nodes. Only the concepts denoted as such may be used that way.

Alias acts as string that triggers a built-in auto-completion menu. If the name is unambiguous (i.e. not a prefix of another item in the menu), an instance of the concept is immediately created. More about menu is discussed in editor aspect of language.



2.5 Auto-completion menu in MPS

Properties hold values that are owned by the concept. They are analogous to UML class properties. [FC] They are either:

* Primitive type – integer, boolean, or string
* Enumeration type – a custom ‘enum data type’ may be created in structure aspect
* Constrained data type - a custom ‘constrained data type’ may be created in structure aspect, which is basically a string compared to a defined matching regular expression

It may be interesting to note that primitive types can be derived from using the other two options.

Children resemble aggregation relationship known from UML. They are other concepts with defined cardinality ([1], [0..1], [1..\*] or [0..\*]) and type. A concrete instance (i.e. a node) must be a child of an exactly 1 other node, with an exception of root nodes. This way their life depends on their parent’s life meaning that removal of a certain node from AST removes all of its children as well as children of their children, recursively down to leaf nodes.

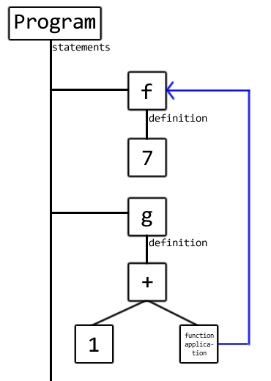
Expressing a relationship between nodes can be also done via references. It is only possible to create a reference to a node if this node already exists in the corresponding AST. [FC] Contrary to children, cardinality can take here only two forms: [0..1] meaning an optional reference and [1] meaning a required reference to a node of the given type.

Where would a reference be used? Consider a following piece of code in Frege:

f = 7

g = 1 + f

We could express the formed AST in many different ways, but let us imagine for the sake of simplicity a root node consisting of statement nodes. Both f = 7 and g = 1 + f are statements. It is easy to imagine 1 + 2 as a tree with node + having 2 children: “1” and “2”. But in the case of 1 + f it is less clear what f is. Using a reference here might be helpful. We already have a statement declaring, what f is, in the AST. In 1 + f we are only applying an existing function f. Such AST might then look like this:



2.6 Simplified AST for statements with a reference

Interface concepts are a mechanism to declare concept characteristics that can be used across several concept types. [FC] Unlike concepts, we cannot define alias for them nor can they extend concepts, only other interfaces. They are mostly used for grouping properties that are commonly used together and passing them onto necessary concepts. [FC]

**Editor**

[**https://confluence.jetbrains.com/display/MPSD33/Editor**](https://confluence.jetbrains.com/display/MPSD33/Editor)

Editor aspect is responsible for rendering and editing ASTs by the user of the language being created. This includes textual and graphical representation of each AST node and certain AST transformation actions. This aspect is what makes MPS a projectional editor, rather than using lexers and parsers to process the user-written code.

****

2.7 Concept editor in MPS

The easiest way to define an editor for a language is to define an editor for each concept (called ‘concept editor’). There may be several different editors defined for a single concept, which offers different views of the same concept for different needs. [FC] If no editor is defined for a given concept, a default one will be provided by MPS.

On the other hand, editor component is an editor responsible for rendering and editing only a part of a node. [FC] It does not focus on any single concept and as such may be reused across several concept editors to render certain parts similarly. [FC]

As most aspects in MPS, editor is too defined by writing a code in a special language provided by MPS – in this case jebtrains.mps.lang.editor. Furthermore, the code is written inside a projectional editor – to define a specific projectional editor for the new language being created.

The editor language is used to specify editor cells. An editor cell is typically used to render a certain portion of a node over a rectangular region in MPS editor window. The types of editor cells include:

* Constant cells – These are used to render keywords and other constant text in editor. [FC]
* Property cells – They render content of a specific property of a concept for which the editor is being defined. Editing such a cell in the editor window for a concrete AST is immediately reflected in the given property of the corresponding AST node. The cell provides automatic binding to the concept’s property.
* Child cells – Child cells delegate the rendering of a specific concept’s child (or a set of children) to their corresponding concept editors. The concrete behavior of such a cell depends on the child’s cardinality:
  + [1] – the editor cell is always present
  + [0..1], [0..\*], [1..n] – child nodes are bound to their corresponding editors and removing a child in MPS editor window results in removing it from the parent node of the corresponding AST as well
  + [0..\*], [1..\*] – the children’s corresponding concept editors are separated by a specified (textual) delimiter
* Referent cells – These cells are used to display an attribute of a referenced node from the given concept (see references in “structure” above). As in case of property cells, they are mapped to a certain property of the referenced node in the AST. However, they can only reflect the property of the original node, but not affect it.

******

2.8 Exemplar usage of the editor defined for Java-like language - changes in the name property of the original node (left) result in immediate re-rendering of the referent cell (right)

* Collection cells – Wrapper-like cells to contain other editor cells. They affect visual arrangement of the cells being rendered. There are 3 types:
  + Horizontal cells
  + Vertical cells
  + Indent cells
* Model access cell – This type of cell is used to modify a state of a node(s). We can define custom getter and setter for the cell, or just a custom getter, to change the behavior of our editor.

There are several other types of editor cells. Here we only described the most-used ones.

To change the way an editor cell is rendered, we can use editor styles. Applying editor style could be described as analogous to applying CSS *(Cascading Style Sheets)* styles to DOM nodes in HTML and XML documents. This allows us to change colors of a text, background color, spacing, padding, functional aspects such as editor cell being editable, read-only, etc.

******

2.9 Editor style for a selected editor cell

While all of this allows us to create a visually appealing editor, it still may not come as a user-friendly, especially to developers used to program in text-based IDEs. To simulate many types of behavior of classic text editors, MPS comes with a notion of actions. These were originally part of Action aspect of MPS, but since they are tightly related to the editor, they were moved in newer versions of MPS into this aspect.

To better explain the need of editor actions, consider an exemplar expression from figure (image?) 2.2: (7 + 1) \* 2 + 3. We already know what an AST would look like for such an expression, however, in MPS, without actions, a user would need to specify the AST him or herself.

First, we would need to add addition binary expression node, then for the right operand, literal ‘3’ while for the left operand another binary expression - multiplication, for which we would need to specify the operands again, and so on.

With a careful implementation of actions, however, we could let the user type the expression from ‘left to right’ while transforming the AST automatically for him or her. To simplify, let us forget about brackets for a while and try to type a slightly different arithmetic expression: 7 – 1 \* 2 + 3.

|  |  |
| --- | --- |
| First, the user types in ‘7’. That is a very simple unary expression and no further work is to be done here. | 2.10 |
| Then, user hits ‘-’. We immediately create a binary expression subtree, where root is the operator ‘-‘. ‘7’ is put as its left operand and we set focus on the right operand, so the user may edit that. | 2.11 |
| User types in ‘1’, which only concludes the editing of the right operand and we are done there. | 2.12 |
| Then, however, follows ‘\*’. We are now editing the right child of AST of the expression “7 – 1”. So we take a look at the parent’s operator’s precedence: - is definitely less precedent than \*. So we create a subtree for binary operator \*, put 1 as its left child and set focus on the right child. The subtree is placed in 1’s stead (i.e. right child of AST of the expression “7 – 1”). | 2.13 |
| Then comes 2, which only finishes editing of the right child of \* subtree. The twist comes, however, when user types in ‘+’. We take a look on the parent’s operator, which is \*. This has a higher precedence, so a new subtree has to be created elsewhere now. The parent of the \* node is, however, -. While – has the same precedence, as +, all of the operators are left associative, which means, we have to create our new subtree even on the higher level. We create the subtree, put the current AST for the expression “7 – 1 \* 2” as its left child and set focus on its right child. Typing ‘3’ only finishes editing of the right child and we are done. | 2.14 |

In the description above, we were always editing a specific editor cell corresponding to a single node in the AST. We always handled an event of writing a specific textual pattern right of a certain editor cell. This is just what MPS allows us to do.

**Transformation menu**

The transformation menu provides a way to describe side-transformations and substitute actions in MPS. For a particular cell in the editor we can specify one of the two mentioned actions.

Side-transformations allow to manipulate the AST when a certain textual pattern is enterd either from the left or right of the current editor cell.

Substitute actions, on the other hand, are triggered, when one node is being substituted by another node. They allow to define a default concept to be used for instantiating new nodes in certain places in the model, or an action to take upon creating a new node of a certain type in certain places, and others. We descibre substitute actions later in this chapter.

<https://confluence.jetbrains.com/display/MPSD20172/Transformation+Menu+Language>

Transformation menu is always created for a specific set of concepts. Similarly to OOP languages, we can create a transformation menu for a concept, which will be also applicable to all of its (either direct or indirect) subconcepts. There are two main types: default and named menus.

Every concept has associated a default transformation menu. If the language designer does not provide one explicitly, a transformation menu defined for the closest super-concept is assumed. [JB-C] If none are defined, a one implicitly defined for BaseConcept is used. A named menu is an additional menu for a concept. Unlike the default menu, it is not applied to a concept directly, but rather has to be attached to a certain editor cell(s) of the given concept manually. Another way to use a named transformation menu is to include it into a different menu via „Include menu“ statement.



2.15 A default transformation menu for a concept

We start by specifying section, i.e. where the transformation should take place. Side transformations have only two options: either they are triggered upon typing a text right of an editor cell, or left of a cell.

Then, we specify a, so called, menu part [JB-C]:

* Action - a simple menu item specifying an action to be performed, its corresponding menu text and applicability. The 3 categories are separated (see figure 2.15).
  + Text – a string that triggers the current action written either left or right of an associated editor cell (depending on the „section“ specification). It can either be a constant or a piece of code that returns a string that triggers the action.
  + Can execute – a piece of code that is executed upon the text trigger. If the code returns false, the current action may not be executed. *(Returning „true“, however, does not indicate that the action should just immediately execute. If, for example, we have another action for the same concept defined with different „Text“ trigger, for which its „can execute“ clause would also return true, ambiguity arises. The „Text“ part is just a trigger to execute any of the actions for which „can execute“ returns true. That is why it is mostly recommended to leave* “*can execute <always>“, which has a different behaviour – the action is triggered exactly when the string from „text“ part is typed in.)*
  + Execute – a code associated with the current action, i.e. what should actually happen.
* Group - a collection of menu items.
* Include ... - include a specific default or named menu. We may also include a transformation menu not aimed for the concept we are currently dealing with. In that case we have to provide the node of that specific type (e.g. be it a parent of a current node, or a completely new node created ad-hoc in the AST, etc.)

There are also other menu parts in MPS we will not be currently dealing with.

Important thing to note when dealing with ~~transformation~~ (all!) menus is that they only apply on constant, property and referent editor cells. If a concept has other children for which it defines child editor cells in the concept editor, then they do not trigger the concept’s side transformations. In that case a language designer has to pay a greater attention to what cells actually trigger which action.



2.16 In Frege, we may define custom algebraic datatypes as follows:

data Maybe a = Just a | Nothing

We call “Just a” and “Nothing” datatype constructors. After each one it is desirable to be able to add a new constructor when typing character ‘|’. But since a constructor can (but does not have to) be followed by one or more type variables, like ‘a’ in ‘Just a’, it is not sufficient to specify the transformation for the ‘DataConstructor’ concept only. Even more troubling, the transformation right of a type variable is desirable only if it happens to be the ‘right-most’ editor cell in the given ‘DataConstructor’. (i.e. we would not want to add a new constructor when typing ‘|’ right of ‘a’ in this case: data Maybe a b = Just a| b ) The solution is to simply include the default transformation for ‘DataConstructor’ into type variable (here: ‘Simpletype’) concept, but only if it happens to be the right-most child of a ‘DataConstructor’.

Substitute actions differ from side transformations. Typically, they are actions from completion menu, which is invoked in MPS by pressing ctrl + space. Since MPS is not a classic text editor, completion menu is a vital part of the tool and a user will find him or herself using it quite often.

A default substitute menu is always provided by MPS when the caret‘s position is in front of a node or the node is selected. *(We mean here a group of editor cells corresponding to a single AST node.)*



2.17 Invoking a substitute menu on a selected node

For that case a menu with all concepts applicable in the given context are displayed, which can substitute the current node. [JB-C] Abstract concepts are, understandably, not displayed as well as none of the concepts, for which their constraints do not allow their presence in the current place. (See constraints aspect for more information). Smart references are treated somewhat otherwise. A smart reference is a term we use for concepts that contain only a single reference and nothing else. Such a concept is not added to the completion menu itself, but rather all of the nodes it could contain a reference to *(i.e. the nodes in the current scope)* are added to the completion menu. Selecting such a node results in creating an instance of the corresponding smart reference with the reference to the selected node.

Similarly to transformation menus, substitute menus are tied to a concept type and can be either default, or named. Named substitute menus can, however, be only included in other menus and on their own have no effect on the editor.

Adding an empty default substitute menu for a concept makes it inelligibile to be included in the default menu provided by MPS. This is a useful way to treat concepts like ‚empty line statement‘. In such an example, we would set that a default ‚statement’ is ‚empty line‘ (in ‚element factory‘ settings for ‚child editor cells‘ in ‚concept editor‘). But since it does not make much sense to replace a statement with an empty line (usually a user would want either to delete the node completely or add a new line above), we will remove it from the completion menu this way.

There are several entries to specify for a substitute menu:

* Include – includes a specified substitution menu.
* Subconcepts menu - includes all subconcepts of the current concept.
* Concept list - adds a specified collection of concepts.
* Substitute action - adds a single substitute action.
* Wrap substitute menu - wraps a specified concept using the provided handler. [JB-C]

There are several more entries to choose from, but for the most common cases these ones are sufficient. We will now take a closer look onto entries „substitute action“ and „wrap substitute menu“, since they are a little more complicated.

Substitute actions allow to define an action to take upon writing a specific textual pattern in place of a node for which the substitute action is being defined. It consists of 3 main parts:

* Create node – a piece of code which returns a node of such a type that the current concept can be substituted for.
* Matching text – a string (or a piece of code which returns a string) that triggers the current substitute action.
* Can substitute – a boolean depending on whether the substitute action can execute. *(Similar paradigm as in „Transformation action“ applies here – returning true for several substitute actions with different „matching text“ still results in ambiguity and the action will not execute on its own, but will rather require user’s decision from the completion menu. If an additional check besides „matching text“ is necessary, it is recommended to check for user-entered string here as well. Otherwise it is best to omit this part completely, which will result in the action triggering only upon exact match of the user-entered text with „matching text“)*

For better undestanding, let us assume an abstract concept „Literal“ with two subconcepts: „Integer“ and „Boolean“. We want to automatically create a node, where a node of type „Literal“ is expected, upon user typing either an integer number (we create an instance of „Integer“ concept) or „true“ or „false“ (we create an instance of „Boolean“ concept). This is done by defining a default substitute menu for concept „Literal“ with 2 substitute actions. In both cases, „matching text“ simply returns any text a user has typed, since there is no single string, that would suit. Instead, we have to check, whether the written string is actually an integer number or „true“/“false“ in „can substitute“ clause. Finally, „create node“ clause simply handles the substituion action.

„Wrap substitute menu“ has a similar behaviour as smart references. In this case, however, a language designer specifies a concrete concept of which node instances should be suggested in the completion menu to substitute the current concept for. This also works for an example, where we have already defined substitute actions for literals, such as numbers, or boolean values. Let us assume such literals inherit from common ancestor – concept „Literal“. However, consider that we are defining a node of type „Pattern“. Now, „Pattern“ is not an ancestor for „Literal“ in terms of inheritance. This means that if we are to allow specifying literals where concept „Pattern“ is expected, we have to create a new concept, a „wrapper“ for concept „Literal“, that inherits from concept „Pattern“. (This is analogous to adapter design pattern.) Let us call it „PLiteral“. Now the problem is, the substitute actions defined for „Literal“ concept do not work here, since „PLiteral“ is expected instead of „Literal“. A simple solution is to define „wrap substitute menu“, which acts as a substitute menu for „Literal“, but as its output a node of type „PLiteral“ is produced instead.



2.18 Wrap substitute menu for concept PLiteral, which wraps concept Literal

From editor actions we mention also Cell Action Map. These are custom defined actions to take upon a user attempt to e.g. remove an editor cell, insert a new one, pressing a backspace key, etc. They are always associated with an editor cell (though several editor cells can share the same cell action map) and a simple event, such as cell removal, insertion, selection, copying and pasting.



