Bachelor Thesis
Bachelor of Science Informatik

Dynamic type analysis of metaprograms

Matthias Krebs

Technische Universität Darmstadt
Fachbereich Informatik
Software Technology Group

Prüfer: Prof. Dr. Mira Mezini
Betreuer: Dr. Sebastian Erdweg

Abgabetermin: 02.01.2015

Erklärung

Hiermit versichere ich gemäß der Allgemeinen Prüfungsbestimmungen der Technischen Universität Darmstadt (APB) §23 (7), die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	(Matthias Krebs)

Abstract

A wishful property for metaprograms is that generated code is well-typed. Yet type systems build context information with a top-down traversal of the AST from a program, but code is generated bottom-up. We propose *binding requirements* as an approach to build contextual information with a bottom-up traversal. This allows the definition of a type system whose information flow is bottom-up, enabling the integration of type systems in code generation. We introduce operators for the union of multiple requirement sets and for the constraint generation for intersecting requirements from two sets. We transform a type system with context for the Simply Typed Lambda Calculus and PCF into a bottom-up typesystem without context. We use the approach on a subset of the language Java, to evaluate the applicability to a real-world language.

Contents

Chapter 1

Introduction

1.1 Motivation

A metaprogram is a program with the ability to treat other programs as data. For example, a compiler takes a program written in a language and translates it to an equivalent program of another language. Properties that are usefull for a metaprogram to fulfill, are that the generated code is well-formed according to the syntax of the target language and that it is well-typed according to the type system of the target language. A developer can be sure when using a metaprogram that fulfills these properties, that the generated code has no unsound behavior during its runtime. A solution for the guarantee that generated code is well-formed accoring to a syntax definition is proposed in [EVMV14].

Code generators construct the abstract syntax tree (AST) of a program bottom-up, which means that the construction starts at the leafs or the innermost fragment of a program. Type systems use usually a type context to keep track of the bound variables of a program and that type context is filled with a top-down traversal of the AST. A type derivation of a program would see first the binding and afterwards the usage of a variable, so that it knows the type of a variable before it may occure in the program.

We propose *bottom-up typechecking* of a program, that uses *binding requirements* instead of a type context. The binding requirements model the same information as the type context, but they can be built with a bottom-up traversal. A binding requirement maps a variable to the type we want that variable to be bound. We use *type variables* as placeholders for actual types, when we can not determine the type of a program fragment directly, e.g. when type checking a variable. We use *constraints* on those type variables, when we get further knowledge of the actual type, e.g. when we find the declaration of a variable, we constrain the required type to be equal to the actual type of the variable. We use *unification* to solve the generated constraints, to guarantee that the type variables can either be unified or to detect type errors for the program.

1.2 Contributions

In this thesis, we make the following contributions:

- We present an approach to eliminate the type context from a type system for the Simply Typed Lambda Calculus. This approach allows us to typecheck each subexpressions of a program in isolation from their corresponding enclosing expression.
- We introduce *binding requirements* as the opposite of a type context. The binding requirements express the type to which a variable must be bound instead of making a proposition of the type to which a variable is actually bound. We use type variables as placeholders if the actual type of an expression can not directly be deducted in the current expression and constraints on those type variables to unify them and obtain their actual type.
- We propose *intermediate unification*, as the process of using unification after each rule application to solve the constraints generated so far.
- We adapt this approach to PCF, to evaluate the applicability to a sligthly more expressive language. We use this typing relation to show a type derivation for a program with intermediate unification.
- We generalize the approach to be applicable to Java. We introduce new constraints to support the features of the language and we extend the unification for rules that solve these new constraint types. We add a new set for class information to the signature of the typing relation, which contains all information belonging to the public interface of a class.

1.3 Structure

The thesis is structure as follows. In chapter 2 we introduce preliminaries to the contributions. We develop in chapter 3 a bottom-up type system for the Simply Typed Lambda Calculus, starting with a naive approach of removing the context from the type rules introduces in chapter 2 and proceeding with the fixing of the problems that arose from this. We adapt the approach from chapter 3 to the language PCF in chapter 4 und use it to demonstrate the usage of the typing relation and the constrain solving. This is generalized to a subset of Java in Chapter 5. We discuss related work in chapter 6 and conclude the thesis in chapter 7.

Chapter 2

Preliminaries

2.1 Spoofax/IMP

The Spoofax language workbench [KV10] is an Eclipse plugin which can be used to design programming languages. It integrates the syntax definition formalism (SDF) [HHKR89] e.g. used for providing basic editor support based on the grammar such as syntax highlighting and code folding and the program transformation language Stratego. Stratego/XT [Vis01] consists of the transformation language Stratego and the XT toolset used not exclusively for parsing and pretty printing.
We use the following constructs of the Stratego language.

Transformation Rules can be used to rewrite the structure of the AST, for example using term replacements.

Transformation Strategies define the traversal order of the AST and can be used to combine transformation rules.

We use Stratego to give an implementation of the bottom-up typechecker for PCF and Java. We use the predefined *bottomup* strategy in these implementations. This strategy applies the rule we pass to it first to the leafs of the AST.

2.2 Down-up type checking

Type checking is often done by applying syntax oriented derivation rules to the abstract syntax tree (AST) of the program to be checked starting at the root node. In the following we present a type system for the Simply Typed Lambda Calculus[Pie02] with natural numbers and addition.

2.2.1 Syntax

The syntax of the Simply Type Lambda Calculus consists of expressions and types. An expression is either a variable, an abstraction or an application as well as natural numbers and addition. A type is either the type $\mathcal{N}$for natural numbers or a function type from one type to another type.

$$
\begin{aligned}
\langle \mathit{Expr} \rangle &::= \langle \mathit{Var} \rangle \mid \lambda\ \langle \mathit{Var} \rangle{:}\langle \mathit{Type} \rangle.\ \langle \mathit{Expr} \rangle \mid \langle \mathit{Expr} \rangle\ \langle \mathit{Expr} \rangle \\
&\quad\mid\ \langle \mathit{Num} \rangle \mid \langle \mathit{Expr} \rangle + \langle \mathit{Expr} \rangle
\end{aligned}
$$

$$\langle \mathit{Type} \rangle ::= \mathcal{N}\ \mid \langle \mathit{Type} \rangle \rightarrow \langle \mathit{Type} \rangle$$

$$\langle \mathit{Num} \rangle ::= 0 \mid 1 \mid 2 \mid \ldots$$

Where an element created with the *Var* production is a character string starting with a lower case letter and possibly ending with a natural number.

Example 2.1. a, x, x0, camelCase0 are productions from Var.

x is a variable.

$\lambda\ x$**:**t**.** e is stating that the variable x is bound to the type t in the enclosed expression e.

$e_1\ e_2$ is stating that the expression e_2 is applied as an argument to the expression e_2. The expression e_1 must be a lambda abstraction to work properly.

2.2.2 The Typing Relation

We now define a typing relation that associates types to expressions (e:T). In order to assign a type to an expression with a variable we need to keep track of the types to which those variables are bound. For this we introduce a typing context Γ which stores variable names and their corresponding types.

Definition 2.2. Γ ::= ∅ | Γ,⟨Var⟩:⟨Type⟩

A context is either empty or a list of key-value pairs where the variable is the key and the type is the value. To lookup the type of a variable in a context Γ we write $\Gamma(x) = T$ stating that the type of x in Γ is T.

$$\text{T-Var}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

$$\text{T-Abs}\ \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda.x : T_1.e : T_1 \rightarrow T_2}$$

$$\text{T-App}\ \frac{\Gamma \vdash e_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}$$

$$\text{T-Num}\ \frac{n \text{ is a number}}{\Gamma \vdash n : \mathcal{N}}$$

$$\text{T-Plus}\ \frac{\Gamma \vdash e_1 : \mathcal{N} \qquad \Gamma \vdash e_2 : \mathcal{N}}{\Gamma \vdash e_1 + e_2 : \mathcal{N}}$$

The type rules shown above are syntax oriented. A syntax oriented type rule is only applicable for one syntactic construct, we achieve this by matching for a syntactic construct in the conclusion of the type rules.
We give a brief overview of the five rules and we will continue with some short examples of the usage.

T-Var

In order to check the type of a variable we must lookup the variable in the typing context Γ. Note that this lookup will fail if the variable is not bound in Γ.

T-Abs

The type of an abstraction is a function type with the type of the argument as the parameter type and the type of the body as the return type. The argument type is explicitly denoted within the syntax. Since the denoted variable of an abstraction is bound in its body, the body will be checked within a context extended by this binding.

T-App

An application expression is valid if the receiver (e1) has a function type and the type of the provided argument (e2) matches the argument type of the function type. The resulting type is then the return type of the function type.

T-Num

The type of a number is $\mathcal{N}$.

T-Plus

Since we only can add numbers, the subexpressions of an addition expression must be both of type $\mathcal{N}$. And the addition of two numbers produce another number, so the type of an addition expression is as well $\mathcal{N}$.

Now that we have seen the typing relation we can start to use it to typecheck some programs. We start with the pure additive program $p_1 = 1+(2+3)$. The first node of the AST of this program is an addition expression. We have just one rule whose conclusion matches and this rule is the rule T-Plus. This rule has two premises, so in order to say that p_1 is of type $\mathcal{N}$we need to check that the subexpressions 1 and $2 + 3$ are of type $\mathcal{N}$. The first subexpression is a natural number and we have just one rule that matches, namely T-Num. This rule does not have a premise and we directly know that 1 has type $\mathcal{N}$. The second subexpression is another addition expression and we need to apply again the rule T-Plus with its two premises and we need to check that 2 and 3 are of type $\mathcal{N}$. Both 2 and 3 are natural numbers and we need to apply rule T-Num to both to know that both expressions have type $\mathcal{N}$. We have checked all premises and we can conclude that p_1 has type $\mathcal{N}$.

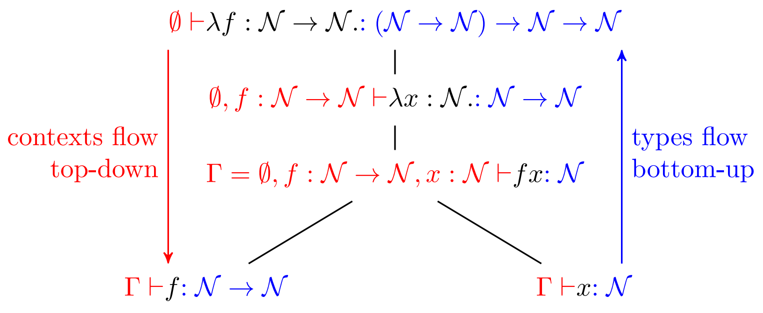
$$\text{T-Plus}\ \dfrac{\text{T-Num}\ \dfrac{}{\emptyset \vdash 1 : \mathcal{N}} \qquad \text{T-Plus}\ \dfrac{\text{T-Num}\ \dfrac{}{\emptyset \vdash 2 : \mathcal{N}} \qquad \text{T-Num}\ \dfrac{}{\emptyset \vdash 3 : \mathcal{N}}}{\emptyset \vdash 2 + 3 : \mathcal{N}}}{\emptyset \vdash 1 + (2 + 3) : \mathcal{N}}$$

In the typechecking process for program p_1 we have seen how typerules are applied to an abstract syntax tree. What we have not seen so far is how the context works. We have startet to check p_1 within the empty context ($\emptyset$) and while we have checked all premises the context has remained unchanged.

We will now typecheck program $p_2 = \lambda f : \mathcal{N} \to \mathcal{N}.\lambda x : \mathcal{N}.(fx)$ and show how the context is build. Lets first describe what the program does. It is a lambda abstraction that takes a parameter f of function type $\mathcal{N}\to\mathcal{N}$and a x of type $\mathcal{N}$as arguments and then applies x to f. So we start to check the program p_2 within the empty context and the first node in the AST is an abstraction. There is just one rule that matches namely T-Abs and we know that the resulting type is some function type and that the first part of the function type must be the denoted type of the parameter f which is $\mathcal{N}\rightarrow\mathcal{N}$. The T-Abs rule has one premise which we have to check and since the variable f is bound in the subexpression of the abstraction we have to regard this fact in the context. We check the subexpression $\lambda x : \mathcal{N}.fx$ in the extended context $\emptyset, f : \mathcal{N} \rightarrow \mathcal{N}$. There is an abstraction and we apply rule T-Abs again. In this abstraction we bind x to $\mathcal{N}$and we must take this into account when we check the subexpression fx, so we extend the context again to $\emptyset, f : \mathcal{N} \rightarrow \mathcal{N}, x : \mathcal{N}$ and we name it Γ. The expression fx is an application and checked within the context Γ. The T-App rule has two premises. First we need to check that f has a function type and second we need to check that x matches the argument part of that function type. So we check f in the context Γ and since f is a variable we use the T-Var rule to lookup the type of f in the context and we know that f has type $\mathcal{N} \rightarrow \mathcal{N}$ because $\Gamma(f) = \mathcal{N} \rightarrow \mathcal{N}$. We check x in the context Γ next and again x is a variable and we use rule T-Var to lookup x in the context and we know that x hast type $\mathcal{N}$ because $\Gamma(x) = \mathcal{N}$. We have checked all premises and must now deduce the type of the original program p_2. We have shown that f has type $\mathcal{N} \rightarrow \mathcal{N}$ which is indeed a function type and that x has type $\mathcal{N}$ which indeed matches the argument part of the function type of f. The resulting type of the application from x to f is then $\mathcal{N}$ and the resulting type of the abstraction which bound x to $\mathcal{N}$ is then $\mathcal{N} \rightarrow \mathcal{N}$ because we checked that the body of the abstraction has type $\mathcal{N}$. We finally conclude that p_2 has type $(\mathcal{N} \rightarrow \mathcal{N}) \rightarrow \mathcal{N} \rightarrow \mathcal{N}$, because we checked that the body of p_2 has type $\mathcal{N} \rightarrow \mathcal{N}$.

$$\text{T-Abs}\ \dfrac{\text{T-Abs}\ \dfrac{\text{T-App}\ \dfrac{\text{T-Var}\ \dfrac{\Gamma(f) = \mathcal{N} \rightarrow \mathcal{N}}{\Gamma \vdash f : \mathcal{N} \rightarrow \mathcal{N}} \qquad \text{T-Var}\ \dfrac{\Gamma(x) = \mathcal{N}}{\Gamma \vdash x : \mathcal{N}}}{\Gamma = \emptyset, f : \mathcal{N} \rightarrow \mathcal{N}, x : \mathcal{N} \vdash fx : \mathcal{N}}}{\emptyset, f : \mathcal{N} \rightarrow \mathcal{N} \vdash \lambda x : \mathcal{N}.fx : \mathcal{N} \rightarrow \mathcal{N}}}{\emptyset \vdash \lambda f : \mathcal{N} \rightarrow \mathcal{N}.\lambda x : \mathcal{N}.(fx) : (\mathcal{N} \rightarrow \mathcal{N}) \rightarrow \mathcal{N} \rightarrow \mathcal{N}}$$

While we have checked program p_2, we have seen two different information flows through the AST. First we have build up the context while the type-checking progresses from the root to the leaves of the AST and second we have deduced the type of p_2 the other way around starting with the type of the leaves and ending with the type of the whole program. The context information flows down and the types flow up the AST.


$\emptyset \vdash \lambda f : \mathcal{N} \rightarrow \mathcal{N}.: (\mathcal{N} \rightarrow \mathcal{N}) \rightarrow \mathcal{N} \rightarrow \mathcal{N}$
$\emptyset, f : \mathcal{N} \rightarrow \mathcal{N} \vdash \lambda x : \mathcal{N}.: \mathcal{N} \rightarrow \mathcal{N}$
contexts flow top-down
$\Gamma = \emptyset, f : \mathcal{N} \rightarrow \mathcal{N}, x : \mathcal{N} \vdash fx: \mathcal{N}$
types flow bottom-up
$\Gamma \vdash f: \mathcal{N} \rightarrow \mathcal{N}$
$\Gamma \vdash x: \mathcal{N}$

Chapter 3

Bottom-up Typechecking

In the previous chapter we have seen a type system for the Simply Typed Lambda Calculus (STLC). We have noticed that there are two different oriented information flows while typechecking a program. The context information was build up top-down from the root to the leaves and the deduction of the type was bottom-up from the leaves to the root. To typecheck a program, we had to traverse the AST two times in different directions. In this chapter, we want to get rid of the top-down traversal that builds up the context information to ensure that all information flow is bottom-up in the typechecking process and we call this bottom-up typechecking.

3.1 Eliminating Context

We have used the context to keep track of the variable bindings of a program, but as the context information has flown top-down we must get rid of it. If we just remove the context from the typerules we have seen so far issues will arise. So let us remove them and see what will happen.

$$\text{T-Var}\ \frac{}{x :?}$$

$$\text{T-Abs}\ \frac{e : T_2}{\lambda.x : T_1.e : T_1 \to T_2}$$

$$\text{T-App}\ \frac{e1 : T_1 \to T_2 \qquad e2 : T_1}{e_1\ e_2 : T_2}$$

$$\text{T-Num}\ \frac{n \text{ is a number}}{n : \mathcal{N}}$$

$$\text{T-Plus}\ \frac{e_1 : \mathcal{N} \qquad e_2 : \mathcal{N}}{e_1 + e_2 : \mathcal{N}}$$

The first problem can be seen directly by looking at the T-Var rule. When checking a variable we can not say what type the variable has, because there is no context to look it up. Second, when we introduce a new variable binding in an abstraction we loose that the variable is bound in the subexpression. The third issue arises when we previously duplicated the context in nodes with two or more premises involving contexts like application. To fix the first problem, we have to introduce type variables as placeholders for our actual types. When we now check a variable x we generate a fresh type variable α and assign this type variable as the type of the variable. We must require that x is somehow bound later (above the current node) in the AST and we remember that we have used α as the type for x. We lost the context and to compensate for the information loss we introduce contextual requirements. The new typing relation associates expressions with types and requirements ($e : T|reqs$).

$$\text{T-Var}\,\frac{\alpha\ fresh}{x : \alpha|\{x : \alpha\}}$$

The second problem can be fixed similar. When we typecheck an abstraction, each time a variable is used in the subexpression we generate a requirement with the T-Var rule shown above. We must ensure that this requirement equals the denoted type. We access the requirements like we looked up in the context. For requirements $reqs$ we use $reqs(x)$ to look up the variable x in $reqs$.

$$\text{T-Abs}\,\frac{e : T_2|reqs \qquad reqs(x) = T_1}{\lambda.x : T_1.e : T_1 \to T_2|reqs - (x : reqs(x))}$$

Note that T_1 is the denoted type of x in the abstraction and thus $reqs(x) = T_1$ is a condition we need to fullfill.

