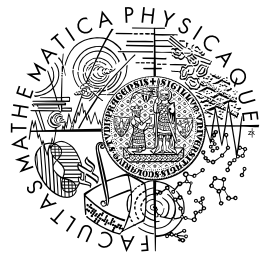

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jiří Klepl

Type Inference and Polymorphism for C

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Miroslav Kratochvíl
Study programme: Computer Science
Study branch: General Computer Science

Prague 2020

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor, RNDr. Miroslav Kratochvíl, for his guidance and active support during my work on this thesis, and for introducing me to functional programming.

Title: Type Inference and Polymorphism for C

Author: Jiří Klepl

Department: Department of Software Engineering

Supervisor: RNDr. Miroslav Kratochvíl, Department of Software Engineering

Abstract: The C language, despite its age, is one of the main languages in systems development. It is valued for giving the user almost complete control over the memory management and the computations the program written in it performs.

However, a large portion of criticism of C arises from the lack of generic programming features. C compensates that by utilizing preprocessor macros, which are prone to user errors.

This problem has been addressed in the early stages of the development of the C++ language, but many systems developers refuse C++ because of its complexity and non-transparency of the code.

We propose a simpler solution by applying the Hindley-Milner type system extended by Haskell type classes and type constructors. We will show that this approach is viable even with minimal changes to the syntax of C, but giving it much higher expressiveness.

Keywords: compilers, programming languages, type systems

Contents

Introduction

We will propose a new programming language, CHM (C with the Hindley-Milner type system), a simple extension of the C programming language. This extension will try to answer the overuse of macros and generic `void *` pointers, which are prone to user errors. Both macros and `void *` pointers provide no type safety, and their misuse can be difficult to debug.

The type safety will be achieved by using type inference and type checking of the Hindley-Milner type system, or more precisely, the Haskell type system based on it.

We will use lambda calculus as an internal representation of the code in the type inference stage as the theory of these type systems is described on it.

Choosing lambda calculus, or its derivatives might seem counterintuitive at first because it is not very efficient in expressing imperative ideas like mutability, for example, but for purposes of type inference, we do not need to know the actual meaning of the code (like for example if a loop stops), we focus on modeling type dependencies in the program. Everything on top of that is just for human readability, debugging purposes, demonstration and for possible extension.

Lambda calculus can quite easily model any C program, but one has to be careful at distinguishing between initialization and assignment as initialization creates binding (in lambda calculus, expressed as a substitution for a bound variable in an abstraction), and assignment is a function that takes two expressions of the same type and returns the first (here we demonstrate that we do not need to model the whole behavior of the program as copying of the value does not have any effect on the types of the arguments).

Chapter 1

Functional programming with Types

1.1 Lambda Calculus

Understanding lambda calculus is important in both the context of type inference and in the context of (functional) programming.

In this thesis, we will use lambda calculus as described in *Lambda Calculi with Types* [2].

Definition 1 (Lambda Calculus Grammar)**.**

$$variable ::= v \mid variable'$$

$$\lambda\text{-}term ::= variable \mid (\lambda\text{-}term\ \lambda\text{-}term) \mid (\lambda\ variable\ .\lambda\text{-}term)$$

In the λ-$term$ grammar rule, the second case is called *application* and the third is called *abstraction*. These two constructs correspond to passing and declaring an argument, respectively.

Definition 2 (Bound Variable and Free Variable)**.** *In the abstraction we call the variable x (its occurrences in the abstraction)* ***bound*** *by the abstraction. If it is not bound we call it* ***free****.*

It is a simple language in which we can formally describe any computational problem.

The whole idea of evaluation of a lambda calculus program is using a simple rewriting rule, β reduction:

Definition 3 (β reduction)**.**

$$(\lambda x.e)e' = e[x := e']$$

[2]

There is one other rewriting rule, α *conversion*, which is used to resolve name collisions, but we will not consider it as we will consider each bound variable uniquely named.

1.1.1 Subexpression

Definition 4 (Subexpression)**.** *Given a well-defined expression e, a subexpression is an expression e' such that: $e = \lambda x.e'$, $e = e'e''$, $e = e''e'$, or $e = e'$, and iteratively from that.*

This definition will extend intuitively to all other syntactic systems.

Writing Conventions Lambda calculi usually use the writing conventions of applications being left-associative, abstractions being right-associative, and the $\rightarrow$ operator (which will be used in typed lambda calculi to denote functions) being right-associative as well.

1.2 Simply-Typed Lambda Calculus

Simply-typed lambda calculus (also $\lambda \rightarrow$), described by Church, is one of the first lambda calculi with a type system. Typing of lambda terms helps in checking the validity of the program, protecting the user from writing programs that make little sense like for example, adding amperes to volts [2] and also it makes it easier to think about the program in the bigger picture giving us information about the usage of the typed entities in some code, typing $f : \alpha \rightarrow \alpha \rightarrow \alpha$, for example, tells us that f is a function taking two arguments of some type and returning a value of the same type. From that alone, we can assume that applying it to two parameters of the same type should be well-defined.

Syntactically it is not any different from the lambda calculus, considering the *à la* Curry variant. [2]

1.2.1 Types and Typing

We will formally define what we consider a type, first we consider a **type variable** similarly to the variables of the lambda calculus:

Definition 5 (Type Variable)**.** *By a type variable, we will mean a symbol defined by the following grammar rule:*

$$type\_variable ::= tv \mid type\_variable'$$

but often, we will use α, β as aliases to arbitrary type variables

Definition 6 (Type in $\lambda \rightarrow$)**.** *For types τ', τ'', a type variable α, we define a type τ in $\lambda \rightarrow$ by the following grammar rule:*

$$\tau ::= \alpha \mid \tau' \rightarrow \tau''$$

where $\tau' \rightarrow \tau''$ is a type of a function taking an argument of the type τ' and returning a value of the type τ''

We can also freely add type constants (like *integers* or *booleans*) to the definition.

Definition 7 (Typing and Assumption in $\lambda \rightarrow$)**.** *Typing is an assignment of a type to an expression written as follows:*

$$e : \tau$$

*where e is an expression and τ is a type. If e is a variable x and we infer other types from it, we refer to such typing as an **assumption** or a **judgement**.*

Sometimes we will write $\Gamma \vdash e : \tau$, meaning that in the context of Γ (set of assumptions) e can get the type τ, more on that in the definition 8.

1.2.2 Inference Rules

Simply-typed lambda calculus uses the following rules, we present them to formally define how to determine whether a certain typing is valid:

Definition 8 (Inference Rules of $\lambda \rightarrow$)**.** *For types τ, σ, expressions e, e', a variable x and a set of assumptions Γ, we define the type system $\lambda \rightarrow$ by the following inference rules:*

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \textit{ (Variable)}$$

$$\frac{\Gamma \vdash e : \sigma \rightarrow \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash ee' : \tau} \ (\textit{Application})$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \sigma \rightarrow \tau} \ (\textit{abstraction})$$

We call Γ either a basis, a context, or a set of assumptions. These names describe pretty well its meaning. In the simply-typed lambda calculus, it is a set of typings of type variables from which we derive the rest of the typings.

The inference rules describe the step-by-step derivation of types where the bottom one follows the top one.

Example

A Step by step example for inferring the type of $id = \lambda x.x$:

1. $\{\} \vdash \lambda x.x :?$
2. $\{\} \vdash \lambda x.x : \sigma \to \tau$ (abstraction)
3. $\{\}, x : \sigma \vdash x : \tau$ (still abstraction; we want to infer τ)
4. $\{x : \sigma\} \vdash x : \sigma$ (var)
5. $\{\} \vdash \lambda x.x : \sigma \to \sigma$ (result of abstraction in the step 2)

Notice here the context changes during the derivation process and that we can give typing to $\lambda \to$ terms without any context, and they can have different typings in different contexts. Also, one term can have multiple possible typings.

1.2.3 Type Substitution, Instantiation and Unification

Definition 9 (Type Substitution). *Type substitution is a process that rewrites (simultaneously) certain type variables into different types. Substitutions will be very important for various definitions and, even more importantly, for type unification, which will be the basis for type inference algorithms in the following type systems.*

For types $\pi_1, \ldots \pi_n$ and type variables $\alpha_1, \ldots \alpha_n$ such that each α_i is distinct, we define substitution S by the following grammar rule:

$$S ::= [\alpha_1 := \pi_1, \ldots \alpha_n := \pi_n]$$

where $\alpha_i := \pi_i$ denotes that π_i replaces α_i.

