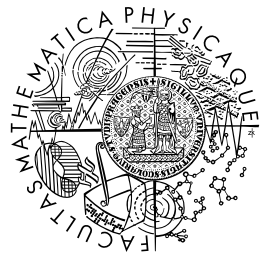

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jiří Beneš

C++ linter based on linear types

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 RNDr. Miroslav Kratochvíl for his generous advice and feedback throughout the entire process of creating this work. I would also like to thank my family and my girlfriend Katka for their relentless support and kind encouragements.

Title: C++ linter based on linear types

Author: Jiří Beneš

Department: Department of Software Engineering

Supervisor: RNDr. Miroslav Kratochvíl, Department of Software Engineering

Abstract: Low-level programming requires careful management of system resources, most notably memory. In C++ programmers are encouraged to follow idioms like RAII and smart pointers to handle resources correctly as violating them leads to unsafe code.

Typed functional programming languages guarantee safe automatic memory management, but are often sub-optimal in handling system resources. A nice, formal solution to handling resources naturally is linear types. Unfortunately, existing languages that support linearity are cumbersome and require explicit, complicated annotations from the programmer.

We bridge the two worlds by exploring a novel combination of C++ and linear types. We describe a new type system with linearity for C++ by using constrained qualified types, while requiring no additional input from the programmer. The applied result of our work is called Lily, a static analysis tool for C++ using the Clang compiler infrastructure. Lily can statically detect large, general classes of issues, some of which are not detected by common state-of-the-art tools for C++.

Keywords: type systems linear types C++ static analysis

Contents

Introduction

C++ is a very complicated language, as witnessed by the over 1500 pages in the 2017 version of the C++ standard [ISO17]. It is reasonable to assume that the average C++ programmer has a mental model of the language which corresponds to a mere subset of the actual standard. This in turn leads to sub-optimal decisions by the programmer, especially with regards to managing resources manually. Mismanagement of resources is both common and costly [Sze+13].

To ease this burden, a rule of thumb called 'Resource Acquisition Is Initialization' (RAII) [Str13] is commonly used. The intention is to pair resource allocation and object initialization together. Similarly, resource deallocation and object deinitialization are also paired together. This results in resource management being tied to a lifetime of an object.

Going even further, the C++ community adopted the concept of ownership via smart pointers — each object is owned either uniquely by an another object, or together by a group of objects. When an object is destroyed, all its owned objects and therefore resources are destroyed as well. This is a valuable low-level compromise between automatic and manual memory management.

On the other end of the spectrum of resource management, there are highly sophisticated functional programming languages which are very high-level and thus quite safe, especially with regards to handling resources. The behavior of such programs is heavily determined by their types, but since the programs are very high-level, one cannot tweak their low-level behavior much. Therefore such languages often rely on complex automatic memory management which is often sub-optimal.

Our contribution is an attempt to bridge together the two worlds, taking the safety in types approach from functional programming and pairing it with the low-level control of C++. The aim is to take a valid, unmodified C++ source file, extract type constraints, infer advanced linear types and interpret the results back in the realm of C++. Note that our goal is to do this while requiring no additional input from the user except for the source file.

In this thesis, we present all the necessary details to achieve said goal in a significant subset of C++. We also present Lily, a proof-of-concept linter for

C++ based on linear types, which can detect several kinds of issues in resource mismanagement statically, i.e. without actually running the code. Although a large subset of these issues is detected by other state-of-the-art C++ linters, we also found several general classes of programs with gross resource mishandling, which are detected by Lily and not by other linters.

In order to do achieve that, we build upon Girard's linear logic [Gir87; Wad93] and Wadler's linear types [Wad90] to model precise types of single-use functions and arguments which must be used exactly once. We also use Morris's linear qualified types [Mor16] to give linear types an accessible façade.

Thesis layout

In the first chapter, we first describe lambda calculus — a simple, yet Turing complete computation model based on functions. After that, type systems are introduced to provide static guarantees about code. We show simply typed lambda calculus, Hindley-Milner type system, then move to an algorithmic interpretation of type systems through constraints based systems and end with qualified types.

The second chapter is about linear types which provide more static guarantees for resources. We introduce general theory about similar substructural type systems, show how we can change the world by using linear types. In the end, we show a type system supporting linearity which is expressible in the previously established model of constrained qualified types.

Lily, a proof-of-concept linter implementation, which uses linear types to describe C++ programs is introduced in the third chapter. We show the design of the linter and note some implementation details and the issues encountered when creating the program. The chapter also describes how linear types are used to statically analyze C++.

The fourth and final chapter focuses on achieved results, comparing it to the state-of-the-art in C++ linters. It also contains a comparison of Lily to similar, related work and points to potential future work.

Appendix of the thesis provides brief instructions on compiling and using the Lily linter.

Chapter 1

Lambda calculus & type systems

1.1 Lambda calculus

We start by defining a small, yet universal, model of computation centered around function abstraction and application called *lambda calculus*. It was originally introduced by Alonzo Church in 1932 [Chu32]. The following description is a summary of [Roj15] and of [Bar+84].

1.1.1 Syntax

Given an expression E depending on x, the abstraction $\lambda x.\ E$ denotes a function $x \mapsto E$. For example, the identity function ($x \mapsto x$) is denoted as $\lambda x.\ x$ in lambda calculus. This corresponds to the concept of anonymous functions in programming languages since our abstractions are also not bound to an identifier.

The syntax of lambda calculus is described in BNF in figure 1.1. Every expression is either a variable, an application of two expressions or an abstraction. We also apply the following notational conventions:

- Parentheses are used to disambiguate expressions — (E) is the same expression as E.
- Function application associates from the left — the expression $E_1\ E_2\ E_3$ is the same expression as $(E_1\ E_2)\ E_3$.
- Our definition of lambda calculus as stated supports only unary abstractions. However, every multi-argument abstraction can be represented as a chain of unary abstractions via a process called *currying*.

  Given a function $f : (X \times Y) \to Z$, currying creates a function $h : X \to (Y \to Z)$, such that $f(x, y) = h(x)(y)$ for all choices of $x \in X$ and $y \in Y$.

Variable	V	$::=$	$x \mid y \mid z \mid \cdots$	
Expression	E	$::=$	V	(variable)
		$\mid$	$E\ E$	(application)
		$\mid$	$\lambda V.\ E$	(abstraction)

Figure 1.1 Syntax of (untyped) lambda calculus

This means that we are justified to use multi-argument abstractions as we can always 'translate' them into unary abstractions.

Therefore we use short-hand notation for multi-argument abstractions, meaning that $\lambda xy.\ E$ is the same expression as $\lambda x.\ \lambda y.\ E$.

- In order to disambiguate the possible meaning of $\lambda x.\ E_1\ E_2$, we say that it is the same as $\lambda x.\ (E_1\ E_2)$ and not $(\lambda x.\ E_1)\ E_2$, meaning that the abstraction body extends as far to the right as possible.

1.1.2 Semantics

Semantically, an abstraction $\lambda x.\ E$ *binds* the variable x in the abstraction body E. All variables that are not bound in an expression are considered *free*. Formally:

Definition 1 (Free variables)**.** *Given a lambda calculus expression E, we define the set of free variables $\mathrm{fv}(E)$ inductively as:*

$$\begin{aligned} \mathrm{fv}(x) &= \{x\} \\ \mathrm{fv}(\lambda x.\ E) &= \mathrm{fv}(E) \setminus \{x\} \\ \mathrm{fv}(E_1\ E_2) &= \mathrm{fv}(E_1) \cup \mathrm{fv}(E_2) \end{aligned}$$

For example:

$$\begin{aligned} \mathrm{fv}(\lambda x.\ x) &= \emptyset \\ \mathrm{fv}(\lambda x.\ x\ y) &= \{y\} \\ \mathrm{fv}(\lambda x.\ x\ (\lambda y.\ y)) &= \emptyset \end{aligned}$$

Next, we would like to define when two expressions are equivalent, i.e. the same programs up to isomorphism via renaming, formally:

Definition 2 (Equivalence)**.** *Lambda calculus expressions E_1 and E_2 are equivalent, notation $E_1 \equiv E_2$, if any of the following conditions hold:*

- *E_1 and E_2 are the same expression*

- *One can be obtained from the other by renaming its bound variables*

For example:

$$(\lambda x.\ E_1)\ E_2 \equiv (\lambda x.\ E_1)\ E_2$$
$$\lambda x.\ x \equiv \lambda u.\ u$$
$$(\lambda x.\ x)\ E \equiv (\lambda y.\ y)\ E$$
$$(\lambda x.\ x)\ E \not\equiv (\lambda x.\ y)\ E$$
$$\lambda x.\ x\ y \not\equiv \lambda x.\ x\ z$$

In order to actually work with expressions in lambda calculus, we need to introduce the concept of *capture-avoiding substitution.* This is a principled form of substitution which can rename bound variables on name collision, formally:

Definition 3 (Substitution)**.** *Given lambda calculus expressions M, N and variable x, we define substituting N for the free occurrences of x in M, notationally $[x := N]M$ as:*

$$[x := N]y \equiv \begin{cases} N, & \textit{if } x \equiv y \\ y, & \textit{otherwise} \end{cases}$$
$$[x := N](E_1\ E_2) \equiv ([x := N]E_1)\ ([x := N]E_2)$$
$$[x := N](\lambda y.\ M) \equiv \lambda y.\ ([x := N]M)\ \textit{if } x \not\equiv y \textit{ and } y \notin \mathrm{fv}(N)$$

The last case in the definition is seemingly not well defined for any possible substitution in an abstraction. For example $[x := y](\lambda y.\ y\ x)$ violates the condition that y shall not be free in y, the right-hand side of the substitution. However, we can first rename the bound variable y to a *fresh* variable u in the abstraction producing $[x := y](\lambda u.\ u\ x)$ which is equivalent according to definition 2 to the original expression. Now u is free in the right side and we can finally substitute to get $\lambda u.\ u\ y$.

Computation in lambda calculus uses a single rule known as β-reduction. Applying an abstraction $\lambda x.\ E_1$ to an expression E_2 results in the abstraction body E_1, where the variable x is substituted for E_1.

$$\lambda x.\ E_1\ E_2 \rightsquigarrow [x := E_2]E_1 \qquad (\beta)$$

1.1.3 Recursion

Since Turing completeness is equivalent to unbounded recursion, we may show that lambda calculus is Turing complete if we can show that it has unbounded recursion.

We can define recursive functions using a special function called the *Y combinator* which calls some function (expression) E and then creates itself again. The Y combinator is defined as:

$$\mathbf{Y} \equiv \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))$$

Notice that applying $\mathbf{Y}$ to some expression E using β yields:

$$\mathbf{Y}\ E = (\lambda x.\ E\ (x\ x))\ (\lambda x.\ E\ (x\ x)))$$

which can be further reduced by using β, yielding:

$$E\ ((\lambda x.\ E\ (x\ x))\ (\lambda x.\ E\ (x\ x)))) = E\ (\mathbf{Y}\ E)$$

Therefore it holds that $\forall E : E\ (\mathbf{Y}\ E) = \mathbf{Y}\ E$, which means that we can use $\mathbf{Y}$ to create boundless recursion.

1.2 Type systems

After the introduction of lambda calculus, Alonzo Church realized that his lambda calculus is logically paradoxical. To fix that oversight, he added simple *types* to his creation [Chu40], in order to restrict the use of inconsistent expressions.

A type is assigned to every lambda calculus expression. Expressions that are unable to be typed are declared invalid. We may observe similar behavior for types in modern strongly-typed programming languages. Such languages may for example ban the addition of a number of type `Int` and a string of type `String` which would otherwise likely lead to doing something the programmer might not have expected.

1.2.1 Formalism

In order to present actual type systems, we first need to introduce a specific formal notation [Car96] that is used to describe said type systems.

A *typing judgment* is an assertion that an expression E has type τ in the *context* Γ, denoted as $\Gamma \vdash E : \tau$. For example we can say that $\varnothing \vdash 1 : \texttt{Int}$, that is 1 has type `Int` or $\varnothing, x : \texttt{Int} \vdash x + 1 : \texttt{Int}$, $x + 1$ has type `Int` if x has type `Int`. We distinguish between *valid* and *invalid* typing judgments, e.g. $\varnothing \vdash 1 : \texttt{Int}$ and $\varnothing \vdash 1 : \texttt{String}$ respectively.

A context Γ is just a collection of bindings. Each binding $x : \tau$ is a pair of a variable x and its type τ. We write $\varnothing$ to denote an empty context and $\Gamma, x : \tau$ to denote a context Γ being extended with a binding where x has type τ. For

example $\Gamma = \varnothing, x : \texttt{Bool}, y : \texttt{Int}$ is a valid context. We denote the domain of a context Γ as $\mathrm{dom}(\Gamma)$, defined as set of all variables in bindings of context Γ, that is $\mathrm{dom}(\Gamma) = \{x \mid (x : \tau) \in \Gamma\}$.

