Type inference

Main seminar, summer term 2011

Sebastian Wiesner

June 10th, 2011

Type inference refers to a variety of techniques to deduce the types of expressions. This article intends to give an overview over type inference techniques, which are established in different programming languages. It starts with a short introduction, which elaborates on the motivation to include type inference in whatever form into a programming language. The next section covers different type inference algorithms: We start with an introduction to type inference by studying the primitive type inference in C#. Following this we discuss the classic Hindley-Milner algorithm for complete type inference as found in Haskell or the ML family of languages. Finally we sketch the more advanced local type inference found in Scala. The article concludes with a discussion of two drawbacks of type inference itself and of certain implementations of type inference.

1 About types, and the need to infer them

The world of programming languages is very diverse, with many conflicting paradigms and concepts coexisting in different programming languages. One concept however is common to almost all programming languages: The concept of *types* and *type systems*. Though the actual implementations greatly vary, almost any language has a type system, and attaches different types to identifiers and expressions, thus classifying identifiers and expressions by the kind of values that are bound to identifiers or computed by expressions. This leads to the definition of *type systems* as given by Pierce [Pie02]:

Definition 1 *A* type system *is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. [Pie02, section 1.1]*

With types being omni-present in a wide variety of programming languages, programmers are somewhat used to types. Interestingly, they are also used to *explicitly* annotate phrases with types, even in situations in which explicit annotations are actually superfluous, because the type could easily be inferred from the context of a phrase. A simple example for such a situation is the following listing, which shows typical variable declarations in Java:

```
int k = 42;
int l = k + 1;
ArrayList<Double> L = new ArrayList<Double>();
```

Obviously, the annotations on the left hand side of the assignment expressions are unnecessary, because the type of the identifier can trivially be inferred from the type of the expression on the right hand side of the assignment expression. Such superfluous type annotations can consequently be omitted without loosing any type information[1]:

```
auto k = 42;
auto l = k + 1;
auto L = new ArrayList<Double>();
```

To support this style of programming, the type checker must be able to *infer* the type of variables from the right hand side of the assignment automatically. In other words, the type checker has to *reconstruct* type information for phrases without explicit type annotations from the context of the phrase. More formally:

Definition 2 *The computation of type information for phrases without explicit type annotations is called* Type inference *or* type reconstruction*.*

The advantage of type inference What is the advantage of type inference? There are basically two reasons to include type inference in a type-checker: [Pot01]:

- As can be seen in the previous examples, type inference saves the tedious task of adding type annotations to every identifier in a program.
- Type inference is also a sort of program analysis, whose results can for instance drive compiler optimizations.

2 Fundamentals of type inference

In order to perform type inference, the type-checker needs formal rules on how to infer the type for each kind of phrase for which inference should be supported. These rules are called *inference rules*:

[1]This listing does not contain valid Java code anymore, as Java does not include any means of type inference. It merely serves as an example of what is possible in a language with type inference.

Definition 3 *An* inference rule*, as shown in equation 1, states, that if* all *of the* premises P_0 *to* P_n *hold, then also the* conclusion C *holds.*

$$\frac{P_0 \quad \dots \quad P_n}{C} \tag{1}$$

For type inference, these rules are defined over *terms.* A *term* is simply a semantic unit of the programming language in question. A term can contain *subterms*, as in `if` $\mathtt{t}_1$ `then` $\mathtt{t}_2$ `else` $\mathtt{t}_3$, where each of t_1, t_2 and t_3 refers to another term. The terms of a programming language generally arise from its formal grammar specification.

3 About different type inference techniques

Equipped with this theoretical background, we will now study three different type inference algorithms found in popular, practically used programming languages.

3.1 Locale type inference in imperative languages

We start with a closer look at the type inference mechanism provided by C# 3.0: *implicitly typed local variable declarations* [Mic07, section 8.5.1]. This inference technique is rather primitive, and uninteresting for any in-depth study of type inference, but serves quite well as a short and easy introduction to type inference rules.

Implicitly typed local variables are declared as follows in C#:

```
var i = 5;
var j = i + 5;
```

Listing 1: Implicity typed local variable declarations in C# 3.0

The general rule[2] is, that the C# type-checker can infer the type of the variable, if the right hand side is an expression with a compile-time type (which excludes "typeless" values like `null`). This corresponds to the following inference rule:

$$\frac{t\colon \tau}{x \leftarrow t\colon \tau} \tag{C-SimpleVarDecl}$$

What does this rule say? It tells the type-checker, that given a term t of type τ, we can infer, that the assignment expression $x \leftarrow t$ and thus the variable x is of type τ, too. It is important to understand that τ does not refer to a concrete type. It is a *type variable*, which stands for all permissible types. A type variable can then be *substituted* or *instantiated* with other types (which may again be type variables):