* 1. Example of cell action map

A cell action map is always defined for a specific concept, but can be used for any editor cell in the corresponding editor of the concept. We then select an event we want to create a handler for. There are around 40 events to choose from, but probably the most important ones are INSERT and DELETE. ‘Can execute’ clause allows to specify a condition to cases where the action is actually applicable. Finally, in ‘execute’ we specify the handler itself. An example of where a ‘cell action map’ could be useful may be concepts which are semantically different, but syntactically similar. For instance, consider in Frege an expression surrounded by brackets and a tuple:

(let x, y, z be functions, e.g. x, y, z :: Int -> Int)

f = (x) 1 -- Expression surrounded by brackets

g = (x, y, z) -- Tuple

While similar, the difference is significant – an expression surrounded by brackets may be used as a ‘function application’, i.e. we may specify arguments right of the expression. On the other hand, such a syntax would be illegal for tuples. This means we only need to allow specifying arguments, if the node is an instance of the concept ‘BracketedExpression’, but not for ‘Tuple’. However, we still want to allow a user to remove all but the last item of a tuple. Since tuple requires at least 2 items, we may do this by automatically converting it into ‘BracketedExpression’ once the second item has been deleted.

To conclude, editor actions provide a flexible way to build a user-friendly editor that mimics many features of a traditional, text-based, editor. However, understandably, it is impossible to allow the completely same behavior, since a user is editing the AST and not the text. This means that almost every editing feature has to be implemented manually. We can, however, optimize the editor for the most common cases at least.

**Behavior**

Behavior aspect allows to, simply said, define methods on concepts. If we take the analogy with OOP further, then structure aspect allows us to declare classes and their fields while behavior aspect allows us to declare and implement their methods, including constructors.

Constructor is a block of code which is executed when a new node of the concept is created. Certain exceptions exist, however, when the constructor is not run. These mainly include creating an instance of a node by using “new node<MyConcept>()” statement (e.g. in editor actions) instead of “new initialized node<MyConcept>()”.

Similar concept of methods as in languages like Java is present here. A concept may be associated with several methods with strictly defined visibility (public, protected, private). Methods that can be overridden in subconcepts have to be marked as virtual. Static methods are not attached to an instance of the concept but rather have to be called on the concept itself. (concept/MyConcept/.staticMethodName()) [FC]

Important characteristic of the behavior aspect is that it allows to traverse the AST being created. We can easily inspect parent and children of any node as well as nodes’ references (see structure –

concept references).

2.20 Behavior aspect in MPS

**Intentions**

These are special user interface elements that allow executing predefined actions in certain places. They usually perform some modification of the current AST. The intentions menu is usually invoked by pressing “alt+enter” and selection is confirmed by pressing “enter” key.

 2.21 Intentions in Frege-IDE example

Standard intentions are defined for a concrete concept. There are also “surround-with” types of intentions, which are similar to the regular ones, but are designed to surround a part of the AST with another construct. [FC]

In regular intentions, we also specify, whether the intention can be executed inside of any of the concept’s children. This allows for example to invoke an intention defined for “DataConstructor” concept even if the caret’s current position is over one of its type variables, e.g.:

data Maybe a = Just a|

Where | denotes the caret’s current position – which is on DataConstructor’s child, ‘TypeVariable’ concept. Next, we specify a textual description of the intention that will be shown to the user upon invoking the intentions menu. “isApplicable” method specifies, whether an intention is applicable for the current state of the AST. If the method returns false, the intention will not be shown in the menu. Finally, “execute” method specifies the intention’s action itself.

 2.22 Example of a concept intention definition

**Constraints**

This aspect lets a language designer to declare constraints that help him or her control where nodes of a language are allowed. [FC] We define constraints always for a specific concept. They allow specifying several attributes each of which has a different meaning.

2.23 Example of a constraint definition which puts restrictions on a referenced node “module”

“Can be child” allows defining a boolean method which returns, whether a node of the current concept can be a child in a specific AST context. [FC] If the method returns false, then such a node will not be suggested in the auto-completion menu. Attributes “Can be parent” and “Can be ancestor” work in a similar way.

Property constraint allows restricting a set of values of a concept property. Additionally, it is possible to define a getter and a setter for the given property.

Referent constraint makes it possible to restrict how references are established to nodes of the concept. [FC] This basically allows restricting what nodes will be referenceable from the given concept. Part “Scope” specifies scope of referenceable nodes. Part “Presentation” specifies how these nodes will be displayed in the auto-completion menu *(their textual representation)*. Scope is an object which defines a list of potential targets that can be referenced. It also helps to locate a suitable target from the given list based on what the user has entered in the corresponding place in the editor. When no scope is provided in the concept’s constraint aspect, all nodes of the appropriate type are considered eligible. The auto-completion menu is filled with nodes based on this rule and the scope, if specified.

2.24 An example of applicable nodes in the function-definition context of Frege-IDE. According to the given scope, variable “x” is part of a different function pattern definition and as such may not be suggested in the auto-completion menu for the definition of function “gg”.

It is important to note that part of the job for a Scope object is to return a reduced list of nodes that conform to the already entered string by a user. Consider an example from figure 2.24. If we were to enter a string “f”, a list would be reduced to contain only item “ff” as far as referenceable nodes are concerned, since “gg” and “y” *(and operators)* does not contain “f” as its substring. *(Note that items like “false” and “if” are not nodes to be referenced, but rather aliases of certain concepts, which we may then create instances from, too.)*

A language designer may specify his or her own implementation of “Scope”. The new class must inherit from the “Scope” class provided by MPS implicitly and implement the necessary abstract methods. Notable methods from the class include:

* **public abstract sequence<node<>> getAvailableElements(@Nullable() string prefix);** Returns all of the nodes from the scope that begin with a string “prefix”.
* **public abstract node<> resolve(node<> contextNode, @NotNull() string refText);** Returns a node, if the entered string “refText” can unambiguously determine a referenceable node from the current scope. (In 2.24 “ff” would be an example of such a string.)

Another point to note is that it is not certain what string is represented a certain node by. That is why the method **resolve** cannot work automatically. Additionally, the string defined in Scope object does not have to be equal to the one specified in “Presentation” part of a link in constraint aspect.

**Typesystem**

Typesystem aspect makes it possible to report semantic errors to the user of the language. [FC] On its highest level, it could be said it contains mechanisms to check for both non-typesystem and typesystem rules.

Non-typesystem rules are called checking rules. These serve a language designer to implement custom semantic error checks. An example in Frege-IDE would be a type statement, which allows users to declare a new custom type. It may contain type variables, but each of these must have a unique name:

 2.25 An example of type statement in Frege-IDE

Error detection like this is easily implemented by using a checking rule for the concept representing the whole type statement.

2.26 An example of a checking rule in Frege-IDE

A checking rule is basically a single method that is executed for a specific concept upon its any change. We tell MPS there is some semantic error by using “error” statement in the checking rule definition. “error” statement accepts a string (a message to display to the user of the language) and a node, which caused the problem. The MPS then underlines the corresponding node with red color in the editor to denote the error. *(Note that if a node’s reference is provided, then the original node is underlined, not the reference.)*

Typesystem rules, on the other hand, offer a declarative way to express rules which support type calculations. [FC] We can let MPS calculate types of expressions in the runtime and upon finding inconsistencies MPS will report error to the user on its own.

[FC – about w/s st:] MPS supports weak and strong sub-typing relations. A weak sub-type relation ta :< tb implies that expression of types ta and tb can be used interchangeably only in special cases. On the other hand a strong sub-type relation ta :<< tb implies ta :< tb and expressions of types ta and tb can be used interchangeably in all situations where expression of either ta or tb can be used. We can compare calculated types by using (both functions return boolean):

* isSubtype(type1 :< type2)
* isStrongSubtype(type1 :<< type2)

Inference rules are created to calculate a type of a node for a given concept. These can also be used to enforce a type, i.e. to perform a type check. An inference rule consists of a name, “applicable for” clause (where we specify a concept which we want to calculate a type for) and a “do“ part, where we define the rules for the current concept. The rules are written in the standard Java-like imperative language used also in the other MPS aspects (such as behavior), but it is extended with statements regarding the typesystem rules. An inference rule for a concept representing an integer literal would look like:

typeof(intLiteral) :==: <int>

On the left side, we have typeof(intLiteral). With this we are simply denoting a type of a node of a concept “IntLiteral”. With the operator :==: we are telling MPS that the type on the left must be the same as the type of the right operand. This is the rule we are specifying. *(Note that it is possible to perform a check only – this has to be specified in the inspector window in MPS for the current operator :==:.)*

Finally, the <int> part denotes a quotation. It is used when a language designer needs to create nodes of the concepts of a language (the language may be even the one he or she is now creating). A standard way of creating a node of a concept would be (an equivalent piece of code):

node<IntTypeNode> itn = new initialized node<IntTypeNode>();

typeof(intLiteral) :==: itn;

Anything displayed inside the quotations symbols (< … >) is what a node would look like as if an editor was used to edit the actual AST (in this case “int” is a textual representation of a node of the concept IntTypeNode). The quotations are most useful when dealing with larger ASTs, where a series of statements of “new initialized node<IntTypeNode>();” and child assignments would be difficult to read. Additionally, “antiquotations” may be used to include some dynamic content. These are denoted by symbols %( … )% inside quotations. For instance:

var firstTypeVar;

typeof(firstTypeVar) :==: <int>;

typeof(intBoolTuple) :==: <( %( firstTypeVar )%, bool )>;

Based on this, this is what an inference rule could look like for a not-expression node (standard “!” operator in Java, C, …):

2.27 An exemplar inference rule for “not-expression” concept

If the type of the original expression is not boolean, MPS will find out and underline the error for us: 2.28 An automatic type check for “not-expression”

Sometimes it is necessary to create custom types that are not present in jetbrains.mps.baseLanguage. For instance, consider a tuple in Frege language. A tuple is basically just a wrapper of 2 or more items of (not necessarily) different types. To create a tuple type, we have to create a new concept, which inherits from jetbrains.mps.baseLanguage.Type concept. We then specify its children. For simplicity, let us assume a tuple may contain 1 or more items rather than 2 or more. In that case, we declare that our “TupleType” concept has [1..n] children of type “jetbrains.mps.baseLanguage.Type”. To be able to use our new “TupleType” in typesystem rules, we have to include the language the concept is defined in, for the typesystem aspect.

 2.28 Specifying included languages for a typesystem aspect in an exemplar project

There are several other operators to use when specifying typesystem rules, but we will mention only the 2 following ones:

:<=: Tells MPS that the type on the left-hand side is a weak sub-type of the type on the right-hand side. [FC]

:~: Tells MPS that the type of operands on either side of the operator are weakly comparable. [FC]

The :~: is useful for the type definition of “equals” operator (“==”): we just specify that the operands on the left and right are weakly comparable and set the resulting type to bool:

typeof(equalsExpression.operandLeft) :~: typeof(equalsExpression.operandRight) ;

typeof(equalsExpression) :==: <bool>;

The custom defined types, however, are not implicitly comparable. Inequations are normally solved in MPS by applying subtyping rules. This means to compare types like “int” and “double”, we would first need to create a subtyping rule, which tells MPS that “int” is a subtype of “double”. For a more complicated type, like tuple, we would, however, need to create a replacement rule. These are used by MPS to solve inequations where subtyping rules may not be used. A replacement rule is always created for 2 concepts (which represent corresponding types) and has a defined body. For the given 2 incomparable types, MPS removes the inequation and simply executes the body in the corresponding replacement rule.

2.29 An example of a replacement rule for tuples in an exemplar project

To see how :<=: operator would be used, let us consider an inference rule for the concept representing binary + operation. Let us imagine that the operation is supported for all of the current types, but the types need to be comparable between each other, i.e. “double + int” is supported and the result is double, but “double + bool” is not. However, “bool + bool” is supported with the result “bool”, whereas “string + string” returns “string”, and so on. To express something like that in MPS, we can simply use the following piece of code:

infer typeof(plusExpression.operandLeft) :<=: typeof(plusExpression);

infer typeof(plusExpression.operandRight) :<=: typeof(plusExpression);

These two statements tell MPS that the result of the plus expression is the most “abstract” type of all of its two operands.

*REDO: We use when concrete only when inspecting the type!*

~~Now, to make things a little more complicated, we may assume our exemplar language on its highest level consists of series of statements each of which is a simple declaration of a variable, for instance:~~

~~i = (1 + 7.14)~~

~~j = (true == false)~~

~~k = !((1, 2.7) == (7, 2.7))~~

~~We can, however, use a declared variable as an expression, e.g.:~~

~~l = k + 1~~

~~With the current implementation, we would find out that MPS did not, in fact, underline the statement as an erroneous one~~ *~~(“bool” + “int” is not defined)~~*~~. Why? The reason is that the type of “k” is not guaranteed to be evaluated before the type of “l”. It is different for expression (true == false), where the expression forms a binary tree and its type can be evaluated “bottom-up”. But the statements “l” and “k” are simply children of a root node and their ordering does not play any role here (we could swap statements “k“ and “l” and still expect MPS to evaluate the types correctly – just like in languages like Java and C#, where we can have a variable of a type that is declared later in the code). We can deal with the problem by using “when concrete” operator. “when concrete” has a form of a block of code and it delays its evaluation until the time the type of the requested argument has been calculated. The plus operator would then look like this:~~

when concrete (typeof(plusExpression.operandLeft) as olType) {

when concrete (typeof(plusExpression.operandRight) as orType) {

infer olType :<=: typeof(plusExpression);

infer orType :<=: typeof(plusExpression);

}

}

It is important to note that the code wrapped inside the “when concrete” operator is executed in a different thread, which means rest of the code of a currently defined typesystem rule will carry on with its execution independently.

**Texgen**

An optional aspect component of a language that allows to define a mapping from concept nodes to text. [FC] We can trigger the text generator for a specific root AST in a model by using a right mouse button in the editor space and selecting ‘Preview generated text’.

Textgen is defined for each concept. In certain areas, it could be compared to the editor. Upon triggering, it starts from the root node and (usually) continues down to the leaf nodes of the AST, handling each node separately. (Upon triggering for multiple root nodes, each one is written to an individual file.)

The textgen consists of a single method written in a Java-like language. It should specify, what should be outputted to the file, by using ‘append’ statement. We can increase indentation for a specific text by using ‘with indent’ block. Additionally, we can call a textgen for another node by using a variant of the ‘append’ statement.

Note that an error arises when trying to call a textgen for a node which does not have the textgen specified. This can be useful when a given AST is currently incomplete or in an erroneous state. That is why abstract concepts usually do not implement it.



2.30 Textgen for a concept in an exemplar project

* Chapter 3: Frege
  + *what is Frege*
  + *Frege vs. Haskell*
  + *Frege language - code examples, what it consists of (high-level)subset related to what we decided to support in “Frege-IDE”, links to frege, github – further reading; here only the basics*

<https://github.com/Frege/frege> [FG - Frege github]

Frege is a Haskell for JVM. It is a purely functional language, has a strong static type system with global type inference and non-strict evaluation. [FG] The language compiles to Java and runs naturally on the JVM. This way it can be used inside any Java project. [FG]

The language was designed by Ingo Wechsung, who named it after the German mathematician, logician and philosopher Gottlob Frege. [WIKI]

<https://wiki.haskell.org/Foreign_Function_Interface#FFI_extension> <https://hackage.haskell.org/package/base-4.11.0.0/docs/Prelude.html>

<https://github.com/Frege/frege/wiki/Differences-between-Frege-and-Haskell> [1]

The syntax of Frege is very close to that of Haskell, with only small differences. In general, Frege could be considered a somewhat smaller subset of Haskell language. Certain features are missing, like the foreign function interface, which allows Haskell to interact with code written in another language. Instead, there are language constructs to make Java types and methods usable. All primitive types are just Java types, and so is String. [FG] The Frege-Prelude library (an equivalent of Haskell-Prelude library which defines many standard types like Maybe, functions, operators (||, &&), etc.) has many functions, type classes and types known from Haskell. However, Frege uses the Java APIs whenever possible, so certain points may feel different. [FG] For instance, implementation of type classes is incomplete and multi parameter type classes are not supported at all. [FG] Additionally, Frege does not support “newtype” declaration (an algebraic data type with exactly one constructor), “deriving” clause for data type declarations, and a few other keywords. A string value in Frege, unlike Haskell, is not a list of characters but an instance of the Java class java.lang.String. [FG] Also, Frege does not have any operator data constructor other than colon (“:”). This, however, allows to define even such custom operators that begin with the colon character (like :-: is a legal custom operator function in Frege). There are several other minor differences between Haskell and Frege; however, they mostly do not affect this work in any way, which is why we will not be describing them. To find out more information, please follow link [1].