Inference rules assert the validity of some assertions based on other assertions that are known to be valid. Each inference rule has a name to the right of the line, premises above the line and a single conclusion below the line. When all the premises are valid, the conclusion is valid as well.

$$\frac{\text{Premise}_1 \quad \cdots \quad \text{Premise}_n}{\text{Conclusion}} \quad [\textsc{rule name}]$$

For example, given A, B as first-order logic variables, we can write the modus ponens rule (also known as the implication elimination rule), summarized as 'assuming A is true and A implies B, deduce B is true' as:

$$\frac{A \quad A \to B}{B} \quad [\text{MP}]$$

When the inference rule is used to assign a type to an expression, it is called a *type rule*, a collection of such type rules is called a *type system*.

A *derivation tree* for a judgment J is a tree of judgments with J at the root at the bottom and some leaves at the top. Each judgment follows from those immediately above it by using some inference rule of the system. A *valid judgment* is one that can be obtained as the root of a derivation tree in a given type system.

The problem of finding a type τ for a given expression E in a given context Γ and a type system so that there exists a derivation tree with $\Gamma \vdash E : \Gamma$ is called *type inference*. The problem of just verifying if there exists a valid judgment for a given type τ, expression E, context Γ and a type system is called *type checking*.

1.3 Simply typed lambda calculus

Knowing some basic type theory, we can finally extend untyped lambda calculus with the most elementary form of types creating *simply typed lambda calculus*, also known as STLC. The syntax of STLC is described in figure 1.2.

We add a set C of type constants of the language — for a real programming language, this would mean its most basic types like `Int` or `Bool`. We also add syntax for types: a type is either a type constant or a function from one type to another. Furthermore, we explicitly annotate abstraction variables with their types.

As a convention, the function type arrow $(\to)$ is right-associative, meaning that $\tau_1 \to \tau_2 \to \tau_3$ stands for $\tau_1 \to (\tau_2 \to \tau_3)$. Especially note that the difference between the following types: $\tau_1 \to \tau_2 \to \tau_3$ and $(\tau_1 \to \tau_2) \to \tau_3$. Whereas the

$$
\begin{array}{lrcll}
\text{Variable} & V & ::= & x \mid y \mid z \mid \cdots & \\
\text{Expression} & E & ::= & V & \text{(variable)} \\
 & & \mid & E\ E & \text{(application)} \\
 & & \mid & \lambda V : \tau.\ E & \text{(abstraction)} \\
 & & & & \\
\text{Type constant} & C & ::= & a \mid b \mid c \mid \cdots & \\
\text{Type} & \tau & ::= & C & \text{(type constant)} \\
 & & \mid & \tau \to \tau & \text{(function type)}
\end{array}
$$

Figure 1.2 Syntax of simply typed lambda calculus

$$
\frac{x : \tau \ \in \ \Gamma}{\Gamma \ \vdash x : \tau} \quad [\textsc{Var}]
$$

$$
\frac{\Gamma, x : \tau_1 \ \vdash E : \tau_2}{\Gamma \ \vdash (\lambda x : \tau_1.\ E) : \tau_1 \to \tau_2} \quad [\textsc{Abs}]
$$

$$
\frac{\Gamma \ \vdash E_1 : \tau_1 \to \tau_2 \quad \Gamma \ \vdash E_2 : \tau_1}{\Gamma \ \vdash (E_1\ E_2) : \tau_2} \quad [\textsc{App}]
$$

Rule set 1 Typing rules for simply typed lambda calculus

former type can be viewed via currying as a function of two arguments of types τ_1 and τ_2 returning a value of type τ_3, the latter type is of a function taking itself a function of type $\tau_1 \to \tau_2$ as an argument.

Let us go over the typing rules of STLC described in rule set 1:

- The first rule [Var] states that in order to infer a type τ for a variable x given context Γ, we need to have the binding $x : \tau$ in Γ. Essentially, we can use any bindings from the context at any time.

- The rule [App] tells us that to infer a type τ_2 for an application $E_1\ E_2$, we first need to infer that the type of E_1 is a function from τ_1 to τ_2 and the type of E_2 is τ_1. Notice that this rule is strikingly similar to modus ponens, which is no coincidence as STLC corresponds to a variant of minimal propositional logic [SU06].

- The last rule [Abs] says that to infer a type for an abstraction like $\lambda x : \tau_1.\ E$, we can assume that the type of x is τ_1. In order to infer the type of E, we'll need to add our assumption that $x : \tau_1$ into the context Γ in which we infer the type of E. This is necessary since the abstraction body

$$\boxed{\frac{}{\Gamma \vdash \mathsf{fix}_\tau : (\tau \to \tau) \to \tau}\ [\textsc{Fix}]}$$

Rule set 2 Typing rules for fix_τ in the simply typed lambda calculus

E might use x. The whole abstraction will then have a type $\tau_1 \to \tau_2$ given the context Γ.

It is straightforward to devise an algorithm using the typing rules in rule set 1, because the rules are *syntax-directed*. We can simply read the rules bottom to top and act accordingly.

Given type constants $C = \{a, b\}$, a derivation tree for $\lambda x : a.\ x$ (the identity function) is:

$$\frac{\dfrac{x : a \in \varnothing, x : a}{\varnothing, x : a \vdash x : a}\ [\textsc{Var}]}{\varnothing \vdash (\lambda x : a.\ x) : a \to a}\ [\textsc{Abs}]$$

Given type constants $C = \{a, b\}$, a derivation tree for $\lambda x : a.\ (\lambda y : b.\ x)$ (known as the K combinator) is:

$$\frac{\dfrac{\dfrac{x : a \in \varnothing, x : a, y : b}{\varnothing, x : a, y : b \vdash x : a}\ [\textsc{Var}]}{\varnothing, x : a \vdash (\lambda y : b.\ x) : b \to a}\ [\textsc{Abs}]}{\varnothing \vdash (\lambda x : a.\ (\lambda y : b.\ x)) : a \to b \to a}\ [\textsc{Abs}]$$

Notice that we cannot produce a derivation tree for any type for the expression $\lambda x : a.\ x\ x$. Such an expression does not have a valid type in STLC even though it is a perfectly valid expression in the untyped lambda calculus from section 1.1.

Another expression which we cannot type is the Y combinator from section 1.1.3 or any of its functional equivalents, that would allow boundless recursion. Specifically, this means that STLC does not support boundless recursion and is not Turing complete. It can even be shown that all STLC programs halt!

However, we might be inclined to re-add recursion to the system, to keep Turing completeness. In order to do that, we introduce a new operator in the syntax called fix_τ. Notice that the operator requires an explicit type annotation.

The semantics of fix_τ are the same as the semantics of the Y combinator — for all expressions E, $\mathsf{fix}_\tau\ E = E\ (\mathsf{fix}_\tau\ E)$. We also add a new simple typing rule into our type system rules, thus producing rule set 2, which simply states that the type of fix_τ is $(\tau \to \tau) \to \tau$.

1.4 Hindley-Milner

Simply typed lambda calculus helped us to express some basic types but not in a reusable way. For example there have many different types for the identity function in STLC!

Consider functions $\lambda x : a.\ x$ and $\lambda x : b.\ x$. These functions have very different types: $a \to a$ and $b \to b$ in STLC but we can certainly see a common pattern — the types always take the form $\tau \to \tau$ for some type τ.

Our goal is to modify our syntax and typing rules in order to express a more general type of the form 'for every type τ, this function has type $\tau \to \tau$'. In addition to that, it is desired that the programmer does not have to specify any explicit type annotations and that the system is algorithmically decidable.

A type system that nicely balances our needs is called Hindley-Milner (shortened as HM) as it was independently discovered by Hindley [Hin69] and by Milner [Mil78] and formalized and proven correct by Damas and Milner [DM82]. This system generalizes types only at specific points determined by the programmer, by convention on top-level declarations. Unlike more powerful systems like System F by Girard [Gir71], which are undecidable [Wel99], Hindley-Milner is algorithmically decidable (specifically EXPTIME-complete [Mai89]).

A new expression called a *let expression* is added to our syntax as depicted in figure 1.3. Its purpose is to be merely a point where a type is generalized; it has no semantical meaning on its own. In fact, let $x = M$ in N is semantically equivalent to $(\lambda x.\ N)\ M$. As per our requirements, an explicit annotation on an abstraction's argument is no longer necessary.

In order to express more general types, we add type variables into the syntax of types. We also distinguish two kinds of types:

- *Monotypes* (monomorphic types) are simple types without any quantifiers, such as `Bool`, `Int` $\to$ `Bool` or α
- *Polytypes* (polymorphic types, also sometimes called type schemes) are types with universal quantifiers at the beginning, that is in prenex form, e.g. $\forall\alpha.\ \alpha \to \alpha$ or $\forall\alpha.\ \forall\beta.\ (\alpha \to \alpha) \to \beta \to$ `Bool`

An important detail is that type variables are monotypes and stand only for monotypes, i.e. some type variable α can <u>not</u> itself stand for a polytype like $\forall\beta.\ \beta \to \beta$.

Note that all quantifiers used are universal ($\forall$) and they appear only at the very beginning of the type. We therefore write $\forall\alpha\ \beta.\ \tau$ instead of $\forall\alpha.\ \forall\beta.\ \tau$ as a syntactic convention. Especially notice that this system does not support more general types such as $(\forall\alpha.\ \alpha \to \alpha) \to$ `Int` — a type which is very different from $\forall\alpha.\ (\alpha \to \alpha) \to$ `Int`. Whereas the former requires a function of type $\forall\alpha.\ \alpha \to \alpha$

Variable	V	$::=$	$x \mid y \mid z \mid \cdots$	
Expression	E	$::=$	V	(variable)
		$\mid$	$E\ E$	(application)
		$\mid$	$\lambda V.\ E$	(abstraction)
		$\mid$	$\mathsf{let}\ V = E\ \mathsf{in}\ E$	(let)
Type variable	X	$::=$	$\alpha \mid \beta \mid \gamma \mid \cdots$	
Type constant	C	$::=$	$a \mid b \mid c \mid \cdots$	
Monotype	τ	$::=$	X	(type variable)
		$\mid$	C	(type constant)
		$\mid$	$\tau \to \tau$	(function type)
Polytype	σ	$::=$	τ	(monotype)
		$\mid$	$\forall \alpha.\ \sigma$	(quantifier)

Figure 1.3 Syntax of HM

as an argument, the other requires a function of type $\alpha \to \alpha$ for some specified α, not for every α. Hindley-Milner system can be extended to support inference of such types at the potential cost of requiring some explicit type annotations, see [Jon+07].

In order to present the type rules for Hindley-Milner, we first need to define a few useful concepts. First off, we introduce free *type* variables in definition 4, similarly to free variables in untyped lambda calculus in definition 1.

Definition 4 (Free type variables)**.** *For a type σ, the set of free type variables, notation $\mathrm{ftv}(\sigma)$, is defined inductively as:*

$$\begin{aligned} \mathrm{ftv}(\alpha) &= \{\alpha\} \\ \mathrm{ftv}(\forall \alpha.\ \sigma) &= \mathrm{ftv}(\sigma) \setminus \{\alpha\} \\ \mathrm{ftv}(\tau_1 \to \tau_2) &= \mathrm{ftv}(\tau_1) \cup \mathrm{ftv}(\tau_2) \\ \mathrm{ftv}(a) &= \emptyset \end{aligned}$$

Additionally, we also need to define a concept of a substitution for type variables. We write $S = [\alpha_1 := \tau_1, \ldots, \alpha_n := \tau_n]$ to express a substitution S which replaces every mention of a type variable α_i by a type τ_i. An empty substitution is denoted as $[]$. By convention, $S\tau$ denotes applying a substitution S in a type τ. The composition of two substitutions, S_1 and S_2 is denoted as $(S_2 \circ S_1)$.

We assume that all substitutions have the following property: $S(S\tau) = S\tau$, i.e. applying a substitution multiple times is equivalent to applying it once. For example, the property does not hold for the following substitution: $[\alpha_1 := \texttt{Bool}, \alpha_2 := \alpha_1]$, therefore we consider it invalid. However, we can often rewrite

a substitution to have the aforementioned property, for example the substitution above can be rewritten as $[\alpha_1 := \texttt{Bool}, \alpha_2 := \texttt{Bool}]$. The property can be rewritten in this way if it passes an occurs check.

We use $\mathrm{mgu}(\tau_1, \tau_2)$ as a function which returns the *most general unifier* of τ_1 and τ_2 — a substitution S such that $S\tau_1 = S\tau_2$. This function can be implemented using Robinson's unification algorithm [Rob65] or similar algorithms.

A consequence of the generality provided by the Hindley-Milner system, a single expression may have multiple valid types. For example with type constants $C = \{\texttt{Bool}, \texttt{Int}\}$, identity function $\lambda x.\ x$ may have type $\texttt{Bool} \to \texttt{Bool}$ or $\texttt{Int} \to \texttt{Int}$, or even $\forall\alpha.\ (\alpha \to \alpha) \to (\alpha \to \alpha)$.