[2]The precise details may be looked up in section 8.5.1 of the C# language specifiction [Mic07]

Definition 4 *A* type substitution *σ is a finite mapping from type variables to types (which in turn may contain type variables). For instance $[X \mapsto T, Y \mapsto U]$ substitutes T for X and U for Y. We write σT to apply the substitution σ to the type T.*

The same type variable may appear on both side of clauses in σ. In such cases, all clauses are applied simultaneously*. For instance, $[X \mapsto Z, Y \mapsto X \to X]$ maps X to Z and Y to $X \to X$, not $Z \to Z$. [Pie02, section 22.1]*

Using this simple rule, we can infer the type of a declared variable, whenever we know the type of the right-hand side expression. This is the case for literals, whose types can be taken from the language specification. In the first declaration in listing 1, the right hand side is an integer literal of type `int`, so we substitute *int* for τ (written $[\tau \mapsto \texttt{int}]\tau = \texttt{int}$) and use the previously defined rule C-SimpleVarDecl:

$$\frac{\texttt{5}\colon \texttt{int}}{i \leftarrow \texttt{5}\colon \texttt{int}}$$

However, what to do, if the right hand side is a complex expression like an operator or a function application? In this case, the type of the right hand side expression cannot be taken from the specification, we have to take the *context* of the expression into account. To do so, we introduce a *type context* or *type environment* [Pie02, section 9.2]:

Definition 5 *A* type context *(or* type environment*) Γ is a* sequence *of variables and their corresponding types. We write $\Gamma, x\colon \tau$ to add a new binding for the variable x to type τ at the end of the context Γ.*

Now the rule C-SimpleVarDecl can be updated to include a type context:

$$\frac{\Gamma \vdash t\colon \tau}{\Gamma, x\colon \tau \vdash x \leftarrow t\colon \tau} \qquad \text{(C-VarDecl)}$$

It enables the type-checker to look up those types, which are necessary to compute the type of t[3]. Consequently it can now infer the type of assignments, even if the right hand side contains expressions, which are not merely literals:

$$\frac{\Gamma \vdash i + 5\colon \texttt{int}}{\Gamma, j\colon \texttt{int} \vdash j \leftarrow i + 5\colon \texttt{int}}$$

The type-checker of C# does not support type inference for any other term than assignment on local variables. Such a type-checker is said to perform *local type inference*:

Definition 6 *Given the set T of all terms of a programming language, type inference, which is only defined for a* real *subset of terms $T' \subset T$, is called* local *type inference. Type inference defined for all terms T is called* complete *type inference consequently.*

[3]In C# this is not done by inference, but simply by using the explicit type annotations and applying the rules of overloading in case of functions or operators.

Similar primitive techniques for local type inference can be found in other imperative languages too, for instance in C++, where the type of a template function is inferred from the type of arguments passed to it [Int03, section 14.8.2].

This simple kind of local type inference has no impact on the type system of a programming language, and can thus be added to almost any conventional, imperative programming language. It would for instance be a suitable addition to Java. If one is willing to accept greater impact on the type system, or even to design a type system for type inference, type inference can be a lot more powerful, and even be complete.

3.2 Complete type inference in functional languages

In this section we will discuss such an algorithm for complete type inference, called *Hindley-Milner type inference* after its inventors J. Roger Hindley and Robin Milner. It appeared first in Standard ML and now forms the base of other popular functional programming languages like Haskell or OCaml.

As said before, inference rules are defined over terms. To define a complete type inference algorithm for such languages, we obviously have to define their basic terms first. This is not very difficult as there are actually only five such terms:

$$\begin{array}{rll}
\texttt{exp} ::= & x & (<\textsc{Var}>) \\
\mid & \lambda x.\texttt{exp} & (<\textsc{Abs}>) \\
\mid & \texttt{exp}_1\ \texttt{exp}_2 & (<\textsc{App}>) \\
\mid & \texttt{let}\ x = \texttt{exp}_1\ \texttt{in}\ \texttt{exp}_2 & (<\textsc{Let}>) \\
\mid & \texttt{if}\ \texttt{exp}_1\ \textit{then}\ \texttt{exp}_2\ \textit{else}\ \texttt{exp}_3 & (<\textsc{If}>)
\end{array}$$

The above definition excludes literals and data type definitions. Such terms are uninteresting when reasoning about type inference, because literals and data constructor applications have an inherent type, which does not need to be inferred. Other terms bindings can be reduced to the above terms in the context of type inference. Top-level bindings for instance can be expressed as a combination of let expressions and lambda abstractions:

```
factorial n = if n == 0 then 1 else n * factorial (n-1)
-- is equivalent to
let factorial = \n.if n == 0 then 1 else n * factorial (n-1)
```

Infering types of recursive definitions When trying to define an inference rule for such recursive definitions, a problem arises: In order to infer the type of the newly bound identifier, we need to infer the type of the right-hand side expression, which in turn requires to infer a type for the newly bound identifier.

