Inhaltsverzeichnis

[1. Introduction: Project Objective 2](#_Toc98367622)

[2. Fundamentals 2](#_Toc98367623)

[2.1. Theoretical Fundamentals 2](#_Toc98367624)

[2.1.1. Typing 2](#_Toc98367625)

[2.1.2. Structural Subtyping 2](#_Toc98367626)

[2.2. Technical Fundamentals and Decisions 2](#_Toc98367627)

[2.2.1. Language, Framework and Dependencies 2](#_Toc98367628)

[2.2.2. Architectural Decisions 2](#_Toc98367629)

[3. Solution Implementation 2](#_Toc98367630)

[3.1. User Input via “TypeConstructionKitComponent” 2](#_Toc98367631)

[3.2. Typing 3](#_Toc98367632)

[3.3. Structural Subtyping 4](#_Toc98367633)

[4. Outlook on future work 4](#_Toc98367634)

[4.1. Replay-able History for TypingTree and SubtypingGraph 4](#_Toc98367635)

[4.2. Persisting User Input in Localstorage 4](#_Toc98367636)

[4.3. User Input via code editor 4](#_Toc98367637)

# Introduction: Project Objective

* Goal of Typing in general (e.g.: manage memory, select correct assembler instruction, avoid certain run-time errors); The purpose of types (http://www.cse.chalmers.se/edu/year/2011/course/TIN321/lectures/proglang-07.html)
* Goal of Tool: Webtool to enable visually accompanied learning/understanding of Typing and Structural Subtyping
* (Note: Tool hosted via static file sharing via Strato on “typing-tutor.tum.endredi.de”)

# Fundamentals

## Theoretical Fundamentals

### Typing

### Structural Subtyping

* Hint: use sets (DS) for explanation; basics: https://www.inf.hs-flensburg.de/lang/algorithmen/grundlagen/menge.htm
* Structural equality ‘=’ as subset relation to structural subtype relation ‘<=’
* TODO: Define wording ‘real subtype’: ‘<’

## Technical Fundamentals and Decisions

### Language, Framework and Dependencies

* Angular (Component-based)
* TypeScript (detail about structural subtyping)
* RxJS (Subject)

### Architectural Decisions

* Client only
* High-level UML Class Diagram about most important classes (AbstractType, AstNode, TypeEnvironment)

# Solution Implementation

* Main focus of tool: perform algorithms on arbitrary complex types. Where do they come from?

## User Input via “TypeConstructionKitComponent”

* First approach integrating library ‘cparse’ failed due to incapability of parsing:
  + Nested structs
  + Higher order functions (Functions taking functions as parameters)
  + Recursive type definitions
* Solution:
  + UI providing base types as atomic building blocks
  + Type constructor templates as toolset for consecutive type construction
  + Output:
    - Larger set of types
    - Feature of generating typdefs and declarations on built types (printed in readonly code-editor)
* Implementation:
  + Lifecycle-hooks of AbstractTypeBubble as a framework enabling extension (provide UML Class Diagram!)
  + States CONSTRUCTION and IDLE
  + AliasPlaceholder as workaround for recursive typedefs (provide example use case)
* Further Consequence
  + Computing valid C-Code from an arbitrary Type not trivial, e.g.

typedef char\* (\*(\*x(int\* (\*(()[]))))[5])();

* + Webservice cdecl.org cannot be used due to lack of support for structs; also workaround not possible for cases where it holds: “Some non-struct type is pointing on a struct” (e.g. [complex pointer/array combo] to pointer to struct) 🡪 toEnglish() would contain struct TODO: Really a problem?
  + However, webservice still useful for testing --> Test driven development
  + List cdecl rules relying on 3-tuple