**Frege language**

We will now provide several examples of writing programs in Frege. The syntax described here was also mostly included in this work. Though recommended to read, people familiar with Haskell or Frege may skip this part.

<https://github.com/Frege/frege/#a-taste-of-frege>

An example of a “hello world” program in Frege:

module Hello where

greeting friend = "Hello, " ++ friend ++ "!"

main args = do

println (greeting "World")

This code would compile to Hello.class and Hello.java with a regular Java entry point method main. Moreover, the Hello.class would have a method public static String greeting(String ...) {...} that one can call from Java (or any other JVM language). [FG]

Just like in Haskell, the function „greeting“ is **pure**, which means it is stateless and does not have any side effects. For the same given input parameters it always returns the same result. [FG] This is a great advantage of functional languages that basically allow the results of such functions to be cached. Function „main“, however, is not pure. Since it corresponds to the „main“ function in Java language, it may produce side effects, like printing to the console (which it happens to do so in this concrete example). [FG] Impure functions being part of a rather larger group (called monads) were not included in this work and it is therefore not possible to write the definition for „main“ function in Frege-IDE.

Frege is a strongly and statically typed language. If the types are not provided by a programmer, they are automatically inferred. To provide types for a function, one can write:

greeting :: String -> String

We call this signature a type annotation of the function greeting. Here, we denote that function „greeting“ accepts a single argument of a type „String“ and returns a result of type „String“. Additional types that are also supported by Frege-IDE include:

* Bool (only values true / false)
* Char (a single utf-8 character)
* Int (integer numbers)
* Double (floating point numbers equivalent to Java double type)
* Tuples
* Lists
* Custom algebraic datatypes

<http://learnyouahaskell.com/syntax-in-functions> <https://www.haskell.org/tutorial/patterns.html> [H]

An important aspect of programming in both Frege and Haskell is pattern matching. When we define a new function, we may define different bodies for different input arguments. To elaborate, consider the following definition of the function charToName:

charToName :: Char -> String

charToName 'a' = "Albert"

charToName 'b' = "Broseph"

charToName 'c' = "Cecil"

charToName \_ = "No Name"

The main idea behind the function is to provide the caller with a (human) name beginning with the given character. (Here we only provide definition for the first 3 characters of English alphabet, albeit it should suffice for the demonstration.) The first line of the program tells us the charToName is a function accepting a single character argument and outputting a string. Then, we provide for each character a specific function’s definition. The wildcard ‘\_’ matches any input. This way if we were to call charToName with, for example, ‘a’, it would return “Albert”, but for ‘z’ we would get “No Name”. The ordering of the definitions is important here – moving the definition for any pattern (i.e. wildcard ‘\_’) above the definition for ‘a’ would result in all calls to charToName returning “No Name”.

Regarding the pattern matching in Frege / Haskell it is important to also mention variables. These “formal parameters” are also patterns – it’s just that they never fail to match a value (similarly to the wildcard ‘\_’). As a "side effect" of the successful match, the formal parameter is bound to the value it is being matched against. [H] For this reason patterns may not contain multiple variables with the same identifier. Based on this, we could create a function returning a second element of any 3-item-tuple. A tuple is just a collection of items of (possibly) different types, separated by comma, e.g.:

(1, 2.7, true, ("hello", 'a'), 'z') is a tuple of 5 items, 4th of which is another tuple of 2 items. Our function would then look like this:

second (\_, x, \_) = x

When calling “second” with a 3-item-tuple argument, the second value is automatically bound to the variable “x”. This is what we then return. We call this mechanism data deconstruction.

Besides tuples, there is also another fundamental data structure to hold multiple values in Frege: a list. A list is a homogenous data structure (i.e. all of its elements need to be of the same type). An example of a function returning a list is as follows:

listExample :: [Int]

listExample = [4, 8, 15, 16, 23, 42]

In this specific example we return a list of integer numbers, which is defined by enumeration. For ordinal data types, however, we can also specify a range of values:

rangeListExample :: [Int]

rangeListExample = ['a'..'z']

rangeListExample is a function which defines a list of 26 characters from ‘a’ to ‘z’.

When thinking of lists, we separate it into 2 parts: head and tail. A head is a single element at the top of the list. Tail is the remaining part. (If the list contains only 1 element, the tail is an empty list.) Such a picture of a list is important, because it allows us to pattern-match it against the data constructor operator “:”:

getTop :: [String] -> String

getTop [] = "No elements"

getTop (x:xs) = x

The function above returns for a list of strings a first element from the given list, if not empty, otherwise it returns “No elements”. The pattern (x : xs) matches a list in this way: x is its first element, xs is the remaining part of the list (tail).

frege> getTop ["hey", "hi", "hello"]

hey

This principle allows us working with lists in an actually useful way; otherwise we would need to match them against an exact pre-defined pattern. Here we provide an example of a function which joins 2 lists into a single one:

listJoin [] ys = ys

listJoin (x:xs) ys = x : (join xs ys)

frege> listJoin [1, 2, 3, 4, 5, 6] [7, 8, 9]

[1,2,3,4,5,6,7,8,9]

Another way of declaring list is using a, so called, list comprehension. This could be thought of as analogy to using SQL, where we define what data we want, input sets (where to get the data from) and condition to filter out the unwanted records. For example, a function returning a list of Pythagorean triplets can be defined as:

pt = [(x, y, z) | x <- [1..15], y <- [1..15], z <- [1..15], x < y, y < z, x\*x + y\*y == z\*z]

frege> pt

[(3, 4, 5),(5, 12, 13),(6, 8, 10),(9, 12, 15)]

Frege-IDE provides (albeit limited) support for all of the options for defining lists.

**Custom types and algebraic data types**

Frege allows to create new types by using ‘data’ statement:

data Days = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday

This introduces a new type (called ‚Days‘) and constructors (‚Monday‘, ‚Tuesday‘, ...) for making

values of that type. [REF] We can work with the new type for example in the following way:

getNextDay :: Days -> Days

getNextDay (Monday) = Tuesday

getNextDay (Tuesday) = Wednesday

…

In this particular example, we have defined only a simple enumeration. However, we can also wrap additional data like in the following example:

data Point = Point Double Double

movePointX :: Point -> Double -> Point

movePointX (Point x y) \_x = Point (x + \_x) y

We have defined a simple type representing a point in a 2D space and a function moving that point by the given value in x-axis. In this case our custom data type has only a single constructor. Notice that the name of the data type and the constructor are treated completely independently by the compiler and therefore may be equal.

The constructor arguments do not necessarily have to be of primitive types. Consider the following more advanced example:

data Shape = Circle Point Double | Rectangle Point Point

surface :: Shape -> Double

surface (Circle \_ r) = pi \* sqr r

surface (Rectangle (Point x1 y1) (Point x2 y2)) = abs (x2 - x1) \* abs (y2 - y1)

This, however, means we can also construct recursive data structures:

data Tree = Nil | Node Int (Tree) (Tree)

To provide more flexibility for custom datatypes, we can also use the notion of type variables:

data Maybe a = Just a | Nothing

In the example above, ‘a’ represents a type variable. We can use any type in its place. Instead of then having to define several datatypes for several different functions, we can just reuse the type while specifying, what ‘a’ is, for each one:

getTopIntList :: [Int] -> Maybe Int

getTopIntList [] = Nothing

getTopIntList (x:xs) = Just x

getTopCharList :: [Char] -> Maybe Char

getTopCharList [] = Nothing

getTopCharList (x:xs) = Just x

Knowing this, the above-mentioned built-in list type could then also be represented by the following datatype definition *(though this is not exactly accurate since Frege/Haskell provides several syntactic constructs to ease working with the lists and also their internal implementation allows for random item access)*:

data List a = EmptyList | Constant a (List a)

On the other hand, we may also define a type synonym. Though similar in certain ways to datatypes, they only wrap a more complex type into a single name. Usage of type variables is also possible.

type Stack a = [a]

pop :: Stack Int -> Stack Int

pop [] = []

pop [x] = []

pop (x:xs) = xs

**Function type**

Functions can also be used as function arguments. Consider, for instance, a list of integer numbers. We may want to apply a certain function to each item of the list (for example, increment it by 1) and return a new list of integers this way. The function applied is unknown to us beforehand, but we can still implement the ‘mapping’ function. We can do it this way:

map :: (Int -> Int) -> [Int] -> [Int]

map \_ [] = []

map ff (x:xs) = (ff x) : (map ff xs)

The function argument is denoted in the annotation by (Int -> Int) . This represents a function accepting a single integer argument and returning a new integer. We then apply this function ‘ff’ to each item of the input list and return the new list.

An important aspect of type annotation is providing an interpreter with ‘unknown type’. We may want to implement the mapping function for all list types, not just the list of integer numbers. Since the implementation of the function is completely equivalent to the mapping function above, the only change is in the function’s annotation:

map :: (a -> b) -> [a] -> [b]

What this says is that the function ‘map’ accepts a function accepting an argument of a certain type (let’s call it type ‘a’), and returning an element of a (possibly) different type (‘b’). Then ‘map’ accepts the list of items of the type ‘a’ and returns a list of items of type ‘b’.

In Frege-IDE, there is only a limited support in typesystem for handling ‘unknnown types’, but it is possible to tell the IDE to perform only a high-level check of the provided arguments *(in this case that the first argument is an unary function and the second and the result of the ‘map’ are lists)*.

**Operators**

Both Haskell and Frege provide a lot of flexibility when it comes to infix operators. There are several standard, built-in, operators, like arithmetic addition (+), subtraction (-), comparison operators (==), etc. It is, however, possible to define almost any custom operators consisting of allowed symbols. These include: # $ % & \* + . / < = > ? @ \ \ ^ | ~ : -

For example, a custom operator +++ for adding two integer numbers while also incrementing the result by 1 may be implemented as follows:

(+++) :: Int -> Int -> Int

a +++ b = a + b + 1

We can then use the newly defined operator in a function:

ff = 2 +++ 3

Since operators are basically only binary functions, there are no major differences between the two. The annotation differs only in obligation to wrap the operator inside brackets.

*(Note that there are several different syntactical ways of providing an operator implementation, but we show here only the supported variant in Frege-IDE.)*

If not specified, the custom operator is by default non-associative and has a precedence of 16. [REF] To change that, we can apply the following statement:

infixl 5 +++

This forces our operator to be left-associative with precedence 5. (The precedence must be an integer number between 1 and 16.) Three modes of associativity exist:

1. left-associativity (statement begins with keyword ‘infixl’)
2. right-associativity (statement begins with keyword ‘infixr’)
3. non-associative operator (statement begins with keyword ‘infix’; there must not be several operators used in a single expression, with exception of using brackets or changing precedence of a sub-expression in any other way)

A combination of several operators with both types of associativity (left and right) with the same precedence in a single expression is not possible and results in a compilation error.

Function application, using constructors, or brackets, have all higher precedence, than 16. This means that in the following expression, ‘ff 2 3’ gets evaluated first regardless of the actually specified precedence of +++ operator:

1 +++ ff 2 3 +++ 7

If we were to specify +++ operator as an argument for a function ff, we would have to put it inside brackets:

1 +++ ff 2 3 (+++) 7

This expression evaluates ‘ff 2 3 (+++) 7’ first, and only then continuing with the remaining part.

**Currying**

Currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument. [WIKI] In Haskell and Frege, a function may use for its implementation another function, while providing only some of its arguments. The technique may be demonstrated by the following example:

multiplyThree :: Int -> Int -> Int -> Int

multiplyThree x y z = x \* y \* z

multiplyByEighteen = multiplyThree 2 9

In the example above, it is interesting to note that we do not have to provide in the implementation of ‘multiplyByEighteen’ any arguments, i.e. this is redundant *(the main idea behind the technique)*:

multiplyByEighteen x = multiplyThree 2 9 x

Currying may be used also with operators, and also applies in cases of a partial function application, as demonstrated by the following example:

max :: Int -> Int -> Int

…

six = (max 4) 6

Expression ‘(max 4)’ applies the function ‘max’ only partially, resulting in a function with annotation ‘Int -> Int’. This is then applied again for argument ‘6’, resulting in a constant integer.

**Where, let**

It is possible to create ad-hoc definitions inside a function definition. Such definitions may be placed inside ‘where’ or ‘let’ code block. The scoping rules prevent the functions created this way to pollute the working namespace, which is useful for creating reusable modules.

The following are 2 implementations of a function, which describes length of a given list:

describeListWhere xs = "The list is " ++ what xs

where

what [] = "empty."

what [x] = "a singleton list."

what ys = "a longer list."

describeListLet xs = "The list is " ++ let

what [] = "empty."

what [x] = "a singleton list."

what ys = "a longer list."

in what xs

frege> describeListWhere [1]

The list is a singleton list.

In the example above, both in ‘let’ and ‘where’ we define a new function, ‘what’, accepting a single list argument. The function ‘what’ cannot be used outside the functions in which it is defined, however. (‘what’ in describeListLet is a different function than the one in describeListWhere)

When working with ‘where’ and ‘let’, a correct indentation is important. In ‘let’, the subsequent function definitions have to be aligned with ‘in’ statement, or have a greater indentation. A similar rule should be kept when working with ‘where’ block, where each definition should be aligned with ‘where’ keyword, or have a greater indentation.

**If, guards**

A similar concept to ternary operator (?:) in many imperative programming languages, which defines a conditional expression, is ‘if’.:

doubleSmallNumber x = if x < 100 then 2 \* x else x

It returns one of the two expressions based on whether the given condition evaluates to true or false. This is an important feature for when a simple pattern matching is not enough, as demonstrated by the example. *(We cannot just enumerate each integer number to provide pattern matching for ‘small’ numbers.)*

A similar concept to ‘if’ are guards. They could be thought of as several ‘if-else’ constructs, but with a rather more readable syntax:

sign x

| x < 0 = (- 1)

| x > 0 = 1

| otherwise = 0

The example above is a standard implementation of signum function. [WIKI-LINK] The ‘otherwise’ keyword denotes a condition that always evaluates to true, given the conditions above that guard evaluated to false. (Therefore the ordering of the guards is important – it makes no sense to put guard with ‘otherwise’ above any other guard.) Unlike a series of ‘if-else’ statements, the ‘otherwise’ guard may not be present. (In a scenario where a user invokes the function for an undefined condition, a simple ‘NoMatch’ exception is thrown – just like when no pattern could be matched.)

**Case**

Case expression is a very similar mechanism to pattern matching. It basically allows to ‘pattern-match’ when already inside a single pattern. To elaborate, consider the following example:

head :: [Int] -> Int

head xs = case xs of

[] -> 0

(x : \_) -> x

The function returns a head of a given integer list. If the corresponding list is empty, 0 is returned instead. The function could be easily rewritten by using standard pattern matching, but in certain scenarios, where we would need to define patterns for several arguments at once, approach with using case expression may be more readable and maintainable. For instance, we could rewrite our function charToName to return a name based on the first character from an input character list. We only have to make sure that the list is not empty, first:

firstCharToName :: [Char] -> String

firstCharToName [] = "No input provided."

firstCharToName (x:xs) = case x of

'a' -> "Albert"

'b' -> "Broseph"

'c' -> "Cecil"

\_ -> "No name"

Note that the usage of the wildcard ‘\_’ is also possible.

**Import, export**

To use functions, operators and data types from other modules, we must first import them. An import statement has the following form:

import frege.prelude.Math (\*\*, log)

Import declarations are processed in the order they occur in the program text. However,

their placement relative to other declarations is irrelevant. Nevertheless, it is considered good

practice to write all import declarations somewhere near the top of the program.[REF]

There are several things to note about import here. First, we always import all of the (exported) definitions from one module into the current workspace. The only thing we can change here is whether we have to use qualified names to reference the corresponding definitions.