Fortunately, there is a way out of this in HM as there is exactly one most general type to give to identity — $\forall\alpha.\ \alpha \to \alpha$. This most general type is known as the *principal type* of an expression. Notice that every other valid type for the identity function can be obtained by substituting for α.

We introduce a partial order $\sqsubseteq$ on types, where $\sigma_1 \sqsubseteq \sigma_2$ means that σ_1 is more general than σ_2, or equivalently σ_2 is more specific than (instance of) σ_1. Formally:

Definition 5 (Type ordering)**.** *For two types $\sigma_1 = \forall\alpha_1 \ldots \alpha_n.\ \tau_1$ and $\sigma_2 = \forall\beta_1 \ldots \beta_m.\ \tau_2$, we say that σ_1 is more general than σ_2, notation $\sigma_1 \sqsubseteq \sigma_2$, if the following rule holds:*

$$\frac{\tau_2 = [\alpha_1 := \tau'_1, \ldots, \alpha_n := \tau'_n]\tau_1 \qquad \beta_i \notin \mathrm{ftv}(\forall\alpha_1 \ldots \alpha_n.\ \tau_1)}{\forall\alpha_1 \ldots \alpha_n.\ \tau_1 \sqsubseteq \forall\beta_1 \ldots \beta_m.\ \tau_2} \ [\textsc{Order}]$$

In the HM type system, if an expression E has a valid type, then there exists a principal type for E (see [DM82] for a proof). This property is called *principal type property*. This property is quite desirable for a type system since all possible types are comparable — for every two possible types σ_1, σ_2 for an expression E either $\sigma_1 \sqsubseteq \sigma_2$ or $\sigma_2 \sqsubseteq \sigma_1$. It is also desired that an algorithm for type inference in a type system with principal type property is able to returns the principal type for an expression.

We finally present a syntax-directed formulation of type rules for HM in rule set 3. Notice that the context Γ is modified so that it now contains bindings $(x : \sigma)$ — pairing a variable to its polytype.

- Rule [Var]: Since the context Γ now holds polytypes, we need to instantiate the polytype σ into a monotype τ such that $\sigma \sqsubseteq \tau$. Otherwise the rule stays the same.
- Rules [Abs] and [App] are the same as in type rules of STLC (rule set 1) — they both work only with monotypes.

$$\frac{x : \sigma \ \in \ \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \ \vdash x : \tau} \quad [\textsc{Var}]$$

$$\frac{\Gamma, x : \tau_1 \ \vdash E : \tau_2}{\Gamma \ \vdash (\lambda x.\ E) : \tau_1 \to \tau_2} \quad [\textsc{Abs}]$$

$$\frac{\Gamma \ \vdash E_1 : \tau_1 \to \tau_2 \quad \Gamma \ \vdash E_2 : \tau_1}{\Gamma \ \vdash (E_1\ E_2) : \tau_2} \quad [\textsc{App}]$$

$$\frac{\Gamma \ \vdash E_1 : \tau_1 \quad \Gamma, x : \sigma \ \vdash E_2 : \tau_2 \quad \sigma = \mathrm{Gen}_\Gamma(\tau_1)}{\Gamma \ \vdash \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \tau_2} \quad [\textsc{Let}]$$

Rule set 3 Typing rules for HM

- We have a new rule — [Let], which is like a combination of [Abs] and [App] as by its semantic definition ($(\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2) \equiv ((\lambda x.\ E_2)\ E_1)$). However, unlike in the aforementioned rules, after inferring a type τ_1, we generalize it into σ and add $(x : \sigma)$ to the context Γ. Generalization $\mathrm{Gen}_\Gamma(\tau)$ is just a closure of τ with respect to Γ, that is $\forall\alpha_1 \ldots \alpha_n.\ \tau$ where the type variables $\{\alpha_1, \ldots, \alpha_n\}$ are exactly $\mathrm{ftv}(\tau) \setminus \mathrm{ftv}(\Gamma)$.

There exists an efficient algorithm called $\mathcal{W}$ [DM82] for inferring types in the Hindley-Milner type system that always infers a principal type. However, we choose to present a different, more general algorithm that is more easily extensible in section 1.5.

Example of a derivation tree for the expression $\mathsf{let}\ x = (\lambda y.\ y)\ \mathsf{in}\ x\ x$ according to rule set 3 is shown in figure 1.4.

1.5 Type constraints

In this section, we simplify the type inference process by splitting it into two phases: First, we generate all *constraints* that describe relations between types from the expression. Then we take all of the acquired constraints, carefully *resolve* them and produce the desired output — the inferred type.

This approach has many advantages. Not only is it simpler than doing everything at once, it is also more extensible and can provide a more general view into the type inference process. We can employ that to produce clear error messages for programmers, since by having a more general view of the process, we remove the left-to-right bias inherent in traditional algorithms like $\mathcal{W}$.

$$
\dfrac{
\dfrac{
\dfrac{x : \tau_1 \in \Gamma}{\varnothing, y : \tau_1 \vdash y : \tau_1}\ [\textsc{Var}]
}{\varnothing \vdash \lambda y.\, y : \tau_1 \to \tau_1}\ [\textsc{Abs}]
\qquad
\dfrac{
\dfrac{
\dfrac{x : (\tau_2 \to \tau_2) \to (\tau_2 \to \tau_2) \in \Gamma}{\varnothing, x : \forall \alpha.\, \alpha \to \alpha \vdash x : (\tau_2 \to \tau_2) \to (\tau_2 \to \tau_2)}\ [\textsc{Var}]
}{\varnothing, x : \forall \alpha.\, \alpha \to \alpha \vdash x\, x : \tau_2 \to \tau_2}\ [\textsc{App}]
\qquad
\dfrac{x : \tau_2 \to \tau_2 \in \Gamma}{\varnothing, x : \forall \alpha.\, \alpha \to \alpha \vdash x : \tau_2 \to \tau_2}\ [\textsc{Var}]
}{\varnothing \vdash \mathsf{let}\ x = (\lambda y.\, y)\ \mathsf{in}\ x\, x : \forall \beta.\, \beta \to \beta}\ [\textsc{Let}]
$$

$$
\dfrac{
\dfrac{
\beta_2\ \mathsf{fresh}\quad \dfrac{\beta_1\ \mathsf{fresh}\quad \dfrac{}{\{y : \beta_1\} \vdash y : \beta_1 \mid \varnothing}\ [\textsc{Var}]}{\varnothing \vdash \lambda y.\, y : \beta_2 \to \beta_1 \mid \{\beta_1 \sim \beta_2\}}\ [\textsc{Abs}]
\qquad
\beta_5\ \mathsf{fresh}\quad \dfrac{\beta_3\ \mathsf{fresh}\quad \dfrac{}{\{x : \beta_3\} \vdash x : \beta_3 \mid \varnothing}\ [\textsc{Var}]}{\{x : \beta_3,\, x : \beta_4\} \vdash x\, x : \beta_5 \mid \{\beta_3 \sim (\beta_4 \to \beta_5)\}}\ [\textsc{App}]
}{\varnothing \vdash \mathsf{let}\ x = (\lambda y.\, y)\ \mathsf{in}\ x\, x : \beta_5 \mid \{\beta_1 \sim \beta_2,\ \beta_3 \sim (\beta_4 \to \beta_5),\ \beta_3 \preccurlyeq_\varnothing (\beta_2 \to \beta_1),\ \beta_4 \preccurlyeq_\varnothing (\beta_2 \to \beta_1)\}}\ [\textsc{Let}]
\qquad
\dfrac{\beta_4\ \mathsf{fresh}\quad \dfrac{}{\{x : \beta_4\} \vdash x : \beta_4 \mid \varnothing}\ [\textsc{Var}]}{\{x : \beta_4\} \vdash x : \beta_4 \mid \varnothing}\ [\textsc{Var}]
$$

Figure 1.4 Example derivation trees for let $x = (\lambda y.\, y)$ in $x\, x$ according to rule set 3 (left) and rule set 4 (right).

$$
\begin{array}{lrll}
\text{Constraint} & C & ::= & \tau_1 \sim \tau_2 & \text{(equality)} \\
& & \mid & \tau \preccurlyeq \sigma & \text{(explicit instance)} \\
& & \mid & \tau_1 \preccurlyeq_M \tau_2 & \text{(implicit instance)}
\end{array}
$$

Figure 1.5 Syntax of constraints

There are many different approaches to type constraint systems, such as the well-known HM(X) system by Odersky, Sulzmann, and Wehr [OSW99]. We follow the more recent approach by Heeren, Hage, and Swierstra [HHS02] since it can represent even more relations between types as constraints.

Unlike the previous systems, the context Γ is absent. Instead an assumption set $\mathcal{A}$ is used which tracks type variables assigned to the occurrences of free variables. Note that contrary to HM, there can be multiple different assumptions for a given variable.

A *constraint set*, denoted as $\mathcal{C}$ is a multiset of constraints. Note that unlike HM(X) [OSW99], our system does not build a tree out of constraints, rather keeping them in a multiset. There are three kinds of constraints (figure 1.5):

- An equality constraint $(\tau_1 \sim \tau_2)$ is used to express that τ_1 and τ_2 should be equal.
- An explicit instance constraint $(\tau \preccurlyeq \sigma)$ expresses that τ is an instance of σ, i.e. $\sigma \sqsubseteq \tau$ according to definition 5. This constraint is useful when the polytype of an expression is known in advance.
- An implicit instance constraint $(\tau_1 \preccurlyeq_M \tau_2)$ is used when the polytype in advance is not known, however it is known that τ_1 is an instance of τ_2 generalized with respect to the set of type variables M, i.e. $\mathrm{Gen}_M(\tau_2) \sqsubseteq \tau_1$.

With that out of the way, we can finally present the constraint generation rules in rule set 4 for type inference:

- Rule [Var]: A new fresh type variable β is created and inserted into the assumption set $\mathcal{A}$. We implicitly assume that we have access to a generator of new fresh variables.
- Rule [Abs]: A new fresh type variable β is created to represent the type of the argument x. New equality constraints are generated for each type variable in the assumption set $\mathcal{A}$ assigned to the argument x, thus ensuring that their types are equal.

  On encountering an abstraction, we locally add β into the set of monomorphic type variables M when inferring the type for its whole subexpression. Note that this instruction is not explicitly expressed in the rule set.

$$\frac{\beta \text{ fresh}}{\{x : \beta\} \vdash x : \beta \mid \varnothing} \quad [\textsc{Var}]$$

$$\frac{\mathcal{A} \vdash E : \tau \mid \mathcal{C} \quad \mathcal{C}' = \mathcal{C} \cup \{\tau' \sim \beta \mid (x : \tau') \in \mathcal{A}\} \quad \beta \text{ fresh}}{\mathcal{A} \backslash x \vdash (\lambda x.\ E) : \beta \to \tau \mid \mathcal{C}'} \quad [\textsc{Abs}]$$

$$\frac{\begin{array}{c}\mathcal{A}_1 \vdash E_1 : \tau_1 \mid \mathcal{C}_1 \quad \mathcal{A}_2 \vdash E_2 : \tau_2 \mid \mathcal{C}_2 \\ \mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2 \quad \beta \text{ fresh} \\ \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \sim \tau_2 \to \beta\}\end{array}}{\mathcal{A} \vdash (E_1\ E_2) : \beta \mid \mathcal{C}} \quad [\textsc{App}]$$

$$\frac{\begin{array}{c}\mathcal{A}_1 \vdash E_1 : \tau_1 \mid \mathcal{C}_1 \quad \mathcal{A}_2 \vdash E_2 : \tau_2 \mid \mathcal{C}_2 \\ \mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2 \backslash x \\ \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau' \preccurlyeq_M \tau_1 \mid (x : \tau') \in \mathcal{A}_2\}\end{array}}{\mathcal{A} \vdash \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \tau_2 \mid \mathcal{C}} \quad [\textsc{Let}]$$

Rule set 4 Typing rules for HM with constraints

- Rule [App]: Again, a new fresh type variable β is created, this time to represent the result of the type application. A new constraint is generated expressing that the types are compatible — that the type of the function τ_1 must be equal to a function type from the type of the argument τ_2 to the type of the result β.
- Rule [Let]: The new constraint here ensures that each use of x in the body of the let expression (E_2) has a type which is an implicit instance of the type τ_1 inferred from E_1. If any uses of x were seen, then they have an entry in $\mathcal{A}$ by [Var]. As stated above, this needs to be an implicit instance constraint since we have not generalized the type τ_1 yet.