Hindley and Milner solved this problem by dividing type inference into two distinct steps. Instead of immediately inferring a concrete type, the inference rules merely generate a set of equations, which describe *constraints* for type variables. After inference is done for all terms, the accumulated constraints are solved to determine the actual types.

3.2.1 Constraint generation

To define constraint generating inference rules for the given terms, we first need a formal definition of what constraints actually are:

Definition 7 *A* constraint set *C is a set of equations $\{\tau_0 = \alpha_0, \dots, \tau_n = \alpha_n\}$. A substitution σ* unifies *an equation $\tau = \alpha$, if $\sigma\tau$ and $\sigma\alpha$ are identical. σ* unifies *C, if it unifies all equations in C. [Pie02, section 22.3]*

Based on this definition, we can now define inference rules for the terms <Var> and <Abs>:

$$\frac{x\colon \tau \in \Gamma}{\Gamma \vdash x\colon \tau \mid_{\emptyset} \{\}} \tag{C-Var}$$

$$\frac{\Gamma, x\colon \tau_1 \vdash t_2\colon \tau_2 \mid_{\chi} C}{\Gamma \vdash \lambda x.t_2\colon \tau_1 \to \tau_2 \mid_{\chi} C} \tag{C-Abs}$$

In these rules we write $\Gamma \vdash t\colon \tau \mid_{\chi} C$ to state, that t is of type τ in the context Γ, whenever the constraints C are satisfied. The χ subscript captures type variables introduced by inference rules. This allows us to impose restrictions upon the choice of type variables to make sure, that whenever we infer the type of an expression from more than one other term, the sub-inferences use a distinct set of type variables, and that newly introduced type variables do not conflict with any other type variable. The above rules leave the constraint set and the set of captured variables untouched, because using existing identifiers or creating abstractions does not impose any constraints or restrictions on type variables.

This is different for <If> and <App>. Both terms impose restrictions on the types of their sub-terms. In case of <If> the condition must be of boolean type, and the branches must be of the same type. In case of <App>, the first term must be of function type. Moreover both terms impose restrictions on the type variables, because they consist of multiple sub-terms. This is reflected by the inference rules C-If and C-App (with $FV(t)$ being the set of free variables in t):

$$\frac{\begin{array}{c}\Gamma \vdash t_1\colon \tau_1 \mid_{\chi_1} C_1 \qquad \Gamma \vdash t_2\colon \tau_2 \mid_{\chi_2} C_2 \qquad \Gamma \vdash t_3\colon \tau_3 \mid_{\chi_3} C_3 \\ \chi_1, \chi_2, \chi_3 \text{ nonoverlapping} \\ C' = C_1 \cup C_2 \cup C_3 \cup \{\tau_1 = \texttt{Bool}, \tau_2 = \tau_3\}\end{array}}{\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3\colon \tau_2 \mid_{\chi_1 \cup \chi_2 \cup \chi_3} C'} \tag{C-If}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1\colon \tau_1 \,|_{\chi_1}\, C_1 \qquad \Gamma \vdash t_2\colon \tau_2 \,|_{\chi_2}\, C_2 \\ \chi_1 \cap \chi_2 = \chi_1 \cap FV(\tau_2) = \chi_2 \cap FV(\tau_1) = \emptyset \\ \alpha \notin \chi_1, \chi_2, \tau_1, \tau_2, C_1, C_2, \Gamma, t_1, t_2 \\ C' = C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \rightarrow \alpha\}\end{array}}{\Gamma \vdash t_1\ t_2\colon \alpha \,|_{\chi_1 \cup \chi_2 \cup \{\alpha\}}\, C'} \quad \text{(C-App)}$$

The last rule is the rule for <Let>:

$$\frac{\begin{array}{c}\Gamma \vdash t_1\colon \tau_1 \,|_{\chi_1}\, C_1 \qquad \Gamma, x\colon \tau_g \vdash t_2\colon \tau_2 \,|_{\chi_2}\, C_2 \\ \tau_g = \mathit{generalize}(\Gamma, \tau_1, C_1) \\ \chi_1 \cap \chi_2 = \emptyset \qquad C` = C_1 \cup C_2\end{array}}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \,|_{\chi}\, C'} \quad \text{(C-Let)}$$

$t_g = \mathit{generalize}(\Gamma, \tau_1, C_1)$ means, that this rule generalizes the type of the binding expression t_1 and uses the generalized type to infer the type of the body t_2.