Case 1: We import everything from a module into the current namespace

import frege.prelude.Math

In this case, we can e.g. reference the Archimedes’ constant in a function definition directly by writing:

circumference r = 2 \* **pi** \* r

Case 2: We import only specified parts from a module into the current namespace

Let us consider the example from above:

import frege.prelude.Math (\*\*, log)

In this case, we import into the current namespace only operator \*\* and function log. To reference the Archimedes’ constant, we now have to write:

circumference r = 2 \* **Math.pi** \* r

We use only the last ‘part’ of the module’s name, i.e. Math (frege.prelude is omitted).

Knowing this, we may force using the module’s name for each referenced definition by importing completely nothing into the current namespace:

import frege.prelude.Math ()

In a case where two imported modules’ names would collide, we can use ‘as’ clause to specify an alias for a given module:

import frege.prelude.Math AS MM ()

…

circumference r = 2 \* **MM.pi** \* r

Case 3: We import all parts from a module into the current namespace except the specified ones

This is done by using ‘hiding’ clause:

import frege.prelude.Math hiding (pi)

Now, for the Archimedes’ constant ‘pi’ we have to use the qualified name, but for everything else we do not have to:

circumference r = 2 \* **Math.pi** \* r

rightTriangleC a b = **sqrt** (a \*\* 2 + b \*\* 2)

It also should be noted that we may import only certain constructors from a given datatype into the namespace. This makes sense from a point where 2 or more constructors could potentially have the same name, but come from different sources. Consider a module ‘ExampleTree’ with the following datatype:

data Tree = Nil | Node Int (Tree) (Tree)

We may import none of the constructors, all of them, or only the specified ones:

-- imports none of the constructors, i.e. we have to reference them by using ET.Node, ET.Nil

import mps.frege.ExampleTree as ET (Tree)

-- imports only the constructor ‘Nil’ into the current namespace

import mps.frege.ExampleTree as ET (Tree(Nil))

-- imports all of the constructors into the current namespace (2 ways)

import mps.frege.ExampleTree as ET (Tree(..))

import mps.frege.ExampleTree as ET (Tree(Node, Nil))

All modules by default import the standard module ‘frege.Prelude’. It contains standard arithmetic operators (+, -, \*, /), comparison and Boolean operators, many datatypes (e.g. Maybe), classes and instances we do not mention in this work and a lot of other definitions. For a complete list, see <http://www.frege-lang.org/doc/frege/Prelude.html>

The similar mechanism is also used for exporting definitions from a module. Not exported definitions may not be referenced at all.

To export all of the definitions, we use:

module Hello where

…

To export only the function ‘greeting’, we can write:

module Hello (greeting) where

…

We may, however, want to export definitions of an imported module as well. For this, we use the keyword ‘module’:

module Hello (**module frege.prelude.Math, module Hello**) where

import frege.prelude.Math (\*\*, log)

Now upon importing module ‘Hello’, we would also import indirectly the module **frege.prelude.Math.**

Note the usage of an item **module Hello.** We used that to denote the exporting of all definitions from the current module ‘Hello’, otherwise only the definitions from the module **frege.prelude.Math** would get exported.

Please, be aware that based on how the platform MPS works, it is, unfortunately, not possible to work with the modules and libraries written in a standard text editor. These have to be rewritten in a specific MPS project *(Frege-IDE)* to make them usable in the corresponding project. That is why we offer in this work only a limited support of the standard module ‘frege.Prelude’.

**Further reading**

Frege and Haskell include many other aspects that are not mentioned in this work. Due to the scale and complexity, most were not implemented, though for an interested reader we recommend visiting the following pages:

Reference: [REF]

[***http://www.frege-lang.org/doc/Language.pdf***](http://www.frege-lang.org/doc/Language.pdf)

*Frege Wiki:*

[***https://github.com/Frege/frege/wiki/\_pages***](https://github.com/Frege/frege/wiki/_pages)

*Frege goodness:*

[***https://dierk.gitbooks.io/fregegoodness/***](https://dierk.gitbooks.io/fregegoodness/)

* Chapter 4: Frege in MPS
  + *Subset related to what we decided to support in “Frege-IDE”*
    - *(Analysis, what parts of the grammar we decided to cut off, what couldn't be cut off, hot it relates to everything)*
  + *Grammar transformation for MPS structure aspect, design of the Frege structure, how and why (analysis + design decisions)*

Frege has rather many syntactic (and semantic) constructs for this work to be able to include them all. We have therefore focused our attention only on the most important features worth implementing, such as function definition and annotation, operators and custom datatypes. Our ideal IDE will have a user-friendly editor that will emulate normal text editing and writing code in the way that most Frege and Haskell developers are used to. This should be accompanied by a context help (sometimes referred to as ‘code completion’), which would allow for referencing already defined functions, operators, variables, etc. Last, but not least, we will strive for a type checker, which would be able to find small mistakes in the code, such as calling a function with illegal arguments, or evaluate type of an expression.

Before delving deeper into this chapter, we recommend reading the chapter 3, which describes Frege syntax and features of the language we will implement.

**Supported subset of Frege**

In this work, we focus our attention only on the most important parts of the Frege language which gained its popularity (or, rather, popularity of Haskell). For the most part, so called ‘syntactic sugars’ are omitted, as well as monads, which make Frege appear less of a functional and a more of an imperative programming language. To also include more advanced features, like context help (references) and type system checking, we also had to keep the complexity of the work reasonably small and thus concepts like classes and instances were omitted as well.

We will now recapitulate the features we will implement, from the high-level point of view.

Generic program structure

A program in Frege has to be properly structured. We expect a module to have a header, depicting its name. We will also try to emulate recommended practices of writing programs in Frege by, for instance, forcing export and import declarations to be written in the top of the program. Since a program is then just a series of definitions, we leave the rest up to the user of the IDE. (Note that we do not intend to implement monads section of the language, so we do not care about the definition of ‘main’ function.)

Import and export

To demonstrate the capabilities of the MPS platform regarding the scoping and code completion area, we intend to implement importing and exporting features of the language, as they are depicted in chapter 3, i.e. allowing a user to reference functions, custom algebraic datatypes, datatype constructors, and operators. The imported module may or may not be aliased (by using ‘as’ clause), and the corresponding statements will occupy top of the program’s layout.

Comments

Comments should be easily applicable, where all of the normal definitions are expected.

Function annotation

We allow a user to specify a type of a function. This will also play a role during the type checking and evaluation, where we will have an easier job to infer the type of a function and its arguments.

Function definition

We must provide a way for a user to define a new function. A function may accept zero or multiple arguments, it consists of the patterns (refer to the chapter 3 – pattern matching), and a ‘right side’ for each pattern definition.

Operator definition

Infix operators are also just functions, but strictly accepting 2 arguments. (Though a result of an operator application may be again a function accepting additional arguments.) An operator definition should be able to populate the namespace with new operators, allow these new operators to be used in expressions and allow the operators to be annotated in the same way a regular function can be.

infix/infixl/infixr

The statements beginning with the ‘infix\_’ keywords specify an operator’s precedence together with its associativity. These have a rather larger impact on the type evaluation of expressions consisting of infix operators, which we will look later in this chapter.

Custom algebraic datatypes

Another applicable statement in a module, ‘data’ declares a new algebraic datatype, which can be later used in functions. The name of a new datatype becomes a new type, whereas the constructors become new values of that type.

Type synonyms

Similar to datatypes, these statements, however, only introduce a new name for a (usually) more complex type.

Standard types and literals

Boolean, character, integer and double are types that are all part of the standard Frege library. Though the standard library defines several other types (such as Float, Decimal, etc.), these were selected due to their prevalence and representative status. Additionally, type ‘String’ is also supported with the standard syntax of using quotation marks, i.e. “Hello, world!”.

Tuples

Standard tuples should be supported.

Lists

There are 3 main ways of defining a list:

* Enumeration ['a', 'b', 'c']
* Range ['a'..'z']
* List comprehension [x | x <- ['a'..'z']]

Due to list comprehension being a little more complex, we will provide only a limited support.

Additionally, however, we have to account for a usage of an operator ‘:’ regarding the lists, which attaches a single list element (head) to the rest of the list (tail): [1] : [[2, 2], [3, 3, 3], [4, 4, 4, 4]]

Function type

In addition to the above, there is another standard type which covers the type of a function. A function accepting two integer arguments and outputting a string has the following type:

Int -> Int -> String

Then, a function which accepts the mentioned function as its single argument and outputs, for instance, a list of strings, has the following type:

(Int -> Int -> String) -> [String]

Where

‘where’ clause allows to provide additional ‘ad-hoc’ definitions inside a definition. These are then visible only to the closest outer definition, like in the following example:

five = 1 + four

where

four = 1 + three

where

three = 3

*(In the example above, the constant function ‘three’ is not visible in the right side of the function ‘five’)*

Similarly to export/import features of the language, we aim to demonstrate the scoping capabilities of our future IDE.

Let

Implementation of let will be similar to that of the ‘where’ clause.

Guards

These provide an alternative way to define a function to the standard ‘assignment definition’ (f = “value”), where each ‘guard’ contains a boolean condition for whether its branch should execute (an analogy to a series of if-else statements).

Case

Case expression allows to ‘pattern-match’ when already inside a definition of a pattern. The most important thing to consider during the implementation will be its scope-providing capabilities, because it may introduce a new variable:

head xs = case xs of

[] -> “x does not exist here”

(x : \_) -> “x does exist here and its value is ” ++ show x

Additional concepts

From the standard language we should also include the statement ‘if’ and definition of lambda functions (anonymous functions defined inside a definition). These together with the above should cover most of the standard usage of the Frege language (not accounting for classes and instances).

Type checking and evaluation

We will implement a simple type checking capability to our IDE, which will infer types of certain expressions and compare types of function definitions to their annotations. Providing a user with a complete type-checking capabilities would require a lot of resources and time, therefore we only intend to implement a rather restricted type evaluation, capable of handling only certain scenarios. The system should be, however, easily extensible and robust enough to demonstrate the potential of the MPS platform.

**Structure**

[Sources: <http://www.frege-lang.org/doc/Language.pdf> [REF] <https://github.com/Frege/frege/blob/master/frege/compiler/grammar/Grammar.ebnf> [GRM] ]

Defining concepts in the structure aspect in MPS for our IDE is the most important part of this work. The concepts are ‘building bricks’ when it comes to working with AST. Every other aspect of our IDE will depend on this part, therefore we have to do a careful analysis.

Working with structure in MPS to a certain extent resembles defining a grammar of a language for a compiler parser *(syntactical analysis)*. To implement it as correctly as possible, we should understand the Frege grammar and how its different parts relate to the actual features of the language.

In this chapter (part?), we are going to delve into certain parts of the Frege grammar, show, what actual features they correspond to and how we transformed them into the MPS concepts. A complete analysis would far exceed the scope of this text, so we will focus only on the most important or otherwise interesting parts we had to deal with.

During our analysis, we used materials from [Link-GRM], which contains an actual grammar used in the official Frege compiler, and the language reference from [Link-REF]. (*The resources are also provided in the appendix.)* The grammar uses mainly the extended Backus-Naur form [WIKI] notation, which we also use in this work. Explanation of the notation is provided below.

Explanation of the EBNF notation

The notation used throughout this work is based on the grammar from [Link-GRM].

*TODO: set colors and/or font and style for distinguishing.*

Non-terminal symbols: e.g. module

Terminal symbols: in apostrophes, e.g. ‘->’

Symbols for regular expressions:

expression\* - token “expression” repeats 0..n times

expression+ - token “expression” repeats 1..n times

expression? - token “expression” may be used 0..1 times

(expression1 expression2 expression3)? - tokens “expression1 expression2 expression3” may be used 0..1 times in exactly this order, whereas all of them have to be used or none

Syntax of the used notation:

token ::= ‘a’ | ‘b’ | ‘c’ – non-terminal symbol ‘token’ may be rewritten to one of the following terminal symbols: ‘a’, ‘b’ or ‘c’

token ::= ‘a’ | ‘a’ token – non-terminal symbol ‘token’ may be rewritten to either a terminal symbol ‘a’, or the terminal symbol ‘a’ and non-terminal symbol ‘token’; i.e. here we see an example of a right recursion – the token ‘token’ generates a sequence of symbols ‘a’ of arbitrary lengths (‘aaa’, ‘aaaaa’, ‘a’, ‘aaaaaaaaaaaa’, …)

Abbreviations: NTS (non-terminal symbol)

Please, note that in many cases we try here to simplify the grammar and omit the irrelevant parts not included in the final project. If a major part is omitted, it is mentioned explicitly.

Generic program structure

A high level view of the module definition is described by the following NTS rule:

module ::= (moduleclause (';' definitions|'where' '{' definitions ‘}')|'{' definitions '}')

Normally the places where a proper indentation is required may be replaced by usage of curly brackets ‘{’, ‘}’ and a semicolon ‘;’. Consider the following example:

foo x =

let s = sin x

c = cos x

in 2 \* s \* c

Which may be rewritten like this:

foo x = let { s = sin x; c = cos x; } in 2 \* s \* c

Parser in most of the compiler implementations works with the second variant, while most of the Haskell and Frege programmers use the style with indentation (which we have actually stuck to in this work). The process of converting the first to second is normally done during the lexical analysis.

Knowing this, the rule for ‘module’ NTS is clearer now. After we define the module and its qualified name, we may or may not use the keyword ‘where’, though it is not otherwise necessary to denote the separation from the rest of the program. After that the module is just a series of definitions. We will demonstrate the NTS rule on the example from chapter 3:

*moduleclause (::= 'module' modulename …) 'where'*

module Hello where

*'{' definitions ‘}'*

greeting friend = "Hello, " ++ friend ++ "!"

main args = do

println (greeting "World")

Before actually implementing a concept in our IDE responsible for the ‘program structure’ (a.k.a. a root concept), we will have a look on ‘definitions’ NTS.

Definitions

‘definitions’ is just a series of ‘definition’ NTS. Though, to be considered legal, there has to be at least 1 in a single module.

definitions ::= definition (';' definition)\* ';'?

A single definition is a substitute for: [REF - pg. 47]

* Import declaration – what is to be imported into the module
* Fixity – specifies associativity and precedence of an infix operator
* Type declaration – declaration of a type synonym
* Data declaration – allows to create custom datatypes
* Class declaration (*omitted in this work)*
* Instance declaration (*omitted in this work)*
* Local definition – annotation and function definition

We have mentioned that it is a good practice to include the import statements in the top of a module definition, before any other statements. We can enforce this by not following the exact Frege grammar, but rather implement our own version of the ‘module’ NTS. It will have to be a root concept (responsible for the overall program structure) with the following children:

* module – corresponds to the first line in the program structure example above; ‘moduleclause’ NTS
* import – corresponds to the import statements, which are part of the ‘definition’ NTS
* definitions – rest of the statements that are part of the ‘definition’ NTS

A possible implementation of the concept represents Figure 4.1 (we named the concept ‘Skeleton’).

Notice that the concept must act as a root concept *(‘instance can be root’)* to be usable.



Figure 4.1.: Skeleton concept implementation

Import

According to the Frege reference, the following rule applies for the importing statement [REF-PG84]:

import ::= 'import' packagename (‘as’? namespace)? ‘public’? importlist?

importlist ::= ‘hiding‘? ‘(‘ (*importitem (*‘,‘ *importitem)\*)?* ‘)‘

This clearly represents the import statement as it was described in chapter 3:

import mps.frege.ExampleTree as ET (Tree(Nil), ->>, traverse)

Unlike in the official Frege compiler implementation, we did not include the visibility *(‘public’ keyword – not used throughout the work)* and aliasing of imported items. (We intend to implement only the aliasing of the imported module.)

The ‘importitem’ NTS represents a function, operator, type, datatype, or a class. Simplified, we get the following rule:

importitem ::= VARID | OPERATOR | CONID(‘(‘ (*member (*‘,‘ *member)\*)?* ‘)‘)?

The distinction between the NTSs are as follows:

* VARID: identifiers beginning with a lowercase symbol (a-z) or an underscore (*functions, variables, type variables, ...)*
* CONID: identifiers beginning with an uppercase symbol(A-Z) *(types, datatypes, constructors, classes, etc.)*
* OPERATOR: a sequence of symbols an operator can consist of (refer to chpt. 3 - Operators)

A ‘member’ enumeration (CONID(‘(‘ (*member (*‘,‘ *member)\*)?* ‘)‘)?) in this case applies only to datatypes and classes (which are both ‘CONID’ tokens according to the Frege grammar). Even though classes are omitted in this work, we allow datatype constructors to be enumerated inside the brackets. Thus for our purposes, a ‘member’ may be only either a ‘CONID’ token (which covers the constructors), or a special symbol ‘..’ denoting an import of all constructors. (Again, refer to the chapter 3 – import/export.)