What is left to fix is the third issue. In the previous type system with context we have duplicated the context when checking an application, because both subexpressions e_1 and e_2 obtained the bindings that were valid in the application. Now this information is lost because we check e_1 and e_2 independend from each other and contain two requirement sets $reqs_1$ and $reqs_2$. We must build the union of these requirement sets and ensure type equality for all variables that are required in both sets to reflect the previous behaviour. Since we ensure the type equality is suffices to keep only one requirement per variable. We use the operator $\uplus$ for the union of requirements that only keeps one requirement per variable. This operator will be defined in detail later. This step must be repeated for rule T-Plus, because it also has two premises with contexts.

$$\text{T-App}\,\frac{\begin{array}{c}\forall x \in dom(reqs_1).\ x \in dom(reqs_2) \implies reqs_1(x) = reqs_2(x)\\ e_1 : T_1 \to T_2|reqs_1 \qquad e_2 : T_1|reqs_2\end{array}}{e_1\ e_2 : T_2|reqs_1 \uplus reqs_2}$$

$$\text{T-Plus}\,\frac{\begin{array}{c}\forall x \in dom(reqs_1).\ x \in dom(reqs_2) \implies reqs_1(x) = reqs_2(x)\\ e_1 : \mathcal{N}|reqs_1 \qquad e_2 : \mathcal{N}|reqs_2\end{array}}{e_1 + e_2 : \mathcal{N}|reqs_1 \uplus reqs_2}$$

We changed all typerules except rule T-Num which remains as it was, because there is no interaction with the context in this rule. All other rules T-Var, T-Abs, T-App and T-Plus have been modified to compensate the loss of context and we altered the information flow for this context requirements to bottom-up.

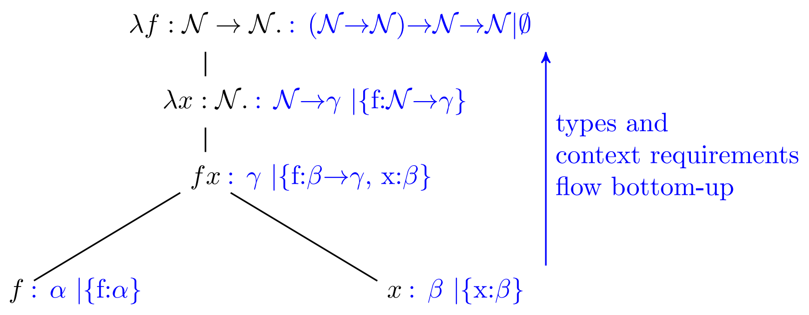
We recall the example program $p_2 = \lambda f : \mathcal{N} \to \mathcal{N}.(\lambda x : \mathcal{N}.fx)$ from the previous down-up typechecker and we typecheck it again with the changed typerules. The first things to check are the expressions f and x. We use for both

expressions the rule T-Var and we generate two fresh typevariables α and β. We assign them as the types for f and x and we also require that f will later be bound to α and that x will later be bound to β.
Then we proceed to the application of x to f. We know from rule T-App that the first expression must have a function type and that the argument type of the function must be the type of the second expression. What we do not know is the return type, so we generate a fresh typevariable γ as the return type. The type of the first expression is α and the type of the second expression is β and we have to ensure that $\alpha = \beta \to \gamma$. The required variables of the two subexpressions are distinct, so we do not need to check for equalitiy at this point and merge the both requirement sets $\{f : \alpha\}$ and $\{x : \beta\}$ together to $\{f : \beta \to \gamma, x : \beta\}$. Note that we have replaced α with $\beta \to \gamma$ in the requirement set to respect our new knowledge of the type. What is left to check are both abstractions and we start with $\lambda x : \mathcal{N}.f\ x$ and we use rule T-Abs. We discover the first variable binding of x to $\mathcal{N}$ and we must ensure that the requirement of x which is β equals $\mathcal{N}$ ($\beta = \mathcal{N}$). Since x was bound in this node, we must remove all binding requirements of x and we obtain the new requirement set $\{f : \mathcal{N} \to \gamma\}$.
The next abstraction node is the whole program p_2. We use rule T-Abs again and we ensure that $\mathcal{N} \to \gamma = \mathcal{N} \to \mathcal{N}$. Since f is bound in this abstraction we remove all requirements of f and we end with the empty requirement set $\emptyset$ and the type $(\mathcal{N} \to \mathcal{N}) \to \mathcal{N} \to \mathcal{N}$ for program p_2.
We observe that we have no more binding requirements to fullfill, because we ended the typechecking with the empty requirement set. This means that we do not have any unbound variables in the program. Every time we used the rule T-Var, we have generated a new requirement and every time a variable was bound, we removed the corresponding requirements with rule T-Abs.
We show the type derivation for p_2 below. The first derivation is for the application and the second derivation is for the two abstractions and uses the result from the first derivation as a premise.

$$\text{T-App}\;\dfrac{\text{T-Var}\;\dfrac{}{f:\alpha|\{f:\alpha\}} \qquad \text{T-Var}\;\dfrac{}{x:\beta|\{x:\beta\}} \qquad \alpha=\beta\to\gamma}{f\ x:\gamma|\{f:\beta\to\gamma, x:\beta\}}$$

$$\text{T-Abs}\;\dfrac{\text{T-Abs}\;\dfrac{f\ x:\gamma|\{f:\beta\to\gamma, x:\beta\} \qquad \beta=\mathcal{N}}{\lambda x:\mathcal{N}.f\ x:\mathcal{N}\to\gamma|\{f:\mathcal{N}\to\gamma\}} \qquad \mathcal{N}\to\gamma=\mathcal{N}\to\mathcal{N}}{p_2:(\mathcal{N}\to\mathcal{N})\to\mathcal{N}\to\mathcal{N}|\emptyset}$$

All the changes to the typerules were made to eliminate the top-down information flow of the context. We examine the information flow of program p_2 by reference to the AST of p_2 and we see that all information flows bottom-up.



$\lambda f:\mathcal{N}\to\mathcal{N}.:$ $(\mathcal{N}\to\mathcal{N})\to\mathcal{N}\to\mathcal{N}|\emptyset$
$\lambda x:\mathcal{N}.:$ $\mathcal{N}\to\gamma$ $|\{$f:$\mathcal{N}\to\gamma\}$
$fx:$ γ $|\{$f:$\beta\to\gamma$, x:$\beta\}$
$f:$ α $|\{$f:$\alpha\}$
$x:$ β $|\{$x:$\beta\}$
types and
context requirements
flow bottom-up

When we typechecked program p_2 we have noticed that in the application node we introduced the type variable γ as the type of the application of x to f and we changed the requirement of f from α to $\beta \to \gamma$. This change was made to respect the new knowledge of the types that we obtained from the T-App rule. So we must be able to alter requirements, to interchange type variables and to replace type variables with actual types.

3.2 Constraints

When we typechecked program $p_2 = \lambda f : \mathcal{N} \to \mathcal{N}.(\lambda x : \mathcal{N}.fx)$ we have done some changes to the typevariables in the application case that were not explicit statet in the T-App rule. We introduced a fresh type variable and used it as the type of the application. We also changed the required type of f from α to $\beta \to \gamma$, because we knew that α which was the type of the first subexpression of the application needed to be of some function type. We also have seen in the abstraction nodes that we used the equality conditions of the T-Abs rule to do the same e.g. when we had to ensure that the type variable β equals type $\mathcal{N}$. What we have actually done was to constrain the more general type variables α and β to some specialized types. In order to be more explicit in typerules we introduce equality constraints for types and we say for two types t_1 and t_2 that $t_1 \doteq t_2$ is the equality constraint for those types. This constraints can be used to unify type variables.

3.3 Merging of requirements

Now that we have introduced constraints, we have to specify the merging of two requirement sets. There are two things to do when we merge requirement sets. We need to generate constraints for those variables that are required to be bound in both sets, because a variable that is used multiple times in the same scope can not be bound to different types. We generate a constraint for all those intersecting requirements with the $\cap_c$ operator.

Definition 3.1 (Intersection constraints for requirements).
For each variable x that is bound to a type t in the first requirement set, if x is bound in the second requirement set t is constrained to be equal to the required type of x in the second set.

$$reqs_1 \cap_c reqs_2 := \{t \doteq reqs_2(x) \mid (x : t) \in reqs_1 \wedge x \in dom(reqs_2)\}$$

We also need to generate a new requirement set which contains for all variables of both sets only one occurence of the same variable, because we have already generated constraints for the variables that occured more than once which can be seen as the union of two requirement sets and we use the $\uplus$ operator for it.

Definition 3.2 (Union of requirements).
Take all requirements from the first set and only those requirements from the second set which are not required in the first set.

$$reqs_1 \uplus reqs_2 := reqs_1 \cup \{x : t \mid (x : t) \in reqs_2 \wedge x \notin dom(reqs_1)\}$$

3.4 Unification

We can build sets of constraints either by adding constraints directly or by the previously introduced intersection operator for requirements ($\cap_c$). The constraints we have are type equality constraints which may contain typevariables. These typevariables must be unified in order to solve the constraint set. Here, an unifier is a substitution for type variables and we introduce a set of rules to compute unifiers for a constraint set.

Definition 3.3 (Type mismatch)**.** For types T_1 and T_2, we say T_1 does not match T_2 ($T_1 \neq_{Prod} T_2$) if T_1 and T_2 were built with a different production.

For example $\mathcal{N} \neq_{Prod} \alpha \to \beta$, because $\mathcal{N}$ and $\to$ are different. We need the following six rules which associate an unifier to a constraint set.

$$\text{Trivial}\ \frac{\{t \doteq t\} \cup C|\sigma}{C|\sigma}$$

$$\text{Orient}\ \frac{\{t \doteq x\} \cup C|\sigma}{\{x \doteq t\} \cup C|\sigma}\ t \notin Typevar$$

$$\text{Decompose}\ \frac{\{s_1 \to s_2 \doteq t_1 \to t_2\} \cup C|\sigma}{\{s_1 \doteq t_1, s_2 \doteq t_2\} \cup C|\sigma}$$

$$\text{Clash}\ \frac{\{s \doteq t\} \cup C|\sigma}{\top}\ s \neq_{Prod} t$$

$$\text{Occur Check}\ \frac{\{x \doteq t\} \cup C|\sigma}{\top}\ x \text{ occurs in } t$$

$$\text{Variable Elimination}\ \frac{\{x \doteq t\} \cup C|\sigma \qquad x \text{ occurs not in } t}{C\{x/t\}|\sigma\{x/t\} \cup \{x/t\}}$$

Trivial
If we have a constraint $T \doteq T$ we can omit it, because it is true.

Orient
If we have a constraint $T \doteq x$ where T is not a type variable, we swap T and x. So all type variables will be oriented left and because of this orientation we have less work in the Variable Elimination rule.

Decompose
If we have a constraint $S_1 \to S_2 \doteq T_1 \to T_2$ we need to check if $S_1 \doteq T_1$ and $S_2 \doteq T_2$ are solvable.

Clash
If we have a constraint $T_1 \doteq T_2$ and the productions from which T_1 and T_2 emerged from, do not match we have found an unsolvable constraint. This is an error in the constraint set and we can break the unification at this point. We have currently the constant $\mathcal{N}$ and for types T and T' the binary $T \to T'$ productions. We will later introduce new types which will be handled similar in the clash rule.

Occur Check

If we have a constraint $x \doteq T$ and x occurs in T we have found an unsolvable constraint, because we cannot substitute x with a type that contains x.

Variable Elimination

If we have a constraint $x \doteq T$, a partial unifier σ, and x occurs not in T we substitute x with T in the remaining constraint set and σ and we add the substitution to σ.

The Robinson Algorithm [Rob65] is a particular application strategy for those rules. It tries to apply the rules in the following order. Trivial, Decompose, Clash, Orient, Occur Check, Variable Elimination. We use the Robinson Algorithm for unification and constraint solving.

3.5 Typing Relation for STLC

We have introduced all techniques needed to fix the issues we had with the first approach of the context elimination and we give a typing relation for STLC that associates expressions with types, requirements and constraints ($e : T|reqs|cs$).

$$\text{T-Var}\ \frac{\alpha\ fresh}{x : \alpha|\{x : \alpha\}|\emptyset}$$

$$\text{T-Abs}\ \frac{e : T_2|reqs|cs}{\lambda x : T_1.e : T_1 \to T_2|reqs - (x : reqs(x))|\{reqs(x) \doteq T_1\} \cup cs}$$

$$\text{T-App}\ \frac{e_1 : T_1|reqs_1|cs_1 \qquad e_2 : T_2|reqs_2|cs_2 \qquad \alpha\ fresh}{e_1\ e_2 : \alpha|reqs_1 \uplus reqs_2|\{T_1 \doteq T_2 \to \alpha\} \cup cs_1 \cup cs_2 \cup (reqs_1 \cap_c reqs_2)}$$

$$\text{T-Num}\ \frac{n \text{ is a number}}{n : \mathcal{N}|\emptyset|\emptyset}$$

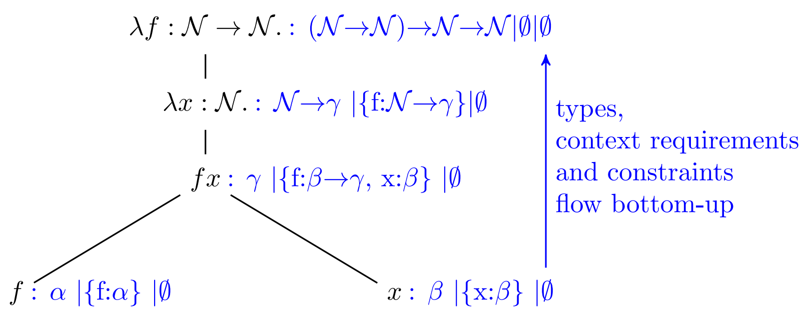
$$\text{T-Plus}\ \frac{e_1 : T_1|reqs_1|cs_1 \qquad e_2 : T_2|reqs_2|cs_2 \qquad cs_{new} = \{T_1 \doteq \mathcal{N}, T_2 \doteq \mathcal{N}\}}{e_1 + e_2 : \mathcal{N}|reqs_1 \uplus reqs_2|cs_1 \cup cs_2 \cup (reqs_1 \cap_c reqs_2)}$$

The only change in the T-Var rule is, that we now produce an additional empty constraint set ($\emptyset$) to respect the new signature of the relation. In the T-Abs rule we remove the requirements for the newly bound variable x and we constrain the required type of x to the actual bound type. The T-App rule has changed the most. We now explicitly generate a fresh type variable and constrain the type of the first subexpression to be a function type with the type of the right subexpression as the argument part and the new type variable as the return part. We also state explicitly that the variables required in both subexpressions need to be constrained equal with the intersection operator $\cap_c$. The T-Plus rule is also affected by the second part of changes of the T-App rule and we moved the check whether the subexpressions have type $\mathcal{N}$ into the constraints, because a type can also be a type variable which first needs to be unified. The changes in the T-Num rule are like in the T-Var rule just to respect the signature of the relation.

For STLC we can use unification already in intermediate steps, because all constraints that are introduced are either solvable with the knowledge we have when we generate them or are unsolvable even with future knowledge.

We check program $p_2 = \lambda f : \mathcal{N} \to \mathcal{N}.(\lambda x : \mathcal{N}.fx)$ again using the new relation with intermediate unification and we analyse the information flow through the program.

We begin the type checking with the variable nodes f and x. We use for both the T-Var rule and generate fresh typevariables. We generate α for f and β for x and we require the variables to be of this types. In the application node we use rule T-App and we generate a fresh typevariable γ as the type of the application and we constrain α to $\beta \to \gamma$. We merge the requirement sets $\{f : \alpha\}$ and $\{x : \beta\}$ together using $\uplus$ and we get the new requirement set $\{f : \alpha, x : \beta\}$. We intersect the requirement sets using $\cap_c$ and since the variables required in the sets are distinct we must not generate a new constraint. We unify the constraint set $\alpha \doteq \beta \to \gamma$ and we get the unifier $\{\alpha/\beta \to \gamma\}$. We apply the unifier to the new requirement set and get $\{f : \beta \to \gamma, x : \beta\}$. The next node above is an abstraction which binds the variable x to the type $\mathcal{N}$. We remove the requirements of x and get $\{f : \beta \to \gamma\}$ as our new requirement set. We constrain the requirement of x which was β to the denoted type $\mathcal{N}$and we solve the constraint set $\{\beta \doteq \mathcal{N}\}$ to get the unifier $\{\beta/\mathcal{N}\}$. We apply the unifier to the requirement set and get $\{f : \mathcal{N} \to \gamma\}$ and we have $\mathcal{N} \to \gamma$ as the type for the abstraction. The final node is again an abstraction which binds f to $\mathcal{N} \to \mathcal{N}$. We remove f from the requirements and we get the empty requirement set $\emptyset$. We constrain the previous requirement of f to $\mathcal{N} \to \mathcal{N}$ and we unify the constraint set $\{\mathcal{N} \to \gamma \doteq \mathcal{N} \to \mathcal{N}\}$ and we obtain the unifier $\{\gamma/\mathcal{N}\}$ with the use of the Decompose rule. We have checked program p_2 to be of type $(\mathcal{N} \to \mathcal{N}) \to \mathcal{N} \to \mathcal{N}$ and since we have ended with an empty requirement set we have no unbound variables and since unification succeeded in all cases we had no unsolvable constraints.



$\lambda f : \mathcal{N} \to \mathcal{N}.:$ $(\mathcal{N}\to\mathcal{N})\to\mathcal{N}\to\mathcal{N}|\emptyset|\emptyset$
$\lambda x : \mathcal{N}.:$ $\mathcal{N}\to\gamma$ $|\{f:\mathcal{N}\to\gamma\}|\emptyset$
$fx:$ γ $|\{f:\beta\to\gamma,\ x:\beta\}$ $|\emptyset$
$f:$ α $|\{f:\alpha\}$ $|\emptyset$
$x:$ β $|\{x:\beta\}$ $|\emptyset$
types, context requirements and constraints flow bottom-up

Note that instead of checking the type to be equal to some type obtained from the outside of the typing relation we have infered the type of the program out of itself. This type inference works not only for this example program, but for every program that can be written in STLC. If the program is not well-typed, the unification would fail and we would know that the program is illtyped.

3.6 Properties of the typing relation

We have presented a bottom-up typing relation for STLC. We want to adress the three following properties of this relation.