## Typing

* + 1. Intro / Manual / Frame / Features
* Expression in input field is parsed by a façade implementation of cparse (having problems parsing complex types, but still good for arbitrary expression 🡪 TODO: maybe add grammar here? Maybe from cparse source code?)
* On change of the entered expression, performTypeCheck (implementation of rules in 2.1.1.) is executed.
* Visualization representation of the algorithm is printed on the right
  + Case 1: Type check on node successful 🡪 print type inference
  + Case 2: Type check has error 🡪 highlight error and fall back to case dependent recovery type, so rest of traversal during call can be performed unrestrictedly
* When indicated, structural subtyping queries upon performing type checks are attached to enquiring steps as references allowing the user to reproduce these substeps.
  + 1. Model: AbstractSyntaxTree -> AstNode -> examples (show UML); TypeEnvoronment
* TODO: Explain “TypeEnvironment”
* TODO: Explain “performTypeCheck(t: TypeEnvironment): AbstractType”
  + failWithTypeError explained (error stored in placeholder type; thought return recovery type so parent ast node can assume a type allowing to unrestrictedly continue the algorithm without propagating the error) [example of this strategy being useful while programming]
    - Strategy “Defined ”
    - Strategy “Wildcard”
      * toEquals() with wildcard always yield true 🡪 Subtyping check true
      * AstNodes (performTypeCheck) have to implement handling wildcard themselves, e.g.
    1. Specification of performTypeCheck() in AstNode

In the following, let ‘<=’ denote the structural subtyping relation, i.e. let t1 <= t2 mean that type t1 is a structural subtype of t2.

* **BinaryExpression**

Let **□** denote the operator represented by the BinaryExpression object. Let lt and rt be the types computed by performTypeCheck() on this.leftType and this.rightType, respectively.

* + Case “□ is =” checks rt <= t (as per typing rule). If true, leftType is returned. Else, assignment is not valid. Therefore, an error is stored and return as if subtyping check was successful
  + Case “□ is arbitrary valid binary operator (+,-,\*,/)” checks structural subtyping in both directions. To obtain a valid typecheck, either lt <= rt or rt <= lt must hold; visually:

|  |  |  |
| --- | --- | --- |
| lt <= rt / rt <= lt | true | false |
| true | lt = rt | lt < rt |
| false | rt < lt | error |

If present, the error is stored. Ultimately, the (“more general”) supertype amongst lt and rt is return. (E.g. the expression “1 + 3.14” yields type float). [Explanation? Maybe reference to real compiler?]

Note: This approach is strongly simplified in contrast to a real compiler, as it disregards weather lt and rt are applicable parameters to **□**. A more sophisticated approach would be storing actual instances of FunctionType (holding exactly two parameters) for each supported binary operator and delegating the performTypeCheck call to a respective instance of CallExpression. However, due to the teaching purpose of this project, the currently implemented solution is considered as adequate.

* **CallExpression**

A CallExpression opject represents a function application of the form *f(p0,…,pn)*.

Let *ft* denote the type computed for *f* and a*pt\_i* the type computed for the parameter applied at index i. Further, let *fpt\_0,…,fpt\_n* be the defined parameter types and *rt* the return type of *ft*.

* **Identifier**
* **IndexExpression**
* **Literal**
* **PrefixExpression**
* **StructAccessExpression**
  + [note to parser translating ‘->’’]
    1. Output visualization by “TypingTreeComponent”
* Recursive angular component (show .html and .ts files)
* Display space allocation by (recursively computed) node width depending on number of children (show screenshots before and after)

## Structural Subtyping

* [Use (Pseudo) Code here]

# Outlook on future work

## Replay-able History for TypingTree and SubtypingGraph

* 2 Phases: “Delegation” (placeholder types here) + “Backpropagation” (types getting defined according to child nodes) TODO: Clear if wording is okay

## Persisting User Input in Localstorage

* Implement Serialization/ Deserialization of AbstractType

## User Input via code editor

* Implement or look for suitable C-Code parser
* Implement Adapter mapping that output to existing Ast structure ( “/ast/ast-nodes/DEPRECATED-AST-NODES/”already covering syntax including various declarations, if-statements, …)
* Finish implementation of performTypeCheck() of that AstNode classes covering
  + Meaningful delegation of performTypeCheck() call to its children and
  + Implement meaningful error handling using failWithTypeError() (maybe even extend to new Error subclasses), e.g.: check if resulting types of ReturnStatements within FunctionDeclaration match with its returnType attribute, i.e. with x being the computed type of an arbitrary ReturnStatement, then x.isStructuralSubtypeOf(this.returnType) is true. Otherwise indicate error, with recoveryType being the expected returnType