Now, to implement the feature, we can mostly follow the official grammar. We create a concept ‘Import’, which has 3 parts:

1. It references an existing module’s name
2. It can (but does not have to) contain ‘as’ clause to allow the imported module’s aliasing
3. It can (but does not have to) contain a list of imported items into the current namespace

An example of such implementation is captured on Figure 4.2. Note that terminal symbol ‘import is not a part of the ‘Import’ concept, but rather of its editor. (It only tells us what the concept should look like, but we do not need it for parsing.) Additionally, we may define the concept’s alias to be ‘import’. This only tells MPS to create an instance of the concept in places where a user types ‘import’ *(our alias).* The concept must be, however, expected in those places (we cannot create an instance of the concept where e.g. an instance of a ‘Module’ concept is expected).

In Figure 4.2., the implementation of the ‘Import’ consists of children concepts ‘ImportAs’ and ‘ImportItems’. ‘ImportAs’ wraps (‘as’? namespace)? part of the ‘import’ NTS rule, whereas ‘ImportItems’ is a (relatively) straightforward implementation of ‘importlist’. Regarding the reference of ‘Module’ (‘packagename’ NTS), this is only one of the options. We need to tell the MPS that we will allow importing of items only from an existing module. *(Understandably, as mentioned in chapter 3, only the modules defined in MPS using our language Frege-IDE, will be importable.)* Since a generic program structure captured in concept ‘Skeleton’ (Figure 4.1) consists of exactly one ‘Module’ child, this a legal approach.



Figure 4.2.: Import concept implementation

Now while ‘ImportItems’ concept may be clear, it consists of 0..n children of ‘ImportItem’ (represents ‘*importitem‘* NTS). We have already showed the NTS rule for the ‘*importitem*’ NTS, but regarding the corresponding concept implementation, we will want to take advantage of references. In this work we know that we may only import either functions, operators, type synonym, datatypes, or datatype constructors. We may therefore create an abstract concept ‘ImportItem’, which is otherwise empty. Then we create several inherited concepts:

* ‘IIFunction’ (for importing functions)
* ‘IIOperator’ (for importing operators)
* ‘IIType’ (type synonyms and datatypes)
* ‘IIConstructor’ (importing constructors)

Why is it enough to create only 1 concept as a substitute for importing both type synonyms and datatypes? The reason is, again, the grammar – there is basically no syntactical difference between the two since they both define a type. Thus, where there is the former, there can also be the latter.

However, so far we have not created any of the concepts from above (meaning there is no concept for defining functions, operators, etc.), so we cannot provide a complete implementation right now, because there is nothing to actually reference. But, we can create, so called, ‘smart references’ for each of the entity to reference, and implement these later. This will prove useful also in the later stages in the implementation, because there are many places we will need to re-use a reference for functions, operators, etc. We will demonstrate our reasoning on the ‘IIType’ concept.

‘IIType’ may reference a type synonym or a datatype. Regardless, both are types, so we only need to create a single smart reference. We will call it ‘TypeReference’ and can leave its implementation empty for now. The smart reference can then be used as a child for the concept ‘IIType’, which covers the referencing part. ‘IIType’, however, also includes the optional datatype member enumeration. We will simply encapsulate the feature into a new concept, ‘IITConstructorList’, which will be its optional child. The implementation, unsurprisingly, is very similar to that of the ‘ImportItems’ concept, hence we will not describe it.

The implementation of the ‘IIType’ concept is shown on the Figure 4.3. An optional and slightly more advanced feature here would be a correct scoping for the constructor list. For example, for a datatype ‘Maybe’, we could want our IDE to allow only to reference constructors “Just” and “Nothing”: data Maybe a = Just a | Nothing

This feature is described in the later part/chapter, **scoping**.



Figure 4.3.: IIType concept implementation (notice the ‘extends’ clause)

Function definition

Possibly the most complex part of the Frege language, function definition covers the area of defining new functions, operators and certain special expressions for defining constants (e.g. (a, b, c) = (1, 2, 3) ).

On its highest level, we could divide it into 2 parts: left and right side. To elaborate, when providing a definition, we always specify a pattern of a sort and an expression corresponding to that pattern. We can see it on the following example from chapter 3:

getTop (x:xs) = x

Here, the statement defines the function ‘getTop’, which returns a first item of a (non-empty) string list. In this case, we have a left side consisting of ‘getTop (x:xs)’, which describes the name of the function and its arguments. On the right side, we have a simple expression ‘x’ referencing an argument from the left side. A slightly different example is the definition of a signum function:

sign x

| x < 0 = (- 1)

| x > 0 = 1

| otherwise = 0

Similarly to the previous case, ‘sign x’ could be considered a ‘left side’ of the function, whereas the rest of the definition the ‘right side’. *(We understand that this is not a completely official convention, but it will help us navigating in the relatively complex Frege grammar)*.

This corresponds to the actual Frege grammar, where the high-level function pattern binding is defined as:

binding ::= lhs rhs

rhs ::= ‘=’ expression (‘where’ declarations)?

| guardedExpressions (‘where’ declarations)?

*(lhs = ‘left hand side’, rhs = ‘right hand side’)*

Left hand side

An experienced Frege / Haskell user knows there are 3 main ways of defining a function:

* Standard function definition, e.g. multiply x y
  + multiply x y = x \* y
* Operator definition, e.g. x :-: y
  + x :-: y = x + y + 1
* Any pattern definition, e.g.: (a, b, c)
  + (a, b, c) = (1, 2, 3)

The chapter 3 does not spend much time explaining the third option, so we will go through the matter briefly. It is completely legal to define a constant function f like this:

[2, f] = [2, 3]

In this case f has a value of 3. However, the following definition is also legal:

[1, f] = [2, 3]

In this case, though, an exception is given upon an attempt to evaluate f in any way.

This means that we can define constants in some interesting ways, like so:

(f1, f2, f3) = (1, 1 + f1, 1 + f2)

On the left side, it is expected a user provides a legal pattern *(the grammar is described below)*, so any combination of variables, types and literals is supported, such as tuple of lists, list of tuples, presence of literals, usage of constructors, and so on. The right hand side must correspond to the pattern provided on the left, but in regards of the grammar, there are (almost) no restrictions. (The compiler will try to evaluate the type of the right side and compare it with the used pattern, however this process is done during the semantic analysis.)

What does the grammar say about the left hand side of the function definition?

lhs ::= VARID patternTerm\*

| patternTerm OPERATOR patternTerm

| pattern

pattern ::= listPattern

| patternTerm

listPattern ::= patternTerm ‘:’ listPattern

| patternTerm

patternTerm ::= VARID

| ‘\_’

| literal

| ‘[‘ (pattern (‘,’ pattern)\*)? ‘]’

| ‘(‘ (pattern (‘,’ pattern)\*)? ‘)’

| CONID (pattern (‘,’ pattern)\*)?

(REF-PG67+40) We have simplified the actual grammar due to not supporting certain features, such as word-like operators (e.g. x ‘plus’ y = x + y) or argument capture; other changes were made for clarity.

We can see in the rule definition of NTS ‘lhs’ that it corresponds to the 3 ways of definition mentioned earlier. ‘patternTerm’ allows us to either:

* define a new variable (VARID): (**a**, **b**, **c**) = (1, 2, 3)
* to use the wildcard (‘\_’): charToName **\_** = "No Name"
* to use a literal: factorial **0** = 1
* to use a list of patterns separated by a colon, or just an empty list: getTop **[]** = "No elements"
* to use empty brackets ‘()’
* to put a single pattern inside the brackets

to use a tuple of patterns, in which case at least 2 patterns have to be inside the brackets: second **(\_, x, \_)** = x

* to apply a datatype constructor (CONID) to optional patterns: surface (**Circle \_ r**) = pi \* sqr r

Additionally, ‘listPattern’ serves to match a list inside patterns, as in the following example:

getTop (**x:xs**) = x

We may, however, specify any number of the first list items, so the following example is also legal (but unlike the example above, this requires the list to have at least 3 items to successfully match):

getTop (**item1 : item2 : item3 : tail**) = item1

In the final work, though, we made several additional simplifications to ease us the work in typesystem and other aspects. Mainly, we require each ‘listPattern’ to be enclosed within brackets. This bears no semantic restrictions and is an actually recommended practice, because it makes the code easier to read. Furthermore, empty brackets make no sense in the restricted set of features we provide, therefore they were not included either.

Important implementation thing to note relates to the datatype constructor application. We will again use the notion of references to allow only the application of existing constructors. This would be implemented by creating a concept acting as a smart reference. The smart reference can, however, only point to a constructor, and we also need to allow specifying arguments. Therefore the smart reference has to be enwrapped inside a new concept (we will name it ‘PConstructor’), which will additionally contain [0..n] ‘pattern’ children acting as the arguments.

In the official Frege compiler implementation, ‘lhs’ is syntactically no different from expressions, since it can be considered as a restricted subset of the latter. The compiler only implements additional checks during parsing. However, in our IDE, it is best to use the grammar mentioned here, since it greatly eases our work with the other aspects of the MPS, especially with the editor and typesystem.

Right hand side

The right hand side of the function definition can be either a single expression optionally followed by ‘where’ declaration, or a series of guards optionally followed by the ‘where’ declaration.

‘where’ declarations are simply function definitions and annotations. These are described later in this chapter.

Guards, in contrast to the single expression variant of function definition, consist of at least 2 expressions and in their most simple form a grammar for them looks like this:

guard ::= '|' expression '=' expression

Expressions

An expression consists of a series of binary expressions (infix operators with operands) and an optional type or ‘forall’ construct [GRM]:

expression ::= binex ('::' (forall|type))?

Even though ‘forall’ is linked to type declaration, it is a part of a more advanced feature in Frege and we will not include it. The ‘type’ NTS, on the other hand, will be described later in the chapter.

‘binex’ NTS represents a series of operands (‘topExpression’) separated by operators. A single operand may take one of the following forms:

* Case expression (TODO: provide reference to chpt. 3)
* Let expression (TODO: ditto)
* Conditional (if expression) (TODO: ditto)
* Lambda (TODO: ditto)
* A series of ‘primary expressions’ separated by a whitespace

To understand the last point, we have to look into the primary expressions first. Primary expressions include terms, monads (omitted in this work) and a usage of (not necessarily) qualified names, e.g. to apply imported functions, or operators. In Frege, terms are literals, lists and tuples; or in other words, values. This is why we can understand the last point as an application – of a function, operator, or a datatype constructor.

six = max 4 (3 + 3)

(In the example above, ‘max’ is a qualified name, being a part of ‘primary expression’. ‘4’ is a term. ‘(3 + 3)’ is again a part of ‘primary expression’, containing a high-level expression ‘3 + 3’.)

A special case of application is a usage of brackets. Consider the following example from the currying section of chapter 3:

six = (max 4) 6

‘max’ is normally a function accepting 2 arguments, but by putting it into a bracket with only one argument, we have effectively created a new function accepting only a single argument. This function then may be applied again, thus leaving us with what we will refer to as ‘brackets application’.

Thus, we can distinguish in our IDE between the different forms of applications and create specialized concepts for the matter. And if the first ‘primary expression’ is a term, it cannot be an application anymore, hence our IDE would not allow to specify any additional arguments.

Figure 4.4. show a concept hierarchy regarding the ‘Top expression’ in Frege-IDE. A ‘primary expression’ may either act as an ‘application entity’ (i.e. what is to be applied) or a term. If an application consists of several arguments, the concept ‘GenericApplication’ is to be used (contains children ‘ApplicationEntity’ and ‘primary expressions’ for arguments), which is a part of ‘Top expression’, thus not allowing mixing with other ‘primary expressions’.

Regarding the expression grammar, it is also interesting to note that the applications are syntactically bound to have the highest precedence regardless of the operators used in an expression. That is why to use an operator as a function argument, it has to be put inside brackets:

1 +++ ff 2 3 +++ 7 = 1 +++ (**ff 2 3**) +++ 7

1 +++ ff 2 3 (+++) 7 = 1 +++ (**ff 2 3 (+++) 7**)



Figure 4.4: Hierarchy for the TopExpression concept in Frege-IDE

Grouped representation

A problem with the function definition in Frege is that it may seem like we are providing several definitions for the same entity. Consider the example:

getTop [] = "No elements"

getTop (x:xs) = x

This is obviously due to the pattern matching mechanism. (For different versions of input arguments we are specifying different bodies of the same function.) In MPS, however, we will need to reference existing nodes in the created AST and this causes a problem. Let us imagine we are defining a new function, which uses the function from the previous example, ‘getTop’:

isEmpty x = getTop x == “No elements”

The question is, which instance of the function definition concept (regarding the ‘getTop’ definitions) does ‘getTop’ (function application in ‘isEmpty’) reference? The answer should be ‘all of them’. This can be achieved in MPS by creating a new concept, which wraps all of the function definitions of the same entity. Fortunately, even in the Frege specification, patterns for the same function have to be defined in one place, together. For instance, this is illegal:

length [] = 0

getTop [] = "No elements"

length (x:xs) = 1 + length xs

(The empty lines between the definitions are allowed, though.)

The ‘grouping’ concept should contain a child, which represents the function’s name, and 1..n patterns together with their right hand side. Furthermore, we will need to change the concept representing function patterns, where the function name is now not a child, but rather a reference.

This small change will need to be incorporated also for the operator definitions. In the third case, where in the left hand side we have ‘pattern’ NTS, there is no grouping necessary – we can define only constant functions this way, so providing different patterns makes little to no sense.

The corresponding concept in Frege-IDE is called ‘FDGrouped’ and was implemented in the similar way.

The last remark regarding the function definition may be, whether should not an annotation ‘create’ a name of the function, which will be later referenced from the function definitions. While this is definitely also an option, in Frege, we can have a function without an annotation whereas the other way around we would get an error during the compilation. We believe the approach we took and described in this chapter is more correct and conforms to the specification, which would not be the case in the latter option.

Types

Types are a vital part of the Frege grammar and play a role in the definition of a function annotation, or when declaring datatypes.

There are several ‘native’ types, which translate directly to the types from JVM. These include:

* Booleans
* Characters
* Strings *(unlike Haskell, in Frege Strings are not lists of character)*
* Numeric types *(In this work we only support Integer and Double)*

These types are part of the Frege-Prelude library and are implicitly imported to all modules. Besides the native types, there are also several built-in algebraic datatypes. We are only going to focus on Tuples and Lists, since these are practically non-implementable using only ‘data’ definition statements. (In theory, Lists could be implemented this way as described in chapter 3, but there are several, widely used, syntactical constructs, which would be missing.)

The remaining type is the function type (see chapter 3), used to denote types of functions.

Types connected to monads, exceptions, and standard datatypes *(e.g. Maybe)* easily implementable using ‘data’ definition statements, were not included in this work.

Before we delve deeper into the Frege types and how to implement them in our IDE, it is important to note that by omitting classes and instances, we lose an important aspect of the Frege language – parametric polymorphism. This feature allows to apply certain functions and operators on a ‘class’ of types, but not all of them. Consider for instance a built-in operator ‘+’. It allows to add any numeric values together. Its annotation would look like this:

(+) :: a -> a -> a

The problem is to specify that ‘a’ represents a numeric type. This is, unfortunately, not possible without classes and instances. We omitted the feature due to the size of the work.

By removing the class specification, we get the following grammar (simplified):

type ::= (typeApplication ‘->’)\* typeApplication

typeApplication ::= simpleType+

simpleType ::= VARID

| CONID

| ‘(‘ type ‘)’

| ‘(‘ type (‘,’ type)+ ‘)’

| ‘[‘ type ‘]’

On its highest level, we can specify a function type (NTS ‘type’). Each part of the function type consists of 1 or more ‘simple types’, which is denoted by ‘typeApplication’ NTS. As the name suggests, this part is responsible for application of a type. Consider the datatype ‘Maybe’, which consists of a single type variable. A type of a function getTopCharList (chapter 3) then looks like this:

getTopCharList :: [Char] -> **Maybe Char**

(The type name may be qualified.)