An example derivation using rule set 4 is depicted in figure 1.4. It produces the following set of constraints:

$$\{\beta_1 \sim \beta_2,\ \beta_3 \sim (\beta_4 \to \beta_5),\ \beta_3 \preccurlyeq_\varnothing (\beta_2 \to \beta_1),\ \beta_4 \preccurlyeq_\varnothing (\beta_2 \to \beta_1)\}$$

Notice that all of the implicit constraints in this set are not limited by any monomorphic variables. If the whole example expression E was wrapped in some abstraction, e.g. $\lambda z.\ E$, the resulting implicit constraints would be limited by the set $\{z\}$.

After constraint generation, our goal is to resolve the generated constraints, specifically to produce a substitution S that *satisfies* every constraint in the resulting set $\mathcal{C}$:

Definition 6 (Constraint satisfaction)**.** *For a given substitution S and a constraint C, satisfaction of C by S is defined as follows:*

$$\begin{array}{lll} S \textit{ satisfies } (\tau_1 \sim \tau_2) & \textit{iff} & S\tau_1 = S\tau_2 \\ S \textit{ satisfies } (\tau_1 \preccurlyeq_M \tau_2) & \textit{iff} & \mathrm{Gen}_{SM}(S\tau_2) \sqsubseteq S\tau_1 \\ S \textit{ satisfies } (\tau \preccurlyeq \sigma) & \textit{iff} & S\sigma \sqsubseteq S\tau \end{array}$$

Note that in general $\mathrm{Gen}_{SM}(S\tau) \neq S(\mathrm{Gen}_M(\tau))$, which means that not all implicit instance constraints are outright *solvable*. We want to solve an implicit constraint $(\tau_1 \preccurlyeq_M \tau_2)$ by converting it to a constraint $(\tau \preccurlyeq \mathrm{Gen}_M(\tau_2))$. However we need to ensure that no other constraints are affected, i.e. we want applying a substitution after generalizing have the same result as first generalizing and then applying a substitution.

Fortunately, Heeren, Hage, and Swierstra [HHS02] found that the equality holds in some cases:

$$(\mathrm{ftv}(\tau) \setminus M) \cap \mathrm{dom}(S) = \emptyset \implies \mathrm{Gen}_{SM}(S\tau) = S(\mathrm{Gen}_M(\tau))$$

Next, we present a definition of active type variables in a constraint set:

Definition 7 (Active type variables)**.** *For a constraint C, the set of active type variables, notation $\mathrm{atv}(C)$, is defined as:*

$$\begin{aligned} \mathrm{atv}(\tau_1 \sim \tau_2) &= \mathrm{ftv}(\tau_1) \cup \mathrm{ftv}(\tau_2) \\ \mathrm{atv}(\tau_1 \preccurlyeq_M \tau_2) &= \mathrm{ftv}(\tau_1) \cup (\mathrm{ftv}(M) \cap \mathrm{ftv}(\tau_2)) \\ \mathrm{atv}(\tau \preccurlyeq \sigma) &= \mathrm{ftv}(\tau) \cup \mathrm{ftv}(\sigma) \end{aligned}$$

For a constraint set $\mathcal{C}$, the set of active type variables $\mathrm{atv}(\mathcal{C})$ is defined as:

$$\mathrm{atv}(\mathcal{C}) = \bigcup_{C \in \mathcal{C}} \mathrm{atv}(C)$$

In order to present an algorithm, we need to define $\mathrm{Gen}_M(\tau)$ and $\mathrm{Inst}(\sigma)$ as implementable functions. Generalization can be simply implemented by its definition: $\mathrm{Gen}_M(\tau) = \forall \bar{\alpha}.\ \tau$ where $\bar{\alpha} = \mathrm{ftv}(\tau) \setminus \mathrm{ftv}(M)$. Instantiation of $\sigma = \forall \bar{\alpha}.\ \tau$ is implemented as generating fresh $\bar{\beta}$ and returning an instantiated monotype $[\alpha_1 := \beta_1, \ldots, \alpha_n := \beta_n]\tau$.

Finally, we can describe an algorithm Solve 1 used to produce a satisfiable substitution S from a set of constraints $\mathcal{C}$. Equality constraints are solved using unification, implicit instance constraints are converted to explicit instance constraints using generalization and explicit instance constraints are converted to equality constraints using instantiation.

The constraints generated by rules from rule set 4 are always solvable in some order and they produce a valid typing if there is one. Notice that the algorithm

$$
\begin{aligned}
&\textsc{Solve}(\varnothing) && = [] \quad \text{(empty substitution)} \\
&\textsc{Solve}(\{\tau_1 \sim \tau_2\} \cup \mathcal{C}) && = \textsc{Solve}(S\mathcal{C}) \circ S \\
& && \qquad \textbf{where } S = \mathrm{mgu}(\tau_1, \tau_2) \\
&\textsc{Solve}(\{\tau_1 \preccurlyeq_M \tau_2\} \cup \mathcal{C}) && = \textsc{Solve}(\{\tau_1 \preccurlyeq \mathrm{Gen}_M(\tau_2)\} \cup \mathcal{C}) \\
& && \qquad \textbf{if } (\mathrm{ftv}(\tau_2) \setminus M) \cap \mathrm{atv}(\mathcal{C}) = \emptyset \\
&\textsc{Solve}(\{\tau \preccurlyeq \sigma\} \cup \mathcal{C}) && = \textsc{Solve}(\{\tau \sim \mathrm{Inst}(\sigma)\} \cup \mathcal{C})
\end{aligned}
$$

Algorithm 1 Algorithm for solving constraints

is non-deterministic — there is no specific order in which we need to solve the constraints, bar the requirement for solvability of an implicit instance constraint.

We show an example of algorithm 1 on the set of constraints obtained from the previous example in figure 1.4:

$$
\begin{aligned}
& \textsc{Solve}(\{\beta_1 \sim \beta_2,\ \beta_3 \sim (\beta_4 \to \beta_5),\ \beta_3 \preccurlyeq_\varnothing (\beta_2 \to \beta_1),\ \beta_4 \preccurlyeq_\varnothing (\beta_2 \to \beta_1)\}) \\
= \ & \textsc{Solve}(\{\beta_3 \sim (\beta_4 \to \beta_5),\ \beta_3 \preccurlyeq_\varnothing (\beta_1 \to \beta_1),\ \beta_4 \preccurlyeq_\varnothing (\beta_1 \to \beta_1)\}) \circ [\beta_1 := \beta_2] \\
= \ & \textsc{Solve}(\{(\beta_4 \to \beta_5) \preccurlyeq_\varnothing (\beta_1 \to \beta_1),\ \beta_4 \preccurlyeq_\varnothing (\beta_1 \to \beta_1)\}) \circ [\beta_3 := \beta_4 \to \beta_5] \circ [\beta_1 := \beta_2] \\
= \ & \textsc{Solve}(\{(\beta_4 \to \beta_5) \preccurlyeq_\varnothing (\beta_1 \to \beta_1),\ \beta_4 \preccurlyeq_\varnothing (\beta_1 \to \beta_1)\}) \circ [\beta_3 := \beta_4 \to \beta_5] \circ [\beta_1 := \beta_2] \\
= \ & \textsc{Solve}(\{(\beta_4 \to \beta_5) \preccurlyeq \forall\alpha.\ (\alpha \to \alpha),\ \beta_4 \preccurlyeq_\varnothing (\beta_1 \to \beta_1)\}) \circ [\beta_3 := \beta_4 \to \beta_5] \circ [\beta_1 := \beta_2] \\
= \ & \textsc{Solve}(\{(\beta_4 \to \beta_5) \preccurlyeq \forall\alpha.\ (\alpha \to \alpha),\ \beta_4 \preccurlyeq \forall\alpha.\ (\alpha \to \alpha)\}) \circ [\beta_3 := \beta_4 \to \beta_5] \circ [\beta_1 := \beta_2] \\
= \ & \textsc{Solve}(\{(\beta_4 \to \beta_5) \preccurlyeq \forall\alpha.\ (\alpha \to \alpha),\ \beta_4 \preccurlyeq (\delta_1 \to \delta_1)\}) \circ [\beta_3 := \beta_4 \to \beta_5] \circ [\beta_1 := \beta_2] \\
= \ & \textsc{Solve}(\{((\delta_1 \to \delta_1) \to \beta_5) \preccurlyeq \forall\alpha.\ (\alpha \to \alpha)\}) \circ [\beta_4 := \delta_1 \to \delta_1] \circ [\beta_3 := \beta_4 \to \beta_5] \circ [\beta_1 := \beta_2] \\
= \ & \textsc{Solve}(\{((\delta_1 \to \delta_1) \to \beta_5) \sim (\delta_2 \to \delta_2)\}) \circ [\beta_4 := \delta_1 \to \delta_1] \circ [\beta_3 := \beta_4 \to \beta_5] \circ [\beta_1 := \beta_2] \\
= \ & \textsc{Solve}(\varnothing) \circ [\delta_2 := \delta_1 \to \delta_1, \beta_5 := \delta_1 \to \delta_1] \circ [\beta_4 := \delta_1 \to \delta_1] \circ [\beta_3 := \beta_4 \to \beta_5] \circ [\beta_1 := \beta_2]
\end{aligned}
$$

Since from figure 1.4 we know that the type of let $x = (\lambda y.\ y)$ in $x\ x$ is β_5, we get a type $\delta_1 \to \delta_1$ after applying the resulting substitution show above and a principal type $\forall\alpha.\ \alpha \to \alpha$ after generalizing.

A full type inferring algorithm InferType 2 may be created from rules of rule set 4 and the solve algorithm Solve 1. First the constraints are generated, then we ensure that there are no variables in $\mathcal{A}$ that are not present in the context Γ as such variables would be unbound. After that, we create an additional set of constraints for variables left in $\mathcal{A}$ — variables that were not defined during the process and ensure that their types match with the types in the context Γ. Then we solve all of the constraints and return the resulting type of the expression. Note that the algorithm can be simply modified to also return the resulting substitution S.

Notice that the algorithm does not necessarily require the step where we add constraints between the assumptions and the context. Without doing that,

$\textsc{InferType}(\Gamma, E) = \textbf{do}$
  $\mathcal{A} \vdash E : \tau \mid \mathcal{C}$
  if $\mathrm{dom}(\mathcal{A}) \not\subseteq \mathrm{dom}(\Gamma)$ **then error**
  else do
    $\mathcal{C}' = \{\tau \preccurlyeq \sigma \mid (x : \tau) \in \mathcal{A}, (x : \sigma) \in \Gamma\}$
    $S = \textsc{Solve}(\mathcal{C} \cup \mathcal{C}')$
    return $S\tau$

Algorithm 2 Algorithm for inferring a type of an expression

the inferred type for a variable x would be $\{x : \alpha\} \vdash \alpha$ — the assumptions would be attached to the inferred type. This method closely resembles the so called principal typings of compositional systems, such as the system by Chitil [Chi01].

Since the algorithm Solve for solving constraints is non-deterministic, we may choose any order in which to solve the constraints. A common approach for emitting clearer error messages is to create a type graph with types as vertices and constraints as various edges. In addition to that, the vertices are split into equivalence classes. If an error occurs, then there exists some path in the graph between the equivalence classes as the witness of the error. After that, we may employ various heuristics to determine which error to report. For more details on this approach, see [HHS02].

1.6 Qualified types

What if we wanted to type an addition function $(+)$ that works both with adding two floats of type `Float` to produce another and with adding two integers of type `Int` to produce another? This would mean that the type of $(+)$ would have to be both $\texttt{Float} \to \texttt{Float} \to \texttt{Float}$ and $\texttt{Int} \to \texttt{Int} \to \texttt{Int}$. What would be the principal type of $(+)$? Surely it cannot be $\forall\alpha.\ \alpha \to \alpha \to \alpha$ since it might not desirable to implement addition for all types!

Similar issue is with adding an equality function $(=)$ that works only for some selected types — types for which equality makes sense semantically. Again, the principal type of $(=)$ would have to be $\forall\alpha.\ \alpha \to \alpha \to \texttt{Bool}$, which contradicts our stance that not all types should be equatable.

A possible solution might to introduce an additional ordering relation $\sqsubseteq^*$ so that every `Int` is an instance of `Float`, thus getting $\texttt{Float} \to \texttt{Float} \to \texttt{Float}$ as the principal type of $(+)$. This additional ordering relation is called *subtyping* but it is not an ideal solution for this problem from our point of view as its type inference can be needlessly complicated.

Type variable	X	$::=$	$\alpha \mid \beta \mid \gamma \mid \cdots$	
Type constant	C	$::=$	$a \mid b \mid c \mid \cdots$	
Type class	Q	$::=$	$\texttt{Num} \mid \texttt{Eq} \mid \cdots$	
Monotype	τ	$::=$	X	(type variable)
		$\mid$	C	(type constant)
		$\mid$	$\tau \to \tau$	(function type)
Predicate	P	$::=$	$Q\ \bar{\tau}$	
Qualified type	ρ	$::=$	$\bar{P} \Rightarrow \tau$	
Polytype	σ	$::=$	$\forall \bar{\alpha}.\ \tau$	

Figure 1.6 Syntax of HM types with qualified types

Our preferred solution, originally introduced by Jones [Jon03] involves a creating *class* of types so that we can restrict the type of equality to be defined only for a *qualified* subset of all possible types. Thus the principal type of equality is $\forall\alpha.\ \texttt{Eq}\ \alpha \Rightarrow \alpha \to \alpha \to \texttt{Bool}$ — the function is defined for all types α which are in the class of equatable types `Eq`, i.e. $\texttt{Eq}\ \alpha$ holds. The principal type of addition would then be $\forall\alpha.\ \texttt{Num}\ \alpha \Rightarrow \alpha \to \alpha \to \alpha$ where `Num` is a class of numeric types.