- Intermediate unification
- Type inference
- Parallelizability

We can use unification at intermediate steps in a derivation to solve the constraints generated so far.
We can solve the constraints for each rule application in isolation. When doing so, we need to substitute the resulting unifier from the constraint solving to the binding requirements and the type we produce as a result. The requirements and the type could contain type variables which were unified and since we do not pass the unifier to the next rule to apply we need to substitute all those type variables that have already been unified. For example when we type checked an expression e and resulted in a type T, requirements $reqs$ and constraints cs, we can use unification to solve the constraints cs which will either result in an unifiert σ or will fail. If it fails we know that the program is illtyped. If it succeeds we must substitute σ into T and $reqs$, because both could contain type variables contained in σ. We would then pass the substituted type T and the substituted requirements $reqs$ and since all constraints have been solved an empty constraint set to the following rule application. This would look as follows in a rule application.

$$\text{Type-Rule}\ \frac{e : T \mid reqs \mid cs}{e : \sigma(T) \mid \sigma(reqs) \mid \emptyset}\ \sigma = \{\text{some substitutions}\}$$

We can infer the type of a program out of itself. There is no need to provide external knowledge to the type rules. We could also omit the explicit type annotation for the parameters of abstractions and infer them.

Implementations of the type relation could be parallelized easily. The type rules do not depend on a global traversion of the abstract syntax tree of the program and we can typecheck each subtree independent from each other.
In an implementation, we could spawn one processing entity per rule application. Each process entity could start running, when the entities spawned for the premises of the type rules have completed. So the first process entities that can start running, are the entities spawned for rules without a premise. We use this rules without premises for the leafs of the AST of a program.

Chapter 4

PCF

We have used STLC as a language so far, but as this language is very limited we extend it to Programming Computable Functions (PCF)[Mit96]. On top of that we add boolean expressions to have more than just one base type in our examples.

4.1 Syntax

All the expressions from STLC are available in PCF. So we still have variables, abstractions and applications and we also keep numbers and addition except numbers can now also be negative. New syntactic constructs are the boolean constants *True* and *False* associated with the connectives for negation ($\neg$) and conjunction (&). We also have the greater ($>$) operator and conditional statements (if-then-else). The last two new constructs are a fixpoint combinator and let bindings. The type for numbers ($\mathcal{N}$) and function types ($\rightarrow$) remain and we add a new bool type ($\mathcal{B}$).

$$\langle \mathit{Num} \rangle ::= \ldots \mid \text{-2} \mid \text{-1} \mid 0 \mid 1 \mid 2 \mid \ldots$$

$$\begin{array}{lcl}
\langle \mathit{expr} \rangle & ::= & \langle \mathit{ID} \rangle \\
& \mid & \lambda\langle \mathit{ID} \rangle \texttt{ : } \langle \mathit{Type} \rangle\texttt{. } \langle \mathit{expr} \rangle \\
& \mid & \langle \mathit{expr} \rangle\ \langle \mathit{expr} \rangle \\
& \mid & \langle \mathit{Num} \rangle \\
& \mid & \langle \mathit{expr} \rangle \texttt{ + } \langle \mathit{expr} \rangle \\
& \mid & \texttt{True} \\
& \mid & \texttt{False} \\
& \mid & \mathit{neg}\langle \mathit{expr} \rangle \\
& \mid & \langle \mathit{expr} \rangle \texttt{ \& } \langle \mathit{expr} \rangle \\
& \mid & \langle \mathit{expr} \rangle \texttt{ > } \langle \mathit{expr} \rangle \\
& \mid & \texttt{if } \langle \mathit{expr} \rangle \texttt{ then } \langle \mathit{expr} \rangle \texttt{ else } \langle \mathit{expr} \rangle \\
& \mid & \text{fix } \langle \mathit{ID} \rangle \texttt{ : } \langle \mathit{Type} \rangle\texttt{. } \langle \mathit{expr} \rangle \\
& \mid & \langle \mathit{ID} \rangle \texttt{ : } \langle \mathit{Type} \rangle \texttt{ = } \langle \mathit{expr} \rangle\texttt{; } \langle \mathit{expr} \rangle
\end{array}$$

$$\begin{array}{lcl}
\langle \mathit{Type} \rangle & ::= & \mathcal{N} \\
& \mid & \mathcal{B} \\
& \mid & \langle \mathit{Type} \rangle \rightarrow \langle \mathit{Type} \rangle
\end{array}$$

Instead of presenting a detailed reduction relation we will show the semantics of the syntax with a few example programs to give a brief overview of the behavior of the new language. We will show the reduction of those programs and will state to what they will normalize. Reduction is the stepwise process of evaluating a program and normalization is the repetition of that, until the program can not be reduced anymore.

We start with a program of a pure boolean expression (True & !False) which would normalize to True, because !False reduces to True and True & True then reduces to True. In the next program we look at the conditional statement. The program (if 1 > 2 then True else False) normalizes to False because 1 is not greater as 2 and therefore the second branch is evaluated. Now that we have more than one base type we can easily write programs that would not behave well. If we change one of the previous expression to (2 & !False) the program can not be normalized to a value because we can not further reduce 2 & True, because 2 is a number and True is a boolean constant. So we must be sure that the typing relation respects that. This problem occurs not only in the conjunction connective. We now look at the fixpoint combinator. We provide a program that sums up all numbers below a given number towards zero. We call that program fixsum to reuse it in the next example. fixsum = fix f:$\mathcal{N}$→$\mathcal{N}$. (λ x:$\mathcal{N}$. if n > 0 then 0 else n+(f (n+(-1)))))
In the last program we will look at the let binding construct. We bind the previously defined fixsum program to a variable *sum* and then use the *sum* variable in the body to call the fixsum program with the value 5.
sum : $\mathcal{N}$→ $\mathcal{N}$= fixsum; sum 5.
This program would normalize to the number 15. We can use the fixpoint combinator to write recursive programs and the let binding construct helps with the reuse of expressions because instead of duplicating an expression we can bind it to a variable and then use the variable multiple times.

4.2 Typing Relation for PCF

Now that we have seen the idea of bottom-up typechecking we define a bottom-up typing relation for PCF. We split the syntax into three categories and discuss the typerules for each category. We split the syntax on the basis of the differences to the syntax we already knew from STLC. The first category contains the syntactic constrcuts that we kept from STLC, which are variables, abstractions, applications, numbers and additions. The second category consists of the boolean constants *True* and *False*, the logical connectives negation and conjunction and the greater operator. The third category is made up of conditional statements, the fixpoint operator and the let binding construct.

For the typerules for the first partition we can reuse the typerules introduced in the previous chapter, because the syntactic constructs are unmodified and therefore we do not have to change the typerules.

The second partition contains the boolean expressions and expressions that are related to the type bool ($\mathcal{B}$). As there are no contextual requirements and constraints for the constants *True* and *False* we can statically assign $\mathcal{B}$ as the type for both expressions.

$$\text{T-True}\ \frac{}{True : \mathcal{B}|\emptyset|\emptyset}$$

$$\text{T-False}\ \frac{}{False : \mathcal{B}|\emptyset|\emptyset}$$

The negation expression has one subexpression which must be of type $\mathcal{B}$, because we can only negate boolean expressions. Like in the T-Plus rule we need to move this into the constraints because of type variables. This holds true for all other expressions with such restrictions. Apart from this there are no new requirements or constraints.

$$\text{T-Not}\ \frac{e : T|reqs|cs}{!e : \mathcal{B}|reqs|cs \cup \{T \doteq \mathcal{B}\}}$$

The conjunction connective has two subexpressions which must be both of type $\mathcal{B}$. And since we have two subexpressions we must merge the requirements and constraints together using the operators $\uplus$ for the union of requirements and $\cap_c$ for the intersection constraints of requirements. For the greater operator which has also two subexpressions we must do the same, except that the subexpressions must be of type $\mathcal{N}$, because we can only compare numbers.

$$\text{T-And}\ \frac{e_1 : T_1|reqs_1|cs_1 \qquad e_2 : T_2|reqs_2|cs_2 \qquad cs_{new} = \{T_1 \doteq \mathcal{B}, T_2 \doteq \mathcal{B}\}}{e_1 \& e_2 : \mathcal{B}|reqs_1 \uplus reqs_2|cs_1 \cup cs_2 \cup cs_{new} \cup (reqs_1 \cap_c reqs_2)}$$

$$\text{T-Gt}\ \frac{e_1 : T_1|reqs_1|cs_1 \qquad e_2 : T_2|reqs_2|cs_2 \qquad cs_{new} = \{T_1 \doteq \mathcal{N}, T_2 \doteq \mathcal{N}\}}{e_1 > e_2 : \mathcal{B}|reqs_1 \uplus reqs_2|cs_1 \cup cs_2 \cup cs_{new} \cup (reqs_1 \cap_c reqs_2)}$$

We have seen all typerules for the second partition and we adapted the methods we used in the type rules for application and addition to other expressions with two subexpressions. Now we look at the third partition and we start with conditional statements. When we want to typecheck conditionals we have three subexpressions from which we must merge the requirements- and constraints sets together and our operators for this are just binary operators. We can exploit the properties of this operators to merge the three sets together. The union of requirement sets is a new requirement set and it removes duplicate variables. So we can use it in an associative way and build the union of the first two requirement sets and use the resulting set to build the union with the third requirement set. The intersection constraints for requirements can not be used in this way, because they generate constraints instead of a new requirement set. What it generates, are equality constraints and as equality is transitive it suffices that we use the operator pairwise. So we generate the intersection constraints for the requirement sets of the first and second subexpression and for the second and third subexpression. We further need to check that the first subexpression of a conditional expression must be of type $\mathcal{B}$ and we must constrain the types of the second and third subexpression to be equal.

$$cs = \{T_1 \doteq \mathcal{B}, T_2 \doteq T_3\} \cup cs_1 \cup cs_2 \cup cs_3 \cup ((reqs_1 \cap_c reqs_2) \cup (reqs_2 \cap_c reqs_3))$$

$$\text{T-Cond}\ \frac{e_1 : T_1|reqs_1|cs_1 \qquad e_2 : T_2|reqs_2|cs_2 \qquad e_3 : T_3|reqs_3|cs_3}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T_2|(reqs_1 \uplus reqs_2) \uplus reqs_3|cs}$$

Now we look at the fixpoint combinator. The fixpoint combinator is similar as the abstraction we already knew except it can be used to write recursive programs. It also introduces a new variable binding which is valid in its body. The difference is that the type subexpression for which we try to find a fixpoint must be the same as the denoted type which stands for the fixpoint.

$$\text{T-Fix}\ \frac{e_1 : T_1|reqs_1|cs_1 \qquad cs_2 = \{T \doteq T_1, reqs_1(x) \doteq T\}}{fix\ x : T.e_1 : T|reqs_1 - (x : reqs_1(x))|cs_1 \cup cs_2}$$

The final sort of expression to look at is the let binding construct. A let binding has two subexpressions. The first subexpression introduces a new variable binding, which is valid in the second subexpression. So we must constrain the type of the first subexpression to be equal to the denoted type of the argument. As the first subexpression initializes the variable binding we must constrain only the requirements of the second subexpression to the denoted type, because we must not initialize a variable with itself. Further we must merge the requirement sets together like we did in the other expressions with two subexpressions and we must remove the requirements of the newly introduced binding like we did with the abstraction and fixpoint combinator.

$$\text{T-Let}\ \frac{\begin{array}{c} cs = \{T \doteq T1, reqs_2(x) \doteq T\} \cup cs_1 \cup cs_2 \cup (reqs_1 \cap_c reqs_2) \\ e_1 : T_1|reqs_1|cs_1 \qquad e_2 : T_2|reqs_2|cs_2 \qquad reqs = reqs_1 \uplus reqs_2 \end{array}}{x : T = e_1; e_2 : T_2|reqs - (x : reqs(x))|cs}$$

4.3 Properties of the typing relation

The properties for the typing relation for STLC hold also for the typing relation for PCF.

We still have intermediate unification. Since the signature of the typing relation is unchanged from the one for STLC, we use the same method for the constraint solving after a rule application.
The typing relation for PCF, like the typing relation which we presented for STLC, can be used for type inference. We do not need to give some external knowledge of the type of a program to the relation in order to infer the type of that program. We could also omit the type denotation in abstractions, fix points and let bindings and infer them.
Implementations of the typing relation can be parallelized as well, using the same approach as in STLC.

4.4 Usage of the typing relation

We now infer the type of the program $p_3 = \lambda x : \mathcal{N}.\ \texttt{if}\ x > 0\ \texttt{then}\ x\ \texttt{else}\ x$ to show the usage of the typing relation with intermediate unification. Program p_3 consists of a conditional expression with an comparison as the condition and variables in the branches enclosed by an abstraction. We begin with the condition which is a comparison and as the greater operator has two subexpressions we first check them. The first subexpression is the variable x and we use rule T-Var. We generate a fresh typevariable α as the type of the expression ($T_{Gt_1} = \alpha$), we require x to be of type α ($reqs_{Gt_1} = \{x : \alpha\}$) and there are no constraints that need to be generated ($cs_{Gt_1} = \emptyset$). The second subexpression is a number and we use rule T-Num to get $T_{Gt_2} = \mathcal{N}$, $reqs_{Gt_2} = \emptyset$ and $cs_{Gt_2} = \emptyset$. We then use rule T-Gt and generate new constraints $cs_{Gt_{new}} = \{\alpha \doteq \mathcal{N}, \mathcal{N} \doteq \mathcal{N}\}$. We merge the requirements together $reqs_{Gt_1} \uplus reqs_{Gt_2} = \{x : \alpha\}$ and we generate the constraints from intersecting requirements $reqs_{Gt_1} \cap_c reqs_{Gt_2} = \emptyset$. Since there are no intersecting requirements we solve the constraint set $cs_{Gt_{new}}$ and get the unifier $\sigma_{Gt} = \{\alpha/\mathcal{N}\}$. We substitute σ_{Gt} to the merged requirements to get $reqs_{Cond} = \{x : \mathcal{N}\}$ and from rule T-Gt we know that the type of the expression $T_{Cond} = \mathcal{B}$. The expression in the then branch is the variable x and we use rule T-Var to get $T_{Then} = \beta$, $reqs_{Then} = \{x : \beta\}$ and $cs_{Then} = \emptyset$. We repeat this for the else branch which is again the variable x and get $T_{Else} = \gamma$, $reqs_{Else} = \{x : \gamma\}$ and $cs_{Else} = \emptyset$. Now we have checked all subexpressions of the conditional expression and we can use rule T-Cond to check the conditional itself. The type of the conditional expression is the type of the then branch which was β. We merge the requirement sets together to get

$$\begin{aligned} reqs_{If} &= (reqs_{Cond} \uplus reqs_{Then}) \uplus reqs_{Else} \\ &= (\{x : \mathcal{N}\} \uplus \{x : \beta\}) \uplus \{x : \gamma\} \\ &= \{x : \mathcal{N}\} \uplus \{x : \gamma\} = \{x : \mathcal{N}\}. \end{aligned}$$

We generate the new constraints $cs_{new} = \{T_{Cond} \doteq \mathcal{B}, T_{Then} \doteq T_{Else}\} = \{\mathcal{B} \doteq \mathcal{B}, \beta \doteq \gamma\}$ and the constraints of the intersecting requirements

$$\begin{aligned} cs_{isect} &= \{(reqs_{Cond} \cap_c reqs_{Then}) \cup (reqs_{Then} \cap_c reqs_{Else})\} \\ &= (\{x : \mathcal{N}\} \cap_c \{x : \beta\}) \cup (\{x : \beta\} \cap_c \{x : \gamma\}) \\ &= \{\mathcal{N} \doteq \beta\} \cup \{\beta \doteq \gamma\} = \{\mathcal{N} \doteq \beta, \beta \doteq \gamma\} \end{aligned}$$

and we build $cs_{If} = cs_{new} \cup cs_{isect} = \{\mathcal{B} \doteq \mathcal{B}, \beta \doteq \gamma, \mathcal{N} \doteq \beta\}$. The next step is to solve cs_{If} to get the unifier $\sigma_{If} = \{\beta/\mathcal{N}, \gamma/\mathcal{N}\}$ and to substitute σ_{If} to β toget $T_{If} = \mathcal{N}$. We finally check the abstraction which binds x to $\mathcal{N}$ with rule T-Abs. We know that the type of the abstraction is $T = \mathcal{N} \to \mathcal{N}$, because the parameter x is bound to $\mathcal{N}$ and the subexpression hast type $T_{If} = \mathcal{N}$. We remove all requirements of x, because it is now actually bound and get $reqs = reqs_{If} - (x : reqs_{If}(x)) = \emptyset$. What is left is to check if the bound variable has been used correctly. Therefore we generate the new constraints $\{reqs_{If}(x) \doteq \mathcal{N}\} = \{\mathcal{N} \doteq \mathcal{N}\}$. We have in fact generated just one constraint which is trivially solved and produces an empty unifier and T contains no type variables so we have unified all type variables generated while type checking. And since $reqs = \emptyset$ we know that there are no unbound variables in the program. The derivation for this program is split into two parts. The first part contains the typing of the comparison. The second part contains the typing of the whole program and the first part is reused as *cond* in it.

$$\dfrac{\text{T-Gt}\ \dfrac{\text{T-Var}\ \dfrac{}{x : \alpha|\{x : \alpha\}|\emptyset} \qquad \text{T-Num}\ \dfrac{}{0 : \mathcal{N}|\emptyset|\emptyset}}{x > 0 : \mathcal{B}|\{x : \alpha\}|\{\alpha \doteq \mathcal{N}, \mathcal{N} \doteq \mathcal{N}\}}}{x > 0 : \mathcal{B}|\{x : \mathcal{N}\}|\emptyset}\ \sigma = \{\alpha/\mathcal{N}\}$$

$$\text{T-Abs}\ \dfrac{\dfrac{\text{T-Cond}\ \dfrac{\dfrac{\mathit{cond} \qquad \text{T-Var}\ \dfrac{}{x : \beta|\{x : \beta\}|\emptyset} \qquad \text{T-Var}\ \dfrac{}{x : \gamma|\{x : \gamma\}|\emptyset}}{\texttt{if}\ \ldots : \beta|\{x : \mathcal{N}\}|\{\mathcal{B} \doteq \mathcal{B}, \beta \doteq \gamma, \mathcal{N} \doteq \beta\}}}{\texttt{if}\ x > 0\ \texttt{then}\ x\ \texttt{else}\ x : \mathcal{N}|\{x : \mathcal{N}\}|\emptyset}\ \sigma = \{\beta/\mathcal{N}, \gamma\mathcal{N}\}}{\lambda x : \mathcal{N}.\ \texttt{if}\ x > 0\ \texttt{then}\ x\ \texttt{else}\ x : \mathcal{N} \to \mathcal{N}|\emptyset|\{\mathcal{N} \doteq \mathcal{N}\}}}{\lambda x : \mathcal{N}.\ \texttt{if}\ x > 0\ \texttt{then}\ x\ \texttt{else}\ x : \mathcal{N} \to \mathcal{N}|\emptyset|\emptyset}\ \sigma = \emptyset$$

4.5 Implementation of the typing relation

An implementation in Stratego of the type system using intermediate unification is available in Appendix A.1. The syntax definition in SDF of this Grammar is provided in A.1.1.