We will write applying S to a type τ as τS (left associative), we will also allow a simplified definition $[\overline{\alpha} := \overline{\pi}]$ meaning $[\alpha_1 := \pi_1, \ldots \alpha_n := \pi_n]$.

Definition 10 (Type Instance). *We will call a type τ a type instance of a type σ if there is a type substitution S such that $\tau = \sigma S$.*

Definition 11 (Type Unifier). *If τ_1, τ_2 are types and S is a substitution, we say that S unifies τ_1 and τ_2 if $\tau_1 S = \tau_2 S$, then we call S their unifier, and we call τ_1 and τ_2 unifiable. [12] (originally for sets of types)*

1.2.4 Principal Typing

Every typeable term e in the simply-typed lambda calculus has principal type τ [2] which is computable from e and for every other valid typing $e : \sigma$ in the same context there is a substitution S such that $\sigma = \tau S$.

The intuitive idea of principal typing of e is that it is a typing general enough to be substitutable into all other possible typings of e but not general enough to be substitutable into any impossible typings. This is usually simplified into 'the most general type of e'.

The example in section 1.2.2 is a great fit for showing this. $\lambda x.x : \sigma$ would be a possible typing general enough to be instantiated into any valid typing but is not a principal one because substitution $[\sigma := \alpha \to \alpha \to \alpha]$ would give use $\alpha \sim \alpha \to \alpha$, which would yield an incomputable infinite type.

Principal typings will be very important later when we introduce more advanced type systems as the principal type of e is the only information about e we have to consider when we see e again, given it has no free variables.

1.2.5 Type Checking, Typeability, and Type Inference in $\lambda \to$

Type checking is the process of validating whether a term can have a given type, whereas typeability is a decision problem of whether the term can be given some type.

Type inference is closely related to typeability but has an additional requirement of giving an actual type for the term, and what we usually are interested in is its principal type.

Both type checking and typeability of the simply-typed lambda calculus are decidable, and its type inference is computable [2].

One thing that is a problem for the simply-typed lambda calculus is that it is not Turing-complete. It is strongly normalizing [2], and thus assuming it being Turing-complete, the type inference would have to solve the halting problem. This problem will be answered by the following type systems.

1.3 System F and Parametric Polymorphism

System F, polymorphic lambda calculus, or second-order lambda calculus introduces parametric polymorphism into the lambda calculus.

Here we can reuse the example of $id = \lambda x.x$ which, given the simply-typed lambda calculus, can have many typings ($id : \tau \to \tau$ for any τ) which in this type system we can state by one typing:

$$id : \forall\alpha.\alpha \to \alpha$$

We call such typings (with the $\forall$ quantifier) **polymorphic** (opposed by **monomorphic**).

The definition of types extends the definition 6 for the simply-typed lambda calculus.

1.3.1 Types, Type Substitution, and Type Instantiation in the System F

Definition 12 (Type in the system F)**.** *Types in this type system are defined by the following grammar rule extending the original definition 6:*

$$\tau ::= \alpha \mid \tau \to \tau' \mid \forall\alpha.\tau$$

τ, τ' are types and α is a type variable; we can also add a case $\tau ::= \iota$, where ι is a type constant (for example, `int`*).*

In the last case ($\forall\alpha.\tau$), we call α (its occurrence of in τ) a **bound** type variable. If the variable is not bound, we call it **free**.

Definition 13 (Type Substiotution and Type Instantiation in the System F)**.** *Type substitution and type instantiation are defined as in the simply-typed lambda calculus (definitions 9 and 10), with the exception that the rewriting does not apply to bound type variables.*

1.3.2 Inference Rules of the System F

Definition 14 (Inference rules of the System F)**.** *System F uses the rules of the simply-typed lambda calculus (definition 8), plus the following two rules:*

$$\frac{\Gamma \vdash e : \forall\alpha.\sigma}{\Gamma \vdash e : \sigma[\alpha := \tau]}\ (\forall\text{-elimination})$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall\alpha.\sigma}\ (\forall\text{-introduction})$$

In ($\forall$-introduction), the type variable α is chosen so it is not free in any any assumptions on which the premise $e : \sigma$ depends.

In system F the type checking is undecidable, and thus any general type inference algorithm is impossible. The Hindley-Milner type system is a restriction of it that allows for a simple type inference algorithm on which we will base ours.

1.4 Hindley-Milner Type System

The Hindley-Milner type system, we will refer to it simply as HM, is a fairly minimalistic extension of lambda calculus, or at least syntactically. Semantically it could be seen rather as a restriction of the system F that answers the undecidability of it. For that, see the subsection *Type Schemes*.

Definition 15 (Expression in HM)**.** *For a variable x and expressions e', e''. We define an expression e in HM have the form given by the following grammar rule:*

$$e ::= x \mid e'e'' \mid \lambda x.e' \mid \textit{let}\ x = e'\ \textit{in}\ e''$$

We skipped two more cases that are not necessary for the type inference. We will describe them in the definition 16, and later we will define them as regular functions.

It differs from the syntax of lambda calculus (and that of the system F) only in the addition of the third clause, called the *let statement*. The meaning of this clause is the same as of $e'[x := e]$, but it allows e to be polymorphic, which is otherwise for subexpressions forbidden in HM. A very similar construct $(\lambda x.e')e$ (using abstraction instead of *let*) would result in all occurrences of x in e' having the same monomorphic type in this system. We will later describe definitions for typings and inference rules that will result in this behavior.

There are two additional expressions completing the language that have no significance to the type system and thus we will not further consider them in any of the typing rules, and in the inference algorithm. They are defined as follows:

Definition 16 (If Then Else and Fix in HM)**.** *Here we give the two cases omitted in definition 15:*

$$e ::= \cdots \mid \textit{fix}\ x.e' \mid \textit{if}\ e'\ \textit{then}\ e''\ \textit{else}\ e'''$$

where '$\cdots$' denotes that this extends the definition 15.

The latter clause has the expected meaning, and the former is defined as the least fixed point of $\lambda x.e$ [8]. In the context of type inference the two clauses can be seen replaced by functions (operators) with the following typings (type schemes) [3]:

$$\textit{fix} : \forall\alpha.(\alpha \to \alpha) \to \alpha$$

$$\textit{if} : \forall\alpha.\ \textit{bool} \to \alpha \to \alpha \to \alpha$$

1.4.1 Types, Types Schemes, and Typing in HM

Types of the HM type system are restricted in that they cannot include quantification of type variables introduced in the F system. This change is then counterbalanced by the introduction of type schemes, which replace types in typings.

Definition 17 (Type in HM)**.** *Types in HM are similar to those in $\lambda \rightarrow$ (definition 6), they are defined by the following grammar rule:*

$$\tau ::= \alpha \mid \iota \mid \tau' \rightarrow \tau''$$

the only difference is it adds a case ι for a primitive type (type constant).

Definition 18 (Type Scheme in HM)**.** *Type schemes are defined similarly to types in the system F. They are introduced, so the type $\forall$ quantification, if there is any (in a typing), is always at the very front of the typing (in the system F, it can be anywhere inside the type).*

For a type variable α and a type τ, we define a type scheme σ by the following grammar rule:

$$\sigma ::= \tau \mid \forall \alpha.\tau$$

We call the former case a monomorphic type scheme and the latter a polymorphic type scheme.

Definition 19 (Typing in HM)**.** *Typing is assigning a type scheme to an expression instead of assigning a type in definition 7.*

Common Typing extensions to HM

In the HM type system, it is common to introduce the following type operators: cartesian products (tuples): $\times$; disjoint sums (tagged unions): $+$; and lists. The existence of these is not required for the type system, and so we will avoid them going forward. Their reintroduction is trivial as they can be described as regular type constructors.

1.4.2 Type Instantiation in HM

The definition of type instantiation carries over from the system F and the simply-typed lambda calculus (definition 10), considering substitution as described in definition 13, but it also applies to *type schemes*.

Definition 20 (Generic Instance in HM)**.** *A notable change to the notion of instances is the definition of a **generic instance** of a type scheme $\sigma = \forall \alpha_1, \alpha_2 \cdots \alpha_n.\tau$ which is defined as $\sigma' = \forall \beta_1, \beta_2 \cdots \beta_m.\tau'$ such that $\tau' = \tau S$ for some type substitution $S = [\alpha_1 := \tau_1, \cdots \alpha_n := \tau_n]$, where type variables β_i are not free in σ. We then write $\sigma > \sigma'$. [3]*

Note that $\sigma > \sigma'$ implies $\sigma S > \sigma' S$, however it does not imply $\sigma S > \sigma$.