In order to be able to work with such types, we have to introduce new type syntax in figure 1.6 so that it supports *qualified types* which are monotypes qualified with *predicates*. Each predicate is formed by applying some monotypes as arguments to a *type class*. We keep a list of valid predicates, for example `Num Int` and `Num Float`. Each type class is identified by a name and arity. Both `Eq` and `Num` have arity 1 as they expect a single monotype applied to them.

Qualified types may be expressed in our constraints-based interpretation of Hindley-Milner. Each predicate is simply interpreted as a constraint by itself. A predicate P is satisfied by a substitution S iff SP is a valid predicate, therefore we solve P by removing it if it appears in our list of valid predicates. At the end of inference, we simply return all unsatisfied predicates and let the algorithm 2 return a qualified type.

We could make predicates much richer by letting the programmer create new type classes and valid predicates (called instances in Haskell), see [Jon03; Jon99] for more details.

Chapter 2

Linear types

As we have shown in the previous chapter, type systems are used to restrict the universe of valid programs. A very common source of errors are memory safety issues. This is especially problematic in the field of computer security [Sze+13].

Consider the following simplified system file interface where `File` is a file handle and `Unit` is a type of a constant $()$ — commonly used in the same way as `void` return type in C++:

$$\begin{aligned} \textsc{Open} &: \texttt{String} \to \texttt{File} \\ \textsc{Write} &: \texttt{String} \to \texttt{File} \to \texttt{File} \\ \textsc{Close} &: \texttt{File} \to \texttt{Unit} \end{aligned}$$

As an invariant, system file interfaces allow writing only into files which have been opened and have not been closed. Additionally, it is expected that the program closes all of the file handles which have been previously opened. Unfortunately, type systems presented in chapter 1 do not honor such invariants as the programmer may forget to close a file handle. Alternatively, the programmer may also *copy* the file handle, close one and then attempt to write into the other one.

There is a similar issue with many system resources — file handles, as shown above, are either open or closed; memory is either allocated or unallocated; locks are either open or closed.

The goal of this chapter is to present a static, general solution to such problems directly in the type system. This way, expressions misusing system resources will produce a type error and will therefore be invalid.

$$\frac{\Gamma_1,\ \Gamma_2\ \vdash E : \tau}{\Gamma_2,\ \Gamma_1\ \vdash E : \tau}\ [\textsc{Exchange}]$$

$$\frac{\Gamma\ \vdash E_1 : \tau_1}{\Gamma, E_2 : \tau_2\ \vdash E_1 : \tau_1}\ [\textsc{Weakening}]$$

$$\frac{\Gamma, E_2 : \tau_2, E_2 : \tau_2\ \vdash E_1 : \tau_1}{\Gamma, E_2 : \tau_2\ \vdash E_1 : \tau_1}\ [\textsc{Contraction}]$$

Rule set 5 Structural rules of Γ

2.1 Substructural type systems

In order to achieve our goal, we first need to explicitly state some of the rules for contexts that we have used implicitly. The rules described in rule set 5 are called *structural* as they determine the structure of the type system:

- Rule of [Exchange] — the context Γ is unordered, meaning that if we can infer a type under Γ, then we can infer the same type under any permutation of bindings in Γ.
- Rule of [Weakening] — adding extra, redundant, unnecessary bindings to Γ is allowed.
- Rule of [Contraction] — any duplicate binding in Γ is ignored, that is if $\Gamma_1 = \varnothing, E : \tau, E : \tau$ and $\Gamma_2 = \varnothing, E : \tau$, then Γ_1 is morally equivalent to Γ_2.

Type systems using only a subset of structural rules are called *substructural*. Although there are many possible combinations, only the following type systems have known useful applications [Pie05, Chapter 1]:

- *Ordered* type systems guarantee that variables are used exactly once, in the same order as they are declared — they do not support any of the structural rules.
- *Linear* type systems guarantee that every variable is used exactly once — they support exchange but not weakening or contraction.
- *Relevant* type systems guarantee that every variable is used at most once — they support exchange and contraction but not weakening.
- *Affine* type systems guarantee that every variable is used at least once — they support exchange and weakening but not contraction.

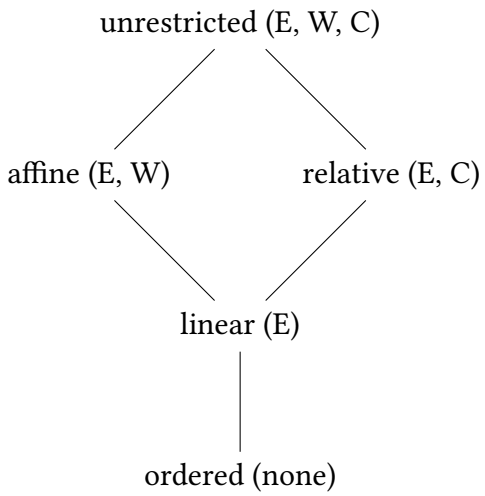

Figure 2.1 Hasse diagram of partial order $\geqslant$ (admits at least the same structural rules as), adapted from [Pie05, Chapter 1]. Each type system has its structural rules abbreviated next to its name — E stands for exchange; W stands for weakening; C stands for contraction.

- *Unrestricted* type systems do not provide any guarantees — they support exchange, weakening and contraction.

For example, the type system of STLC (rule set 1) is an unrestricted system as its rules implicitly uses all of the structural rules in rule set 5, see [Wad93] for a proof.

Notice that each system has some subset of all the possible structural rules. We can partially order type systems T_1, T_2 using a relation $T_1 \geqslant T_2$ that holds iff T_1 admits at least the same structural rules as T_2. Since $\geqslant$ is a partial order, we can present it in the style of a Hasse diagram for easier reference in figure 2.1. Note that there are more possible combinations which are not shown in the diagram, such as type systems supporting only contraction and weakening.

2.2 Changing the world

Not only can linear types make system resource interfaces safer, there is another advantage to linear types in a functional setting. Consider the following interface for an array of type `Array` with elements of type `Value`:

$$\textsc{Lookup} : \texttt{Int} \to \texttt{Array} \to \texttt{Value}$$
$$\textsc{Update} : \texttt{Int} \to \texttt{Value} \to \texttt{Array} \to \texttt{Array}$$

Notice that UPDATE does not modify the array — it returns a new array. This means that we have to copy every value from the previous array, which might be suboptimal. Essentially, we cannot, as Wadler [Wad90] put it, *change the world* in functional programming; we can only return a new copy of the world. For a more optimal experience, we would like to ensure that we *destructively* update the old world, that is to produce a new one by a simple straightforward modification.

This is yet another application of a type system, which supports linear types. Note that the system does not have to be strictly linear. In fact, we could split all the values into two categories:

- (strictly) *linear* — for values which have a single, owning reference, therefore requiring no garbage collection
- *unrestricted* — for values that may have many references pointing to them, therefore requiring actual garbage collection

If our type system allows us to choose and enforce which values are linear and can enforce the linearity constraints, we may allow safe destructive updates. In order to do such updates, we need two things:

- Disallow duplication — If we ban duplication of linear values, we can guarantee that we have at most one live reference pointing to the value, i.e. that there is at most one reference to an array.
- Disallow discarding — If we ban discarding of linear values, we can guarantee that there is at least one live reference pointing to the value. This means that garbage collection is not required, the programmer will have to allocate and deallocate linear values explicitly.

In the previously shown unrestricted type systems, the following is a valid judgment even though it does not use the variable in its assumptions:

$$x : \texttt{Array} \vdash () : \texttt{Unit}$$

The *product* type $\alpha \times \beta$ is the type of a pair expression $(x,\ y)$, where the type of x is α and the type of y is β. Similarly to the previous judgment, the following is also a valid judgment in unrestricted type systems even though it uses the variable in the assumption twice:

$$x : \texttt{Array} \vdash (x,\ x) : \texttt{Array} \times \texttt{Array}$$

As we stated before in section 2.1, linear type systems guarantee that every variable is used exactly once. This means that both of the judgments above are invalid in a linear type system.

Linear types are closely related to linear logic by Girard [Gir87]. This variant of logic formalizes well known rules of working with resources. For example, if a person has two coins on entering a candy store which only sells lollipops for two coins, the person has either the lollipop or the two coins upon leaving the store, but not both.

Unfortunately adding linear types to a programming language is not so easy. The properties desired from a type system are the following:

- Complete type inference — no need for any explicit annotations from the programmer; the system can infer a type for any expression without any external help
- Decidable type inference — the process of inferring a type should be decidable
- Principal type property — there exists a single most general type for any expression; introduced in section 1.4
- No syntax changes — the programmer should not have to determine by themselves which functions are linear and which are not

2.3 Linear qualified types

Notice that the essence of duplication and discarding from the previous section is captured by the following two functions called Dup and Drop respectively:

$$\textsc{Dup} = \lambda x.\ (x,\ x)$$
$$\textsc{Drop} = \lambda x.\ ()$$

If there is a meaningful implementation of both operations for every value of some type τ, we claim that the type τ is *unrestricted*. Otherwise, we claim that the type is *linear*. Since we need to distinguish between different kinds of functions, we denote linear function types as $(\tau_1 \multimap \tau_2)$ and unrestricted function types as $(\tau_1 \to \tau_2)$. Note that a linear function is a *single-use function* in our interpretation, unlike in other linear systems like Linear Haskell [Ber+17], where it signifies a function which uses its argument exactly once.

To formalize this idea, Morris [Mor16] uses qualified types from section 1.6 to create a new type class for unrestricted types denoted as `Un`. That is, a predicate `Un` τ holds if τ is unrestricted. Every type in `Un` supports the aforementioned

operations Dup and Drop. Note that both of the operations need to have unrestricted function types. Therefore the types of the operations are as follows:

$$\textsc{Dup} : \forall\alpha.\ \texttt{Un}\ \alpha \Rightarrow \alpha \multimap \bullet\ \alpha \times \alpha$$
$$\textsc{Drop} : \forall\alpha.\ \texttt{Un}\ \alpha \Rightarrow \alpha \multimap \bullet\ \texttt{Unit}$$

As per our requirements, we shall not force the programmer to write these operations by themselves explicitly. We infer their use from type declarations after the type inference process is finished.

Let us examine the type for the following expression $\lambda(f,\ x).\ f\ x$ — simply takes a pair of function and argument and applies the function to the argument. In an unrestricted type system, this expression has a type $(\alpha \multimap \beta) \times \alpha \multimap \beta$. However in a type system with linearity, we have more choices for each function arrow that appears in the type, thus getting four types which are seemingly incomparable:

$$(\alpha \multimap \beta) \times \alpha \multimap \beta \qquad (\alpha \multimap \bullet\ \beta) \times \alpha \multimap \beta$$

$$(\alpha \multimap \beta) \times \alpha \multimap \bullet\ \beta \qquad (\alpha \multimap \bullet\ \beta) \times \alpha \multimap \bullet\ \beta$$

In order to solve this issue, we introduce another type class for function types — Fun. A predicate $\texttt{Fun}\ f$ means that a type variable f is a function type: either $\multimap\bullet$ or $\multimap$. We also write $\alpha \xrightarrow{f} \beta$ for a function type parameterized by f, where f is a type variable for which $\texttt{Fun}\ f$ holds. This change applied to the first element of the pair reduces possible types for the expression to two:

$$\texttt{Fun}\ f \Rightarrow (\alpha \xrightarrow{f} \beta) \times \alpha \multimap \beta$$

$$\texttt{Fun}\ f \Rightarrow (\alpha \xrightarrow{f} \beta) \times \alpha \multimap \bullet\ \beta$$

Finally, we reintroduce a shorthand notation: we write $\alpha \to \beta$ for a type $\alpha \xrightarrow{f} \beta$ with a fresh f and the predicate $\texttt{Fun}\ f$. We can now state that the single most general type of $\lambda(f,\ x).\ f\ x$ is $(\alpha \to \beta) \times \alpha \to \beta$ — the same type as it had in an unrestricted system, though now with a different meaning.