The type rules are implemented with the transformation rule $generateConstraints$ in A.1.3. As each type rule matches for concrete syntax in the conclusion, the transformation rule matches for abstract syntax. The merging of the results from the premises with integrated unification is implemented in the transformation rule $mergeUnify$.

Helper rules and rules for accessing the results of the constraint generation are available in A.1.2.

The implementation is also available via Github in the repository *bottomup-pcf*[1].

[1]`https://github.com/gnush/bottomup-pcf`

Chapter 5

Java

In the previous chapter, we adapted the methods of bottom-up typechecking introduced for STLC to PCF. We presented PCF as an extension for STLC and we have seen that we could reuse the patterns used in the typing relation for STLC. Now we want to look at Java, a language that is actually used for writing large programs.

The Java grammar and the semantics of it is defined in the Java Language Specification. [GJSB05] The grammar contains an expression language, statements and classes. We will present a typing relation for the stated parts of the grammar, but we will omit some constructs which will be mentioned in the sections that correspond to the sort of the construct.

5.1 Types

In Java we have two different kind of types, namely primitive types and reference types. Primitive types are boolean and the numeric types. The numeric types are divided into two families, the integral- and the floating point types. Integral types are byte, short, int, long and char. The floating point types are float and double.

Definition 5.1 (Primitive type families).
We define sets for the numerical types, the integral types, the floating point types and the boolean type.

$$\begin{aligned}
T_{Num} &:= \{byte, short, int, long, char, float, double\} \\
T_{Int} &:= \{byte, short, int, long, char\} \\
T_{Float} &:= \{float, double\} \\
T_{Bool} &:= \{boolean\}
\end{aligned}$$

The reference types are divided into three kinds, class types, interface types and array types. The type of a class- or interface is the class- or interface name and an array type is a type followed by empty brackets ([ ]).

Example 5.2 (Types).

- int[ ] is an array type with the base type int.

- C is the typename of class C.
- I is the typename of interface I.
- C[ ] is an array type wit the base type C.

Java expressions with two or more subexpressions allow type widening for the subexpressions. We can for example add a value of type int to a value of type long, which will result in a value of type long. The value of type int which is 32-bit is widened to a value of type long which is 64-bit. So we do not loose any information, because all values of the int type can be represented as a value of the long type. We define a widening relation for primitive types:

Definition 5.3 (Primitive type widening)**.**
Primitive types can be widened as follows:

- byte to short, int, long, float, double
- short to int, long, float, double
- char to int, long, float, double
- int to long, float, double
- long to float, double
- float to double

Type widening for reference types is based on subtyping and in Java subtyping is based on class inheritance. If a class C_1 with type T_1 inherits from another class C_2 with type T_2, then T_1 is a subtype of T_2 and T_2 is a supertype of T_1. We write $T_1 <: T_2$ for subtyping and $T_1 :> T_2$ for supertyping. The same holds for interfaces and the subtyping relation is the reflexive transitive closure of the direct inheritance. We define type widening for reference types as follows.

Definition 5.4 (Reference type widening)**.**
A reference type T_1 can be widened to another reference type T_2, if T_1 is a subtype of T_2.

Expressions do not support only type widening, they support also boxing and unboxing. We do not conver that in our typing relation, but we describe the process briefly. In Java each primitive type has a corresponding reference type. For example the appropiate reference type for the primitive type boolean is Boolean. Boxing is the process of converting primitive types to their corresponding reference type and unboxing vice versa.

5.2 Expressions

The functionality of a Java program is based upon expressions, because they can cause side effects such as variable assignments. We present a relation that associates an expression with a type, binding requirements, binding declarations, constraints and class requirements. We note that we now have binding declarations in the relation. This is because variable declarations can be in the scope of not just the current subtree. We will see this when we typecheck block statements. When we typecheck an expression with more than one subexpression, we need to merge the declarations together. Since all declarations are valid in the node above the current expression, because an expression can not define a new scope we can just build the normal set union for the declarations. The set union will remove duplicates and we need to check if a single variable has been declared more than once and if so, we must generate a constraint for that. Note that this constraint will be unsolvable.

Definition 5.5 (Constraint for multiple declarations)**.**
For a variable x and a set of types T, x *declared* T is the unsolvable constraint which states that x has been declared to the types in T.

Definition 5.6 (Constraints for intersecting declarations)**.**
For each variable x that is declared to some type T_1 in the first declaration set and for each variable y that is declared to some type T_2 in the second declaration set. If $x = y$, then we generate a multiple declaration constraint x *declared* $\{T_1, T_2\}$.

$$\begin{aligned} decls_1 \cap_{decl} decls_2 := \{x \ declared \ \{T_1, T_2\} \mid \ & y : T_1 \in decls_1 \wedge \\ & x : T_2 \in decls_2 \wedge \\ & x = y\} \end{aligned}$$

We already stated in the previous section that Java Expressions support type widening, but we can only constrain types to be equal. Since type widening can not be translated into equality we need new constraints. We need two new sorts of constraints. First we need a constraint that states that two types can be primitive widened. Second we need a constraint that states that two types can be widened and that constraints another type to be equal to the widening of those two types. We can restrict the type widening to primitive widening, because the expressions that need them, work only on primitive types.

Definition 5.7 (Constraints for type widening)**.**
The new constraints are:

- For types T_1 and T_2: $T_1 \ widen_{prim} \ T_2$.
- For types T_1, T_2 and T_3: $T_1 \doteq (T_2 \ widen_{prim} \ T_3)$

The type widening works for both directions.

Since we introduced new constraints we must adapt the constraint solving to support them. So we add the following new rules for unification.

Definition 5.8 (Unification for widening constraints)**.**

$$\frac{\{T_1 widen_{prim} T_2\} \cup C \mid \sigma}{C \mid \sigma} \ T_1, T_2 \text{ are widenable}$$

$$\frac{\{T_1 \doteq (T_2 widen_{prim} T_3)\} \cup C \mid \sigma}{\{T_1 \doteq T_3\} \cup C \mid \sigma} \ T_2 \text{ is widenable to } T_3$$

$$\frac{\{T_1 \doteq (T_2 widen_{prim} T_3)\} \cup C \mid \sigma}{\{T_1 \doteq T_2\} \cup C \mid \sigma} \ T_3 \text{ is widenable to } T_2$$

We have defined constraints for type widening, but some Java expressions can operate only with some type families. Multiplication for example works only with values of numeric type and results again in a value of a numeric type. So we must be able to constrain types to a specific type family. We again define a new kind of constraint and the corresponding rule for unification.

Definition 5.9 (Type family constraint)**.**
For a type T and a set of types F: $T \epsilon F$ is the constraint that is solvable if T is a member of F.

$$\frac{\{T \epsilon F\} \cup C \mid \sigma}{C \mid \sigma}\; T \in F$$

We define a typing relation for Java expressions which associates Java Expressions with a type, binding requirements, binding declarations, constraints and class information ($e : T \mid reqs \mid decls \mid cs \mid clzz$). The class information are used to track members of classes, whereas the binding requirements are used to track variables.

5.2.1 Literals

We start with type rules for literals. In Java a literal is a representation of a value of a primitive type, the string type or the null type. We can differ those literals by their syntactic form. We have the following literals for primitive types: integer, floating point, boolean and character. If an integer literal is suffixed with the letter L or l is of type long and otherwise if the literal is not suffixed it is of type int. For floating point literals this is similar. If a floating point literal is suffixed with F or f it is of type float and otherwise if it is not suffixed or it is suffixed with D or d it is of type double.

$$\text{T-Int}\ \frac{i \text{ is integer literal} \qquad i \text{ has no suffix}}{i : \mathit{int} \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}$$

$$\text{T-Long}\ \frac{i \text{ is integer literal} \qquad i \text{ has suffix L or l}}{i : \mathit{long} \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}$$

$$\text{T-Float}\ \frac{f \text{ is floating point literal} \qquad f \text{ has suffix F or f}}{f : \mathit{float} \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}$$

$$\text{T-Double}\ \frac{f \text{ is floating point literal} \qquad f \text{ has suffix D, d or none}}{f : \mathit{double} \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}$$

We do not have such restrictions for boolean and character literals, because each of them can be only of a single type. There are exact two values we can represent with a boolean literal, true and false which have type boolean. Character literals can represent a single Unicode letter and have type char.

$$\text{T-True}\ \frac{}{\mathit{true} : \mathit{boolean} \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}$$

$$\text{T-False}\ \frac{}{\mathit{false} : \mathit{boolean} \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}$$

$$\text{T-Char}\ \frac{}{\text{'}c\text{'} : char \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}$$

We have the string- and the null literal left. The string literal represents a constant sequence of Unicode characters and has the type of class String. The null type is somewhat special, because it has no name, but it can be cast to any reference type. The only syntactic construct that is of the null type is the null literal and because of its characteristics we generate a fresh type variable for it.

$$\text{T-String}\ \frac{}{\text{"}s\text{"} : String \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}$$

$$\text{T-Null}\ \frac{\alpha\ fresh}{null : \alpha \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}$$

5.2.2 Names

Names are used to refer to entities that are declared in a program. Such a declared entity can be a package, class type, interface type, member of a reference type or local variable. A name is either a simple name or a qualified name. A simple name is a single identifier and a qualified name is a sequence of identifiers separated with ".". A declaration that introduces a name has a scope. The scope is that part of the program text within the declared entity can be referred by a simple name. Packages and reference types have members which can be referred to with a qualified name $N.x$, where N is a simple- or qualified name and x is an identifier. If N names a package, then x is a member of that package which is a class or interface type or a subpackage. If N names a reference type, then x is a class, an interface, a field, or a method. We define a class requirement for membership.

Definition 5.10.
For types T_1, T_2 and identifier x

$$T_1 \text{ hasMember } x : T_2$$

requires that type T_1 has a member x with type T_2.

Names are syntactically classified and we cover expression names and method names. Expression names are used for refering to local variables or field in the scope of the name. Method names are used fo refering to methods and can appear only in method invocation expressions. We split the typing for expression- and method names into two type rules for each of them. One for simple names and one for qualified names.

We begin with the type rules for expression names. If an expression name is a single identifier, it refers to a local variable or field. So we generate a fresh type variable as the type for the name and we add a binding requirement from the name to the type variable.

$$\text{T-ExprName}\ \frac{\alpha \text{ fresh}}{x : \alpha \mid \{x : \alpha\} \mid \emptyset \mid \emptyset \mid \emptyset}$$

For the type rule for expression names that are qualified names, we have to generate class requirements for the sequence of identifiers in the name. The first identifier in the sequence must be a local variable or a field and we generate a fresh type variable for it. We require the identifier to be bound to that type variable. The type of the expression name is the type of the last identifier of the sequence, which is the member we want to access.

$$clz = \{\alpha_i \text{ hasMember } x_{i+1} : \alpha_{i+1} \mid i \in [1, n-1]\}$$

$$\text{T-QExprName}\ \frac{\alpha_i \text{ for } i \in [1,n] \text{ fresh}}{x_1.x_2.\cdots.x_n : \alpha_n \mid \{x_1 : \alpha_1\} \mid \emptyset \mid \emptyset \mid clz}$$

Method names act similar as expression names, except that they do not refer to local variables or fields but to methods. So we first define a new class requirement for methods.

Definition 5.11.
For types T_1, T_2 and identifier x

$$T_1 \text{ hasMethod } x : T_2$$

requires that type T_1 has a method x with type T_2.

For method names with a single identifier, we generate two fresh type variables, one for the current class and one for the type of the method. We require the first type variable to be bound to "this", which is a special keyword refering to the current object.

$$\text{T-MethodName}\ \frac{\alpha, \beta \text{ fresh}}{x() : \beta \mid \{\mathit{this} : \alpha\} \mid \emptyset \mid \emptyset \mid \{\alpha \text{ hasMethod } x : \beta\}}$$

For qualified method names, we generate membership requirements for the sequence, except for the last pair for which we generate a method requirement.

$$\begin{aligned} clz = {} & \{\alpha_i \text{ hasMember } x_{i+1} : \alpha_{i+1} \mid i \in [1, n-2]\} \\ & \cup \{\alpha_{n-1} \text{ hasMethod } x_n : \alpha_n\} \end{aligned}$$

$$\text{T-QMethodName}\ \frac{\alpha_i \text{ for } i \in [1,n] \text{ fresh}}{x_1.x_2.\cdots.x_n() : \beta \mid \{x_1 : \alpha_1\} \mid \emptyset \mid \emptyset \mid \{clz}$$

5.2.3 this

The keyword *this* refers to the current instance under observation. The type of *this* is the type of the class within it occured.
Since we do not know in which class we currently are, when typechecking a *this* reference, we have to generate a fresh type variable as the type for it. We then require that the "this" is bound to that type variable. This is safe, because we can not declare a variable or field with the name "this". We can fullfill this requirement when we come to the point of typechecking a class declaration, because we learn the actual class name from that declaration.

$$\text{T-This}\ \frac{\alpha \text{ fresh}}{\mathit{this} : \alpha \mid \{\mathit{this} : \alpha\} \mid \emptyset \mid \emptyset \mid \emptyset}$$

5.2.4 Operators

We have defined type rules for literals and names and we proceed with the operators and we have already stated that Java expressions may support type widening. The operators are mostly the expressions for which this holds true. So when we typecheck an operator we will most likely generate widening- instead of equality constraints.

We first look at the comparison operators (<, <=, > and >=) which act on numeric values. So both subexpressions must be of some numeric type and the two types must be compatible in the way that a type widening is possible. The comparison operators results in a value of type boolean. We merge the binding requirements as we did in the previous chapters and we build the union of the binding declarations and generate a new failure constraint if a variable has been declared more than once with the new intersection operator for declarations ($\cap_{decl}$).

$$\begin{aligned} cs = \{&T_1 widen_{prim} T_2, T_1 \epsilon T_{Num}, T_2 \epsilon T_{Num}\} \\ &\cup (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2) \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-Less}\ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1 < e_2 : boolean \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dcls_2 \mid cs \mid clz_1 \cup clz_2}$$

$$\begin{aligned} cs = \{&T_1 widen_{prim} T_2, T_1 \epsilon T_{Num}, T_2 \epsilon T_{Num}\} \\ &\cup = (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2) \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-LessEq}\ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1 <= e_2 : boolean \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dcls_2 \mid cs \mid clz_1 \cup clz_2}$$

$$\begin{aligned} cs = \{&T_1 widen_{prim} T_2, T_1 \epsilon T_{Num}, T_2 \epsilon T_{Num}\} \\ &\cup = (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2) \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-Gt}\ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1 > e_2 : boolean \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dcls_2 \mid cs \mid clz_1 \cup clz_2}$$

$$\begin{aligned} cs = \{&T_1 widen_{prim} T_2, T_1 \epsilon T_{Num}, T_2 \epsilon T_{Num}\} \\ &\cup = (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2) \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-GtEq}\ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1 >= e_2 : boolean \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dcls_2 \mid cs \mid clz_1 \cup clz_2}$$

There are two comparison operates we have not covered yet, namely the equality (==) and inequality ($! =$) operators. These operators are not restricted to act on numeric types, but we have still type widening for the operands. So we do not generate constraints that limit the types to a numeric one, but generate a type widening constraint.

$$\begin{aligned} cs = \{&T_1 \ \mathit{widen}_{prim} \ T_2\} \\ &\cup (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2) \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-Eq}\ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1 == e_2 : \mathit{boolean} \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dcls_2 \mid cs \mid clz_1 \cup clz_2}$$

$$\begin{aligned} cs = \{&T_1 \ \mathit{widen}_{prim} \ T_2\} \\ &\cup (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2) \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-NEq}\ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1 ! = e_2 : \mathit{boolean} \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dcls_2 \mid cs \mid clz_1 \cup clz_2}$$

Example 5.12.
We use the already defined type rules to typecheck the expression

$$1.0f < 5$$

We check the literal $1.0f$ with rule T-Float, because it is a floating point value suffixed with f. We obtain type $float$ and no requirements, declarations and constraints. We then use rule T-Int to check 5, because it is an integer literal with no suffix. We obtain type *int* and again no requirements, declarations and constraints. We finally use rule T-Less and generate new constraints $cs_{new} = \{float \ widen_{prim} \ int, float \ \epsilon \ T_{Num}, int \ \epsilon \ T_{Num}\}$ and since we generated no requirements and declarations for the subexpressions we have no constraints from the merging $cs_{merge} = \emptyset$. We end with the type *boolean* and the constraints cs_{new}. What is left to do is to solve the constraints. We take the first constraint $float \ widen_{prim} \ int$ and since *int* can be widened to $float$ this constraint is fullfilled. The next constraint $float \ \epsilon \ T_{Num}$ requires that $float$ is a numeric type which holds true. The final constraint $int \ \epsilon \ T_{Num}$ requires that *int* is a numeric type which holds again. There are no constraints left and since we had no type variables we have an empty unifier and the type of $1.0f < 5$ has type boolean.

$$\cfrac{\text{T-Less}\ \cfrac{\text{T-Float}\ \cfrac{1.0f \text{ is float value}}{1.0f : float \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset} \qquad \text{T-Int}\ \cfrac{5 \text{ is int value}}{5 : int \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}}{1.0f < 5 : boolean \mid \emptyset \mid \emptyset \mid cs_{new} \mid \emptyset}}{1.0f < 5 : boolean \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}\ \sigma = \emptyset$$

We knew that the comparison operators resulted in a value of type boolean, so we were fine generating constraints that just check if the operand types could be widened. When type checking numerical operators like multiplication we do not know the resulting type of the operator directly, because the resulting type depends on the operand types. The multiplication of an *int* value to a $float$ value will for example result in a $float$ value, because *int* can be widened to $float$. So we generate the combination of the widening- and equality constraints for these operators.

For multiplication we generate a fresh type variable α as the resulting type. We then constrain α to be equal as the widening of the operand types and the operand types to be some numeric types. We proceed with the merging of the requirements and declarations with the already known operators. We omit the freshness condition for type variables in the type rules from now on and keep in mind that this condition is there implizitly for each type variable introduced in a type rule.

$$\begin{aligned} cs_{new} &= \{\alpha \doteq (T_1 \ widen_{prim} \ T_2), T_1 \ \epsilon \ T_{Num}, T_2 \ \epsilon \ T_{Num}\} \\ cs_{merge} &= (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2) \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-Mul} \ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1 * e_2 : \alpha \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dlcs_2 \mid cs_{new} \cup cs_{merge} \mid clz_1 \cup clz_2}$$

We can reuse this pattern for all numeric operators except plus, because plus can be used also to concatenate strings. So we need a special treatment for plus. Since plus can be used as a numeric operator or as string concatenating, we need a new constraint which checks for primitive type widening if the operants are not of type String. The string concatenation functionality can also be mixed with primitive types, this is the case when one of the operants is of type String.