1.4.3 Type Inference in HM

Definition 21 (Inference rules of HM)**.** *For a variable x, expressions e, e', a set of assumptions Γ (Γ_x stands for the set of assumption Γ with any assumption about x removed), type schemes σ, σ', types τ, τ', and a type variable α, we define the HM type system by the following inference rules:*

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \textit{ (Variable)}$$

$$\frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash ee' : \tau} \textit{ (Application)}$$

$$\frac{\Gamma_x \cup \{x : \tau'\} \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau' \to \tau} \textit{ (Abstraction)}$$

$$\frac{\Gamma \vdash e : \sigma \quad \sigma > \sigma'}{\Gamma \vdash e : \sigma'} \textit{ (Instantiation)}$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \textit{ not free in } \Gamma}{\Gamma \vdash e : \forall \alpha.\sigma} \textit{ (Generalization)}$$

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma_x \cup \{x : \sigma\} \vdash e' : \tau}{\Gamma \vdash \textit{let } x = e \textit{ in } e' : \tau} \textit{ (Let Polymorphism)}$$

The *instantiation* rule replaces the $\forall$-elimination rule of the system F. Apart from that, the *let polymorphism* rule is the only real extension of the rules, balancing the type restriction introduced before.

We will introduce the algorithm W 2. It produces the *principal type scheme* for the given program (expression). But before we describe the algorithm, we need to introduce the type unification algorithm, which is used in its definition.

Most General Unification

One of the main parts of the type inference algorithm is the type unification, and since we are usually after principal types, the most general unification.

Definition 22 (Most General Unifier)**.** *The most general unifier of types τ, τ' is a unifier S such that for every other unifier U of τ and τ' there exists a substitution R such that $U = S \circ R$. [3]*

We will further require the algorithm to use only variables from τ and τ'.

We present an algorithm 1, which is a variation of Robinson's unification algorithm. We will use a modification of this algorithm even in the next type system.

Algorithm 1 Unification Algorithm [12]

function $mgu(\tau, \tau')$
  $S_0 \leftarrow \epsilon$
  for $k = 0$ to ∞ **do**
    if $\tau S_k = \tau' S_k$ **then return** S_k
    end if
    Let τ_k, τ'_k be leftmost corresponding well-formed sub-expressions in which τ and τ' differ ▷ if the expressions are represented by syntactic trees, τ_k and τ'_k have the same relative path from their respective roots - then leftmost means having the lowest inorder rank).
    if τ_k is a variable not occurring in τ'_k **then**
      $S_{k+1} \leftarrow S_k[\tau_k := \tau'_k]$
    else if τ'_k is a variable not occurring in τ_k **then**
      $S_{k+1} \leftarrow S_k[\tau'_k := \tau_k]$
    else
      Terminate with an *occurrence check* error
    end if
  end for
end function

The Type Assignment Algorithm W

Before we describe the algorithm itself, we first need to define the closure of a type τ under assumptions Γ:

Definition 23 ($close_\Gamma(\tau)$)**.**

$$close_\Gamma(\tau) := \forall \alpha_1, \ldots \alpha_n . \tau$$

where $\{\alpha_1, \ldots \alpha_n\} := free(\tau) \setminus free(\Gamma)$*,* $free$ *stands for the set of free variables, see the definition 2.*

The algorithm 2 is a function W which takes a pair (Γ, e) where Γ is a set of assumptions and e is an expression in the context of Γ. Then W returns a pair (S, τ) such that:

$\Gamma S \vdash e : \tau$ and furthermore $close_{\Gamma S}(\tau)$ is a principal type scheme of e under ΓS.

1.4.4 Extensions of the HM Type System

Pure HM allows for just parametric polymorphism. However, without any ad-hoc polymorphism or subtyping, this means there is only one principal type for

Algorithm 2 The algorithm W [8]

function W(Γ, e)
  if e is a variable $\wedge$ $e : \forall\overline{\alpha}.\tau' \in \Gamma$ **then**
    let $\overline{\beta}$ be a list of *new* type variables s.t. $|\overline{\beta}| = |\overline{\alpha}|$
    $(S, \tau) \leftarrow (\epsilon, \tau'[\overline{\alpha} := \overline{\beta}])$
  else if e **matches** $e_1 e_2$ **then**
    $(S_1, \tau_1) \leftarrow W(\Gamma, e_1)$
    $(S_2, \tau_2) \leftarrow W(\Gamma S_1, e_2)$
    let β be a *new* type variable
    $S_3 \leftarrow mgu(\tau_1 S_2, \tau_2 \rightarrow \beta)$
    $(S, \tau) \leftarrow (S_1 \circ S_2 \circ S_3, \beta S_3)$
  else if e **matches** $\lambda x.e_1$ **then**
    let β be *new* type variable
    $(S, \tau_1) \leftarrow W(\Gamma_x \cup \{x : \beta\}, e_1)$
    $\tau \leftarrow (\beta S \rightarrow \tau_1)$
  else if e **matches** let $x = e_1$ in e_2 **then**
    $(S_1, \tau_1) \leftarrow W(\Gamma, e_1)$
    $(S_2, \tau) \leftarrow W(\Gamma_x S_1 \cup \{close_{\Gamma S_1}(\tau_1)\}, e_2)$
    $S = S_1 \circ S_2$
  else
    Terminate with an error stating that e is not typeable
  end if
  return (S, τ)
end function

every symbol in the program and just one definition. This is useful in cases where an algorithm (or a function) works on parameters of arbitrary types. A good example of that could be the K combinator (sometimes called 'const function'). We call these algorithms generic.

The problem is that some functions work only on a particular type, for example, basic arithmetic functions. For that reason, we would like to be able to say that it makes sense to add two integers, or two real numbers, but not adding two boolean values.

A trivial extension would be the introduction of tagged unions, see *Typing extensions* in 1.4.1. If we were to define the $+$ operator as $(+) : Int + Real \rightarrow Int + Real \rightarrow Int + Real$, we solve the problem with booleans, but one different problem comes to the surface: we would like to say that adding two integers yields an integer whereas adding two real numbers yields a real number, one more argument could also be that we would like to forbid adding integers to real numbers (maybe we want to require all data conversions be explicit).

So what we would like is some way of expressing that $(+)$ takes two parameters of a type we can perform addition on and that the function yields us a result of the same type. This can be achieved by **type classes**.

Type Classes and their Methods and Instances

The idea of type classes (parametrized by generic types) is that we put constraints on types of some variables (or functions) we use in our algorithms. Those constraints tell us that the usage of these algorithms is valid only in the context where the constraints are satisfied (this allows us to implement otherwise impossible generic algorithms, for example, sorting, that would be nonsensical for some types).

And also, the functions defined inside a type class (we will call them **methods**) can have a specialized implementation for each *instance*. This was the main motivation for their creation: Eq class with a method $==$ which needs to be specialized for each type. [4]

Instances of type classes (parametrized by a type instance of the type of the corresponding type class) are what allows for ad-hoc polymorphism in a very well controlled manner [15], they are a limited (but powerful) alternative to classic overloading (we will describe overloading shortly). It differs from classic overloading in that all instance *methods* have to have typings that are instances of the principal typing of the generic definition. Furthermore, the types of instances are not allowed to overlap (share a common type instance - this can be proven by the unification algorithm) - this "no overlapping" rule will be important for instantiation (generating code with concrete monotypes: overlapping would mean having, for the given parameters, two implementations simultaneously).

Type classes can be added to the algorithm 2 W with only little changes, for example, a simplified version of the implementation presented in Typing Haskell in Haskell (THIH) [6].

Subtyping and Overloading

Subtyping and overloading are slightly different approaches to the same problem we discussed with specifying what types the $(+)$ function can act on.

If we want to use either of those, we have to constrain them somehow, or they make type inference undecidable. We will consider the variant defined by professor Geoffrey S. Smith [13].

Subtyping, however, does not solve that problem very well as if we specify that Int is a subtype of $Real$, then we have to accept that $(+)$ applied to two real operands can still return an integer. That does not seem that bad from the mathematical point of view, but it can lead to implicit data conversions, which

can be undesirable, especially if we were to use the type system in the context of low-level programming.