The ‘simpleType’ NTS rule denotes one of the following options:

* Usage of a type variable (VARID)
* Usage of a type name (CONID)
* Type in brackets *(May be required for syntactical reasons, but semantically the type of the expression is the same, as the type inside the brackets.)*
* Tuple type
* List type

Types are used in several places throughout the Frege grammar, but we will mention 2 main representatives: function annotation and declaration of new datatypes.

In function annotation, we provide a name of the function (or operator), we want to specify the type for. It is even possible to specify the type for several items at once:

(+), (-), (\*) :: Double -> Double -> Double

The right side of the definition corresponds to the types grammar described in this chapter (/ part?). Important thing to note here is the possibility to declare new type variables:

map :: (**a** -> **b**) -> [**a**] -> [**b**]

This is an important distinction from the datatype declaration, where on the right side, only existing type variables may be used:

data Tree **a** = Nil | Node **a** (Tree **a**) (Tree **a**)

(The left side of the definition declares a type variable ‘a’. This then can be used on the right side, where constructors are defined.) We will want to incorporate the behavior by using references in concept declaration.

Datatype declarations are, however, even more complex in the regards that they define a new type name, which then may be referenced (in the example above it is ‘Tree’). This is why in the corresponding implementation, the name of the declared datatype must be a standalone concept and not a property, because a new type name is also created by usage of the declaration of a type synonym. (type Stack = [Int]). Comparison of the concept implementation is shown on Figure 4.5.



Figure 4.5: Comparison of the type synonym and datatype concept implementation. Both of the concepts use ‘TypeName’ child to allow specifying the name, and can define new type variables that are then used by their respective right sides (represented by ‘equalTo’ and ‘parts’ children).

This also tells us how the concept ‘TypeReference’ metioned in the ‘Import’ part/chapter should be implemented. The concept should specify a reference to TypeName of cardinality [1]. There are no other children, nor properties defined in the smart reference.

As for the final implementation of the types, the distinction between datatype declaration and function annotation regarding the type variable usage forces us to take one of two paths:

* We can create two sets of concepts for types, each with a different implementation of type variable usage. In one set, only declaration of new type variables would be possible. In the second, type variables may only be referenced.
* We could implement two concepts inheriting from ‘simpleType’: one for creating type variables and one for referencing them. Using the Constraints aspect in MPS, we are then able to allow or restrict the concepts from occurring in specific subtrees.

The second option is less time-consuming to implement and poses fewer problems. First, it is syntactically correct (two different sets of concepts would mean there is a different grammar for the types in datatype declaration, than in function annotation, which is not true – the type variables are simply checked during semantic analysis.), and second, the actual type evaluation in typesystem aspect of MPS would be easier to implement. (Not to mention the maintainability of such a solution.)

What remains are the core types. For these, again, 2 options are possible:

* We can ‘hard-code’ the native types into the IDE, creating a concept inheriting from ‘SimpleType’ for each of the types mentioned at the beginning of the chapter/part.
* We can define the types using the datatype declaration statements inside a new module. This module will be implicitly ‘imported’ to all of the other modules a user will create (in the sense of providing a scope for its definitions and declarations).

While the second option may seem easier to maintain, there are actual benefits to the first approach, which have to do with typesystem aspect of MPS. Specifically, it will be difficult to link the type names to the types of the corresponding literals. Hence, we opted for the first approach.

**Editor**

Defining the editor for our IDE consists of defining visual appearance for each AST node *(in general)* and providing a user with a way of manipulating the AST in a user-friendly manner.

As mentioned, MPS is a projectional editor and thus it is not possible for a user to work with the code in text form directly. Every function, name, variable, type etc. is in some way associated with a specific AST node, which has to be presented to the user in some way.

In this chapter, we will show how the editor aspect in MPS allows us to tackle many different challenges we came across when implementing the UI part of our IDE. We will also present a couple of non-trivial problems we had to deal with, and what approach we took when solving them.

Nodes’ visual appearance

Appearance of the most of the concepts is rather straightforward to implement. We have already analyzed and implemented the structure aspect in the previous section, so it is relatively clear which concept is connected to which feature. Frege is a text-based programming language, thus creating concept editors involves only specifying a correct set of strings (constant cells) and/or correct set of other type of cells mentioned in the chapter 2 (property, child and referent cells) on correct places.

We will demonstrate the implementation on an example for datatype concept. A datatype declaration is represented by ‘Data’ concept inheriting from the ‘Definition’ concept (see Figure 4.5). It consists of a name, type variable declarations and a non-empty set of constructors. We can see an example of the concept editor implementation on Figure 4.6 *(at the bottom)*.



Figure 4.6: An example of a concrete AST as displayed in an editor, compared to the concept editor cells

In Frege, the datatype declaration has to begin with the keyword ‘data’. In our editor, it is a constant cell with a strictly defined string ‘data’. We may additionally apply editor styles to denote the keyword by changing its color and making the text bold. The keyword is then followed by the datatype’s name. As mentioned in the previous section, the name must not be a property, but rather a child node due to implementation of the type reference concept. In the editor, we simply use a child editor cell pointing to the editor of the ‘TypeName’ concept. Type variable declarations and a set of constructors are both lists of child nodes. For these we have to use a (horizontal) collection editor cell. They again, as in the case of the datatype’s name, refer to their corresponding concept editors, which are used when editing their values. The editor implementation for the concept representing a datatype constructor is depicted on Figure 4.7. We need to, however, also specify in inspector, that while the type variables are separated by a simple whitespace, the constructors actually need to use the vertical bar symbol |. The delimiters are added by MPS automatically when using the editor.



Figure 4.7: An example of a concrete constructor in AST (top) together with its concept editor implementation at the bottom

Side transformation menus

Side transformations allow us to change a concrete AST in a specific manner upon a user writing a certain text either right or left of an editor cell.

Let us take for example a list expression in right-hand side of the function definition. We have mentioned there are three main ways to define a list:

* By enumerating elements: ['a', 'b', 'c']
* By specifying a range of elements: ['a', 'c', .. 'z']
* By specifying a, so called, list comprehension: [x \* y | x <- [1..5], y <- [3..7]]

For each of the three methods there is a specific concept associated with the list definition. Now we may want to allow a user a seamless transformation from the first to second method upon entering .. right of the last element in the list. To do that, we need to know how both of the concepts are implemented.

Enumeration list concept is a simple term that contains [0..n] children of expression type. Its editor consists of two constant cells representing square brackets [ and ], and a (horizontal) collection cell representing the children items of the list. The range list concept is based on the enumeration list, but it requires to have at least one child right of the ..symbol. Additionally it contains [0..1] children of expression type right of the mentioned symbol, for specifying the upper bound. Figure 4.8. represents the concept editor implementation for the concept. The upper bound (upTo) is preceded by a question mark symbol to denote that the cell is to be displayed only if the upper bound is actually specified.



Figure 4.8: Concept editor for the concept representing a range list expression

The implementation of the transformation is rather straightforward. We put a default transformation menu on the concept associated with the list item (expression) with the following properties:

* Text triggering the transformation should be “..”
* Can execute part may be left to <always> to prevent the other texts from triggering the current transformation
* In execute part we specify that the new instance of the concept associated with the range list expression should copy the items from the former instance of enumeration list concept and then replace it completely. These are the items that will go before the ..symbol. In most cases the user actually wants to specify the upper bound as well, which is why we then create a new instance for the upper bound node and set the focus on the node.

The problem with the current approach is that the concept that triggers the transformation is ‘expression’. ‘Expression’, however, may be a child of many different concepts, not all of which are list enumeration concepts. We need to, therefore, include the relevant condition in the transformation menu. A complete implementation is depicted on Figure 4.9.



Figure 4.9: Default transformation menu for expression concept, which also acts as an enumeration list concept child item

Transformation menu inclusion pattern

As described in the previous section, we have to be careful what actions we assign to what concepts. If a concept may act as a child of several different parents, we have to allow the defined transformation menu actions only for the relevant ones.

There is, however, a different problem related to the parent-child relationship. Let us take a look at the ‘GenericApplication’ concept from the previous chapter [add ref]. It inherits from ‘TopExpression’ concept, so for all of the transformation menus defined for TopExpression to also work in ‘GenericApplication’, we have to include them. The implementation is illustrated on Figure 4.10.



Figure 4.10: Default transformation menu for ‘GenericApplication‘ concept

This, however, is not sufficient. ‘GenericApplication’ contains these children:

* [1] ApplicationEntity (what is to be applied, e.g. a function, operator, expression enclosed in brackets, etc.)
* [0..n] PrimaryExpression (arguments to pass for the application entity)

Its editor consists solely of the childrens’ editors and does not contain any constant, property or referent cells whatsoever (see implementation provided on Figure 4.11). Thus, all of the transformation menus have to be included in the children concepts, too.



Figure 4.11: Concept editor for ‘GenericApplication‘ concept

The idea to implement this for the children concepts is as follows:

* The parent of the current node has to be an instance of ‘GenericApplication’ concept.
* The current node’s concept editor has to be the right-most cell in the ‘GenericApplication’ concept editor. *(So, we are either an application entity and the arguments are empty, or we are the last argument in the corresponding ‘GenericApplication’ instance.)*
* If the above is true, we include the default transformation menu intended for ‘GenericApplication’ concept, and in cases when it is triggered, it will be applied to the parent node.

Figure 4.12 provides an example of such implementation for the concept ‘PrimaryExpression’.

To elaborate, ‘PrimaryExpression’ inherits from ‘TopExpression’, which means it has to include all of the transformation menus that were intended for ‘TopExpression’, first. Then, the type of arguments in ‘GenericApplication’ is ‘PrimaryExpression’, so we include the transformation menu for the case the current ‘PrimaryExpression’ node is the last argument in an application.

A similar approach would be used for the ‘ApplicationEntity’ concept.

Now, obviously, ‘PrimaryExpression’ has several subconcepts, all of which need to include its default transformation menu. And if there was a subconcept of ‘PrimaryExpression’ consisting of one or more children, whose editors would be used at the right of the current concept editor, the children would also need to include its default menu recursively with the similar approach we described in this chapter.



Figure 4.12: Default transformation menu for the concept ‘PrimaryExpression’, which also acts as an argument in ‘GenericApplication’ concept

We used this pattern throughout most of the implementation of transformation menus, since a direct implementation for concrete leaf concepts (which actually can trigger the transformation, as they carry the required type of editor cells, i.e. property, referent and constant cells) would be very difficult to maintain and prone to many mistakes.

Substitute actions

We use substitute actions whenever we need to allow a user to substitute certain AST nodes for another nodes. Mostly we need to allow a seamless substitution of an abstract concept to a concrete one. We will now take a look on how literals should work in our IDE and demonstrate the usage of substitute actions on them.

A literal is an abstract concept which inherits from ‘term’. It basically carries a simple value in expressions. It has these subconcepts:

* StringValue: Contains a single property of a string type. The value is enclosed with quotes “ in regards to how the editor looks like.
* CharValue: Represents a single character. The property is of a custom constrained type which allows only a single character to be typed, otherwise the node will not be validated correctly. Similarly to StringValue, in concept editor, the value is enclosed with apostrophes ‘.
* BooleanValue: Is an abstract concept. The editor has a single cell pointing to the content of the concept’s alias. This allows us to simply inherit from the concept, create two concrete concepts TrueValue and FalseValue with aliases ‘true’ and ‘false’ that will represent the actual value.
* IntegerValue: A concept with a single property. Its editor contains only a single property cell pointing to the mentioned property.
* DoubleValue: Same as IntegerValue, but the property is of a different type to allow floating-point numeric values instead of only the integer ones.

Now we may imagine a situation where there is a focus on an abstract expression AST node, i.e. the node is not set yet to anything at the moment. How will we enter any of the possible values?

In case of string and character, the situation is easy. We know that before entering a string value in an expression, a normal user would probably enter a quotation mark first, then enter the string value and finally finished with typing the right quotation mark. This means that quotation mark can serve as a trigger for entering string values. This is done by declaring alias of the StringValue concept to “. We will use the similar approach for CharValue concept. Additionally, it is clear that Boolean values will work the same way due to the aliasing mechanism. For these concepts, defining special substitute actions is actually not necessary.

IntegerValue and DoubleValue cannot be triggered by a concrete string as in the previous example. For these, we need to define a default substitute action, which will be associated with the Literal concept.

Since we are defining a substitute menu for the Literal concept, we have to include its subconcepts menus first, otherwise we would lose string, char and Boolean value triggers. Then, we define two substitute actions, one for integers and the second for floating-point numbers.

* In ‘create node’ part, we specify what node is to be created and that its property should be equal to the text entered by the user.
* ‘Matching text’ should return the text entered by the user, since there is no single string that could be associated with an integer or a floating-point number.
* ‘Can substitue’ will test the entered text for not being empty and whether it matches the regular expression pattern for entering integer or double values. To prevent the ambiguity between integers and doubles, we will also make sure, regarding the floating-point numbers, that the entered text does not match the regular expression for integer numbers and thus has to be, indeed, a double value.

The corresponding implementation is depicted on Figure 4.13. This approach not only handles the case of a seamless substitution of an abstract expression node to ‘IntegerValue’, but also a substitution of ‘IntegerValue’ to ‘DoubleValue’. For example, upon examination of the node ‘27’ in the following definition by pressing ‘alt + x’ we can see that the node is an instance of the ‘IntegerValue’ concept: myvalue = 27

Upon adding .1 to the expression, we can see that the node was substituted for an instance of ‘DoubleValue’ concept: myvalue = 27.1



Figure 4.13: A default substitute menu for the ‘Literal’ concept

To additionally prevent the DoubleValue and IntegerValue from populating the substitute menu when editing an expression AST node, we may define empty default substitute menus for them. This is, however, optional and bears no significant change on the usage of our IDE.

Wrap substitute menu

Wrap substitute menu is a special type of substitution menu, where we need to populate the menu (defaultly invoked by pressing ‘ctrl + space’) by instances of a different concept, than we will actually substitute the current AST node for.

To demonstrate, we will take a look at the left hand side of the function definition. We know from the previous chapter that it consists of patterns. According to the Frege grammar, we may replace NTS ‘patternTerm’ for a literal. Now, literals are part of the ‘Literal’ abstract concept which is part of the expressions. This means we have literals already implemented for the right hand side of the function definition (together with their corresponding concept editors and associated editor actions) and implementing the same concepts for the left hand side would be meaningless and difficult to maintain. To avoid the code duplication, we will create a new concept representing literals for the patterns: ‘PLiteral’. The concept inherits from ‘Pattern’ and contains a single child, which is the actual ‘Literal’ concept instance. When typing a value (e.g. an integer), we could expect the similar behavior as in the case of the actual literals in expressions, i.e. that the abstract ‘Pattern’ will get immediately substituted for the corresponding concept carrying the value (e.g. ‘IntegerValue’). That is, however, not possible without some work, since ‘Pattern’ may be substituted only for ‘PLiteral’ concept, and only then its child of an abstract ‘Literal’ type can be substituted for a concrete literal value. MPS, however, allow us to define a wrapper around the ‘Literal’ concept, to include all of the substitutions for ‘Literal’ concept into ‘Pattern’ as well, while automatically encapsulating such an AST node inside a new instance of ‘PLiteral’.

An implementation is depicted on Figure 2.18. As it can be seen, most of the work is automatically done by MPS platform itself, we only specify the handler, i.e. how to wrap the given node.

On a side note, regarding the ‘Literal’ concept, since now it can also be a child of ‘PLiteral’, according to our inclusion pattern, it should, among other things, also include the default transformation menu intended for ‘PLiteral’ (when it happens to be its child). If we do not include the menu, certain transformations would not be applicable, such as transformation of a pattern inside brackets to a tuple pattern upon typing a comma ,: ff (1) => ff (1**, 2**)

Another example of a wrap substitute menu usage is the ‘PConstructor’ concept (todo: ref to prev chpt). Since the concept contains the smart reference to the constructor as a child, we cannot use its referencing capabilities directly. Instead, we would need to create an instance of the ‘PConstructor’ concept first and only then we would be able to pick an available constructor. To create an instance of the ‘PConstructor’ automatically, we have to define a wrap substitute menu for ‘PConstructor’ concept in a similar manner.