Example 5.13 (Usage of "+")**.**
We present some examples of the proper usage of the plus operator.

- 1+2 will be evaluated to 3.
- "foo" + "bar" will be evaluated to "foobar"
- "foo" + 2 will be evaluated to "foo2"
- 2.2 + "foo" will be evaluated to "2.2foo".

We define a new constraint widenString which will model these properties. Since we do not know the resulting type of plus statically the constraint will also be combined with equality as we have seen with the widened equality constraint introduced for the numerical operators (e.g. in the multiplication rule).

Definition 5.14.
For types T_1, T_2 and T_3: $T_1 \doteq (T_2 \ widenString \ T_3)$ is the constraint that is solvable either if T_2 and T_3 are primitive types and can be widened or if T_2 or T_3 is String.

$$\frac{\{T_1 \doteq (T_2 \ widenPrim \ T_3)\} \cup C \mid \sigma}{\{T_1 \doteq String\} \cup C | \sigma} \ T_2 \text{ or } T_3 \text{ is String}$$

$$\frac{\{T_1 \doteq (T_2 \ widenPrim \ T_3)\} \cup C \mid \sigma}{\{T_1 \doteq T_3\} \cup C | \sigma} \ T_2 \text{ is widenable to } T_3$$

$$\frac{\{T_1 \doteq (T_2 \ widenPrim \ T_3)\} \cup C \mid \sigma}{\{T_1 \doteq T_2\} \cup C | \sigma} \ T_3 \text{ is widenable to } T_2$$

Now we reuse the pattern from the multiplication operator. We use the new constraint to handle string concatenation for the types of both operants and we generate a fresh type variable as the resulting type. We also constrain the operant types to be of a numeric type or String. The merging of the requirements and declarations is handled as in the previous rules.

$$\begin{aligned} T_{SNum} &= \{String\} \cup T_{Num} \\ cs_{new} &= \{\alpha \doteq (T_1\ widenString\ T_2), T_1 \lessdot T_{SNum}, T_2 \lessdot T_{SNum}\} \\ cs_{merge} &= (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2) \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-Plus}\ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1 + e_2 : \alpha \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dcls_2 \mid cs_{new} \cup cs_{merge} \mid clz_1 \cup clz_2}$$

We have seen the idea of type checking numeric operators in the rule T-Mul and the specially treated plus operator in T-Plus. We reuse these directly for the type checking of the binary numeric operators division (/), remainder (%), minus (−)

$$\begin{aligned} cs_{new} &= \{\alpha \doteq (T_1\ widen_{prim}\ T_2), T_1 \lessdot T_{Num}, T_2 \lessdot T_{Num}\} \\ cs_{merge} &= (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2) \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-Div}\ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1 / e_2 : \alpha \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dlcs_2 \mid cs_{new} \cup cs_{merge} \mid clz_1 \cup clz_2}$$

$$\begin{aligned} cs_{new} &= \{\alpha \doteq (T_1\ widen_{prim}\ T_2), T_1 \lessdot T_{Num}, T_2 \lessdot T_{Num}\} \\ cs_{merge} &= (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2) \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-Rem}\ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1\ \%\ e_2 : \alpha \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dlcs_2 \mid cs_{new} \cup cs_{merge} \mid clz_1 \cup clz_2}$$

$$\begin{aligned} cs_{new} &= \{\alpha \doteq (T_1\ widen_{prim}\ T_2), T_1 \lessdot T_{Num}, T_2 \lessdot T_{Num}\} \\ cs_{merge} &= (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2) \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-Minus}\ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1 - e_2 : \alpha \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dlcs_2 \mid cs_{new} \cup cs_{merge} \mid clz_1 \cup clz_2}$$

and the integer bitwise operators (&, |, ∧) with the extend of type boolean as a possible operant type and we define $T_{BN} := T_{Bool} \cup T_{Num}$ as the set containing the numeric types and boolean.

$$\begin{aligned} cs_{new} &= \{\alpha \doteq (T_1\ widen_{prim}\ T_2), T_1 \lessdot T_{BN}, T_2 \lessdot T_{BN}\} \\ cs_{merge} &= (reqs_1 \cap_c reqs_2) \cup (decls_1 \cap_{decl} decls_2) \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-BitAnd}\ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1\ \&\ e_2 : \alpha \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dlcs_2 \mid cs_{new} \cup cs_{merge} \mid clz_1 \cup clz_2}$$

$$\begin{aligned} cs_{new} &= \{\alpha \doteq (T_1 \ widen_{prim} \ T_2), T_1 \epsilon T_{BN}, T_2 \epsilon T_{BN}\} \\ cs_{merge} &= (reqs_1 \cap_c reqs_2) \cup (decls_1 \cap_{decl} decls_2) \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-BitOr} \ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1 \mid e_2 : \alpha \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dlcs_2 \mid cs_{new} \cup cs_{merge} \mid clz_1 \cup clz_2}$$

$$\begin{aligned} cs_{new} &= \{\alpha \doteq (T_1 \ widen_{prim} \ T_2), T_1 \epsilon T_{BN}, T_2 \epsilon T_{BN}\} \\ cs_{merge} &= (reqs_1 \cap_c reqs_2) \cup (decls_1 \cap_{decl} decls_2) \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-BitXOr} \ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1 \ \wedge \ e_2 : \alpha \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dlcs_2 \mid cs_{new} \cup cs_{merge} \mid clz_1 \cup clz_2}$$

We also can adapt this pattern for the unary operators as increments and decrements. Since we have just one subexpression we do not need type widening for the one operant type. As in the binary numeric operators we do not know the resulting type and we generate a fresh type variable for it. There is no merging involved, again due to the fact that there is just one subexpression.

$$\text{T-PreIncr} \ \frac{e : T \mid reqs \mid decls \mid cs \mid clz \qquad cs_{new} = \{\alpha \doteq T, T \epsilon T_{Num}\}}{{+}{+}e : \alpha \mid reqs \mid decls \mid cs \cup cs_{new} \mid clz}$$

$$\text{T-PostIncr} \ \frac{e : T \mid reqs \mid decls \mid cs \mid clz \qquad cs_{new} = \{\alpha \doteq T, T \epsilon T_{Num}\}}{e{+}{+} : \alpha \mid reqs \mid decls \mid cs \cup cs_{new} clz \mid}$$

$$\text{T-PreDecr} \ \frac{e : T \mid reqs \mid decls \mid cs \mid clz \qquad cs_{new} = \{\alpha \doteq T, T \epsilon T_{Num}\}}{{-}{-}e : \alpha \mid reqs \mid decls \mid cs \cup cs_{new} \mid clz}$$

$$\text{T-PostDecr} \ \frac{e : T \mid reqs \mid decls \mid cs \mid clz \qquad cs_{new} = \{\alpha \doteq T, T \epsilon T_{Num}\}}{e{-}{-} : \alpha \mid reqs \mid decls \mid cs \cup cs_{new} \mid clz}$$

Java also knows a conditional expression (e_1 ? e_2 : e_3) which evaluates similar to the conditional expression we had in PCF, except that the types of the subexpressions e_1 and e_2 can be widenable now. So we constrain the type of e_1 to boolean and we generate a type widening constraint for the types of e_2 and e_3. Since we do not know the resulting type statically we generate a fresh typevariable for it. The requirement merging and the constraint generation for intersecting declarations is done pairwise.

$$\begin{aligned} cs_{new} &= \{T_1 \doteq boolean, \alpha \doteq (T_2 \ widen_{prim} \ T_3)\} \\ cs_{rqs} &= (rqs_1 \cap_c rqs_2) \cup (rqs_2 \cap_c rqs_3) \\ cs_{dcls} &= (dcls_1 \cap_{decl} dcls_2) \cup (dcls_2 \cap_{decl} dcls_3) \\ cs &= cs_1 \cup cs_2 \cup cs_3 \\ reqs &= (rqs_1 \uplus rqs_2) \uplus rqs_3 \\ decls &= dcls_1 \cup dcls_2 \cup dcls_3 \\ clz &= clz_1 \cup clz_2 \cup clz_3 \end{aligned}$$

$$\text{T-CondExpr}\ \frac{e_i : T_i \mid rqs_i \mid dcls_i \mid cs_i \mid clz_i \text{ for } i \in \{1,2,3\}}{(e_1\ ?\ e_2 : e_3) : \alpha \mid rqs \mid dcls \mid cs_{new} \cup cs_{rqs} \cup cs_{dcls} \cup cs \mid clz}$$

Now we look at primitive casts. The cast operator can convert an integral value to a value of any numeric type. The type in which we want to cast is enclosed by parantheses followed by the expression we want to cast.

Example 5.15 (Usage of casts)**.**
The cast operator can be used to explicitly convert numeric types.

- (long) 10 casts the value of type int (10) to a value of type long. This is a type widening.
- (int) 10L casts the value of type long (10L) to a value of type int. This can not be done with type widening, because the long type has a wider value range than the int type. This can lead to a loss of information if the value we want to cast is bigger than what the destinated type can hold.
- (char) 65 casts the int value 65 to a value of type char ('A').
- (int) 1.4f casts the float value 1.4f to the int value 1. Here we loose precision, because the float value will be floored to fit into the int type.

As we have seen in the examples, casts can be used also to narrow down a type instead of just widen it. A type narrowing is the opposite of a type widening and luckily the widening constraint we have already checks for widening in both directions. So we generate a widening constraint for the denoted type and the type of the expression to cast. The resulting type of a cast is the denoted type.

$$\text{T-PrimCast}\ \frac{e_1 : T_1 \mid reqs_1 \mid decls_1 \mid cs_1 \mid clz \qquad cs_{new} = \{T\ widen_{prim}\ T_1\}}{(T)\ e_1 : T \mid reqs_1 \mid decls_1 \mid cs_1 \cup cs_{new} \mid clz}$$

The next expressions we want to look at are assignments. We only cover normal assignments (=), as the functionality of the other kinds of assignments (+=, -=, ...) can be achieved by combining them with the corresponding operators. E.g. $x \mathrel{+}= x$ is semantically equivalent to $x = x + x$. Assignments also support type widening, but as the evaluation of an assignment is directed the widening is allowed in one direction only. This also was the case with casts, but as casts additionally allowed us to narrow down a type we were able to use the widening constraint we already had. Yet another property of an assignment is, that it can be used with reference types, which is also not supported in the existing widening constraint. So we define a new widening constraint, that is directed and allows both, primitive and reference types.

Definition 5.16. For types T_1 and T_2, T_1 *widen* T_2 is the constraint that is solvable if T_1 can be primitive widened to T_2 or if T_1 is a subtype of T_2. The unification rules for this are:

$$\frac{\{T_1\ widen\ T_2\} \cup C \mid \sigma}{C \mid \sigma}\ T_1 \text{ is widenable to } T_2$$

$$\frac{\{T_1\ widen\ T_2\} \cup C \mid \sigma}{C \mid \sigma}\ T_1 <: T_2$$

Now that we have defined the new constraint, we use it in the type rule for assignments. Assignments have two subexpressions, first the expression denoting the variable to which we want to assign and second the expression whose value we want to assign to that variable. For the assignment to succeed, we have to widen the type of the expression to the type of the variable.

$$\begin{aligned} cs_{new} &= \{T_2 \ \mathit{widen} \ T_1\} \\ cs_{merge} &= (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2) \\ cs &= cs_{new} \cup cs_{merge} \cup cs_1 \cup cs_2 \end{aligned}$$

$$\text{T-Assign}\ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1 = e_2 : T_1 \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dcls_2 \mid cs \mid clz_1 \cup clz_2}$$

The last family of expressions we look at are the expressions involving arrays, such as array access and array creation. An array can be accessed with brackets and an integral number. Arrays can either be created by a new array expression or with an array initialization.

Example 5.17. For an array *arr* of type int.

- *arr* = new int[3] initializes *arr* with an empty array of type int with the length 3. Note that int values are initialized with 0, so empty means that all elements are 0. For reference types, all elements would be the null value.
- *arr* = $\{1, 2, 3\}$ initializes *arr* with an array of type int with the length 3. The elements of the array are initialized with 1, 2 and 3 in that order.
- *arr*[2] accesses the third element of *arr*.

We begin with the type rule for array access. We have two subexpressions. First the expression which denotes the array we want to access and second is the position of the element in that array which we want to get. We must ensure that the type of the first expression is some array type. So we generate a fresh type variable and constrain the type of the first expression to an array type whose element type is the generated type variable. The resulting type of an array access is the element type of the array for which we generated a typevariable. As we can only use integer values for the position declaration, we have to constrain the type of the second expression to be widenable to int. This widening is directed and we reuse the widen constraint we introduced for assignments.

$$\begin{aligned} cs_{new} &= \{T_1 \doteq \alpha[\,], T_2 \ \mathit{widen} \ \mathit{int}\} \\ cs_{merge} &= (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2) \\ cs &= cs_{new} \cup cs_{merge} \end{aligned}$$

$$\text{T-Access}\ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e_2 : T_2 \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{e_1[e_2] : \alpha \mid rqs_1 \uplus rqs_2 \mid dcls_1 \cup dcls_2 \mid cs \cup cs_1 \cup cs_2 \mid clz_1 \cup clz_2}$$

Now that we know how to typecheck an array access, we inspect the creation of an array. We already have seen that there are two possibilities to create an array. First, with the new keyword which creates an array with the denoted dimensions and length and initializes the elements with the default values of that type. Second, providing a concrete initialization for the elements from which the length of the array is deducted. These concrete initialization creates a one dimensional array, but the initializations can be nested to create multi-dimensional ones.

We start with the type rule for the first possibility, the new array construct. The resulting type of such an array creation is an array type, whose dimension fits to the length of the provided dimensions list. The provided dimensions list is a non empty list, so we have at least a one dimensional array.

Example 5.18. The type of "new $int[1][1][1]$" is $int[\,][\,][\,]$.

For a dimension list of length n we have n subexpressions denoting the length of the corresponding dimensions of the array. The types of this subexpression must be widenable to the type int, because the length of an array can be only an integral (32-bit) number. So we generate for the type of each subexpression a directed widening constraint from that type to int. For the merging of the requirements and declarations we have to adapt what we did for the conditional with three subexpressions. Since the union of the requirements is left-associative, we can build the union of many requirements by using the operator $\uplus$ linear. The intersection constraints of the requirements and declarations need again to be build pairwise with the operators $\cap_c$ and $\cap_{decl}$.

$$
\begin{aligned}
cs_{new} &= \{T_i \; widen \; int \mid i \in [1,n]\} \\
cs_{merge} &= \Big(\bigcup_{i\in[1,n-1]} rqs_i \cap_c rqs_{i+1} \Big) \\
&\cup \Big(\bigcup_{i\in[1,n-1]} dcls_i \cap_{decl} dcls_{i+1} \Big) \\
rqs &= \biguplus_{i\in[1,n]} rqs_i \\
dcls &= \bigcup_{i\in[1,n]} dcls_i \\
cs &= \bigcup_{i\in[1,n]} cs_i \\
clz &= \bigcup_{i\in[1,n]} clz_i
\end{aligned}
$$

$$
\text{T-ArrNew}\;\frac{e_i : T_i \mid rqs_i \mid dcls_i \mid cs_i \mid clz_i \text{ for } i \in [1,n]}{\text{new } T[e_1]...[e_n] : T\underbrace{[\,]...[\,]}_{n \text{ times}} \mid reqs \mid decls \mid cs_{new} \cup cs_{merge} \cup cs \mid clz}
$$

The array creation using an initialization provides a list of expressions which define the elements of the array. These types have to be equal, so we constrain the type of each subexpression to be equal to the type of the following subexpression except for the last type. With the transitivity of equality we get that all types are constrained equal. The resulting type is then an array type with the type of one subexpression as the element type. Since we have constrained the types of all subexpressions to be equal, we can pick the type of the first subexpression as element type.

$$\begin{aligned}
cs_{new} &= \{T_i \doteq T_{i+1} \mid i \in [1, n-1]\} \\
cs_{merge} &= \Big(\bigcup_{i \in [1,n-1]} rqs_i \cap_c rqs_{i+1} \Big) \\
&\cup \Big(\bigcup_{i \in [1,n-1]} dcls_i \cap_{decl} dcls_{i+1} \Big) \\
rqs &= \biguplus_{i \in [1,n]} rqs_i \\
dcls &= \bigcup_{i \in [1,n]} dcls_i \\
cs &= \bigcup_{i \in [1,n]} cs_i \\
clz &= \bigcup_{i \in [1,n]} clz_i
\end{aligned}$$

$$\text{T-ArrInit}\ \frac{e_i : T_i \mid rqs_i \mid dcls_i \mid cs_i \mid clz_i \text{ for } i \in [1, n]}{\{e_1 ... e_n\} : T_1[] \mid reqs \mid decls \mid cs_{new} \cup cs_{merge} \cup cs \mid clz}$$

Example 5.19.
We previously stated that the type of "new $int[1][1][1]$" is $int[\,][\,][\,]$. Now that we have defined the type rules for array creation, we want to show that this is indeed the type of the expression.

We typecheck each subexpression 1 with rule T-Int. This results in the type *int* and no requirements, declarations and constraints.

$$\text{T-Int}\ \frac{}{1 : int \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}$$

As there is no interaction with the requirements, declarations or constraints involved in the typing of a constant number, we check it once only and use the result as a premise in the T-ArrNew rule application. We have *int* as the type of the three subexpressions and we generate the new constraints *int widen int*, *int widen int* and *int widen int* to ensure that we use only integer numbers for the length definition of the array. Since we have no requirements, declarations or constraints in the premises, there are no new constraints from merging. We combine the newly generated constraints to a set to obtain $\{int\ widen\ int\}$.

$$\text{T-ArrNew}\ \frac{1 : int \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset \qquad 1 : int \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset \qquad 1 : int \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}{\text{new } int[1][1][1] : int[\,][\,][\,] \mid \emptyset \mid \emptyset \mid \{int\ widen\ int\} \mid \emptyset}$$

We have ended the type checking with no requirements, so there are no unbound variables in the expression. What is left to do, is to solve the constraint *int widen int*. Since the types are actually equal this constraint holds true and we know that the type of the expression “new *int*[1][1][1]” is in fact *int*[ ][ ][ ].

5.3 Statements

In the previous section we have discussed expressions, which where responsible for the behavior of programs e.g. for their side effects. We now look at statements, which control the execution order of a program. Statements do not have values and as they have no values they do not need a type assignment. We define a typing relation that associates statements with binding requirements, binding declarations, constraints and class requirements ($stm \vdash rqs \mid dcls \mid cs \mid clz$).

We begin with the type rules for two simple statements, the empty statement (;) and the expression statement (e;). Afterwards we proceed with conditional statements, loops and blocks.