Overloading, giving a symbol more definitions, then is not only undecidable in the context of parametric polymorphism of HM but it also completely contradicts the ideas of parametric polymorphism, where a function's type should describe the function and its effect on the program (in languages like Haskell or ML, the type also describes data dependencies). [11]

1.5 Overloading with Type Classes

Type classes, as stated before, are good for stating an algorithm makes sense only for some types or that it can have a different implementation for each type, usually to enhance performance by writing specialized algorithms for common or critical types instead of writing a generic slow algorithm.

The usefulness of classes can be shown on 'Ord'. This constraint requires that two variables of type α s.t. $Ord(\alpha)$, we can say 'α is (in) Ord'. But not everything can be partially ordered, take directed graphs, for example, where the reachability of one vertex from another is in 'Ord' only if the graph is a directed acyclic graph, in which case we can use algorithms expecting 'Ord' on problems concerning reachability, but not otherwise.

'Ord' can also show how we could want different instances for different types. Let us say 'Ord' has one method $<=$ of type $a \to a \to Bool$ with the expected behavior.

For two 'Int's we just compare them and return whether the first is lower than the other, but if we have, let us say 'Customer's, with different Id numbers as keys in some map (or dictionary), then we do not need to compare whole customers, but just their Ids.

1.6 Haskell Type System

In this section, we will describe an older version of Haskell, as presented in THIH [6], without any pattern-matching and other features not relevant to the type system. The currently used Haskell type system is significantly more complicated and based on constraint solvers to support various new type system features that are not relevant for the goals of this thesis.

The syntax is that of HM but extended by the possibility of giving explicit typings (with type schemes, see the definition 27) and defining/using type constructors and type classes. See THIH [6] for an actual implementation we will use in the CHM compiler (C with Hindley-Milner).

1.6.1 Type Constructors

Haskell generalizes the HM type system and the HM syntax mostly via the notion of type constructors. If they take no type arguments, we call them nullary. Those are the ones we already call *types* in the HM type system. We call a type constructor n-ary when it takes exactly n type arguments to form a (nullary) value type. For example, the type constructor $(\rightarrow)$ is binary, and it constructs the type of a function.

We can formally define type constructors similarly to function applications:

Definition 24 (Type Constructor in the Haskell Type System)**.** *A type constructor is a pair $TC : k$, where TC is the name of the type constructor, and k is its* kind*. But we usually refer to them only by their name.*

We will perform application on them. This application has a special semantic meaning for each applied type constructor, for example: list a is a list of variables of the type a, and $a \rightarrow b$ ($\rightarrow$ is used in the infix *notation) is a function taking an argument of the type a and returning a value of the type b.*

1.6.2 Kinds

The next thing the Haskell type system adds to the HM type system is the notion of kinds. Kinds are used for the classification of type constructors and for checking the validity of their usage in the program.

Kinds are related to types similarly to how types are related to functions and variables, and they follow their syntactic rules but with type constructors instead of values. The most simple kind $*$ (star) represents all types capable of having a value: all legal types from the HM type system would fall into this category.

Then we have kinds of a form $k_1 \rightarrow k_2$ which take a type of a kind k_1 and return a type of kind k_2. For example a $(\rightarrow)$ type constructor has a kind $* \rightarrow (* \rightarrow *)$, usually written with considering right associativity as $* \rightarrow * \rightarrow *$. This means $(\rightarrow)$ takes two types and returns a new type - this reflects the fact that a type of a function is defined by its parameter's type and its return type.

Kinds are less complex then types as they are defined recursively only from the two rules defined in the previous two paragraphs.

For example, we can define a very common type constructor $List : * \rightarrow *$, which takes a type a and constructs a type representing a linked list of values of type a. We can use this type constructor to show the reason behind the notion of kinds as it would not make much sense for a value to have a type just $List$ without it being applied to any type: "a list of what?"

Definition 25 (Kinds)**.** *For kinds k', k'' a kinds k is grammatically defined as follows:*

$$k ::= * \mid k' \rightarrow k''$$

1.6.3 Types and Type Schemes in the Haskell Type System

We will define types formally as:

Definition 26 (Type in the Haskell Type System)**.** *The types are defined, quite similarly to the definition 17. For types τ', τ'', a type variable α, and a type constructor c (with k being its kind), we define a type τ by the following grammar rule:*

$$\tau ::= \alpha \mid c : k \mid \tau'\tau''$$

Note that we consider the $(\rightarrow)$ operator (used in the previous definition) a regular type constructor of the kind $* \rightarrow * \rightarrow *$.

Type schemes will be defined similarly to those of the HM type system, differing only in what we consider a type and in that there can be constraints that the types have to satisfy. It should also be noted that only types of the $*$ kind can be assigned to values.

Definition 27 (Type Scheme in the Haskell Type System)**.** *For a qualified type qt and n distinct type variables $\alpha_1, \ldots \alpha_n$, we define a type scheme σ by the following grammar rule:*

$$\sigma ::= \forall\alpha_1, \ldots \alpha_n.qt \mid qt$$

Qualified Types and Predicates

A qualified type is a type, with an added set of predicates every instance of it has to satisfy.

Definition 28 (Qualified Type and Predicate)**.** *For distinct type variables $\beta_1, \ldots \beta_m$ such that $\{\beta_1, \ldots \beta_m\} \subseteq \{\alpha_1, \ldots \alpha_n\}$ ($\alpha_1 \ldots \alpha_n$ are taken from the definition 27, qualified types occur always in the context of type schemes), type classes $C_1, \ldots C_m$, and a type τ, we define a qualified type qt by the following grammar rule:*

$$qt ::= \{\beta_1 \in C_1, \beta_2 \in C_2, \ldots \beta_m \in C_m\} \Rightarrow \tau \mid \tau$$

*We call the pair $\beta_i \in C_i$ a **predicate (constraint)**, it means that C_i is required to have an instance for β_i.*

Going forward, we will allow the following form for type scheme: $\forall\{\}.qt = qt$, and for qualified types: $\{\} \Rightarrow \tau = \tau$. This simplifies other definitions so we do not need to consider separate cases (for qualified types, of having some predicates, or none).

Type Substitution in the Haskell Type System With the introduction of qualified types, we should explicitly state that the type substitution applies to qualified types, just as if the predicates' type parameters were a part of the type itself.

Type Unification in the Haskell Type System The addition of type constructors and type classes does not require any change to the definition of type unification. The only notable fact is that unification can succeed only if the unified types share kinds.

1.6.4 Type Inference Rules of the Haskell Type System

The type inference rules for the Haskell type system are almost the same as the type inference rules for the HM type system, but we have to add rules for the inference of the type class constraints:

Definition 29 (Inference rules of the Haskell Type System)**.** *For types τ, τ', expressions e, e', type variable x, set of type variables $\bar{\alpha}$, assumption set Γ (where Γ_x stands for the set of assumptions without any assumption about x), and predicate sets $\mathbf{P}, \mathbf{P}'$, the HM system with predicates is defined by the following set of inference rules:*

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \textit{ (Variable)}$$

$$\frac{\Gamma \vdash e : \mathbf{P} \Rightarrow \tau' \to \tau \quad \Gamma \vdash e' : \mathbf{P}' \Rightarrow \tau'}{\Gamma \vdash ee' : \mathbf{P} \cup \mathbf{P}' \Rightarrow \tau} \textit{ (Application)}$$

$$\frac{\Gamma_x \cup \{x : \mathbf{P}' \Rightarrow \tau'\} \vdash e : \mathbf{P} \Rightarrow \tau}{\Gamma \vdash \lambda x.e : \mathbf{P} \cup \mathbf{P}' \Rightarrow \tau' \to \tau} \textit{ (Abstraction)}$$

$$\frac{\Gamma \vdash e : \sigma \quad \sigma \geq \sigma'}{\Gamma \vdash e : \sigma'} \textit{ (Instantiation)}$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \textit{ not free in } \Gamma}{\Gamma \vdash e : \forall \alpha.\sigma} \textit{ (Generalization)}$$

$$\frac{\Gamma \vdash e : \forall \bar{\alpha}.\mathbf{P} \Rightarrow \tau \qquad \Gamma_x \cup \{x : \forall \bar{\alpha}.\mathbf{P} \Rightarrow \tau\} \vdash e' : \mathbf{P}' \Rightarrow \tau'}{\Gamma \vdash \textit{let } x = e \textit{ in } e' : \mathbf{P} \cup \mathbf{P}' \Rightarrow \tau'} \textit{ (Let-polymorphism)}$$

Those rules are the same as those of the HM type system with only a trivial difference that complex expressions retain predicates of all of their subexpressions.