Cell action map

Side transformation menus allow us to transform a certain AST node to another when a specific text is written right or left of an editor cell. We may want to, however, put an event handler on a specific editor cell in cases when it is selected, removed, or otherwise manipulated with. For these cases we use the cell action maps.

In Frege-IDE, most of the scenarios, in which we have to use cell action maps, are simply connected to an editor cell deletion. In the previous subchapter, where we described side transformation menus in our IDE, we demonstrated, how we could allow a user to easily transform a list defined by enumeration to a ranged list. Now we could also allow a backwards transformation from the ranged list to enumeration upon a deletion of the ..symbol. Since the given symbol is a constant editor cell, we can create an action map for the cell right away. We select the ‘DELETE’ action as the trigger and implement the transformation in a similar manner, as for side transformation menus. The final implementation is depicted on Figure 4.14.



Figure 4.14: Cell action map for the ranged list constant editor cell depicting the range (..) symbol

Seamless definitions

In the previous chapter we have seen that a definition may be one of the following:

* Import declaration
* Fixity
* Type synonym and datatype declaration
* Annotation and function definition

Except annotation and function definition, all of the above definitions begin with a certain keyword that makes them easily distinguishable (infix, type, data). That is why a simple alias inside the concepts implementation is enough to provide us with a relatively user-friendly interface – for instance, upon typing ‚data‘, it is clear that the definition must be a datatype declaration, so the MPS may create an instance of the corresponding concept.

Annotation and function definition are more complicated and require a careful analysis. There are several observations, which will help us:

* When we type a new identifier, it is not yet clear whether the definition will be an annotation, or a function definition. There must be an ambiguous step (concept) in the process.
* When we begin our definition with symbol [, or a constructor’s name, or a literal, then it is clear we are providing a function or operator definition. **1** +-+ 1 = 2
* When we begin our definition with symbol (, then it is ambiguous, but has only two outcomes:
  + We may be trying to annotate an operator. By typing any operator symbol, we can immediately transform the corresponding node to annotation instance, since a function definition will never contain such an element. **(+**-+) :: Int -> Int -> Int
  + When typing anything else (from what is allowed to be inside patterns), we know it has to be a function definition.
    - Examples:
    - **(a**, b) = (1, 2)
    - **(1**, c) = (1, 2)
    - **([**a, b], c) = ([1, 2], 3)

Let us now examine the possible outcomes when an identifier is specified first:

* If an identifier is followed by a comma, it has to be an annotation, since we are clearly adding new items to collectively annotate. **identifier1,** identifier2 :: Int
* If an identifier is followed by symbol ::**,** it is again, an annotation. identifier :: Int
* If any other operator symbol is types, it is an operator definition. **identifier :-**: b = 0 (Colon symbol : on its own may not serve as a custom operator, so when typed **:** we have to wait for what comes after.)
* When typing symbol =, we are defining a constant function definition, or an operator definition (for an operator that begins with symbol =). This is ambiguous and we will let the user decide.
* When typing anything else (from the set of symbols allowed in patterns), we are providing arguments to a function definition. **identifier (**a, b) = a \* b

This is a rather complex decision tree we have to implement if we want to account for each case, so we will only describe the implementation briefly. All definitions are concepts that inherit from the abstract ‘Definition’ concept. We can create a concept ‘FunctionDefinition’, which will represent what we get when trying to type either a symbol [, or a constructor’s name, or a literal. It is not a ‘real’ function definition, which also has to consist of the right hand side. Instead, it is rather an ‘incomplete definition’ (we can underline the node with red color to denote an error using the typesystem MPS aspect). The ‘FunctionDefinition’ is composed of a single child representing a pattern, which has its own editor actions that allow the node to be transformed to the final definition form.

To allow a seamless substitution of the ‘Definition’ node to ‘FunctionDefinition’, we define a wrapper for the ‘FunctionDefinition’ for all of the three mentioned cases:

* List pattern (handles the input symbol [)
* Constructor reference in a pattern (‘PConstructor’ concept)
* Literal in a pattern (‘PLiteral’ concept)

This works, because whenever an instance of the ‘Definition’ concept is expected (therefore ‘FunctionDefinition’ is also applicable), we may instead use one of the three options mentioned above and the handler will automatically wrap the node into an instance of the ‘FunctionDefinition’.

We will handle the case for ( symbol in a similar manner. We create a concept representing the brackets, which are empty at first (this is, again, an ‘incomplete definition’). The right transformation menu associated with the left bracket then handles both of the possible outcomes:

* We create two transformation actions.
* The first one is triggered by any character that is from the allowed operators. If the entered character does not conform to this condition, we return any operator character. (See Figure 4.15 for implementation details.) This action transforms the node to annotation.
* If anything else is typed (the text part has to return an empty string “”), the node is to be transformed to ‘FunctionDefinition’.



Figure 4.15: Text part of the transformation action associated with the concept representing empty brackets

The last case is handled by creating another new concept and defining an appropriate substitution of the ‘Definition’ to the new concept. The substitute action has to check the user-typed text for whether it actually can act as an identifier and that it does not equate to one of the reserved words (e.g. true, false, data, type, infix, etc). After that we have to implement all of the cases from the observations above as a right transformation menu for the new concept. The approach is no different from the one described in this section, so we will not go through it.

An interesting problem is to transform the identifier to a reference in case the identifier is followed by symbol ::. While this problem is related to scoping (this is described in the following chapter), if we have a list of available nodes representing the function names, it is sufficient to find the one that is a best match for the user-entered identifier. We pick one and simply create a new node referencing the picked node.

This whole approach is just one of many, since there is no exact set of rules for how the IDE should behave. We took this path as an experiment which we will refer to later in the evaluation chapter. On the one hand we allow a user to type the Frege code as he or she may be used to (from left to righ), but now we offer only a limited assist in referencing existing functions and operators when defining annotations. There are advantage and disadvantages to both.

Adding operators inside expressions

Frege language allows users to define custom infix operators with an almost arbitrary precedence. The precedence, however, may be changed at any time and handling such an event properly would be complicated to implement. Instead, we opted for keeping expressions in linear structures in contrast to a tree that would need to be recomputed and reshaped each time a precedence of an operator is changed. Evaluating type of an expression is therefore left to the typesystem aspect, where a special algorithm must be used to handle various scenarios.

To allow adding operators, we need to obtain a set of available operators first. This topic is described in the chapter that deals with scopes. Once we have it, we need to define right transformation menu for the operands. What we want to achieve is to add a new operator and then set focus to a new operand node upon typing an operator a user wants to use. However, there may be a case when we have two operators where one is a prefix of another. Consider the operators + and ++. When a user types an expression “x+”, it is not clear, whether he or she will continue by adding another + symbol, or the typed operator is finished. Fortunately, MPS simplifies the scenario by automatically handling ambiguous transformation menu actions once the user-entered text does not conform to any of the triggering ‘text’ parts anymore. This means that once a user types “x+y”, it is clear that the text that should trigger one of the two actions, was “+” (the last one possible). Thus, the corresponding action is activated.

Somewhat unpleasant feature of MPS is that we cannot specify in ‘text’ part of a transformation menu action a list of strings, but rather only a single string that can trigger the action. We have to, therefore, return a closest operator that begins with the user-entered operator string. In the example above, it would, however, immediately trigger the action for a single “+” operator once the user writes “x+”, since there is no ambiguity MPS would know of. To force the ambiguity, we will create a second action with the same handler, but the text triggering the action is now the second closest operator that begins with the user-entered string. If such an operator does not exist, we may return an arbitrary string which does not begin with the currently user-typed text.

To implement the feature, we have used a custom trie-like data structure. We give the trie the user-entered text and find out if there is a leaf node with the same prefix as the given input. If yes, we return the operator, if not, we return ‘Illegal pattern’. Figure 4.16 represents such a trie with the blue nodes denoting ‘leaves’. The available operators are ‘+’, ‘++’ and ‘+-+’ and the input string is ‘+--’ in the first case, ‘+-’ in the second and ‘+-+’ in the third. In the first case the trie does not contain the string, so it returns ‘Illegal pattern’. In both the second and third case, the trie returns ‘+-+’, however, the action is executed only in the third case where there is an exact match of the user-entered text and the string returned from the trie. The ambiguity may be forced only in the case the entered text is equal to ‘+’. In that case, for the second action, we may return either ‘++’ or ‘+-+’.



Figure 4.16: A visualization of the trie used to implement the feature for adding operators in expressions

Final remarks

While the editor aspect of MPS provides powerful options to allow the creation of a user-friendly IDE, it takes a considerable amount of time and resources. Rather than analyzing different usage scenarios of the language, it may be sometimes easier to define a set of executable actions (intentions) which a user may choose from to create the required AST nodes. As it has been mentioned, since the MPS does not rely on lexers and parsers to process the code, we have to take a reverse approach and that is costly.

**Code completion**

An important feature of many great IDEs is a context-aware code completion. This feature speeds up development process mainly by putting less pressure on a programmer’s memorization of the identifiers, members and so on. It usually has a form of a list (Figure 2.5). A lot of simplified implementations only include a list of members or all possible words that were written in the corresponding code. Context-aware code completion differs from such implementations by providing a list of only those constructs that are actually ‘visible’ in the given context. Consider the following example of functions in Frege:

length (y:ys) = 1 + length ys

getTop (x:xs) = x

On the right hand side of the getTop function definition a user would not be able to pick e.g. a variable ‘y’ from the code completion menu, since ‘y’ is a local variable of the function ‘length’. The only identifiers, in this context, the menu would be populated with, would be ‘getTop’, ‘x’, ‘xs’ and ‘length’ (if we ignore the possibility of other functions being defined in the module, the imported definitions and implicitly imported definitions from Frege-Prelude library).

To implement context-aware code completion in MPS platform, several mechanisms are available: concept references, constraints and scopes. In this chapter, we will analyze the overall problem with code completion, discuss possible solutions and provide several examples.

Scopes

The high-level approach to restricting set of available nodes in a certain context is relatively simple.

Let us look at datatype and type synonym declarations. We have already explained in the chapter 4.1 (Structure) why their names are connected to a specific node and not a simple property. The node is an instance of ‘TypeName’ concept and a type reference is a simple concept containing only a reference to ‘TypeName’. (Thus, it is a ‘smart reference’.) What we want to do is to restrict certain type names that are referenced from the smart reference based on the context. We will therefore define constraints for the concept, specify its presentation (it should simply return the referenced type name) and the scope, which is basically a list of all ‘TypeName’ instances that are referenceable in the given situation.

Providing a scope for the concept, however, requires a proper analysis. It would be tedious and unmanageable to specify a list of referenceable nodes for each smart reference concept individually. Instead, we can take a different approach and proclaim certain concepts to be ‘scope providers’. What we mean by this is that they contain certain children which can be referenced. In our example, both datatype and type synonym declarations are scope providers, because they contain a child, instance of the ‘TypeName’ concept, which can be further referenced by other concepts. These concepts, upon invocation, will provide a list of nodes that are available to them.

With this approach of scope providers, we can actually form a hierarchy. Both described concepts inherit from ‘Definition’ and are therefore children of the ‘Skeleton’ concept. ‘Skeleton’ could be considered a single whole Frege module. It only makes sense upon requesting a list of available ‘TypeName’ instances from ‘Skeleton’ concept, for ‘Skeleton’ to delegate the call to each its datatype and type synonym declaration and return the merged lists.

This hierarchy is, however, not completely correct, since certain concepts must not provide their children to the ‘outside’ world. A good example is a function definition, which specifies several arguments in the form of variables, but the variables are referenceable only from its own right hand side. (refer to example at the beginning of this chapter) The name of the function still needs to be propagated to the parent scope provider, however.

Based on this, we can implement the constraints aspect of each smart reference by delegating the work of an actual scope creation to their closest ancestor, which can do so. An exemplar implementation is provided on Figure 4.17. Since a scope provider may provide several different list of nodes (e.g. ‘Skeleton’ may provide a list of all type names, but also function names, list of operators, and so on), we should be able to tell, which concept we are actually interested in. In case of type reference, we simply want to obtain a list of all referenceable instances of ‘TypeName’ concept, so that is what we specify. In our work, the concepts, which can create a scope, implement interface ‘ScopeProvider’.

Figure 4.17: An exemplar implementation of the type reference link constraints

Scope hierarchy pattern

In the most cases a scope provider concepts needs to know who is requesting a scope and what the relation between the two is. For instance, the function definition may provide its name and a list of its argument identifiers to its right hand side, but to the outside, only its name may be returned. A similar mechanism may be used for concepts enwrapping ‘where’ and ‘let’ language constructs.

However, since the function definition is the first scope provider in the hierarchy for a requester from its right hand side, now it also has to provide its parent scope. If we go back to the example at the beginning of this chapter, we can see that the right hand side of the function ‘getTop’ can use the identifier of the function ‘length’ and vice versa. In this case the nodes are siblings in the corresponding AST (both are children of the ‘Skeleton’ concept), but generally speaking, all of the pattern scopes, up to the root concept, have to be merged together to provide the correct set of referenceable nodes. (An example could be an imported function from another module, which would be provided by a concept higher in the ‘scope-providing’ hierarchy, than is the sibling.)

We can illustrate the pattern on the Figure 4.18. The figure depicts an AST of a simple Frege module with two functions:

f x y = x + y

g = 0

Function definition is a scope provider. It can create a scope, where it enwraps its arguments and its name. Thus, for the right hand side of the function ‘f’, it provides the arguments ‘x’ and ‘y’, which are then referenced. This is because the function definition knows, where the request came from, and can properly react. If the request for scope came from the ‘outside’ (e.g. ‘Skeleton’), it would only return its own name.

‘Skeleton’ concept is also a scope provider. It provides nodes that were either implicitly or explicitly imported to the module (‘+’ operator is part of the implicitly imported Frege.Prelude library), while also delegating scope creation to its children. This means that based on the illustration below, we know, that ‘Skeleton’ provides the following nodes:

* Everything implicitly imported from Frege.Prelude library, such as ‘+’ operator definition.
* Function names ‘f’ and ‘g’, which are actually provided by the corresponding function definitions themselves, to which ‘Skeleton’ concept only delegates its request. The function definitions, based on the request coming from a parent, know that they may only return their own names, but not the arguments.

Now we can derive, what referenceable nodes the code completion menu for the right hand side of the function ‘f’ is populated with:

* Request for scope to the function definition of ‘f’ comes from its right hand side, which means the definition will return a list of its arguments.
* Subsequently, the definition will delegate the scope creation to its parent.
* The closest parent of the function definition of ‘f’, which is also a scope provider, is the root node and instance of ‘Skeleton’ concept.
* The root node will create the scope as described above.
* The two scopes are merged and returned to the right hand side of the function ‘f’.



Figure 4.18: A visualization of the AST for a single Frege module with two function definitions

To help us implement the pattern, we have created an interface ’DCScopeProvider’ (extends the built-in ‘ScopeProvider’). It contains two main methods, one called by children and one called by parent scope providers. The default implementation, which may be overridden using behavior MPS aspect, is to always include for children only the parent scope, which is delegated up to the root node, and then subsequently called down (i.e. called by parent scope providers), usually back to the original scope provider. In the latter method, the concept returns a concrete scope it provides.

In special cases, such as the function definition concept, we may override the method called by children to also include additional nodes (such as function arguments). The method invoker may be also further inspected, which gives us a lot of flexibility and allows us to react appropriately in almost any scenario.

Import and export

The concept that provides a scope of imported function identifiers, operators, datatypes and other definition names, is the root concept ‘Skeleton’. Upon a request, it constructs two scopes:

* Scope of everything that was imported from other modules.
* Scope of the current module, i.e. function identifiers, operators, etc.

The merged scopes are then returned.

In the both cases, the scope creation is actually delegated to the children concepts. However, the creation of the imported scope significantly differs from the latter, which we have described in the previous section.

The ‘Import’ concept contains a reference to a ‘Module’ concept. From that, it can get to the other module’s root node (instance of ‘Skeleton’ concept) and subsequently request the whole module’s scope. While the main idea works, there are two main obstacles:

* A module may restrict what it is actually willing to export.
* Not all entities are to be imported into the main namespace.

The first point is solved by iterating the enumerated items in the ‘module’ statement. If no items are provided, we create the scope for the whole module. If exported items are specified, a scope is created for each one and then merged into a single list.