5.3.1 Simple Statements

We start with the empty statement, which does nothing. Since it does nothing we create an empty result in the type rule for it.

$$\text{Empty}\ \frac{}{;\ \vdash \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}$$

Expressions can be enclosed by a statement. These expression statements join the behavior with the execution order of a program. Expression statements have exact one subexpression and the result of the typing of that subexpression is used as the result of the statement, except the type which is dropped.

$$\text{Expr}\ \frac{e : T \mid rqs \mid dcls \mid cs \mid clz}{e;\ \vdash rqs \mid dcls \mid cs \mid clz}$$

5.3.2 Conditional statements

The conditional statement “if”, either allows the control if a particular statement will be executed or not or the branching of two statements, executing one branch but not both. We have two syntactic constructs for this. First, the if-then statement, which executes its substatement if the condition holds true or skips the execution of the substatement if the condition holds false. Second, the if-then-else statement, which allows the choice of execution between two statements. If the condition holds true, it evaluates the first branch and the second branch vice versa.

Since the condition is an expression and we evaluate it to make a binary choice, we must constrain the type of the expression to boolean or the corresponding reference type Boolean. We then must merge the requirements and declarations together. We have used the same sort of requirements and declarations for statements and expressions and we can therefore reuse the existing operators to merge requirements for statements and expressions together.

$$\begin{aligned} cs_{new} &= \{T \epsilon \{boolean, Boolean\}\} \\ cs_{merge} &= (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2) \\ cs &= cs_{new} \cup cs_{merge} \end{aligned}$$

$$\text{If}\ \frac{e : T \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad stm \vdash rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{\text{if } e \text{ then } stm \vdash rqs_1 \uplus rqs_2 \mid dcls_1 \cup dcls_2 \mid cs \cup cs_1 \cup cs_2 \mid clz_1 \cup clz_2}$$

$$\begin{aligned} cs_{new} &= \{T \epsilon \{boolean, Boolean\}\} \\ cs_{merge} &= (rqs_e \cap_c rqs_1) \cup (rqs_1 \cap_c rqs_2) \\ &\quad \cup (dcls_e \cap_{decl} dcls_1) \cup (dcls_1 \cap_{decl} dcls_2) \\ rqs &= rqs_e \uplus rqs_1 \uplus rqs_2 \\ dcls &= dcls_e \cup dcls_1 \cup dcls_2 \\ cs &= cs_{new} \cup cs_{merge} \cup cs_e \cup cs_1 \cup cs_2 \\ clz &= clz_e \cup clz_1 \cup clz_2 \end{aligned}$$

$$\text{IfElse}\ \frac{e : T \mid rqs_e \mid dcls_e \mid cs_e \mid clz_e \quad stm_i \vdash rqs_i \mid dcls_i \mid cs_i \mid clz_i,\ i \in \{1, 2\}}{\text{if } e \text{ then } stm_1 \text{ else } stm_2 \vdash rqs \mid dcls \mid cs \mid clz}$$

5.3.3 Local Variable Declarations

Local variable declaration statements can be used to declare one or more local variable names. A local variable declaration statement can only occur directly in a block or in the head of a for statement, this is a syntactical restriction. The scope of a so declared variable is the rest of the block in which it occured. A local variable declaration consists of the type to which the variables are bound and a list of declarators. A declarator is an identifier and optionally an expression used as the initialization for the variable. The name of a local variable is the identifier that appears in the declarator.

Example 5.20 (Local variable declarations)**.**
We show two blocks which both define the variables i, j and k of type *int*. The variables i and j are initialized to 1 and the variable k is not initialized at all. The first block uses three local variable declaration statements and each of them declare one variable. The second block uses just one local variable declaration statement for the same task.

```
{
  int i = 1;
  int j = 1;
  int k;
}
{
  int i=1, j=1, k;
}
```

We will not cover the mixed initialization of variables in the type rules for local variable declaration statements. So if we want to declare two initialized variables and one not initialzed variable as in the previous example, we have to write at least two local variable declaration statements. One statement for the variables with initializations and one statement for the variable without initialization.

We begin with the type rule for a local variable declaration statement without initializations for the variables. We have a denoted type and a list of identifiers. We declare each identifier to be bound to the denoted type. Note that we now declare new variables, so we do not need to use the binding requirements, instead we add them to the binding declarations. Since we do not have subexpressions we do not need any constraints.

$$\text{LocDclNoInit}\ \frac{dcls = \{x_i : T \mid i \in [1,n]\}}{T\ x_1, \cdots, x_n; \vdash \emptyset \mid dcls \mid \emptyset \mid \emptyset}$$

If we want to typecheck a local variable declaration statement with initializations for the declared variables, we can extend the previous rule. We now have a denoted type and a list of pairs of identifiers and expressions. We still declare each identifier to be bound to the denoted type. In addition we constrain the type of each subexpression to be equal to the denoted type, because we can only initialize a variable with an expression of the same type as the type of the variable. We then merge the multiple results from the subexpressions together.

$$\begin{aligned}
cs_{new} &= \{T \doteq T_i \mid i \in [1,n]\} \\
cs_{merge} &= \Big(\bigcup_{i \in [1,n-1]} rqs_i \cap_c rqs_{i+1} \Big) \\
&\quad \cup \Big(\bigcup_{i \in [1,n-1]} dcls_i \cap_{decl} dcls_{i+1} \Big) \\
cs &= cs_{new} \cup cs_{merge} \cup \Big(\bigcup_{i \in [1,n]} cs_i \Big) \\
dcls &= \{x_i : T \mid i \in [1,n]\} \cup \Big(\bigcup_{i \in [1,n]} dcls_i \Big)
\end{aligned}$$

$$\text{LocDclInit}\ \frac{e_i : T_i \mid rqs_i \mid dcls_i \mid cs_i \mid clz_i \text{ for } i \in [1,n]}{T\ x_1 = e_1, \cdots, x_n = e_n; \vdash \biguplus_{i \in [1,n]} rqs_i \mid dcls \mid cs \mid \bigcup_{i \in [1,n]} clz_i}$$

5.3.4 Loops

Loops can be used to repeatedly execute a program fragment. We will cover while statements, do statements and basic for statements. We do not cover abrupt completion with break or continue.

The while statement contains an expression and a statement. The expression is evaluated first. If the value of the expression is true, the statement is executed. This will be repeated until the value of the expression is false. So the expression is the condition which decides if we break the loop or not and we therefore constrain it to be of type boolean or Boolean.

$$\begin{aligned}cs_{new} &= \{T \;\epsilon\; \{boolean, Boolean\}\}\\ cs_{merge} &= (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2)\\ cs &= cs_1 \cup cs_2 \cup cs_{new} \cup cs_{merge}\end{aligned}$$

$$\text{While}\;\frac{e : T \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad stm \vdash rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{\text{while } (e)\ stm \vdash rqs_1 \uplus rqs_2 \mid dcls_1 \cup dcls_2 \mid cs \mid clz_1 \cup clz_2}$$

The do statement contains a statement and an expression. We notice that this is the reverse order as in the while statement. The statement is executed first and the decision if we repeat it, is done afterwards by the evaluation of the expression. We again constrain the type of the expression to be of type boolean or Boolean.

$$\begin{aligned}cs_{new} &= \{T \;\epsilon\; \{boolean, Boolean\}\}\\ cs_{merge} &= (rqs_1 \cap_c rqs_2) \cup (dcls_1 \cap_{decl} dcls_2)\\ cs &= cs_1 \cup cs_2 \cup cs_{new} \cup cs_{merge}\end{aligned}$$

$$\text{Do}\;\frac{stm \vdash rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad e : T \mid rqs_2 \mid dcls_2 \mid cs_2 \mid clz_2}{\text{do } stm \text{ while } (e) \vdash rqs_1 \uplus rqs_2 \mid dcls_1 \cup dcls_2 \mid cs \mid clz_1 \cup clz_2}$$

The basic for statement consists of a variable declaration for initialization, an expression used as condition, some expressions used for updating and a statement. The variable declaration is executed first and just once. Then the condition is evaluated. If the condition is evaluated to true, the statement is executed followed by the update expression. This is repeated until the expression is evaluated to false. We constrain the type of the condition to boolean or Boolean.

This is the first statement in which a variable declaration can occur. The variable declaration can occur in the initialization of the for statement and the scope of the declared variables are the remaining expressions and the statement. We first build the union for requirements ($\uplus$) and generate the constraints of intersecting requirements ($\cap_c$) of the requirement sets obtained from the subexpressions and the substatement. We obtain a new requirement set, which contains all requirements from the scope of the variable declaration. We obtain also a set of constraints containing the type equality constraints for variables required in more than one set. We then generate the final requirement set with the requirement union of the requirements of the scope with the requirements from the declaration. This order is to ensure, that the newly declared variables are not used to initialize itself.
The next step is to remove all requirements that are now actually declared and to constrain the required type to be equal to the declared type of a variable. We remove all that requirements from the obtained set that are declared in the declaration set from the initialization part of the for statement. Since the elements of the declaration set are of the same form than the elements of the requirement set, we can reuse the $\cap_c$ operator to generate the needed equality constraints.
Since there can not flow any declarations from a subexpression or statement to

the statement enclosing this one, we generate an empty set for the declarations of the for statement. So we do not keep the declared variables from this statement, which means that we can currently not detect if one of these variables is declared by a statement enclosing the current one. This holds for all statements that can introduce new variable declarations.

$$\begin{aligned}
rqs_{dcl} &= \{x : T \mid (x : T) \in \underbrace{\Big(\biguplus_{i \in [0,n]} rqs_i \Big)}_{rqs_{scope}} \wedge x \notin dom(dcls_d)\} \\
rqs &= rqs_d \uplus rqs_{dcl} \\
cs_{new} &= \{T_0 \lessdot \{boolean, Boolean\}\} \\
cs_{merge} &= \Big(\bigcup_{i \in [0,n-1]} rqs_i \cap_c rqs_{i+1} \Big) \\
&\quad \cup \Big(\bigcup_{i \in [0,n-1] \cup \{d\}} dcls_i \cap_{decl} dcls_{i+1} \Big) \\
&\quad \cup (rqs_{scope} \cap_c dcls_d) \\
cs &= cs_{new} \cup cs_{merge} \cup \Big(\bigcup_{i \in [0,n] \cup \{d\}} cs_i \Big)
\end{aligned}$$

$$\text{For}\ \frac{dcl \vdash rqs_d \mid dcls_d \mid cs_d \mid clz_d \quad e_i : T_i \mid rqs_i \mid dcls_i \mid cs_i \mid clz_i,\ i \in [0,n]}{\text{for } (dcl;\ e_0;\ e_1, \cdots, e_n)\ stm \vdash rqs \mid \emptyset \mid cs \mid \bigcup_{i \in [0,n] \cup \{d\}} clz_i}$$

Example 5.21.
We typecheck the basic for statement:

```
for (int i=0; true; i++) ;
```

The statement declares a variable of type int and initialized it to zero. Since the condition is *true* it will repeat infinitely long, increasing each time the value of the variable by one. The body of the statement is the empty statement.

We start with the type checking of the subexpressions. We have a local variable declaration, the constant true, an increment and the empty statement. We then use the results of this to typecheck the for statement.

Since the variable declaration has an initialization, we use rule LocDclInit. The variable *i* is initialized to 0 and we use rule T-Int to obtain the type *int*. We then generate a new constraint $int \doteq int$ for the denoted type of the declaration and the type of the initialization and we add the declaration $i : int$. We can solve the constraint $int \doteq int$ and remain with an empty constraint set.

$$\text{LocDclInit}\ \cfrac{\text{T-Int}\ \cfrac{}{0 : int \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}}{\cfrac{\text{int } i = 0; \vdash \emptyset \mid \{i : int\} \mid \{int \doteq int\} \mid \emptyset}{\underbrace{\text{int } i = 0; \vdash \emptyset \mid \{i : int\} \mid \emptyset \mid \emptyset}_{dcl}}\ \emptyset}$$

We use rule T-True for the constant *true*, which results in the type *boolean* and no requirements, declarations, constraints, or class information.

$$\text{T-True}\ \frac{}{\underbrace{\mathit{true} : \mathit{boolean} \mid \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}_{\mathit{cond}}}$$

The subexpression of the increment is a variable and we use T-ExprName to typecheck it. We generate a fresh type variable α and require that the name i is bound to α.
We then use rule T-PostIncr to typecheck the increment with the result from the subexpression in the premise. We generate a fresh type variable β as the resulting type of the increment. We constrain β to be equal to the resulting type of the subexpression α and that β is some numeric type.
We solve the constraints and obtain the unifier $\{\beta/\alpha\}$. We substitute the unifier to the requirements to obtain $\{i : \beta\}$. We remain the constraint $\beta \epsilon T_{Num}$, because we can not solve it yet.

$$\text{T-PostIncr}\ \cfrac{\text{T-ExprName}\ \cfrac{}{i : \alpha \mid i : \alpha \mid \emptyset \mid \emptyset \mid \emptyset}}{\cfrac{i{+}{+} : \beta \mid i : \alpha \mid \emptyset \mid \{\beta \doteq \alpha, \beta \epsilon T_{Num}\} \mid \emptyset}{\underbrace{i{+}{+} : \beta \mid i : \beta \mid \emptyset \mid \{\beta \epsilon T_{Num}\} \mid \emptyset}_{\mathit{update}}}\ \{\alpha/\beta\}}$$

We use rule Empty for the empty statement.

$$\text{Empty}\ \frac{}{\underbrace{;\ \vdash \emptyset \mid \emptyset \mid \emptyset \mid \emptyset}_{\mathit{stm}}}$$

We have checked all subexpressions and statements and we use rule For to typecheck the for statement. We use the previous derivations as the premises of the rule.
We first generate the constraint $\mathit{boolean} \epsilon \{\mathit{boolean}, \mathit{Boolean}\}$ for the type of condition.
What is left to do, is to merge the requirements and declarations together. We build the union of all requirements from the premises which were not the variable declaration and obtain $\{i : \beta\}$. We then remove all newly declared variables from this set and since i is declared in the for statement, we obtain the empty requirements ($\emptyset$). Since all requirements except one are empty, we do not generate new equality constraints for variables that are required more than once, but we have to generate equality constraints for the variables required in the scope of the declaration. The union of requirements of the scope is $\{i : \beta\}$ and the declarations are $\{i : \mathit{int}\}$, so we generate the equality constraint $\beta \doteq \mathit{int}$. We build the union of the newly generated constraints and the constraints from the premises and obtain the constraints $\mathit{cs} = \{\beta \epsilon T_{Num}, \mathit{boolean} \epsilon \{\mathit{boolean}, \mathit{Boolean}\}, \beta \doteq \mathit{int}\}$.

$$\text{For}\ \frac{\mathit{dcl} \qquad \mathit{cond} \qquad \mathit{update} \qquad \mathit{stm}}{\text{for (int } i = 0\text{; true; } i{+}{+}\text{) ; } \vdash \emptyset \mid \emptyset \mid \mathit{cs} \mid \emptyset}$$

We have ended the type checking with a set of constraints which we need to solve. We can solve the constraint $\mathit{boolean} \epsilon \{\mathit{boolean}, \mathit{Boolean}\}$, because $\mathit{boolean} \in \{\mathit{boolean}, \mathit{Boolean}\}$. The next constraint we can solve is $\beta \doteq \mathit{int}$. We add the substitution β/int to the unifier and apply the substitution to the constraint $\beta \epsilon T_{Num}$. We obtain $\mathit{int} \epsilon T_{Num}$ which we solve next, because $\mathit{int} \in T_{Num}$. We have solved all constraints and obtained the unifiert $\{\beta/\mathit{int}\}$.

$$\dfrac{\dfrac{\dfrac{\{boolean \epsilon \{boolean, Boolean\}\} \cup \{\beta \epsilon T_{Num}, \beta \doteq int\} \mid \emptyset}{\{\beta \doteq int\} \cup \{\beta \epsilon T_{Num}\} \mid \emptyset}}{\{int \epsilon T_{Num}\} \mid \{\beta/int\}}}{\emptyset \mid \{\beta/int\}}$$

5.3.5 Blocks

A block is a sequence of statements within braces. This sequence will be executed from first to last. The side effects and variable declarations that occured in the execution of the first statements are visible in all statements that will be executed accordingly.

Since variable declarations are visible in statements that will be later executed in the sequence, we must respect this in the type rule.
We obtain all binding requirements, binding declarations, constraints and class information from the substatements and we have to merge them together in the right order, because each statement can be a local variable statement.
We process the sequence from last to first, because we process then the requirements of a variable before its declaration. We take the last result from the sequence and merge it to the forelast result, we then remove both of them and put the newly generated requirements and constraints in the place of them. We repeat this until we only have one result left, which will then be the result for the block. If the sequence of the block is empty, we generate an empty result for it.

Definition 5.22.
We define an operation $merge_{block}$. This operation processes a sequence of results $(res_1, \cdots, res_{n-1}, res_n)$ in the above described order. Each result res_i of this sequence consists of binding requirements (rqs_i), binding declarations $(dcls_i)$, constraints (cs_i) and class information (clz_i).
Each step of this merging includes the following operations.

- The union for requirements $rqs_{n-1} \uplus rqs_n$.
- The generation of equality constraints for variables that are required in both sets $(rqs_{n-1} \cap_c rqs_n)$.
- The generation of equality constraints for variables that are required in rqs_n which are declared in $dcls_{n-1}$. The declarations of $dcls_n$ are omitted, to detect self declarations. This is safe even for statements which are no local variable declarations, because the declarations of these statements are empty since they can not introduce new variables.

Afterwards we must remove all variables which are declared from the newly generated requirements, to keep just the requirements that are not already declared.

$$\begin{aligned}
merge_{block}() &:= \emptyset \mid \emptyset \mid \emptyset \mid \emptyset \\
merge_{block}(res) &:= res \\
merge_{block}(res_1, \cdots, res_n) &:= merge_{block}(res_1, \cdots, res_{n-2}, (rqs \mid dcls \mid cs \mid clz)) \\
& \quad rqs = rqs_{n-1} \uplus rqs_n \\
& \quad dcls = \emptyset \\
& \quad cs = (rqs_{n-1} \cap_c rqs_n) \cup (dlcs_{n-1} \cap_c rqs_n) \\
& \quad clz = clz_{n-1} \cup clz_n
\end{aligned}$$

We use this operation in the type rule for blocks. We do not have to generate additional constraints, as blocks have no further restrictions.

$$\text{Block}\ \frac{stm_i \vdash rqs_i \mid dcls_i \mid cs_i \mid clz_i \text{ for } i \in [1, n]}{\{stm_1 \cdots stm_n\} \vdash merge_{block}(rqs_i \mid dcls_i \mid cs_i \mid clz_i \text{ for } i \in [1, n])}$$

5.4 Fields

A field is a member of a class. They are variables which are globally visible in the entire class, so the scope of a field is the class in which it is declared. We assume all fields to be public. This means, that all fields are also visible from the outside of the class in which they were defined.