1.6.5 The Type Inference Algorithm for the Haskell Type System

The type inference algorithm has to account for the bottom-up propagation of predicates that the inferred types have to satisfy. The basis for the inference algorithm is the algorithm W 2. For the definition, see THIH [6]. In THIH, `tiExpr` is the part of type inference that encompasses the extension of the algorithm W. The definition in THIH differs slightly from what we have discussed in this chapter in that it does not consider just the syntax from the definition 15 with the extensions we presented, but it also considers some features of the Haskell language (for example pattern matching).

Chapter 2

Type inference and polymorphism for C

2.1 Reasons for Polymorphism Features

The C language is one of the main languages for system development. It is valued for its simplicity, for its transparency, and for giving the user high control over the memory management.

But then C lacks any support for polymorphism and generic programming. This is often compensated by extensive use of macros and `void *` pointers, which do not offer any type safety and thus can be easily misused.

Generic (`void *`) pointers can point to any data, and thus they can be used in some "generic" code. This makes them a very strong tool, and almost every program relies on them (for example: `malloc`, `free`; for some examples see [9]). However, these pointers carry no type information and so to use these pointers. This means that either the user has to know the actual types when they use them, or the type has to be passed explicitly.

One example of using void pointers can be 'malloc', which, without any type inference, requires the user to calculate the required themselves (this can seem to be trivial to do, but it still generates some code noise, and that is a possible source of bugs).

As we stated before, one other common way of implementing generic algorithms and data structures is by using **macros**. However, if we implement a data structure as a macro, all required instances of it and dedicated procedures working on them have to be instantiated explicitly by the user. This can lead to many hard to find bugs.

Thankfully many modern compilers are macro-aware, and so if there is a compile-time bug in the program, the compiler can create a user-friendly error

output (for example, GCC).

The lack of any type checking in the context of macros makes the possible bugs manifest only when their implementations are actually used in some code, and sometimes only in certain contexts, as the compiler, otherwise, should not even assume a macro represents a code.

As for generic pointers, a bug usually becomes apparent only during runtime and that by the corruption of the memory.

One real-life example of making "generic" data structures via macros is klib's `__KHASH_TYPE` macro (in './khash.h') [1]:

```
#define __KHASH_TYPE(name, khkey_t, khval_t) \
    typedef struct { \
        khint_t n_buckets, size, n_occupied, upper_bound; \
        khint32_t *flags; \
        khkey_t *keys; \
        khval_t *vals; \
    } kh_##name##_t;
```

If we were to create an instance of this structure where `khkey_t=int` and `khval_t=float`, we would have to use the following macro to instantiate the structure explicitly: `__KHASH_TYPE(name, int, float)` and also we would have to instantiate all its functions, though we can wrap this into yet another macro.

This approach also requires an explicit mention of the instance in the corresponding function calls.

For example, in this case, we would clear the structure by calling `void kh_clear_<name>(const kh_<name>_t *h);` where `<name>` stands for the name we have given for the instance. So if we were to refactor a part of the program and change the type of something, we would have to change the name as well, and sometimes create a new instance by hand or delete the old one if it is no longer useful - which we would have to investigate ourselves. In large projects, this can cause a lot of problems in case of even a little miscommunication.

Using type checking and type inference described in this thesis has an effect very similar to the effect of instantiating definitions via macro expansion, but with the benefit of brevity, catching bugs early (due to checking the implementation when it is defined) and having fewer requirements on mindfulness of the user. All of these effects can lead to less error-prone code.

Adding generic programming features to C thus can help in debugging programs written in this language and to make writing generic code more user friendly and easier to program in and maintain.

2.1.1 Why the Hindley-Milner Type System

The C language has great support for the procedural programming paradigm with records and data are clearly separated from computation in the C code.

There have been many attempts to add object-oriented features (most notable being C++).

Adding the features of the HM type system does not change the core nature of the language. It just facilitates creating generic abstractions, and with the addition of type classes, it adds support for both simple and expressive ad-hoc polymorphism.

We will present only slightly modified syntax of the C language, which will give a type-safe alternative for many cases where we would otherwise use the aforementioned `void *` pointers or macros.

2.2 λ_C Modelling C Types in HM

In this section, we will specify how we will model the type system of C in the HM type system so we could extend it. We will use λ_C to represent the projection of C types to HM formally (we will then extend this function for expressions). The modeling is quite straightforward, but we will give our reasoning and explanation for each case after giving a formal definition.

2.2.1 Integral Types

Definition 30 (λ_C for integral types)**.** *If t is an built-in type recognized by the C language, we will set:*

$$[[t]]_{\lambda_C} = t$$

.

All main C integral types (like `int` or `char`) can be trivially modelled as type constants, we can model signed and unsigned versions as either two separate types or by using `unsigned` and `signed` type constructors. Though to make the system as consistent as possible in this matter, we propose considering sequences of type specifiers like, for example, `unsigned long long int` as single type constants. This means that type specifier sequences containing `unsigned` would be illegal if they are illegal in C (same for the other type specifiers).

As a side note, this is not considered by the CHM compiler proposed in this thesis as it not an important issue for the demonstration of the effect of the HM type system and polymorphism on the C language.

2.2.2 Pointers

Definition 31 (λ_C for pointers)**.**

$$[[t\ *]]_{\lambda_C} = (*)\ [[t]]_{\lambda_C}$$

where $(*)$ *is the pointer type constructor.*

We argued that certain sequences of type specifiers (naming integral types) should be considered as single type constants. But pointers do not fit this mindset, and we want to use type constructors for them. That is because, whereas `unsigned long long int` and `long long int` can be considered unrelated types, we want to dereference pointers easily and a generic dereference function would be impossible if we considered types of pointers unrelated to their value types.

We will model the pointers by applying $(*)$ type constructor applying to the types of values they point to (`int *` will be modeled as `(*) int`).

This gives the reference `&` and dereference `*` functions obvious types $\& : \forall a.a \to (*)a$ and $* : \forall a.(*)a \to a$.

2.2.3 Const specifiers

One problem with C types in the context of the HM type system is that C uses the `const` type specifier, and its representation in HM cannot be compatible with its C meaning. We will discuss it later in this section.

We propose that `const` has to be handled separately from the type system and so we will omit the `const` specifier in this thesis and leave it for future work.

One reason for the incompatibility is that we can assign type 'a' to 'a' and also 'const a' to 'a', but not the other way around in the latter case. Keeping this behavior, and considering `const a` a type, would imply introducing subtyping to the type system (`const a` being a subtype of `a`), and this has its own issues [11].

Handling the `const` specifier similarly to the way we handle pointers (this would mean explicit conversion when assigning from `const` types) would make a trivial assignment of `const` type to a non-`const` unnecessarily verbose (which is usually considered a bad language design). Then we would need to prevent assigning `const` to another `const` of the same type, which brings even more issues, either making the code even more verbose or checking it during instantiation, which is the thing we want to avoid by type checking.

2.2.4 Functions

Definition 32 (λ_C for functions)**.**

$$[[t(p_1, \ldots p_n)]]_{\lambda_C} = ([[p_1]]_{\lambda_C}, \ldots [[p_n]]_{\lambda_C}) \to [[t]]_{\lambda_C}$$

note that here the parentheses around parameters p_1 through p_n on the left side of the assignment are syntactical. $\rightarrow$ is the function type constructor.

Types of C functions can be modeled in the following way:

We will consider the parameters $p_1, \ldots p_n$ of the given function being a tuple of type T, and the return value of type R, then the type of the function will be modeled as $T \rightarrow R$.

Calls to the function will be modeled as applying the function to the tuple of parameters.

2.2.5 Structs and Unions

Definition 33 (λ_C for structs)**.** *If s is a struct recognized by the C language, we will set $[[s]]_{\lambda_C} = s$.*

Structs and Unions are each considered a different type even if they share all of their members. This approach is common among all languages based on C.

2.3 Modelling C

Here we will use λ_C introduced in the previous section to denote transformation from C to the syntax of the Haskell extension of lambda calculus, this transformation can be done in a multiple of ways, and we will present only one such way, not necessarily the most efficient one.