Next, let us examine the type for uncurried application — $\lambda f.\ \lambda x.\ f\ x$. In an unrestricted system, it has a type $(\alpha \to \beta) \to \alpha \to \beta$. Notice that the type of the function depends on linearity of the argument substituted for f in the expression. Similarly to our previous dilemma, this expression has six incomparable types in a system with linearity, which reduces to just two types using the newly introduced Fun predicate and the related sugar:

$$\texttt{Fun}\ f \Rightarrow (\alpha \xrightarrow{f} \beta) \to \alpha \xrightarrow{f} \beta$$

$$(\alpha \multimap \bullet\ \beta) \to \alpha \multimap \beta$$

Notice that $(\texttt{Fun}\ f,\ \texttt{Fun}\ g) \Rightarrow (\alpha \xrightarrow{f} \beta) \to \alpha \xrightarrow{g} \beta$ is almost the most general form but it allows a scenario where the first argument of the function is linear but the returned function is not. What we want to say is that in the proposed type, f allows for at least the same structural rules as g. We have already encountered this concept in section 2.1 — it is the $\geqslant$ partial order! We can introduce this order directly into the typing process as a type class! Therefore the most general type of uncurried application is:

$$(\texttt{Fun}\ f,\ \texttt{Fun}\ g,\ f \geqslant g) \Rightarrow (\alpha \xrightarrow{f} \beta) \to \alpha \xrightarrow{g} \beta$$

Note that the $\geqslant$ order is not limited only to comparing function types, but it can be used to compare a type and a function type, such as in the most general type of the K combinator $\lambda x.\ \lambda y.\ x$, which is:

$$(\texttt{Un}\ \beta,\ \texttt{Fun}\ f,\ \alpha \geqslant f) \Rightarrow \alpha \to \beta \xrightarrow{f} \alpha$$

We now present typing rules in rule set 6 for the system described above by Morris [Mor16]. Our presentation is different from the original by using type constraints as described in section 1.5, since it is not only easier to implement it, but most of the concepts from type constraints can be elegantly reused.

First off, we add a 'f fresh function' judgment, which is equivalent to 'f fresh' except for the fact that it emits a $\texttt{Fun}\ f$ constraint. Next, we need to lift the definition of $\geqslant$ into assumptions, that is $\mathcal{A} \geqslant \alpha$ is defined as $\{\sigma \geqslant \alpha \mid (x : \sigma) \in \mathcal{A}\}$. Then we need to define $\mathrm{un}.\ \mathrm{un}(\mathcal{A}_1, \mathcal{A}_2)$ simply emits the constraint $\texttt{Un}\ \sigma$ for every assumption $(x : \sigma)$ for all x which are both in $\mathcal{A}_1$ and in $\mathcal{A}_2$. This can also be expressed as a restriction of $\mathcal{A}_1 \cup \mathcal{A}_2$ to the variables which are present in $\mathcal{A}_1 \cap \mathcal{A}_2$. Finally, we introduce $\mathrm{wkn}(x, \sigma, \mathcal{A})$ which either does not emit any constraints if $(x, \alpha) \in \mathcal{A}$, otherwise it emits $\texttt{Un}\ \sigma$.

Let us go over the rules in rule set 6, explaining the differences from plain constraints-based Hindley-Milner (rule set 4):

- Rule [Var]: Unchanged from rule set 4.
- Rule [Abs]: The returned function type is parameterized by fresh type variable f. In addition to the constraints generated previously, we also account for the possibility that x was not used in the abstraction body E — we use $\mathrm{wkn}(x, \beta, \mathcal{A})$ to do so as it adds $\texttt{Un}\ \beta$ if x is not present in the assumptions $\mathcal{A}$. That means that we have not encountered x in the body E. Next, we use $\mathcal{A} \backslash x \geqslant f$ to ensure that every variable encountered in the body E supports at least the same structural rules as f does.
- Rule [App]: Again, a new fresh function type variable f is created. We also emit constraints $\mathrm{un}(\mathcal{A}_1 \cap \mathcal{A}_2)$, which ensure that all of the variables that

$$\frac{\beta \text{ fresh}}{\{x : \beta\} \vdash x : \beta \mid \varnothing} \quad [\textsc{Var}]$$

$$\frac{\begin{array}{c}\mathcal{A} \vdash E : \tau \mid \mathcal{C} \\ \beta \text{ fresh} \quad f \text{ fresh function} \\ \mathcal{C}' = \mathcal{C} \cup \{\tau' \sim \beta \mid (x : \tau') \in \mathcal{A}\} \cup \mathrm{wkn}(x, \beta, \mathcal{A}) \cup \mathcal{A}\backslash x \geqslant f\end{array}}{\mathcal{A}\backslash x \vdash (\lambda x.\ E) : \beta \xrightarrow{f} \tau \mid \mathcal{C}'} \quad [\textsc{Abs}]$$

$$\frac{\begin{array}{c}\mathcal{A}_1 \vdash E_1 : \tau_1 \mid \mathcal{C}_1 \quad \mathcal{A}_2 \vdash E_2 : \tau_2 \mid \mathcal{C}_2 \\ \mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2 \\ \beta \text{ fresh} \quad f \text{ fresh function} \\ \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \sim \tau_2 \xrightarrow{f} \beta\} \cup \mathrm{un}(\mathcal{A}_1 \cap \mathcal{A}_2)\end{array}}{\mathcal{A} \vdash (E_1\ E_2) : \beta \mid \mathcal{C}} \quad [\textsc{App}]$$

$$\frac{\begin{array}{c}\mathcal{A}_1 \vdash E_1 : \tau_1 \mid \mathcal{C}_1 \quad \mathcal{A}_2 \vdash E_2 : \tau_2 \mid \mathcal{C}_2 \\ \mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2\backslash x \\ \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau' \preccurlyeq_M \tau_1 \mid (x : \tau') \in \mathcal{A}_2\} \\ \mathcal{C}' = \mathrm{un}(\mathcal{A}_1 \cap \mathcal{A}_2) \cup \mathrm{wkn}(x, \tau_1, \mathcal{A}_2)\end{array}}{\mathcal{A} \vdash \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \tau_2 \mid \mathcal{C}'} \quad [\textsc{Let}]$$

Rule set 6 Typing rules for qualified linear types

occurred both in E_1 and E_2 are unrestricted — the predicate `Un` α holds for every variable of type α in the intersection of the two assumptions.

- Rule [Let]: Yet again, we emit $\mathrm{un}(\mathcal{A}_1 \cap \mathcal{A}_2)$ to ensure that variables occurring both in E_1 and E_2 are unrestricted. We also use $\mathrm{wkn}(x, \tau_1, \mathcal{A}_2)$ to ensure that x is unrestricted if it was not used in $\mathcal{A}_2$.

We can reuse the algorithm Solve 1 to solve the generated constraints and the algorithm InferType 2 to infer a principal type in the described system. Solving predicate constraints is described in section 1.6.

Chapter 3

Lily

In this chapter, we present the design choices and a description of the implementation of the linter *Lily* which uses linear types from chapter 2 to suggest code improvements to given C++ source code.

Lily can be split into four major parts: parsing, elaboration, type inference and linting. The program itself is written in the programming language Haskell with heavy use of optics and type-level programming. This chapter goes over the implementation of each of the major parts, highlighting encountered problems and their respective solutions.

3.1 Parsing and semantic analysis

3.1.1 Using Clang

Because C++ is a very complex language, as witnessed by its multiple competing Turing complete systems (templates, constexpr, the language itself) and by its undecidable parsing, we chose to use bindings to the `Clang` C++ compiler by Lattner [Lat08].

However because `Clang` does not have direct bindings for Haskell, we had to jump through several hoops in order to connect `Clang` written in C++ to Lily written in Haskell. First, we use `libClang`, an official C library in the `Clang` suite with bindings for a subset of the functionality offered by `Clang`.

As of the time of writing (`Clang` version 10), the subset of `Clang`'s functionality supported by `libClang` is very small. For example, even though `Clang` does recognize different kinds of binary and unary operators, there is no way to get this information from `libClang`. Furthermore, it is not desirable for us to modify `libClang`, because that would force every user to compile the whole `Clang`/LLVM suite when compiling Lily.

Next, we use a modified version of the `clang-pure` [Chi20] library, which itself connects to the `libClang` library and provides a nice interface to `Clang` in Haskell. The library uses type-level programming (dependent sums) for greater type safety and optics for a simpler way to manage heavily nested structures. However, it is again not very feature-rich as it provides only a subset of the functionality provided by `libClang`. We added several new functions to the `clang-pure` library in our own modified version that we intend to incorporate to the original library.

All together, `clang-pure` is used to lex and parse most of the C++ source code, create a weakly linked Abstract Source Tree (AST) with nodes called *cursors*, perform semantic analysis in order for us to be able to find all references of a function/variable or jump to a definition of a function/variable. If any of these processes result in an error, Lily informs the user of the error and does not progress any further.

3.1.2 Untying the knot

In order to do anything productive with the AST, we need to identify all functions and their relations. Specifically, recursive functions and groups of recursive functions are a potential problem, since we need to employ the fix idiom mentioned in section 1.3 to infer the type properly.

We therefore generate a directed graph where vertices are individual C++ functions and methods and there is an edge from a function f_1 to another function f_2 if there is any call to f_2 in f_1. The result is a directed graph of dependencies between functions.

For example, for the C++ program in listing 1, we create the dependency graph in figure 3.1. Notice that we can easily split the graph into *strongly connected components*, each component having a different color in the figure. Now if we sort the components in *reverse topological order*, that is f_1 comes before f_2 if there is an edge from f_2 to f_1, we get a good order to process the components in. Note that there can be a component of size one which is by itself a recursive group, for example the `factorial` function.

Everything described in this subsection is provided in the `Clang` module in Lily, especially the `Clang.Function` and `Clang.Struct` modules.

As mentioned previously in section 3.1.1, the features provided by the `clang-pure` library are lacking. In order to remedy some of the failures, we created two small parsers — one for parsing unary and binary operations located in `Clang.OpParser` and one for parsing member/field references in `Clang.MemberParser`. We think that this functionality should be included by default in `libClang`.

Listing 1 A C++ program with multiple different kinds of function dependencies

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    }

    return n * factorial(n-1);
}

int even(int n);
int odd(int n);