5.4.1 Field Access

A field access is an expression. We can access fields of objects or arrays, so we can only use field accesses on reference types. We have already seen that it is possible to refer to a field with a simple name and that we can not be sure if we access a field or a local variable when doing so. We will treat these field accesses with a simple name when we typecheck field declarations.

A field access has one subexpression and an identifier, separated with ".". The subexpression denotes an object or array. The identifier is the name of the field we want to access in this object or array. The type of the object or array must have a field with the denoted name.
Since a field is a special kind of member, we define a new class requirement for fields.

Definition 5.23.
For types T_1, T_2 and identifier x

$$T_1 \text{ hasField } x : T_2$$

requires that type T_1 has a field x with type T_2.

In the type rule for field access, we have to generate a $hasField$ requirement. We know from the syntax the name of the field we want to access and from the premise the type of the object that needs to have the field. What we do not know is the type of the field, so we generate a fresh type variable as the type of the field and generate a $hasField$ requirement for these information.

$$\text{T-Field}\ \frac{e : T \mid rqs \mid dcls \mid cs \mid clz \qquad \alpha \text{ fresh}}{e.x : \alpha \mid rqs \mid dcls \mid cs \mid \{T \text{ hasField } x : \alpha\} \cup clz}$$

We can also access fields from the class we inherited from with the special keyword *super*. Such a super field access has no subexpression. As the instance from which we want to access the field is already known, we do not get the type of the instance from the super field access expression.
We have already seen the special keyword *this* and we will treat the keyword *super* similar. We generate a fresh type variable and require *super* to be bound to that type variable. We adapt the previous type rule for field access. We interchange the type of the subexpression with the generated type variable for the keyword *super* in the $hasField$ requirement.

$$\text{T-SuperField}\ \frac{\alpha, \beta \text{ fresh}}{super.x : \beta \mid \{super : \alpha\} \mid \emptyset \mid \emptyset \mid \{\alpha \text{ hasField } x : \beta\}}$$

5.4.2 Field Declaration

A field declaration introduces a new variable of a class and we can not have two fields with the same name.
The syntax of a field declaration is similar to the syntax of a local variable declaration. A field declaration can define a visibility, but we assume all fields to be public visible. We further restrict field declarations that we can declare only one field per field declaration. Field declarations may be directly initialized. A field declaration must occur only in the body of a class declaration.
Fields belong to the public interface of a class, so we have to remember which fields are declared in a class. We already have a class information $hasField$, but we used this to require a field to be present in a class. We define a new class information for fields provided by a class.

Definition 5.24.
For types T_1, T_2 and identifier x

$$T_1 \text{ providesField } x : T_2$$

states that type T_1 provides a field x with type T_2.

We begin with the type rule for field declaration without an initialization. We have to generate an information stating that the current class provides a field with the denoted name and type. As we do not know the name of the class when typechecking a field declaration, we generate a fresh type variable and require it to be bound to *this* instance. We then use this type variable as the class type in the $providesField$ information.

$$\text{FieldDcl}\ \frac{clz = \{\alpha \text{ providesField } x : T\} \qquad \alpha \text{ fresh}}{\text{public } T\ x; \vdash \{this : \alpha\} \mid \emptyset \mid \emptyset \mid clz}$$

We adapt this for the type rule for field declarations with an initialization. We now have a subexpression whose type we constrain to be equal to the denoted type of the field declaration. We have to merge the requirements from the subexpression to the new requirement for *this*.

$$cs = \{T \doteq T_1\} \cup (\{this : \alpha\} \cap_c rqs_1) \cup cs_1$$
$$clz = \{\alpha \text{ providesField } x : T\} \cup clz_1$$

$$\text{FieldDclInit}\ \frac{e_1 : T_1 \mid rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1 \qquad \alpha \text{ fresh}}{\text{public } T\ x = e_1; \vdash \{this : \alpha\} \uplus rqs_1 \mid dcls_1 \mid cs \mid clz}$$

5.5 Constructors

Objects are instances of classes and we use constructors to create them. We have constructor declarations and instance creations.

5.5.1 Instance Creation

We can create instances of a class with the *new* keyword. The syntax of an instance creation is the keyword *new* followed by an identifier and a list of expressions. The identifier is the name of the class from which we want to create an instance. The class must have a constructor whose number and types of parameters match the length and types of the list of expressions.
We define a new class information for constructors.

Definition 5.25.
For a natural number n and types T, T_1, $\cdots$, T_n

$$T \text{ hasConstructor } [T_1, \cdots, T_n]$$

requires type T to have a constructor with n parameters with types T_1 to T_n.

We have to generate a *hasConstructor* requirement in the type rule for instance creation. The class the constructor must have is denoted, the number of parameters is the number of subexpressions and the types of the parameters are the types of the subexpressions.

$$\begin{aligned}
rqs &= \biguplus_{i \in [1,n]} rqs_i \\
cs_{merge} &= \Big(\bigcup_{i \in [1,n-1]} rqs_i \cap_c rqs_{i+1} \Big) \\
&\quad \cup \Big(\bigcup_{i \in [1,n-1]} dcls_i \cap_{decl} dcls_{i+1} \Big) \\
cs &= cs_{merge} \cup \Big(\bigcup_{i \in [1,n]} cs_i \Big) \\
clz &= \{C \text{ hasConstructor } [T_1, \cdots, T_n]\} \cup \Big(\bigcup_{i \in [1,n]} clz_i \Big)
\end{aligned}$$

$$\text{T-New}\ \frac{e_i : T_i \mid rqs_i \mid dcls_i \mid cs_i \mid clz_i}{\text{new } C(e_1, \cdots, e_n) : C \mid rqs \mid \bigcup_{i \in [1,n]} dcls_i \mid cs \mid clz}$$

5.5.2 Constructor Declaration

A constructor declaration has a head and a body. The head contains a modifier for the visibility of the constructor, the name of the current class and a parameter list. As we assume all constructors to be public, the access modifier will be always "public". The body of a constructor can contain an explicit constructor invocation and a block statement. An explicit constructor invocation can be the invocation of another constructor of the current class or the invocation of a constructor of the superclass.

We have to remember the declared constructors, to be able to check if a constructor that is required by an instance creation indeed exists. We define a new class information for the constructors provided by a class.

Definition 5.26.
For a natural number n and types T, $T_1, \cdots, T_n$

$$T \text{ providesConstructor } [T_1, \cdots, T_n]$$

states that the type T provides a constructor with n parameters with the types T_1 to T_n.

We have to consider two things in the type rule for constructor declarations. First, that the scope of the parameters of the constructor is the body of the constructor. Second, that the current class provides a constrcutor with the denoted number and types of parameters.

We start with the type rule for a constructor declaration without an explicit constructor invocation.
We create a new set of pairs for the parameters and each pair contains the name and the type of one parameter. We use this set as a declaration set and generate equality constraints for the intersecting variables in this set and the requirement set from the body. We further remove all requirements that are bound as a parameter.
For the information that the current class provides a new constructor, we generate a *providesConstrcutor* information. The type of the class is denoted as the name of the constructor and the types of the parameters are denoted in the parameter list. We further require the current instance to be bound to the denoted type of the class, because we can declare only a constructor with the same type as the current class.

$$\begin{aligned}
param &= \{x_i : T_i \mid i \in [1, n]\} \\
cs_{merge} &= (param \cap_c rqs_1) \cup (param \cap_{decl} dcls_1) \\
cs &= cs_{merge} \cup cs_1 \\
rqs &= \{x : T \mid (x : T) \in rqs_1 \land x \notin dom(param)\} \\
clz &= \{C \text{ providesConstructor } [T_1, \cdots, T_n]\} \cup clz_1
\end{aligned}$$

$$\text{Cons}\ \frac{stm \vdash rqs_1 \mid dcls_1 \mid cs_1 \mid clz_1}{\text{public } C(T_1\ x_1, \cdots, T_n\ x_n)\{stm\} \vdash \{this : C\} \uplus rqs \mid \emptyset \mid cs \mid clz}$$

We adapt this approach for the type rules for constructor declarations with an explicit constructor invocation. We have a constructor invocation of the current (*this*) or the super class in the body before the statement. This constructor invocation has a list of expressions providing the values of the parameters. These subexpressions are also in the scope of the parameters from the constructor declaration.
We begin with the type rule for an explicit invocation of a constructor from the current class. We modify the two steps from the previous typerule.
We have to require the existence of a constructor from the current class, with the appropriate number and types used in the invocation. Since we do not invoke the constructors with the class names, we generate a fresh type variable for the type of the class and require the current class to be bound to that type variable. We further have to build the requirements from the scope of the parameters, before we generate equality constraints for parameters whose variables are required in the scope.

$$\begin{aligned}
param &= \{x_k : T_k \mid k \in [1,n]\} \\
scope &= \biguplus_{l\in[0,m]} rqs_l \\
cs_{merge} &= (param \cap_c scope) \cup \Big(\bigcup_{l\in[0,m-1]} rqs_l \cap_c rqs_{l+1} \Big) \\
&\quad \cup (param \cap_{decl} dcls_0) \cup \Big(\bigcup_{l\in[0,m-1]} dcls_l \cap_{decl} dcls_{l+1} \Big) \\
cs &= cs_{merge} \cup \Big(\bigcup_{l\in[0,m]} cs_l \Big) \\
rqs &= \{this : \alpha\} \uplus \{x : T \mid (x : T) \in scope \wedge x \notin dom(param)\} \\
clz &= \{C \text{ providesConstructor } [T_1, \cdots, T_n] \\
&\qquad , \alpha \text{ hasConstructor } [S_1, \cdots, S_m]\} \cup \Big(\bigcup_{l\in[0,m]} clz_l \Big)
\end{aligned}$$

$$\text{ConsThis}\ \frac{e_j : S_j \mid rqs_j \mid dcls_j \mid cs_j \mid clz_j \quad stm \vdash rqs_0 \mid dcls_0 \mid cs_0 \mid clz_0}{\text{public } C(\underbrace{T_i\ x_i}_{i\in[1,n]})\{this(\underbrace{\ e_j\ }_{j\in[1,m]});\ stm\} \vdash rqs \mid \emptyset \mid cs \mid clz}$$

We repeat this for the type rule for the constructor declaration with an explicit constructor invokation from the superclass. We now have to require the fresh type variable to be bound to the superclass instead of the current one.

$$\begin{aligned}
param &= \{x_k : T_k \mid k \in [1, n]\}\\
scope &= \biguplus_{l \in [0,m]} rqs_l\\
cs_{merge} &= (param \cap_c scope) \cup \Big(\bigcup_{l \in [0,m-1]} rqs_l \cap_c rqs_{l+1} \Big)\\
&\quad \cup (param \cap_{decl} dcls_0) \cup \Big(\bigcup_{l \in [0,m-1]} dcls_l \cap_{decl} dcls_{l+1} \Big)\\
cs &= cs_{merge} \cup \Big(\bigcup_{l \in [0,m]} cs_l \Big)\\
rqs &= \{super : \alpha\} \uplus \{x : T \mid (x : T) \in scope \wedge x \notin dom(param)\}\\
clz &= \{C \text{ providesConstructor } [T_1, \cdots, T_n]\\
&\quad , \alpha \text{ hasConstructor } [S_1, \cdots, S_m]\} \cup \Big(\bigcup_{l \in [0,m]} clz_l \Big)
\end{aligned}$$

$$\text{ConsSuper}\ \frac{e_j : S_j \mid rqs_j \mid dcls_j \mid cs_j \mid clz_j \quad stm \vdash rqs_0 \mid dcls_0 \mid cs_0 \mid clz_0}{\text{public } C(\underbrace{T_i\ x_i}_{i \in [1,n]})\{super(\underbrace{\ e_j\ }_{j \in [1,m]});\ stm\} \vdash rqs \mid \emptyset \mid cs \mid clz}$$

5.6 Classes

A class declaration defines a new reference type. All classes, except Object, are extensions from a single other class.
Fields, methods and constructors are declared in the body of a class. The declared fields and methods are the members of that class. The scope of a member of a class is the entire class body.
We do not cover abstract classes, enums, interface implementations and generics. We assume all classes to be declared public.

The syntax of the head of a class declaration contains a visibility modifier, the name of the class and the name of the superclass. The name of the superclass is optional, but we assume it to be present at all times. This is no restriction, because each class without this annotation implicitly inherits from the class Object. Since we assumed all classes to be public, the visibility modifier will always be "public".
The body of a class consists of a list containing field declarations, method declarations, constructor declarations and also nested class declarations. Not all of these elements may be present in every class body and we do not cover method declarations.

We have to take multiple actions in the type rule for class declarations.
We gained knowledge of the current- and the superclass names, so we can resolve all requirements that we made to *this* and *super* in the body of the class declaration.
We are also at the point where we can merge class information from the body together. We have to bring the required fields together with the provided fields and we have to do this for the constructors as well.

We first show the basic type rule and proceed with the evolution of the resulting requirements, declarations, constraints and class informations.

$$\text{Class}\ \frac{\mathit{body}_i \vdash \mathit{rqs}_i \mid \mathit{dcls}_i \mid \mathit{cs}_i \mid \mathit{clz}_i \text{ for } i \in [1,n]}{\text{public class } C \text{ extends } E\{\ \mathit{body}_1, \cdots, \mathit{body}_n\ \} \vdash \mathit{rqs} \mid \mathit{dcls} \mid \mathit{cs} \mid \mathit{clz}}$$

For the references to *this* and *super*, we have to first merge the requirements from the scope of the class. The scope of the class is the body. We generate equality constraints for occurences of *this* and *super* with the requirements from the scope. Afterwards, we have to remove all requirements for both keywords, because we have found the declaration of *this* and the denoted type for *super*.

$$\begin{aligned}
\mathit{scope} &= \biguplus_{i\in[1,n]} \mathit{rqs}_i \\
\mathit{class} &= \{\mathit{this} : C, \mathit{super} : E\} \\
\mathit{cs}_{\mathit{class}} &= \mathit{class} \cap_c \mathit{scope} \\
\mathit{rqs}_{\mathit{class}} &= \{x : T \mid (x : T) \in \mathit{scope} \wedge x \notin \{\mathit{this}, \mathit{super}\}\}
\end{aligned}$$

For the merging of provided- and requested fields from the body, we have to generate equality constraints in a similar way as for the binding requirements and declarations. We have to constrain the types of field names that are both, provided and requested, to be equal, because each field can only have one type. If a name is both, requested and provided, as a field of the same type, then we constrain the types of the names to be equal. We remember that we can access fields from the current class also with a simple name and that we were not able to distinguish the field access with a simple name from the access to a local variable. Since we are now on the class level, each remaining requirement must be a field access, because we have no local variables directly in a class declaration. So we add a field requirement to the current class for each unresolved binding requirement. We remove all field requirements from the current class afterwards.

$$\begin{aligned}
\mathit{clz}_{\mathit{body}} &= \bigcup_{i\in[1,n]} \mathit{clz}_i \\
\mathit{fields} &= \{C \text{ hasField } x : t \mid (x : t) \in \mathit{rqs}_{\mathit{class}}\} \\
\mathit{cs}_{\mathit{fields}} &= \{T_1 \doteq T_2 \mid (T \text{ hasField } x : T_1) \in (\mathit{clz}_{\mathit{body}} \cup \mathit{fields}) \\
&\qquad \wedge (T \text{ providesField } x : T_2) \in \mathit{clz}_{\mathit{body}}\} \\
\mathit{clz}_{\mathit{fields}} &= \{c \mid c \in \mathit{clz} \wedge c \neq (C \text{ hasField } x : T)\}
\end{aligned}$$

We repeat this step for constructors, but we have to constrain the types of the parameters to be equal. Since constructors do not have names, we can only differ between constructors by the class to which they belong and the number of parameters they got.

$$\begin{aligned}
\mathit{cs}_{\mathit{cons}} &= \{T_i \doteq S_i \mid (T \text{ hasConstructor } [T_1, \cdots, T_n]) \in \mathit{clz}_{\mathit{fields}} \\
&\qquad \wedge (T \text{ providesConstructor } [S_1, \cdots, S_n]) \in \mathit{clz}_{\mathit{fields}} \\
&\qquad \wedge i \in [1, n]\} \\
\mathit{clz}_{\mathit{cons}} &= \{c \mid c \in \mathit{clz}_{\mathit{fields}} \wedge c \neq (C \text{ hasConstructor } [T_1, \cdots, T_n])\}
\end{aligned}$$

We have resolved the *this* and *super* references and we have generated constraints for the fields and the constructors. We now can bring these information together and merge the remaining results from the body.
Since we treated all unresolved binding requirements as a field access, the resulting binding requirements set is empty. This means that we can not use the binding requirements to detect unbound variables, but we can use the unresolved *hasField* information to detect unbound fields.
We generate constraints for the requirements from the body, like we have done it in all type rules so far. We build the union of these constraints with the constraints generated in the previous steps.

$$\begin{aligned}
rqs &= \emptyset \\
dcls &= \bigcup_{i \in [1,n]} dcls_i \\
cs_{merge} &= \Big(\bigcup_{i \in [1,n-1]} rqs_i \cap_c rqs_{i+1} \Big) \\
cs &= cs_{class} \cup cs_{fields} \cup cs_{cons} \cup cs_{merge} \\
&\quad \cup \Big(\bigcup_{i \in [1,n]} cs_i \Big) \\
clz &= clz_{cons}
\end{aligned}$$

We have merged all information together and we obtain *rqs*, *dcls*, *cs*, and *clz* as the result of the type rule. We have already noted, that the requirements *rqs* are empty and we need to use the remaining field requirements from *clz* to detect unbound (global) variables.
We can get the set of unbound variables, by filtering the remaining *hasField* information of the current class.

Definition 5.27.
The set of unbound variables of a class C, can be determined from the resulting class information *clz* of the typechecking of C.

$$unbound = \{x : T \mid (C \text{ hasField } x : T) \in clz\}$$

Note that the set *unbound* includes not only the names of the unbound variables, but also the types from the context they are used.
We can further assume the resulting set *dcls* to be empty, otherwise we would have some invalid syntax, because we have used a local variable declaration outside of a block- or a for statement and the syntax does not allow such a usage.

5.7 Properties of the typing relation

If we have typechecked a class declaration and the set *unbound* is empty, we obtain the following knowledge about the program.

- The public interface of the class we have typechecked.

- Minimal requirements to the public interfaces of classes used in the body of the class we have typechecked.