The second point poses some challenges, since all of the items are actually imported, but some of them can be accessed only by using a qualified name. This means that the ‘Import’ concept has to be able to provide two different scopes, one for all of the items from the referenced module (for a usage with qualified names), and one for the items that are enumerated, or in case of using ‘hidden’ clause, not enumerated (items that are imported into the current namespace).

Additionally, the ‘Import’ concept has to provide a very specific scope for its children as well. First, it should allow to import only the other modules and not the one currently being defined. Thus, a scope containing the sibling module names has to be created. Then, we have to provide the scope for the items enumeration, which will describe, what items will be actually imported into the current namespace. This, understandably, depends on the selection of the module to be imported. In each case, the scope is created by the ‘Import’ concept itself and cannot include its parent scope for obvious reasons. Therefore, the hierarchical scope pattern described in the previous section does not fully apply here. (In a similar manner, the concept ‘IIType’ has to create its own scope for the ‘IITConstructorList’ depending on the datatype it references.)

It remains to explain, how the two scopes, that ‘Import’ concept provides, are used.

The second scope containing only the items imported into the current namespace is the default scope that the ‘Import’ provides. This is then used by ‘Skeleton’, merged with the rest and returned to the requester. The first scope for a usage with qualified names, has to be used explicitly.

In scenarios, where it made sense, special concepts were created to incorporate the usage of qualified names. For instance, in an expression where a certain kind of application is expected (function application, operator application, etc.), we may use ‘ImportedEntityApplication’. The concept consists of two parts, which include the alias of an imported module and the application entity, which is referenced from that module. In a similar manner as the ‘Import’ concept, ‘ImportedEntityApplication’ creates a custom scope for the application entity child, based on the scope provided by the corresponding ‘Import’ node. From the editor perspective, a wrapper substitute menu is defined to create an instance of the concept whenever an import alias is typed by a user. Figure 4.19 shows a usage of the concept on a concrete example. The code completion menu is populated only by the identifiers from the corresponding imported module ‘mps.frege.examples.Lists’.



Figure 4.19: Code completion menu when using a qualified name

Implicitly imported library Frege.Prelude

A library with several standard functions and operators has been created to simulate the behavior of an implicit library Frege.Prelude. The module is simply labeled as ‘Default’ in Frege-IDE (mps.frege.Default).

Every new module imports the library implicitly upon its construction by searching the visible modules and finding the one with the corresponding name. The module is then imported with a ‘hidden’ flag, which makes it invisible in the editor.

**Typesystem**

To implement a simple type checking capability for our IDE, we used the typesystem aspect in MPS. The current feature has some limitations and supports only type evaluation of the functions, where an annotation is provided. In this chapter, we will go through several examples, describe the problems we encountered during the implementation and evaluate the final result.

Types

Several concepts have to be created to be used in the MPS typesystem aspect to denote the types of expressions. As analyzed in previous chapters, we need to implement at least the following types:

* Numeric types (to demonstrate the capabilities, we will restrict ourselves only to Integer and Double types)
* Bool
* Char
* String (in Frege, it is not a list of Char, thus a new separate type is required for the feature)
* List
* Tuple
* Function type

Additionally, we will create a special concept called ‘Undecidable’ to denote the types of expressions we could not or did not want to infer the types of.

An example of an implementation for the concept representing type of a tuple is captured on Figure 4.20. The concept has to inherit from Type concept in j.mps.baseLanguage. The type of a tuple is given by types of its items, therefore we also need to provide the children for the concept.

To specify the relations between the new types we have to create sub- or super-typing rules, and replacement rules. The relationship between Double and Int is rather straightforward, and regarding the implementation, we only create the sub-typing rule for the concept representing the Int type. There, we specify that the super-types of Int are composed only of our new type Double.

Tuples, functions and lists are more complicated. For the two tuples to be comparable, they need to have the same size and, subsequently, their items need to be comparable as well (recursively, e.g. in case the item of a tuple is a tuple again). The replacement rule is implemented in a Java-like language, by iterating the children items of the two tuples to be compared. If the corresponding pair of items does not satisfy the sub-typing relationship, we will manually report an error to the user.



Figure 4.20: Implementation of the concept representing type of a tuple

Infix expressions

In Frege, a user may define custom operators with different associativity and precedence. An operator is just a function accepting two arguments and returning a result. A result of such function may, again, be a function accepting additional arguments.

Frege differs from most programming languages such as C# or C++ in the sense the languages allow mostly only operator overloading, that is, adjust the behavior of the operators for different input types. The syntax tree generated by the parser in the corresponding language compiler, however, stays the same. That is not the case in Frege. Here, a user may create a completely new operator of an arbitrary length, altering the syntax tree that is generated for the expressions using the new operator.

We have already mentioned the problems associated with the custom operators and editors, and why we opted for keeping the expressions linearized instead of using a tree. Now we have to parse the expressions using the information a user gives us by typing the ‘infix’ statement, where he or she specifies the new operator’s associativity and precedence.

The plan to implement the expression type evaluation is as follows:

* Evaluate the types of the sub-expressions recursively, such as expression enclosed within brackets, or expressions inside terms (tuples, lists).
* The remaining is a linearized series of operands, for which we know the type of, and operators.
* Gather the data for the operators used inside the expression, namely precedence and associativity.
* Construct a temporary binary expression tree for the current expression.
* Get the type for each node representing an operator, in the corresponding binary tree
* Evaluate the type of the tree.

The MPS offers a declarative way to implement the type evaluation of AST nodes and thus we do not have to be excessively concerned about recursive approach. Furthermore, to ease the implementation of the last point, we can actually create the temporary expression tree based on new MPS concepts, implement the type evaluation of those (again, declaratively) in the typesystem aspect and let the platform perform the computation itself.

The concept representing usage of infix operators inside the expressions is ‘OperatorReference’. We can implement in the behavior aspect of the concept methods for returning the used operator’s associativity and precedence, simply by searching the corresponding AST and selecting the first ‘fixity’ declaration, which is associated with the current operator. If no ‘fixity’ declaration is available, we return a default value.

In a similar manner, we can implement the type retrieval of the operators. ‘OperatorReference’ points to the operators’ actual function definition. We can inspect the right hand side of the definition (it is an expression), evaluate its type and return it.

What remains, is constructing the binary expression tree for the given expression based on the used operators’ precedence and associativity. For this, we used a derivation of a standard algorithm for translating infix expressions to postfix, which uses a single stack. Our algorithm uses two stacks and iterates through the elements of the expression three times in total.

Before we delve into the algorithm, let us analyze the conditions given by Frege language regarding the operators’ associativity.

* There are three types of associativity: left, right and non-associative operators.
* There must not be operators with different associativity and the same precedence in sequence in a single expression.
* Non-associative operator (or non-associative operators with the same precedence) must not be used in sequence in an expression. (Note that when separated by another left or right-associative operators, the expression is usually legal.)

The corresponding algorithm works like this:

1. Iterate the elements of the expression from left to right and put them on the first stack. Keep the track of the precedence of the last operator and if an operator with a lower precedence is encountered, evaluate the items on the first stack.
2. We evaluate only the items on the first track, where the operators have a strictly higher precedence, than the newly encountered operator. These operators are basically surrounded from both the left and the right side of the expression by the operators with lower precedence, which means they will definitely get evaluated first, therefore we can form a sub-tree for them.
3. Keep track of the last operator’s precedence and associativity. If you encounter two consequent operators with the same precedence, but different or no associativity, report an error to the user.
4. Since we are now iterating the elements in the first stack, we are encountering items in the reversed order. Therefore if we run into the operator with the right associativity, we can form a binary sub-tree right away and put its root node on the second stack. (We need to be careful with the reversed order of the operands.) If we hit a left-associative operator, we only put the item on the second stack, because the sub-tree will need to be formed from the other side. We do this until we hit an operator with a different precedence.
5. Once we hit in the first stack an operator with a different precedence, then the last one, we know, the new operator will have a smaller precedence. This is because of the first point, where we take new operators only in the increasing order. That means the items popped from the first stack can be evaluated in the final phase, since they have a higher precedence and need to get evaluated first.
6. In the final evaluation phase, we iterate the items in the second stack, therefore the order of the items is the same as in the original expression. Here, the operators are left-associative, so we simply take operands and an operator, form a sub-tree, and move onto next items. Then, we take the formed sub-tree, the new operand and the new operator, and form a sub-tree again. This way we get a tree representing evaluation of items from left to right.
7. We return back to the first point and continue with the items in the former expression. Once the items are depleted, we move onto the second point again.

To illustrate, let us consider an example with three operators:

* + left-associative, precedence 1
* \* left-associative, precedence 3
* . right-associative, precedence 2

The type which the operators work on is not important at the moment. Consider the following expression:

"1" + "2" + "3" + "4" \* "5" \* "6" . "7" . "8" . "9"

We know that the operator \* has the highest precedence, so the expression "4" \* "5" \* "6" gets evaluated first. The operator is left-associative, which means we can place brackets in the following way and not alter the expression’s syntax tree:

"1" + "2" + "3" + (("4" \* "5") \* "6") . "7" . "8" . "9"

Then, the operator . is the one with the highest precedence. It is right-associative, so we get the following equivalent expression:

"1" + "2" + "3" + ((("4" \* "5") \* "6") . ("7" . ("8" . "9")))

Finally, the result would look like this:

((("1" + "2") + "3") + ((("4" \* "5") \* "6") . ("7" . ("8" . "9"))))

Now if we take a look on how the algorithm works, we iterate through the items until the operator . is encountered. That is because . has a lower precedence than \*, but \* has a higher precedence, than +.

**Expression:** . "7" . "8" . "9"

**First stack:** "6" \* "5" \* "4" + "3" + "2" + "1"

**Second stack:** (empty)

Then we evaluate the items in the second stack. Since operator \* is left associative, we put everything onto the second stack until operator + is reached, which has a lower precedence.

**Expression:** . "7" . "8" . "9"

**First stack:** + "3" + "2" + "1"

**Second stack:** "4" \* "5" \* "6"

From the second stack, a binary tree is formed by taking elements in order as they come. Figure 4.21 represents such a tree formed in this stage.



Figure 4.21

The algorithm then continues with the first point, taking rest of the elements until all of them are depleted and moves onto second stage.

**Expression:** (empty)

**First stack:** "9" . "8" . "7" .*tree*(("4" \* "5") \* "6")+ "3" + "2" + "1"

**Second stack:** (empty)

In the second stage, we move through the . operators. Being right-associative, the tree is formed immediately, until the operator + is encountered.

**Expression:** (empty)

**First stack:** *tree*((("4" \* "5") \* "6") . ("7" . ("8" . "9")))+ "3" + "2" + "1"

**Second stack:** (empty)

Finally, the rest of the elements are pushed onto the second stack, the tree is formed in the final stage and returned. Figure 4.22 depicts the resulting tree.



Figure 4.22

Function application

The function type is similar to that of the tuple, but it contains 1..n children, which represent the function’s arguments. The last child could be thought of as the function’s return type, but due to currying, this is not necessarily true (several last children could represent together the function’s return type).

In the concept ‘GenericApplication’, the application entity is a node that should carry an associated type. In this work, we completely omitted the user defined types (type synonyms and datatypes), which means we will not be dealing with the constructor application. Application entity for operators and functions is fairly simple – since it references a concrete function or operator definition, we only have to take a look at the type of its right hand side. As for the application entity representing an expression enclosed within brackets, we have to evaluate the contained expression and get its type.

For the type inspection, we have to use ‘when concrete’ block. The application entity may be of a function type, but also does not have to be (consider for example a literal enclosed within brackets). If it is not, we have to make sure no arguments were specified for the application, otherwise that is an error. The resulting type of the application is then the type of the application entity itself.

In case the application entity has a function type, it needs to be properly handled. First, we check the types of the provided arguments against the types of the expected arguments. We also need to check the amount of provided arguments does not exceed the amount of expected arguments for the application entity. It is possible to specify fewer arguments, however, since that is the currying mechanic.

The returned type of the whole application is based on whether all of the arguments of the function have been depleted. If so, we will return the type of the last child of the corresponding function type. If that is not the case, we construct a new function type, insert all of the remaining arguments’ types into it and return the new type.

We can visualize this on an example with the following function definition:

ff :: Char -> Bool -> Int

ff x y = 0

In this case, we are not really interested what the function does and why it does not actually use the variables ‘x’ and ‘y’ in any way. We only care about the type of its arguments as described by the corresponding annotation, and its return type.

Let us consider the usage of the function in the expression ff ‘a’ true. Since all of the arguments were depleted, the type of the expression is obviously Int, as that is what the function returns. However, the expression ff ‘a’ has a type of a function, which could be described by the following annotation:

ff ‘a’ :: Bool -> Int

Note that the similar approach works in the case with infix expressions, where we apply an operator with left and right operands as its arguments.

Function definition type inference

There are two main, relatively independent, parts when it comes to type inference of a function or an operator. The resulting type of either is given by their respective right hand sides. On the other hand, types of the arguments is not always unambiguously inferable. We can take for example a function, which provides a result independently of the given arguments:

ff x y = 0

Though in some cases it is possible to deduce the type of the arguments based on their usage, as demonstrated by the following example:

getTop :: [String] -> String

…

ff x y = getTop [x, y]

(It is clear that the arguments ‘x’ and ‘y’ need to be of the type String based on their usage as arguments to the function getTop.)

This kind of type inference is relatively complex to implement and we did not include it in the final work. Instead, we opted for a reasonable compromise:

* If an annotation is provided for a function or an operator, it is considered to be the actual type of that definition. The resulting type is only checked against the very type of the corresponding right hand side of the definition.
* If the annotation is not provided, the type of the function arguments is ambiguous and is never checked (the inferred type is set to ‘undecidable’). The resulting type of the function is then based on the type of its right hand side.

In the example above, the inferred type of the function ‘ff’ without an annotation would be:

ff :: undecidable -> undecidable -> String

If we were to specify the annotation for ‘ff’ with arguments being of a different type, than String, we would only report an error that the arguments of an illegal type were provided. Figure 4.23 captures such example in Frege-IDE.



Figure 4.23: Example of an illegal arguments usage

Arguments type decomposition

In the previous section, we described that we want the types of function arguments be inferred from the corresponding annotation, if it is provided. Retrieving the type of an annotation is relatively straightforward and a similar problem has been already described in the current chapter (we simply search the current AST for the required annotation, and if available, get its type).

A certain challenge is posed by the fact that a variable, which we intend to infer the type for, may be included deep within a sub-tree representing a certain argument. Let us consider the following function ‘gg’:

gg :: [[Char]] -> [Char]

gg [[1], ['a', 'b', x]] = [x, x, x]

We have to account for several things. First and foremost, we need to know the type of the variable ‘x’, because it is also used in the expression in the right hand side. We need to check that the expression [x, x, x] conforms to the type [Char]. Secondly, we want to make sure that the whole argument is correct, which in this case, it is not: [1] inside the first argument is definitely not of the type [Char].

To do this, we will need to take the type of the corresponding argument from the annotation, and decompose it, as we will delve into the sub-tree associated with the argument. We can illustrate our reasoning on the example above. We know that the argument [[1], ['a', 'b', x]] corresponds to the type [[Char]]. This means that each item of the list has to correspond to the type [Char], i.e. [1] has to have the type [Char] together with the ['a', 'b', x]. Going further down, we check that the type of the literal 1 corresponds to Char as well as the types of items 'a', 'b' and x. Since literals already carry types on their own, the attempt to declare the type Char for the literal 1 will MPS evaluate as an error, whilst the variable ‘x’ will be simply assigned the given type.

To implement the feature, each pattern concept has to inspect the type its parent assigned it. If the type corresponds to the pattern, the type is decomposed and its children types distributed between the children of the corresponding pattern. The root pattern concept takes its type from the annotation, and in case there is none, the ‘undecidable’ type is taken.

We can show how the algorithm works on the mentioned example. The list pattern ['a', 'b', x] is a child of the list pattern [[1], ['a', 'b', x]]. With the help of ‘when concrete’ block the pattern waits until its type, provided by its parent, is resolved. The corresponding type is [Char]. Upon inspection, the type is, indeed, a list, which correctly corresponds to the given pattern. We decompose the type, select Char, and infer the type of the children 'a', 'b' and x to be Char. In case the type given by the parent list was ‘undecidable’, ‘undecidable’ type would be also assigned to the children items.