int odd(int n) {
    if (n == 0) return 0;
    else return even(n - 1);
}

int even(int n) {
    if (n == 0) return 1;
    else return odd(n - 1);
}

int square(int n) {
    return n * n;
}

int uses_even(int n) {
    return even(n) * 2;
}
```

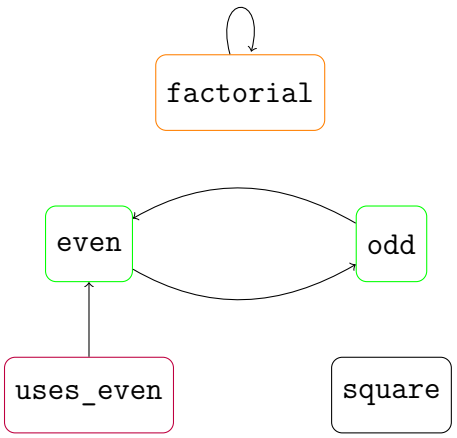

Figure 3.1 The dependency graph created from the program in listing 1

$$
\begin{array}{llll}
\text{Variable} & V & ::= & x \mid y \mid z \mid \cdots \\
\text{Expression} & E & ::= & V & \text{(variable)} \\
 & & \mid & E\ E & \text{(application)} \\
 & & \mid & \lambda V.\ E & \text{(abstraction)} \\
 & & \mid & \mathsf{let}\ V = E\ \mathsf{in}\ E & \text{(let)} \\
 & & \mid & \mathsf{if}\ E\ \mathsf{then}\ E\ \mathsf{else}\ E & \text{(if)} \\
 & & & & \\
\text{Declaration} & D & ::= & \mathsf{fun}\ V = E & \text{(non-recursive declaration)} \\
 & & \mid & \mathsf{rec}\ [\overline{\mathsf{fun}\ V = E}] & \text{(recursive declaration group)} \\
 & & \mid & \mathsf{struct}\ [\overline{\mathsf{fieldname}\ :\ \tau}] & \text{(struct declaration)}
\end{array}
$$

Figure 3.2 Syntax of Lily Core

3.2 Elaboration

The first version of Lily did not have a separate elaboration phase as it directly inferred type over C++ AST. This was problematic for several reasons. Since the C++ AST is highly irregular, it was not trivial to adapt the type rules directly. By performing *elaboration* — a process in which we translate the C++ AST into a small, intermediate representation, we gain a core language which is much easier to reason about. Inspired by the Glasgow Haskell Compiler (GHC), which calls its intermediate representation language 'GHC Core', the intermediate representation language for Lily is called 'Lily Core', or simply 'Core'.

The syntax of Lily Core (as shown in figure 3.2 is quite simple by design as a smaller language is much easier to analyze. Remember that C++ source code is elaborated into Core after the dependency analysis is complete — this results in explicit recursion group handling of declarations. The language also contains builtins such as `this` or `new` keywords and a construct for C++ literals.

We have not created any parser for Core since it is only an internal representation. However, Lily can pretty-print any valid Core expression to the user.

Even though all C++ functions and variables have an explicit type annotation, Lily mostly does not use it during the elaboration phase. The only places type information from C++ is used in the elaboration phase is in typing literals and most builtins in order to avoid any invisible implicit type conversions. It is also present in the definition of struct fields.

Elaboration is only implemented in Lily for a very small subset of C++, namely top-level functions and structures, `if` statements, variable declarations and function calls. However we believe that the chosen subset represents the very core of the C++ programming language and supporting a non-trivial subset

of C++ should not be hard. As an example, C++ includes statements like `while` and `for` which are not directly supported in Lily. But Lily supports recursion and iteration can be expressed as recursion.

Lily stores a parent `Clang` cursor for every expression, thus allowing us to produce nice error messages with approximate locations in the C++ source code even when working with Core. The inner representation of Core is inspired by the 'Trees that Grow' representation as introduced by Najd and Jones [NJ17].

As an example, the functions `factorial` and `square` from listing 1 are elaborated into the following Core declarations:

$$
\begin{aligned}
\text{fun square } n &= \text{builtin\_Mul}_{\texttt{Int}\to\texttt{Int}\to\texttt{Int}}\ n\ n \\
\text{rec fun factorial } n &= \\
&\quad \text{if builtin\_EQ}_{\texttt{Int}\to\texttt{Int}\to\texttt{Bool}}\ n\ 0 \\
&\quad\ \text{then } 0 \\
&\quad\ \text{else builtin\_Mul}_{\texttt{Int}\to\texttt{Int}\to\texttt{Int}}\ n \\
&\quad\quad (\text{factorial } (\text{builtin\_Sub}_{\texttt{Int}\to\texttt{Int}\to\texttt{Int}}\ n\ 1))
\end{aligned}
$$

The implementation of Core in Lily is located in the `Core` module — the definition of the syntax is in `Core.Syntax` and the elaboration is located in `Core.Elaboration`. The elaboration process itself is mostly a limited version of the continuation-passing style conversion as described by Appel [App07] in blocks. This is done because C++ returns explicitly with the `return` keyword while Core returns implicitly as all non-top-level terms are expressions.

Note that elaboration is not as simple as it seems since the AST provided by `clang-pure` is highly irregular with constructs like `FirstExpr`. This obfuscates the actual AST and makes the pattern matching code much more complicated as we sometimes have to rely on the order of the children of a cursor which could change between `Clang` versions.

3.3 Type inference

First off, let us go over the syntax of types in figure 3.3. There are just three type classes used in type inference which were inherited from section 2.3. Similarly, function type variables from the same chapter are used to signify a type variable which stands for a function arrow. The set of type constants is populated by basic C++ types such as `Int` or `Bool` and by all struct/record types that the programmer supplied in their code.

Note that even though it is not required, our implementation keeps track of the type of each type called a *kind*. But since neither type application nor custom higher-kinded types except from the function type variables are created,

Type variable	X	$::=$	$\alpha \mid \beta \mid \gamma \mid \cdots$	
Function type variable	F	$::=$	$f \mid g \mid h \mid \cdots$	
Type constant	C	$::=$	$\texttt{Int} \mid \texttt{Bool} \mid \cdots$	
Type class	Q	$::=$	$\texttt{Un} \mid \geqslant \mid \texttt{Fun}$	
Monotype	τ	$::=$	X	(type variable)
		$\mid$	F	(function type variable)
		$\mid$	C	(type constant)
		$\mid$	$\tau \xrightarrow{F} \tau$	(function type)
Predicate	P	$::=$	$Q\ \bar{\tau}$	
Qualified type	ρ	$::=$	$\bar{P} \Rightarrow \tau$	
Polytype	σ	$::=$	$\forall \bar{\alpha}.\ \tau$	

Figure 3.3 Type syntax of Lily Core

Constraint	C	$::=$	$\tau_1 \sim \tau_2$	(equality)
		$\mid$	$\tau \preccurlyeq \sigma$	(explicit instance)
		$\mid$	$\tau_1 \preccurlyeq_M \tau_2$	(implicit instance)
		$\mid$	P	(predicate constraint)

Figure 3.4 Syntax of constraints in Lily

we think that it is not strictly necessary to keep track of kinds. However it might be necessary for possible extensions of Lily.

3.3.1 Constraint generation

In order to generate constraints, we use the constraints formulation of linear qualified types as seen in rule set 6 with additional rules for builtins, literals and the if expression. The type syntax is implemented in the `Type.Type` module together with facilities for substitution and getting free type variables of a type. Lily does not provide a parser for types since all types are inferred but the syntax is well defined and it should not be hard to add one if desired.

There are four kinds of constraints in Lily (figure 3.4) — equality, explicit instance, implicit instance or a predicate constraint. Lily creates a list of valid predicates before the constraint generation starts. A predicate is solved when it matches one of the valid predicates. Note that the inference system is not perfect — it does not support any proper subtyping, which means that neither C++ inheritance nor implicit casting directly.

During the whole type inference process, Lily keeps track of *reasons* why

something happened; tracing every step through the inference. This has proven to be very useful both for explaining why Lily chose the returned type for an expression and for debugging purposes. Every constraint therefore has one of the many reasons why and how it was created. Both constraints and reasons are implemented in the `Type.Constraint` module.

Remember that when constrained types were introduced, we assumed access to a generator of fresh type variables. This is implemented in the module `Control.Monad.Fresh` which creates a general monadic abstraction for generating fresh identifiers based on the state monad abstraction. Only later is it specialized to create fresh type variables in the module `Type.Fresh`.

Type classes and predicates are handled in the `Type.Class` module with support for multi-parameter type classes and even functional dependencies as described by Jones [Jon00]. The design of this module is heavily inspired by Jones [Jon99].

The actual constraint generation uses type rules from rule set 6. It is located is in module `Type.Infer` using a custom monad that allows to read monomorphic variables, create fresh type variables and raise helpful errors. The rules are written in a simple, declarative fashion — simply transcribed from the rule set. Note that there are some unshown, yet obvious rules for builtins, such as type for the `new` operator.

3.3.2 Constraint solving

After constraints are generated, we need to solve them in the module called `Type.Solve`. The solving algorithm is unchanged from the Solve 1 algorithm shown previously.

We also add a constraint simplifier as shown in figure 3.5 which runs several times until a fix-point is found. The constraint simplifier for predicate constraints always terminates because the number of constraints is always reduced by at least 1 every time it runs.

The unification algorithm mgu needed in the Solve algorithm is located in the module `Type.Unify`. Its design is inspired by the algorithm described by Jones [Jon99]

3.4 Linting

We have finally inferred a type for some C++ function, now we want to produce linearity-related suggestions called *lints* for the programmer. In this stage, the inferred type is compared with the *checked* type obtained from `Clang`. Whereas

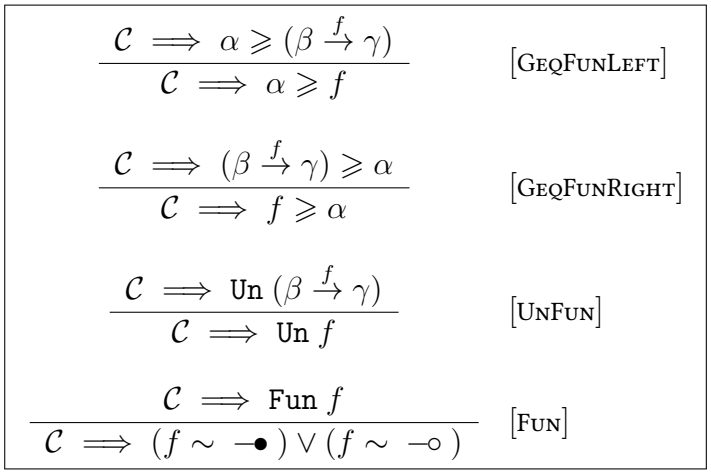

Figure 3.5 Predicate simplification in Lily

before the inference was completely independent of `Clang`, now we employ the `Clang.Type` module to convert a C++ type into our type representation.

In order to model lvalue and rvalue references as specified by the C++ standard [ISO17], we use the `LRef` α and `RRef` α types respectively. To compare the inferred and checked type, we instantiate the inferred polytype into a monotype and a list of predicates and then we use a one-way unification similar to mgu introduced in section 1.4, with the added caveat that both lvalue and rvalue references to type τ are unifiable with τ. One-way unification is implemented in the `Type.Unify` module.

The result of one-way unification is a valid substitution if there is one, which is applied to the predicates. Then we attempt to solve the predicates. If any predicates are not solvable, they are reported to the user.

The most common type class that appears in the inferred type we are comparing against is `Un`, which signifies that the type is being discarded or duplicated. Specifically, linear types should never have a `Un` constraint. We consider predicates `Un` (`LRef` α) for all α as solvable since lvalue references are copyable, whereas `Un` (`RRef` α) and `Un` (`Ptr` α) are not solvable as rvalue references and pointers should act linearly.

All of the described functionality is implemented in the module `Lint`, together with custom suggestion messages for common sources of suboptimal behavior. Note that some suggestions might be a false positive since Lily does not analyze the structure of the type and views a struct with one `Int` field the same way as a struct with megabytes of data, but it recommends against duplicating and discarding for both structs.

As an example, consider the program in listing 2 which exhibits unwanted

Listing 2 C++ program with a *use-after-free* problem — it allocates a pointer, then deallocates it but then returns an invalid pointer from the function

```
int* use_after_free() {
    int* ptr = new int;
    delete ptr;
    return ptr; // returns (uses) after deleting!
}
```

```
The following predicate could not be satisfied!
    Un (Ptr Int)
at location: examples/useafterfree.cpp:1:1

This means that a pointer is either duplicated or discarded!
```

Figure 3.6 Output of Lily's linting for listing 2

behavior as a linear resource is used after deallocation. Since Lily infers a following type which requires that a pointer is unrestricted, it returns a helpful message to the user indicative of the aforementioned issue as shown in figure 3.6.

Chapter 4

Results and discussion

4.1 Results

We created the Lily linter as a proof-of-concept implementation of combining linear qualified types with a constraints generating approach applied to linearity in C++, without the need for the programmer to annotate or modify the program in any ways. Ideally the goal was to create a tool which would find issues that other linters are not able to.

Although the scope of Lily is limited to a small subset of actual C++, it nevertheless can untangle complex linear behaviors including things like mutual recursion to accurately find actual issues.

Note that we err on the side of false positives as it outputs mere suggestions for the actual programmer which are then to be examined by a human, e.g. reporting that a type is unrestricted even though it might not be a problem for example for small structs which are passed in only a small number of registers.

We found a program (listing 3) which uses mutual recursion to create a situation where the linear resource (a pointer in our case) is used only if the argument n is even, otherwise we silently drop it. This results in the output in figure 4.1 from Lily. Note that the same program can be implemented with rvalue references and the use of `std::move` instead of pointers, but Lily can still detect the fact that one of the functions uses its argument in non-linear ways.

Listing 3 C++ program with mutual recursion and passing a pointer, available as `examples/mutuallyrecursive.cpp` in Lily

```cpp
int even(int n, int* importantptr);
int odd(int n, int* importantptr);

int odd(int n, int* importantptr) {
    if (n == 0) return 0;
    else return even(n - 1, importantptr);
}

int even(int n, int* importantptr) {
    if (n == 0) return *importantptr;
    else return odd(n - 1, importantptr);
}
```

```
The following predicate could not be satisfied!
    Un (Ptr Int)
at location: examples/mutuallyrecursive.cpp:4:1

This means that a pointer is either duplicated or discarded!
```

Figure 4.1 Output of Lily's linting for listing 3

Although this problem is obvious from a linearity perspective, widely used C++ linters such as `cppcheck` or `clang-tidy` from the `Clang` ecosystem are not able to detect it, which successfully fulfills our goal.

Similarly, Lily can point out that a linear resource was already moved, see listing 4. This mistake is not really caught by other linters — they only report that the variable `x1` is not used. However Lily can detect the actual error as shown in figure 4.2.

Listing 4 C++ program with mutual recursion and passing a pointer, available as `examples/doublemove.cpp` in Lily

```cpp
#include <utility>

int doublemove(int&& x) {
    int x1 = std::move(x);
    int x2 = std::move(x); // moving ‘x‘ again!
    return x2;
}
```

```
The following predicate could not be satisfied!
    Un (RRef Int)
at location: examples/doublemove.cpp:3:1