For keeping the model simple, we will allow ourselves to disregard the imperative structure of the code, most notably, we will ignore whether a statement is inside a block following a certain control statement, or it is outside of such block - this is not relevant for type checking nor type inference.

We will also allow ourselves to reorder statements as long as they are inside the same function (or in the global scope).

Models of some C constructs can be as follows (grammar is taken from [9]):

2.3.1 Expressions and Statements

Definition 34 (λ_C for binary operators)**.** *We will model most expressions very intuitively:*

$$[[add\_expr + mul\_expr]]_{\lambda_C} = [[add\_expr]]_{\lambda_C} + [[mul\_expr]]_{\lambda_C}$$

similarly for unary (and ternary) operators

This example demonstrates the most simple case where we can model the C construct directly, but there are more tricky examples like the following one (an initialization in a context of a function body):

Definition 35 (λ_C for binary initializations)**.**

$$[[decl\_specs\ ID = init; \ldots]]_{\lambda_C} = (\lambda\ ID : [[decl\_specs]]_{\lambda_C}.[[\ldots]]_{\lambda_C})[[init]]_{\lambda_C}$$

(The dots represent the rest of the function body)

Initialization creates binding (as stated before), so we have to model that by creating a new abstraction and put the whole part of the function's body that follows this initialization (this, in effect, means that all the return values of these nested functions will share the same type as the function which contains them - the function the assignment appears in).

This contrasts with the assignment operation, which just requires the parameters to be of the same type (this will be the type of its return value as well). This means that all initializations have to be represented as abstractions (or equivalently to that), while all assignments can be considered the trivial case.

A similar thing is switch statements where the switch "call" can be modeled as initialization to an anonymous variable and the single case statements as equality comparisons to this variable.

Apart from switch statements and assignments, we can model almost everything in a similar fashion to the first example. One notable exception being return statements, which will be handled separately.

Control Statements

We can completely ignore the structure of control statements, disassembling them into separate statements - however, we can give their conditions and bodies special names.

The only requirement is that the expression in the condition has to be *zero-comparable*. This implies a type class for types that have a zero (null) value (pointers, integers, booleans). However, even that is arguable because requiring the user to write an explicit zero comparison (`x == 0` in the case of an integer, or `x == null` in the case of a pointer) can be seen as a feature supporting readability. We will leave this decision for future work.

2.3.2 Function definitions

First, we will define how we model functions where there are no inner initializations (and switch statements):

We will model those functions as an abstraction of a sequences of nested let clauses (each for every statement in the function body, every time assigning to a fresh new variable) where the innermost is of a form let $\ldots$ in e where e is of the form return e'' e' where e'' is one of the return expressions and e' is defined as e without considering e'', the last e' when we have no more return expressions is a new virtual constant of the return type. The return function has the typing $\forall a.a \to a \to a$ and it is used to connect the types of return expressions.

Then we will generalize this definition for the function where there are some inner initializations:

First, we move all initializations before all other statements, and then we model them as shown in section 2.3.1.

2.4 Extensions to the C language

We showed we could model the C language constructs in Haskell so we can easily apply type inference and from inferred types do instantiation of generic code (we will add to the language) to its intended monotypes.

We will extend the language by *type classes* and *type variables*, the syntax and decisions behind it will be explained in the following chapter.

Chapter 3

CHM compiler

Our implementation of the CHM compiler uses preprocessing and postprocessing by the GCC compiler, with the intermediate representation being the C language. GCC runs the preprocessor and then compiles the fully instantiated code.

Using GCC for parsing and the "second stage" of the compilation allows us to focus on polymorphism, type inference, and type checking instead of dealing with actually interpreting the code, but that brings its toll on the efficiency of the whole process, and a slightly different approach should be considered in future works.

3.1 Language extensions

The C language is extended by type variables so we can define polytyped functions and structures and by type classes, which offer a way of controlled ad-hoc polymorphism. Type classes allow for defining specific (more efficient) implementations to generic algorithms or for restricting types of parameters.

We will borrow the application of these extensions from the Haskell language. However, the two languages in question are very different. In Haskell, there is a clear distinction between variables and non-variables and between values and types, but in C, we do not have this distinction grounded in the language, and we want to preserve it this way as much as possible. So we will require every type variable be declared before it is used (like in the listing 1).

So to preserve the syntactic principles of the C language, we will use a syntax similar to languages like C++, Java, C#, etc. Those languages have become very popular, and so it should not be a bad choice.

Listing 1 CHM Type Class with an Implicit Type Parameter

```
<a>
class Cloneable
{
    a *clone(a *);
}
```

Listing 2 CHM Type Class with an Explicit Type Parameter

```
<a>
class Cloneable<a>
{
    a *clone(a *);
}
```

Listing 3 CHM Type Class Instance

```
instance Cloneable<int>
{
    int *clone(int *value)
    {
        int *copy = new();
        // new is a generic function replacing malloc
        *copy = value;
        return copy;
    }
}
```

Listing 4 CHM Polytype Function

```
<a>
a shallow_copy(a value)
{
    return value;
}
```

3.1.1 Polytyped Definitions

CHM Headers

In the listing 1 we see the part `<a>`, declaration of the type variable `a`, similar to C++ templates and Java generics, always preceding the rest of the definition and with a scope local to the definition. This part, a `chm_header`, has the following grammar rule:

Definition 36 (CHM header)**.**

`chm_header` := `<type_decls>` | `<type_decls : constraints>` | `< : constraints>`

where the `type_decls` *is a list of declared type variables;* `constraints` *is a list of predicates the type variables have to satisfy and new type definitions (which are very similar to typedefs).*

The predicate `constraints` follow principles similar to Haskell. Notable use specific to CHM is they allow for member accessors, see the section 3.4.2 *record fields*.

Type Classes, Their Instances, and Structs

The listing 1 shows how we would define a simple **type class**, here we declare only one type we use in declarations of the class's methods and so it is implicitly considered its type parameter as well. The type parameter can be specified explicitly following the name of the class, which we can see in the listing 2.

We will then use a similar syntax for application of types to the class's **instances**, an example given in the listing 3.

For **structs** we will use the same syntax as we use for type classes.

Functions

Functions will use the very same syntax (see the listing 4), however with slightly different behavior, they do not have any type parameters, and instead, they have type schemes such that their qualified types (containing the predicates specified following the type declarations) are quantified over the declared types.

3.2 Parsing

Parsing of the code is handled by the language-c project [5], originally meant for C99 (and later for C11), which is slightly modified into language-chm to support the extensions we introduce.

This project is used for the output of a pretty-printed C code as well. This output is then inputted to GCC for finalizing the compilation.

3.3 Type Inference

For the type inference, we use the THIH project [6], but using it sensibly required some modifications to be made to it as the original THIH project is not meant to be used as an efficient implementation.

Using THIH required a significant pre/postprocessing of the code described in the previous chapter.

All type inference principles are explained in the first chapter. The basis for the implementation is then explained in Typing Haskell in Haskell [6].

3.4 Code Processing

The main parts of preprocessing the code are in the AST-to-THIH project. Post-processing is then handled in the CHM-instantiate project, which also passes the code to AST-to-THIH for preprocessing. Both of these projects are specifically built for this thesis as part of the CHM compiler.

3.4.1 Monomorphization and Instantiation

The process of instantiation of functions begins with monotype functions and then continues recursively when we determine the monotypes of all occurrences of polytype functions called from them, similarly for types (and namely structs).

We will describe the process only on functions, but the principles of it carry to structures as well.

We start with monotype functions as we can already determine their monomorphic type. We replace all polymorphic symbols (functions, structs, and types) in their definitions with unique symbols (distinct even if the original symbols were the same), each assigned a new unique type variable.

We run the type inference algorithm on this rewritten definition, and then if we successfully determined each monotype, we instantiate each of the original symbols given the new inferred type (unless we have done so already - this implies we have to keep a record of all instances).

Instantiation of polytyped functions is done by simply replacing the quantified type variables with the inferred types. From there, the process repeats for each new instance as if the instance was a regular monomorphic function. Type class methods are an exception to this, which we will discuss in the very next

part, but the only difference is that there is one more step before we instantiate them.