The public interface of the class we have typechecked, consists of all fields and constructors of that class. These information are preserved in the type rule for class declarations *Class*.
If we typecheck a class C, we have to filter all provided fields and constructors of class C from the class information we obtain with the rule *Class*

Example 5.28.
We typecheck class C with the following declaration.

```
public class C{
  int i;
  float j;

  public C(int i, float j){
    this.i = i;
    this.j = j;
  }
}
```

We obtain the following class information from rule Class

$$\begin{aligned}\{\ & C \text{ providesField } i : int,\\ & C \text{ providesField } j : float,\\ & C \text{ providesConstructor } [int, float]\ \}\end{aligned}$$

which is the public interface of class C.

We get the minimum requirements for other classes in a similar way. Instead of filtering the class information for the provided fields and constructors of the typechecked class C, we have to filter for the required fields and constructors of the other classes.

Example 5.29.
We typecheck class E with the following declaration.

```
public class E extends C{
  public E(int i, float j){
    super(i, j);
  }
}
```

We obtain the following class information from rule Class

$$\begin{aligned}\{\ & E \text{ providesConstructor } [int, float],\\ & C \text{ hasConstructor } [int, float]\ \}\end{aligned}$$

The public interface of E is $\{E$ providesConstructor $[int, float]\}$ and the minimum requirements to the public interface of C is $\{C$ hasConstructor $[int, float]\}$. We can validate that the previous shown class C indeed fullfills these requirements.

We can also use intermediate unification for Java. It may happen that we can not solve all the constraints generated so far, but we can distinguish between unsolvable constraints and constraints that may be solvable with future knowledge. If we find an unsolvable constraint, we have found an type error. If we find a constraint for which we can not decide at the moment if it is solvable or not, we skip it in the unification process and keep it for future solving.

5.8 Implementation of the typing relation

An implementation of the presented typing relation for Java can be found on Github in the JavaLang repository[1].

[1]`https://github.com/gnush/JavaLang`

Chapter 6

Related Work

Modular Specification and Dynamic Enforcement of Syntactic Language Constraints when Generating Code
In [EVMV14] Erdweg et al. propose dynamic checking of language-specific invariants on generated code at construction time. They introduce typesmart constructors to dynamically enforce the well-formedness of generated code according to the syntax of the language. Typesmart constructors can be used with any metaprogramming system and typesmart constructors can be derived automatically from the syntax definition of a language.

Erdweg et al. also integrated support for typesmart constructors into the runtime system of Stratego, allowing the direct usage of typesmart constructors in place of regular constructors. This achieves the global invariant that all generated code is well-formed, without the need of adapting the transformations to explicitly use typesmart constructors.

Polymorphic Bytecode: Compositional Compilation for Java-like Languages
Ancona et al. define in [ADDZ05] compositional compilation as the ability to compile source code fragments in isolation for Java-like languages. A polymorphic form of bytecode is introduced to obtain compositional compilation for Java. Polymorphic bytecode contains type variables and is equipped with a set of constraints on those type variables. Standard bytecode can be obtained by substituting the type variables with class names satisfying the coressponding constraints.
They develop a typing and linking algorithm to illustrate compositional compilation for Java. In the typing algorithm, a class is compiled in isolation, generating polymorphic bytecode for the class and constraints in the depending classes. The linking algorithm either produces standard bytecode from a collection of polymorhic bytecode fragments or fails.

Chapter 7

Summary

7.1 Conclusion

In this thesis we have presented a way to eliminate the context from a type system, in order to ensure that all information flows bottom up.

We have introduced binding requirements as the opposite to a typing context. Where a typing context provides information about the types of variables, the binding requirements were used to ask the variables to be "later" bound to a specific type. We have used type variables as placeholders, when we have not known the specific type of a variable or expression. We have used constraints on these type variables for unification.

We have used these tools to define bottom-up typing relations for the Simply Typed Lambda Calculus, PCF and a subset of Java.

7.2 Future Work

We have two main goals for the future. First, we want to evaluate the performance of the Stratego implementation of the bottom-up typechecker for PCF with a Stratego implementation of a regular typechecker with context, as well as implementing a parallelized version. Second, we want to extend the typing relation for Java. Possible extensions are method invocations, method declarations, interfaces and static fields as well as allowing more than the *public* access modifier.

Furthermore, we could try to generalize the presented approach to transform a given typing relation with context automatically into a bottom-up typing relation.

Bibliography

[ADDZ05] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: Compositional compilation for java-like languages. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 26–37, New York, NY, USA, 2005. ACM. URL: `http://doi.acm.org/10.1145/1040305.1040308`, `doi:10.1145/1040305.1040308`.

[EVMV14] Sebastian Erdweg, Vlad Vergu, Mira Mezini, and Eelco Visser. Modular specification and dynamic enforcement of syntactic language constraints when generating code. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 241–252, New York, NY, USA, 2014. ACM. URL: `http://doi.acm.org/10.1145/2577080.2577089`, `doi:10.1145/2577080.2577089`.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf&mdash;reference manual&mdash;. *SIGPLAN Not.*, 24(11):43–75, November 1989. URL: `http://doi.acm.org/10.1145/71605.71607`, `doi:10.1145/71605.71607`.

[KV10] Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. ACM. URL: `http://doi.acm.org/10.1145/1869459.1869497`, `doi:10.1145/1869459.1869497`.

[Mit96] John C. Mitchell. *Foundations of Programming Languages.* MIT Press, Cambridge, MA, USA, 1996.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, USA, 2002.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965. URL: `http://doi.acm.org/10.1145/321250.321253`, `doi:10.1145/321250.321253`.

[Vis01] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, RTA '01, pages 357–362, London, UK, UK, 2001. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=647200.718711`.

Appendix A

Type system implementations

A.1 PCF

A.1.1 Language Specification

```
module TypedLambda
imports Common

exports
  context-free start-symbols
    Module

  lexical syntax
    "False"  -> ID {reject}
    "True"   -> ID {reject}
    "and"    -> ID {reject}
    "module" -> ID {reject}
    "Bool"   -> ID {reject}
    "Num"    -> ID {reject}
    "if"     -> ID {reject}
    "then"   -> ID {reject}
    "else"   -> ID {reject}
    "fix"    -> ID {reject}

  context-free syntax
    "module" ID {Expr ","}*
        -> Module {cons("Module")}
    ID
        -> Expr   {cons("Var")}
    Expr Expr
        -> Expr   {cons("App"), left}
    "\\" ID ":" Type "." Expr
        -> Expr   {cons("Abs")}
    "True"
        -> Expr    {cons("True")}
    "False"
        -> Expr    {cons("False")}
    "~" Expr
        -> Expr    {cons("Not")}
    Expr "and" Expr
        -> Expr    {cons("And"), left}
    INT
        -> Expr    {cons("Num")}
    Expr "+" Expr
        -> Expr    {cons("Add"), left}
    "if" Expr "then" Expr "else" Expr
        -> Expr    {cons("Cond")}
    Expr ">" Expr
        -> Expr    {cons("Gt"), left}
    "fix" ID ":" Type "." Expr
        -> Expr    {cons("Fix")}
    ID ":" Type "=" Expr ";" Expr
        -> Expr    {cons("Let")}
    "(" Expr ")"
        -> Expr    {bracket}
    "Bool"
        -> Type    {cons("Bool")}
    "Num"
        -> Type    {cons("TNum")}
    Type "->" Type
        -> Type    {cons("Function"), right}

  context-free priorities
    { ID -> Expr } >
    { "~" Expr -> Expr } >
    { Expr "and" Expr -> Expr
      Expr "+" Expr -> Expr
      Expr ">" Expr -> Expr } >
    { "\\" ID ":" Type "." Expr -> Expr
      "fix" ID ":" Type "." Expr -> Expr
      ID ":" Type "=" Expr ";" Expr -> Expr } >
    { Expr Expr -> Expr } >
    { "if" Expr "then" Expr "else" Expr -> Expr }
```

A.1.2 Helpers

```
module typecheckBase

imports
    include/TypedLambda

signature constructors
    CEq : Type * Type * String -> Constraint
    TVar       : STRING -> Type

rules
    // applies an mgu to a type
    // (t, mgu) -> t
    app_mgu: (t, []) -> t
    app_mgu: (t, [(x, t')|mgu])
        -> <app_mgu> (t1, mgu)
        where t1 := <tSubst> (x, t, t')

    // substitute x in t with t'
    tSubst: (x, Bool(), t') -> Bool()
    tSubst: (x, TNum(), t') -> TNum()
    tSubst: (x, Function(t1, t2), t') ->
        Function(t1', t2')
        where t1' := <tSubst> (x, t1, t');
              t2' := <tSubst> (x, t2, t')
    tSubst: (x, TVar(x), t') -> t'
    tSubst: (x, TVar(y), t') -> TVar(y)

    // if var occurs in term true, else false
    occurs: (var, Bool())     -> <fail>
    occurs: (var, TNum())     -> <fail>
    occurs: (var, TVar(var)) -> <id>
    occurs: (var, TVar(_))    -> <fail>
    occurs: (var, Function(t, t'))
        -> <if <occurs> (var, t)
             then id
             else <occurs> (var, t') end>

    // union of binding requirements
    bUnion: (xs, ys) -> <bUnion> (<conc> (xs, ys))
    bUnion: [] -> []
    bUnion: [(x, t)|bs] -> [(x, t)|<bUnion> bs']
        where bs' := <filter(not(?(x, _)))> bs
    bUnion = debug(!"bUnion: "); fail

    // forall x in list. (elem, x)
    mk-tuples: (elem, list)
        -> <map(\ x -> (elem, x) \)> list

    // list of bindings -> list of tuples of bindings
    mk-isectCallList: [] -> []
    mk-isectCallList: [x|xs]
        -> <conc> (<mk-tuples> (x, xs),
                   <mk-isectCallList> xs)

    // intersecting binding requirements
    isectConstraints: ([], ys) -> []
    isectConstraints: ([(x, t)|xs], ys)
```

```
        -> <union> (c, <isectConstraints> (xs, ys))
        where c := <filter(?(x, _) ; \ (_, t')
            -> <mk-err> (t, t',
                <conc-strings> ("binding ", x)) \)> ys

    constraints: (cs, _)       -> cs
    constraints: (cs, _, _, _) -> cs
    bindings: (_, bs)       -> bs
    bindings: (_, bs, _, _) -> bs
    mgu: (_, _, mgu, _) -> mgu
    errs: (_, _, _, errs) -> errs
    filter-bindings = map(bindings)
    filter-constraints = map(constraints)
    filter-mgu = map(mgu)
    filter-errs = map(errs)

    // type, type, string -> CEq
    mk-err: (expected, actual, node)
        -> CEq(expected, actual, err)
        where err := <concat-strings> ["Expected ",
                                        <type-to-string> expected,
                                        " but got ",
                                        <type-to-string> actual,
                                        " in ", node]

    type-to-string: Bool() -> "Bool"
    type-to-string: TNum() -> "Num"
    type-to-string: Function(t1, t2)
        -> <concat-strings> ["(", <type-to-string> t1, " -> ",
                                  <type-to-string> t2, ")"]
    type-to-string: TVar(t) -> t

    // list of tuples -> string
    mk-unbound-vars-message: bindings
        -> <conc-strings>  ("Unbound variables",
          <foldl(\ (x, xs)
           -> <concat-strings> [xs, " ", <Fst> x] \)> (bindings, ""))
```

A.1.3 Constraint generation

```
module typecheckIntermediate

imports
  include/TypedLambda
  semantic/typecheckBase

rules
  typeinfer: Module(x, e*) -> <map(typeinfer)> e*
  typeinfer: e -> <if equal(|<length> b, 0)
    then if equal(|<length> errs, 0)
```

```
          then !(t, c, b, mgu)
          else !errs
        end
    else ![<mk-unbound-vars-message> b|errs] end>
    where (t, (c, b, mgu, errs))
      := <bottomup(try(generateConstraints))> e;
      c := <map(\ CEq(t1, t2, err)  // map mgu
          -> CEq(<app_mgu> (t1, mgu), // over cs
                 <app_mgu> (t2, mgu), err) \)> c

gen = bottomup(try(generateConstraints))

generateConstraints: True()
  -> (Bool(), ([], [], [], []))
generateConstraints: False()
  -> (Bool(), ([], [], [], []))
generateConstraints: Num(_)
  -> (TNum(), ([], [], [], []))

generateConstraints: Not((t, (c, b, mgu, err)))
  -> (Bool(), (cs, b, mgu, <union> (err, errs)))
  where (cs, mgu, errs) :=
    <unify> ([CEq(t, Bool(), "Not arg")|c], [], mgu, []);
    b := <map(\ (x, t) -> (x, <app_mgu> (t, mgu)) \)> b


generateConstraints: And((t1, c1), (t2, c2))
    -> (Bool(), cRes)
    where cs := [<mk-err> (Bool(), t1, "And 1st arg"),
                 <mk-err> (Bool(), t2, "And 2nd arg")];
          cRes := <mergeUnify> ([c1, c2], cs)

generateConstraints: Add((t1, c1), (t2, c2))
  -> (TNum(), cRes)
  where cs   := [<mk-err> (TNum(), t1, "Add 1st arg"),
                 <mk-err> (TNum(), t2, "Add 2nd arg")];
        cRes := <mergeUnify> ([c1, c2], cs)

generateConstraints: Gt((t1, c1), (t2, c2))
  -> (Bool(), cRes)
  where cs   := [<mk-err> (TNum(), t1, "Gt 1st arg"),
                 <mk-err> (TNum(), t2, "Gt 2nd arg")];
        cRes := <mergeUnify> ([c1, c2], cs)

generateConstraints:
  Cond((tCond, cCond), (tThen, cThen), (tElse, cElse))
  -> (tRes, cRes)
  where  c1 := <mk-err> (Bool(), tCond, "condition of If");
    c2 := <mk-err> (tThen, tElse, "branch of If");
    cRes := <mergeUnify> ([cCond, cThen, cElse], [c1, c2]);
    tRes := <app_mgu> (tThen, <mgu> cRes)

generateConstraints: Var(x) -> (ty, ([], [(x, ty)], [], []))
  where ty := TVar(<newname> "T")

generateConstraints: App((t1, c1), (t2, c2)) -> (tRes, cRes)
  where t     := TVar(<newname> "T");
    cs    := [<mk-err> (t1, Function(t2, t), "application")];
    cRes := <mergeUnify> ([c1, c2], cs);
    tRes := <app_mgu> (t, <mgu> cRes)

generateConstraints: Abs(x, t1, (t2, c))
  -> (Function(t1, t2), cRes)
  where cRes := <mergeUnify> ([c], [], (x, t1));
    t1   := <app_mgu> (t1, <mgu> cRes);
    t2   := <app_mgu> (t2, <mgu> cRes)

generateConstraints: Fix(x, t1, (t2, c)) -> (t1, cRes)
  where cRes := <mergeUnify> ([c], [], (x, t1));
         t1   := <app_mgu> (t1, <mgu> cRes)

generateConstraints: Let(x, t, (t1, c1), (t2, c2)) -> (t2, cRes)
  where cs    := [<mk-err> (t, t1, "binding of Let")];
        cRes := <mergeUnify> ([c1, c2], cs, (x, t));
        t2    := <app_mgu> (t2, <mgu> cRes)

unify: ([CEq(t, t, _)|cs], mCs, mgu, errs)
  -> <unify> (cs, mCs, mgu, errs) // trivial
unify: ([CEq(Function(t1, t1), Function(t2, t2), err)|cs],
          mCs, mgu, errs)
  -> <unify> (cs, mCs, mgu, errs) // decompose
  where cs := <concat> [[CEq(t1, t2, err), CEq(t1, t2, err)], cs]

unify: ([CEq(Bool(), Function(_, _), err)|cs], mCs, mgu, errs)
  -> <unify> (cs, mCs, mgu, [err|errs]) // clash
unify: ([CEq(Function(_, _), Bool(), err)|cs], mCs, mgu, errs)
  -> <unify> (cs, mCs, mgu, [err|errs])
unify: ([CEq(TNum(), Bool(), err)|cs], mCs, mgu, errs)
  -> <unify> (cs, mCs, mgu, [err|errs])
unify: ([CEq(Bool(), TNum(), err)|cs], mCs, mgu, errs)
  -> <unify> (cs, mCs, mgu, [err|errs])
unify: ([CEq(Function(_, _), TNum(), err)|cs], mCs, mgu, errs)
  -> <unify> (cs, mCs, mgu, [err|errs])
unify: ([CEq(TNum(), Function(_, _), err)|cs], mCs, mgu, errs)
  -> <unify> (cs, mCs, mgu, [err|errs])

unify: ([c@CEq(TVar(x), TVar(y), err)|cs], mCs, mgu, errs)
  -> <unify> (cs, [c|mCs], mgu, errs) // skip, for intermediate

unify: ([CEq(t, TVar(x), err)|cs], mCs, mgu, errs)  // orient
```

```
  -> <unify> ([CEq(TVar(x), t, err)|cs], mCs, mgu, errs)

unify: ([CEq(TVar(x), t, err)|cs], mCs, mgu, errs)
  -> <if <occurs> (x, t) // variable elimination
         then <unify> (cs, mCs, mgu, [err|errs])
         else <unify> (cs, mCs, mgu, errs)
       end>
where cs  := <map(\ CEq(t1, t2, err)
    -> CEq(<tSubst> (x, t1, t),
           <tSubst> (x, t2, t), err) \)> cs;
  mCs := <map(\ CEq(t1, t2, err)
    -> CEq(<tSubst> (x, t1, t),
           <tSubst> (x, t2, t), err) \)> mCs;
  mgu1 := <map(\ (y, t1)-> (y, <tSubst> (x, t1, t)) \)> mgu;
  mgu := [(x, t)|mgu1]

unify: ([], mCs, mgu, errs) -> (mCs, mgu, errs)

// list, list -> resultSet
mergeUnify: (c, cs) -> (constraints, bindings, mgu, errs)
  where bindings := <filter-bindings; flatten-list; bUnion> c;
    constraints := <conc> (cs, <filter-bindings;
      mk-isectCallList; map(isectConstraints); flatten-list> c);
    errors := <filter-errs; flatten-list> c;
    (constraints, mgu, errs) :=
      <unify> (constraints, [], [], errors);
    bindings :=
      <map(\ (x, t) -> (x, <app_mgu> (t, mgu)) \)> bindings

// list, list, (var, type) -> resultSet
mergeUnify: (c, cs, (x, t))
  -> (constraints, bindings, mgu, errs)
  where cNew := ([], [(x, t)], [], []);
    c    := [cNew|c];
    bindings     := <filter-bindings; flatten-list; bUnion> c;
    bindings    := <remove-all(?(x, _))> bindings;
    constraints := <conc> (cs, <filter-bindings;
      mk-isectCallList; map(isectConstraints); flatten-list> c);
    errors       := <filter-errs; flatten-list> c;
    (constraints, mgu, errs) :=
      <unify> (constraints, [], [], errors);
    bindings :=
      <map(\ (x, t) -> (x, <app_mgu> (t, mgu)) \)> bindings
```