This means that a rvalue reference
is either duplicated or discarded!
```

Figure 4.2 Output of Lily's linting for listing 4

Note that there are many different examples included with Lily in the `examples/` folder!

4.2 Related work

There are many different substructural type systems for both functional and imperative programming languages. We present some of these systems which are comparable or related to Lily.

Quill [Mor16] is the most important influence on Lily — it describes adding linear types using qualified types into a Haskell-like programming language. It is also presented together with a proof of soundness of its algorithm.

Linear Haskell [Ber+17] (LH) is a proposal to add linear types to Haskell. Unlike Quill, it uses 'linear arrows' $\multimap$ — functions that *use* their arguments exactly once. Unlike in our system, the programmer is forced to write programs in a continuation-passing style to ensure proper linearity. The proposal contains neither a formal description of its inference algorithm, nor a proof of its properties. We believe that the direct style of programming supported by Lily is easier to reason about for both functional and imperative programmers.

Rust [MK14] is the best known system with linear and affine types. It uses the concepts of ownership and borrowing to ensure safe low-level programming. All types in Rust are affine by default, but every type may implement a trait (similar to a type class) like `Copy`, `Clone` or `Drop` to unlock the potential of linear and unrestricted systems. Unlike Lily, Rust does not support full, principal type inference, since its regions (lifetimes) are potentially very complex.

Affe [RT19] was developed in parallel with our system — it shares many features of Quill and Lily while adding a proper system for region inference and borrowing for a ML-like language, thus well modelling both shared immutable and exclusive mutable references. Affe formalizes the $\geqslant$ relation as a relation between kinds (the types of types) instead of a type class. It also uses a constraints system, but unlike Lily which uses a system based on the work of Heeren, Hage, and Swierstra [HHS02], Affe uses a system based on the HM(X) system of Odersky, Sulzmann, and Wehr [OSW99].

4.3 Future work

There are many directions for future work, both from a practical (Lily-related) and a theoretical perspective:

Larger subset of C++ We have presented just a small subset of real C++ as described by ISO [ISO17]. We have omitted important features of C++ such as inheritance (subtyping), object methods, overloaded operators and implicit conversions. We believe that most of said properties of the C++ language could be supported by Lily. Whereas some changes are quite trivial, e.g.

adding support for overloaded operators or C++ lambdas, some changes might require restructuring the type system, e.g. subtyping.

Especially inheritance might require full *existential* types, that is types with a $\exists$ quantifier. With some care, support for existential types while retaining the important principality of our type inference should be possible, see the work by Munch-Maccagnoni [MM18]. GHC solves existential types by adding a implication constraint into its system as described in the OutsideIn(X) system [Vyt+11].

Borrows Lily does not support an important part of resource management — borrowing. This would allow us to support not only idioms like `std::unique_ptr`, but also to infer regions akin to Rust [MK14]. This change would introduce a shared, immutable borrow similar to a const lvalue reference and a exclusive, mutable borrow similar to a nicely behaving lvalue reference. Unlike Affe [RT19], C++ does not have a syntax for borrowing at call site, but only in the function definition, so the system in Affe is not directly applicable for a syntax-driven type system for C++.

Affine types The system described in this thesis infers the `Un` type class for unrestricted types. As Morris [Mor16] noted, this type class could be split into two smaller type classes, one allowing duplication, another allowing discarding. This would lead to having possibly four kinds of arrows.

Exceptions Lily does not acknowledge the existence of exceptions in C++. This behavior could be mitigated by introducing affine types as indicated in the previous point or by implementing a version of *algebraic effects* which have been proven [BP15] to help in similar settings. The work of Dolan et al. [Dol+17] might be useful for this task.

Mutability Lily does not really consider the difference between mutable and immutable types. We did not prioritize this as other C++ state-of-the-art linters are more than capable of doing this analysis. In order to bring proper mutability support into Lily, a new type class would have to be added for representing mutability and possibly another $\geqslant$-like relation 'is more mutable than', similarly to how unrestrictness is handled.

Linearity of a user-created type Every user-created type is considered linear in Lily. This is not necessarily true in C++, where linearity may be expressed via the deletion of the copy constructor and copy assignment. Unfortunately, `libClang` does not provide a way of determining such conditions.

Compiler optimizations We believe that after inferring which types are linear and which are not, Lily could automatically perform linearity-aware compiler optimizations, e.g. in optimizing closures.

Conclusion

The goal of this thesis was to combine the world of C++ with the world of linear types, whilst not requiring any actual changes to the source code of the program. We think that we have managed to achieve this goal as the presented techniques can be applied to a substantial subset of C++.

Using our prototype implementation in the Lily linter, we have successfully applied our work to a small subset of the language, detecting various issues. Although small, we think that the chosen subset of C++ is the very core of the C++ language and other concepts like iteration using `for` loops can be easily converted into recursion supported by Lily.

Using Lily, we were even able to find issues that the commonly used state-of-the-art C++ linters could not.

It should be reasonably easy to extend Lily to support a much larger subset of the C++ programming language. With some luck and quite a lot of programming work, we think that Lily could be extended to support even a non-trivial subset of the notoriously complicated C++ standard library provided the concerns presented in section 4.3 are addressed.

We think that the ideas presented in this thesis are applicable not only for detecting issues but also for creating a nice, formal model of the linearity-related semantics of C++ for the everyday C++ programmer.

Bibliography

[App07] Andrew W Appel. *Compiling with continuations*. Cambridge University Press, 2007.

[Bar+84] Hendrik P Barendregt et al. *The lambda calculus*. Vol. 3. North-Holland Amsterdam, 1984.

[Ber+17] Jean-Philippe Bernardy et al. "Linear Haskell: practical linearity in a higher-order polymorphic language". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–29.

[BP15] Andrej Bauer and Matija Pretnar. "Programming with algebraic effects and handlers". In: *Journal of logical and algebraic methods in programming* 84.1 (2015), pp. 108–123.

[Car96] Luca Cardelli. "Type systems". In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 263–264.

[Chi01] Olaf Chitil. "Compositional explanation of types and algorithmic debugging of type errors". In: *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*. 2001, pp. 193–204.

[Chi20] Patrick Chilton. *clang-pure*. `https://github.com/chpatrick/clang-pure`. 2020.

[Chu32] Alonzo Church. "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics* 33.2 (1932), pp. 346–366. ISSN: 0003486X. URL: `http://www.jstor.org/stable/1968337`.

[Chu40] Alonzo Church. "A formulation of the simple theory of types". In: *The journal of symbolic logic* 5.2 (1940), pp. 56–68.

[DM82] 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.

[Dol+17] Stephen Dolan et al. “Concurrent system programming with effect handlers”. In: *International Symposium on Trends in Functional Programming*. Springer. 2017, pp. 98–117.

[Gir71] Jean-Yves Girard. “Une extension de L’interpretation de gödel a L’analyse, et son application a L’elimination des coupures dans L’analyse et la theorie des types”. In: *Studies in Logic and the Foundations of Mathematics*. Vol. 63. Elsevier, 1971, pp. 63–92.

[Gir87] Jean-Yves Girard. “Linear logic”. In: *Theoretical computer science* 50.1 (1987), pp. 1–101.

[HHS02] BJ Heeren, Jurriaan Hage, and S Doaitse Swierstra. *Generalizing Hindley-Milner type inference algorithms*. 2002.

[Hin69] R. Hindley. “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60. ISSN: 00029947. URL: http://www.jstor.org/stable/1995158.

[ISO17] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Fifth. 2017, p. 1605. URL: https://www.iso.org/standard/68564.html.

[Jon00] Mark P Jones. “Type classes with functional dependencies”. In: *European Symposium on Programming*. Springer. 2000, pp. 230–244.

[Jon03] Mark P Jones. *Qualified types: theory and practice*. Vol. 9. Cambridge University Press, 2003.

[Jon+07] Simon Peyton Jones et al. “Practical type inference for arbitrary-rank types”. In: *Journal of functional programming* 17.1 (2007), pp. 1–82.

[Jon99] Mark P Jones. “Typing Haskell in Haskell”. In: *Haskell Workshop*. Vol. 7. 1999.

[Lat08] Chris Lattner. “LLVM and Clang: Next generation compiler technology”. In: *The BSD conference*. Vol. 5. 2008.

[Mai89] Harry G Mairson. “Deciding ML typability is complete for deterministic exponential time”. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 382–401.

[Mil78] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348 – 375. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(78)90014-4. URL: http://www.sciencedirect.com/science/article/pii/0022000078900144.

[MK14] Nicholas D Matsakis and Felix S Klock. “The Rust language”. In: *ACM SIGAda Ada Letters* 34.3 (2014), pp. 103–104.

[MM18] Guillaume Munch-Maccagnoni. “Resource Polymorphism”. In: *arXiv preprint arXiv:1803.02796* (2018).

[Mor16] J Garrett Morris. “The best of both worlds: linear functional programming without compromise”. In: *ACM SIGPLAN Notices* 51.9 (2016), pp. 448–461.

[NJ17] Shayan Najd and Simon Peyton Jones. “Trees that Grow.” In: *J. UCS* 23.1 (2017), pp. 42–62.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. “Type inference with constrained types”. In: *Theory and practice of object systems* 5.1 (1999), pp. 35–55.

[Pie05] Benjamin C Pierce. *Advanced topics in types and programming languages*. MIT press, 2005.

[Rob65] John Alan Robinson. “A machine-oriented logic based on the resolution principle”. In: *Journal of the ACM (JACM)* 12.1 (1965), pp. 23–41.

[Roj15] Raúl Rojas. “A tutorial introduction to the lambda calculus”. In: *arXiv preprint arXiv:1503.09060* (2015).

[RT19] Gabriel Radanne and Peter Thiemann. “Kindly Bent to Free Us”. In: *arXiv preprint arXiv:1908.09681* (2019).

[Str13] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

[Sze+13] Laszlo Szekeres et al. “Sok: Eternal war in memory”. In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 48–62.

[Vyt+11] Dimitrios Vytiniotis et al. “OutsideIn(X): Modular type inference with local assumptions”. In: *Journal of Functional Programming* 21 (2011), pp. 333–412. URL: https://www.microsoft.com/en-us/research/publication/outsideinx-modular-type-inference-with-local-assumptions/.

[Wad90] Philip Wadler. “Linear types can change the world!” In: *Programming concepts and methods*. Vol. 3. 4. Citeseer. 1990, p. 5.

[Wad93] Philip Wadler. “A taste of linear logic”. In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 1993, pp. 185–210.

[Wel99] J.B. Wells. “Typability and type checking in System F are equivalent and undecidable”. In: *Annals of Pure and Applied Logic* 98.1 (1999), pp. 111 –156. ISSN: 0168-0072. DOI: https://doi.org/10.1016/S0168-0072(98)00047-5. URL: http://www.sciencedirect.com/science/article/pii/S0168007298000475.

Appendix A

Using Lily

To compile the software from scratch, it is recommended to use the Nix package manager to handle all dependencies. Otherwise, the Cabal package manager is also supported, but the dependency management is more complex.

Note that the source code with a detailed tutorial in the `README.md` file is also freely available on GitHub at `https://github.com/jiribenes/lily`.

Compiling with Nix

First off, unzip the attachment `lily.zip`. Nix can be obtained at the following web page: `https://nixos.org`. After installing Nix, use the following command to compile Lily:

```
nix-build --attr lily default.nix
```

The resulting executable is located in `result/bin/`.

Compiling with Cabal

The software was tested with Cabal version 3.0 and GHC version 8.8.3. In order to run Lily, you will need these versions or newer. GHC and Cabal can be obtained for all popular platforms from official sources by using the `ghcup` tool at `https://www.haskell.org/ghcup/`.

You need a recent version of LLVM and Clang. The software was tested with LLVM and Clang version 9. In order to install LLVM and Clang on Debian-based Linux systems, use the tutorial located at `https://apt.llvm.org/`.

You might need to set the following environment variables for `clang-pure`. The values on the right hand side of the assignment might depend on the location of your LLVM and Clang installations:

```
export CLANG_PURE_LLVM_LIB_DIR=/usr/lib/llvm-9/lib;
export CLANG_PURE_LLVM_INCLUDE_DIR=/usr/lib/llvm-9/include;
```

First off, unzip the attachment `lily.zip`. Next, having all the necessary prerequisites, use the following command to compile Lily:

```
cabal new-build lily
```

In order to run Lily, use the following command where `$ARGS` are user-supplied arguments:

```
cabal new-run lily -- $ARGS
```

Running Lily

The rest of the tutorial assumes that invoking `$LILY` is the way to call Lily as the actual command might vary according to the chosen installation style.

Use the following command to show a help screen:

```
$LILY --help
```

Lily supports three commands:

- `elaborate`: This command simply translates a C++ source code file into Lily Core and pretty-prints the Core to the standard output.
- `infer`: This command infers types for a C++ source code file and pretty-prints the inferred types to the standard output.
- `lint`: This command reports issues found in the C++ source code file and pretty-prints a list of issues and notes to the standard output.

An example usage of Lily follows — linting a file `examples/mutuallyrecursive.cpp` which is included with Lily:

```
$LILY lint examples/mutuallyrecursive.cpp
```

By default, Lily does not know where your C++ standard library or any other includes are located. Simply add `--` after a target filename. All arguments after that are directly passed on to Clang. This means that the following invocation is the same as the one above, but it also tells Clang to include the `include/` folder.

```
$LILY lint examples/mutuallyrecursive.cpp -- -I include/
```