Type Class Method Instantiation

All type classes are just language constructs, they are not manifested in the resulting code in any shape or form, and thus they count as a zero abstraction feature. There are some built-in type classes (for example, `Add` which states that two variables of a type that satisfies it can be added), implicit type classes for each field of a structure (and a union), and then user-defined type classes, showed before in this chapter. The implemented built-in and implicit type classes are mainly for demonstration, and they are subject to change in the future.

Definitions of type classes are stored in the compiler as lists of their methods. Methods act like regular polytypes until we instantiate them.

When a method is instantiated, the compiler checks its instances and determines which one (and if any) matches the inferred type. Then it instantiates this one method instance just as if it was the only definition. Note that type class instances are required not to overlap because of this step of instantiation.

Comparison to C++ Template Instantiation

One could say this project just mirrors template instantiation from C++, but any such resemblance is just superficial. C++ template instantiation is not strict in type signatures, and different instances can have non-matching types (not having a principal typing for the API of the instances). This goes against parametric polymorphism. C++ type system, on the other hand, has its own advantages, for example, variadic templates, which are impossible to implement in the HM type system.

3.4.2 Record Fields

Implementing record fields (struct members) was challenging as the way record fields work in C has no real equivalent in the context of functional programming, although Haskell's object notation or ML's records come pretty close to it.

Record fields are implemented in such a way that all fields of a certain name have to share the exact same type. This is a slightly restricted requirement of each field having a unique name.

The reason for such requirements is that accessing a field is a function taking the record and returning the field. The record has to have this field, which is easily solved by a type class (parametrized by the record's type and named after

the field) the compiler implicitly generates when the field is defined. This type class has exactly one method, which is the accessor of the given field.

This approach perfectly matches the syntax of C, where we know the name of the field and the variable which is supposed to have such a field. The field's type is not specified when we access it, so it has to be inferable from the variable alone. And the easiest way to satisfy that is to require each field have a unique name (this is how it is implemented in Haskell and ML, for example).

3.4.3 Recursion limits

It is indeterminable whether the instantiation of something continues forever. We can prove this using the Post correspondence problem.

So there is a limit of type depth set to 500 (this limit should not affect any sensible program, but if need be, it can be changed using the `--depth` option). If some type exceeds this limit, a compilation error will be outputted, stating what the compiler tried to instantiate at that time.

3.5 Code output

The CHM compiler implementation outputs a GCC-compilable code that interprets the CHM code without any runtime overhead.

The user can use the `--inner` to see the inner Haskell representation of the CHM code (it blurs some of the semantics unnecessary for type inference and leaves others to facilitate the readability of it). This can be used for debugging the compiler implementations.

It should be noted that names of symbols of polytype functions used in the C interpretation of the CHM code do not match their CHM counterparts as the names are mangled, which currently complicates debugging of CHM programs (this, however, can be solved easily on the debugger side).

Chapter 4

Conclusion and Future Work

4.1 Conclusion

We have shown that it is possible to efficiently model the type system of the C language and to model most of the language's constructs in the language of the lambda calculus variant derived from Haskell.

Section 4.2 then showcases a code that might be used in some projects using the CHM language. See appendix C for more code examples that implement some common standard library features (functions we can encounter in the C++ standard library, for example).

The implementation of the CHM compiler provides several test files, each beginning with a description stating what the file tests and what is the program's expected behavior.

The CHM compiler, as it is currently implemented, serves only demonstrative purposes. Its implementation is partially incomplete and not fit for any use in software development. Yet, all code examples in this thesis are correctly parsed and interpreted based on the definitions in the previous chapters.

4.2 Possible Use in System Development

Here we will show the usefulness of this language in some existing code from popular system development projects.

One area where it would be especially useful to support polymorphic code is *polymorphic data structures*. We have discussed their implementation in C using mainly macros and a possibility of alternative CHM type-safe implementations in chapter 2. A concrete example of this is polymorphic lists.

We can encounter such lists, for example, in the linux kernel, their implementation is defined in *./include/linux/list.h* [14].

Simple polymorphic lists can be implemented as follows (note that this implementation is simplified and the push function is suboptimal for nontrivial data):

```
<a : b ~ struct List : <a> >
struct List
{
    b *next;
    a value;
};
```

Then we can implement its functions as follows:

```
<a : b ~ struct List : <a> >
b *push(b *head, a value)
{
    b *item = new();
    // new() is an STL function returning malloc(sizeof(b))
    item->value = value;
    item->next = head;
    return item;
}

<a : b ~ struct List : <a> >
b *pull(b *head)
{
    b* next = head->next;
    delete(head);
    return next;
}
```

In C we could create a similar structure using BSD `<queue.h>`'s [10] `SLIST_` macros, which is yet another example, quite commonly used. Using this library requires us to create each list type instance separately and in invocations of its functions it requires us to pass additional parameters. For example SLIST's equivalent of our `pull` function, `SLIST_REMOVE_HEAD`, takes as an additional `entries` argument (a structure member of the entry node that connects the elements constructed by the `SLIST_ENTRY` macro) on top of the expected `head` argument.

4.3 Future Work

If this project gains popularity, future work should extend the STL library, and a new compiler should be implemented, which would not rely on GCC. That would make it easier to implement new features and optimize the process of compilation.

Incomplete Implementation The provided CHM compiler implements the features crucial for the demonstration of the effect type inference and polymor-

phism would have on development in C. Those features are (generic) functions, pointers, structs, control statements, and almost all expressions.

There are currently implemented only some of the built-in types (the main ones) as their support is not detrimental for the demonstration, and it was not possible to make a complete implementation in time.

Some more notable constructs not implemented by the current compiler are typedefs, enums, and type-checking of zero-comparability in (control statement) conditions. All of those should be implemented in future implementations of the compiler.

Compiler Errors The current implementation of the CHM compiler outputs error messages stating the reason behind the error and the location in the code that was the source of the error. That is not the case for errors that happen during the type inference phase of compilation as the THIH [6] project would need some more changes to the definition of expressions (that model the code) to know from where in the code they originate. But without some extensive rework of the compiler, such an implementation would break the compiler's modular nature as the locations are implemented in the language-chm project (language-c, see [5]).

Future implementations of the compiler should massively refactor the code, change the representation of the compiled code, and make the errors more readable and more specific.

4.3.1 Future Features

Quality of Life Changes In the current implementation of the compiler, we allow declaration of type variables only at the CHM headers of the various definitions. Which is enough for demonstration, but in a more convenient-to-use implementation, the language should allow declaration of local type variables even inside function definitions. Also, it should provide a notion of anonymous/implicit type variables (similar to C++'s `auto`) which would decrease the verbosity of the code in many cases.

Better support for record fields The current specification of record fields in CHM is that if two record fields of different records have the same name, they shall have the same type as well (which can be ambiguous), see the section 3.4.2 *Record Fields*.

In the future language specification, the specification for the polymorphic fields should be that their types are equally dependent on the type parameters of the struct which they are fields of and for all possible types of those parameters,

they still share the type (example: if one field has a type that is the last type parameter of the parent struct, every other field sharing its name have to be typed after the last type parameter of its parent struct as well, ditto for second to last, etc.; and similarly for all composite types).

There should be a compiler option that would forbid fields from sharing their names, at all, and another one that would just trigger a warning unless the compiler is not able to determine which one to use in a certain context.

Then another sharing of names of fields should be defining fields in type family definitions, where all specializations would have these fields without any warning or error even under the effect of the option stated above. Different specializations would then still be able to add their own field on the top of the shared ones. The implicit `Has_<field>` class would apply here as well.

Modules Having many structures with even more fields and procedures would create a high probability of name collisions. The "include" system from the C language then becomes an issue for larger projects or even smaller ones wanting to use some foreign code.

And therefore, this language could prosper from having a module system akin to many modern languages (like Haskell), which would make it possible to export only parts of the code we want to export or give them qualified types (both to avoid the name collisions).

Data type specializations Data type specialization is not currently supported, but a great candidate matching the current nature of the language would be type families (for example, used in Haskell).

Their addition would be hugely beneficial for the language, just like the partial specialization of struct templates is for C++, as those mechanisms can be considered equivalent in many cases of use.

Their use would be especially helpful for optimizing implementations, for example:

If we want to remember arbitrary integral numbers with repetition, we could use a map from numbers to their counts, however for some very limited ranges (or for booleans), we could implement the semantically equivalent data structure using static arrays few counters.

Bibliography

[1] MIT Attractive Chaos. *Klib: a Generic Library in C*. Github Repository. 2020. URL: https://github.com/attractivechaos/klib.

[2] Henk P Barendregt. "Lambda calculi with types". In: (1992).

[3] Luis Damas and Robin Milner. "Principal type-schemes for functional programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1982, pp. 207–212.

[4] Cordelia Hall et al. "Type classes in Haskell". In: *European Symposium On Programming*. Springer. 1994, pp. 241–256.

[5] Benedikt Huber et al. *Language.C*. Github Repository. 2020. URL: https://github.com/visq/language-c.

[6] Mark P Jones. *Typing haskell in haskell*. Vol. 7. 1999.

[7] Jiří Klepl. *CHM Compiler*. Github Repository. 2020. URL: https://github.com/jiriklepl/Bachelor-Thesis.

[8] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.

[9] IS Organisation. *ISO/IEC 9899: 1999 Programming Languages-C*. 1999.

[10] FreeBSD Hypertext Man Pages. *QUEUE*. 2007.

[11] Jens Palsberg. "Overloading is NP-complete". In: *Logic and Program Semantics*. Springer, 2012, pp. 204–218.

[12] John Alan Robinson. "A machine-oriented logic based on the resolution principle". In: *Journal of the ACM (JACM)* 12.1 (1965), pp. 23–41.

[13] Geoffrey S Smith. "Polymorphic type inference with overloading and subtyping". In: *Colloquium on Trees in Algebra and Programming*. Springer. 1993, pp. 671–685.

[14] Linux Torvalds et al. *Linux kernel source tree*. Github Repository. 2020. URL: https://github.com/torvalds/linux.

[15] Philip Wadler and Stephen Blott. “How to make ad-hoc polymorphism less ad hoc”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 60–76.

Appendix A

The CHM Language Grammar

This is the grammar for the CHM language. The meaning of all constructs define here is defined in chapter 3 (*CHM Compiler*).

A.1 CHM headers

Polymorphic functions and structures are constructed using special annotations (called CHM headers) following grammar:

```
chm_header = '<' type_decls '>'
           | '<' type_decls ':' constraints '>'
           | '<' constraints '>'

type_decls = type_decl
           | type_decls ',' IDENT

constraints = constraint
            | constraints ',' constraint
constraint = new_type
           | predicate

new_type = IDENT '~' type
predicate = CLASS '<' type '>'

type = DECLARATOR
     | DECLARATOR ':' '<' types '>'
types = DECLARATOR
      | types ',' DECLARATOR
```

This is a simplified version of a piece of the actual grammar used in the implementation.

- 'IDENT' is a new identifier

- 'CLASS' is the name of a type class
- 'DECLARATOR' is a declarator from the C language (a type)

A.2 Classes and Their Instances

The other addition CHM brings to C are classes. They are defined (simplified version) as follows:

```
class = chm_head 'class' IDENT '{' ext_decls '}'
      | chm_head 'class' IDENT < typevar > '{' ext_decls '}'

instance = 'instance' IDENT '<' type '>'
               '{' ext_decls '}'
           chm_head 'instance' IDENT '<' type '>'
               '{' ext_decls '}'
```

`ext_decls` are declarations and definitions which would be legal in the global scope. However, it is not allowed to put declarations in instances, and all their definitions have to be first declared in the matching classes.

A.3 Polymorphic Functions and Structures

Polymorphic Functions and Structures are defined as regular C constructs preceded by *CHM headers*:

```
CHM_function_definition  = chm_header function_definition
CHM_function_declaration = chm_header function_declaration
CHM_struct_definition    = chm_header struct_definition
```

We omit **unions** as a lot of grammars (including ours) consider them syntactically equivalent to structs.

Appendix B

Mangling of Types, Structs, and Functions

The CHM compiler creates a precompiled source code, which is then compiled by GCC. The precompilation process involves instantiation of polymorphic language constructs, which leads to multiple occurrences of certain definitions in the precompiled code (each with a different typing).

We know which of the symbols with the same name are connected to which (again, determined by having the same typing). To encode those connections into the precompiled code, we use a mangling of names by appending reversible textual representations of the typings.

Choosing this approach, over a simple numbering of the instances, facilitates both the implementation (as we do not need to propagate the mangled names in the compiler) and possible debugging and reverse-engineering of the resulting precompiled code. A downside is that this approach significantly inflates the code.

Note that we have to apply the mangling function only to instances of polymorphic symbols.

B.1 The Mangling Function

Here we describe a simplified definition (algorithm 3) of the mangling function that is used by the compiler. In its definition, we will use *concat* function to concatenate strings, and *length* function to get the decimal representation of their length.

Algorithm 3 The Name-Mangling Algorithm

1: **function** MANGLE(τ)
2: **if** τ is a built-in C type **then return** C representation of τ
3: **else if** τ is a function of the form $(\tau'_1, \ldots \tau'_n) \to \tau''$ **then**
4: params $\leftarrow$ concat(mangle(τ'_1), …mangle(τ'_n))
5: ret $\leftarrow$ mangle(τ'')
6: **return** concat('TF', length(params), params, length(ret), ret)
7: **else if** τ is a pointer to τ' **then**
8: **return** concat('TP', length(mangle(τ')), mangle(τ'))
9: **else if** τ is a type application of the form $\tau'\tau''$ **then**
10: left $\leftarrow$ mangle(τ')
11: right $\leftarrow$ mangle(τ'')
12: **return** concat('TA', length(left), left, length(right), right)
13: **else if** τ is a type constructor named TC **then**
14: **return** concat('TC', length(TC), TC)
15: **end if**
16: **end function**

Appendix C

The STL Library

In every language, there are many functions and patterns used in almost every program. We can expect the same happening to the CHM language if it becomes popular.

We present a demonstrative *STL* library implementation, which should give an example of how some functions provided by it might look. This library is implicitly linked with every program.

The *STL* library is split into two files `stl.h`,which can be included to CHM sources, and `stl.c`, which implements some IO functions currently unimplementable in the CHM language.

C.1 Functions Facilitating Memory Management

Here we will present functions `new` and `delete` for polymorphic memory allocation and disposal, respectively; and a function `nullptr` which serves as polymorphic `NULL` pointer.

The following code is taken from `stl.h`:

```
extern void *malloc(size_t);
extern void free(void *ptr);

<a>
a *nullptr()
{
    return (a *)NULL;
}

<a>
a *new()
{
    return (a *)malloc(sizeof(a));
```

```
}

<a>
void delete(a *where)
{
    return free((void *)where);
}
```

C.2 Functions Facilitating IO

Here we show how the *STL* library could implement a generic `print` function. It is a method of the `Print` type class, as every type requires a specific implementation of how it should be printed.

The implementation shown here takes the object that is to be printed by a pointer, which should not matter for trivial types, as the functions will likely be inlined by GCC, but for larger objects (requiring a nontrivial implementation of printing), passing the object by value might cause unwanted overhead.

The following code is again taken from `stl.h`:

```
// the following two functions implemented in stl.c
extern void __print_char(char);
extern void __print_int(int);

<a>
class Print {
    void print(a *);
}

instance Print<char> {
    void print(char *a)
    {
        __print_char(*a);
    }
}

instance Print<int> {
    void print(int *a)
    {
        __print_int(*a);
    }
}
```

Appendix D

Using the CHM Compiler

In this appendix, we will discuss how to use the demonstration implementation of the CHM compiler.

D.1 Getting the Compiler

Its source code can be viewed and downloaded via Github [7].

Note that the compiler requires all its submodules to be up-to-date. That can be achieved using the following command to clone the repository:

```
git clone --recursive \
  https://github.com/jiriklepl/Bachelor-Thesis
```

D.2 Build Dependencies

The main dependencies of the compiler are *ghc* (at least version 8.10) and *cabal-install* (at least version 3.2). It depends on many packages automatically acquirable via *cabal-install* which can be found in each sub-project's `.cabal` file.

D.3 Building

The package can be built with either of the following commands:

```
make build

# or
cabal new-build all
```

D.4 Usage

The compiler can be run by either of the following commands (note that <arguments> stands for the arguments passed to the compiler):

```
./run.sh <arguments>

# or
cabal new-run CHM-main <arguments>
```

The first one might require adjusting the read and execution permissions on the `run.sh` file.

The implementation is for testing and development of the language and thus the directory it is run in has to contain the files `stl.c` and `stl.h`, which implement the *stl* library implicitly linked with every program.