3.2.2 Generalization

During *generalization* (written as $\mathit{generalize}(\Gamma, \tau, C)$), the constraints in a given constraint set C are solved (*unified*). The result of applying the solution to the given type τ is then quantified with respect to the given type context Γ. The result is a *generalized principal type.*

Unification The first step of generalization is *unification* of the constraint set C (written $unify(C)$). The result of unification is a type substitution σ, computed as follows [Pie02, section 22.4]:

1. If C is empty, $\sigma = []$
2. Otherwise, with $C = C' \cup \{\tau = \tau'\}$ (again $FV(\tau)$ is the set of free variables in τ):
   a) $\tau = \tau'$: return $unify(C')$
   b) $\tau = \alpha$ and $\alpha \notin FV(\tau')$: return $unify([\alpha \mapsto \tau']C') \circ [\alpha \mapsto \tau']$
   c) $\tau' = \alpha$ and $\alpha \notin FV(\tau)$: return $unify([\alpha \mapsto \tau]C') \circ [\alpha \mapsto \tau]$
   d) $\tau = \tau_1 \rightarrow \tau_2$ and $\tau' = \tau_1' \rightarrow \tau_2'$: return $unify(C' \cup \{\tau_1 = \tau_1', \tau_2 = \tau_2'\})$

In all other cases, unification fails. If unification succeeds, the resulting type substitution σ is the *principal unifier*, which is the most general solution to the set of constraints.

This unifier is now applied to τ, yielding the *principal type* τ_p, which is consequently the most general type for a certain expression. Now all *bound* variables α_1 to α_n remaining in this principal type are *generalized* into a *type scheme* $\forall \alpha_1, \ldots, \alpha_n\colon \tau_p$. Free type variables, which are also contained in the type context Γ, are real constraints from the environment and must *not* be generalized.

The key difference between a simple type variable and a type scheme is that a type scheme can be *instantiated* with new, *fresh* type variables on every occurrence. This means, that a type scheme can result in two *different* simple types on two different occurrences.

Let-Polymorphism By generalizing the types of bound identifiers in C-Let, we can exploit this property of type schemes to add *parametric polymorphism* to the type system, that is reusing the same implementation of a function for different types. By simply instantiating a function's type scheme with different concrete types on each application, we can apply the function to many different concrete types. This style of polymorphism, called *Let-polymorphism*, is the base of the powerful generic function libraries provided by functional languages like Haskell.

3.2.3 Extensions to Hindley-Milner

This kind of type inference, and the type system it creates, can be extended with various other techniques, especially to support type inference for complex data structures and to add other kinds of polymorphism. A notable extension are *type classes* in the Haskell language [HHPJW96], which add *ad-hoc polymorphism*[4] to the type system. This is generally considered a success story.

The combination of *subtype polymorphism*, as implemented in common object-oriented languages like Java, and Let-polymorphism however did not yet succeed, though some interesting research has been done [Pot01]. A key problem is, that subtyping adds an ordering over types to define the subtype relationship between types. With such an ordering, the set of constraints does not longer consist of simple equations, which can be solved efficiently with unification algorithms, but instead a set of *inequalities.* Solving such inequalities requires at least cubic time. Moreover a principal unifier is not guaranteed to exists for constraint sets containing inequalities. Describing the set of valid types must consequently take the whole constraint set into account, which can grow complex, containing a lot auxiliary type variables. Such a system is hardly comprehendable to the programmer.

3.3 Advanced local type inference

For this reason, some languages, mainly Scala [CGLO06], abandon the idea of complete type inference, and instead only implement local type inference, however with the aim to provide the same level of comfort as languages with complete inference. This allows to combine subtype polymorphism with Hindley-Milner style type inference.

[4]Ad-hoc polymorphism basically refers to subsuming different implementations under the same identifier, and choosing the appropriate implementation for a given type.

The key insight is, that explicit type annotations are acceptable in some situations, either because they serve as useful documentation (like for top-level bindings[5]) or because a the situation only rarely occurs in practical use of a language. Based on this observation, one can figure out situations in which inference is mandatory or at least desirable [PT00]:

- Type arguments in applications of polymorphic functions must be inferred. Bound variables of top-level functions however are explicitly annotated, because in this case type annotations serve as useful documentation and are common style in languages with complete inference, too.
- Parameter types of anonymous functions should be inferred, if possible.
- Types of local bindings should be inferred.

While these needs obviously require a much more sophisticated type inference algorithm than the primitive one discussed in section 3.1, they are simple enough to avoid many of the problems mentioned in section 3.2.3.

3.3.1 Local type argument inference

With subtyping local type inference for polymorphic function has to take subtype constraints into account. To define a rule for this inference, we first need to define the *subtype relation*: We write $\tau_1 <: \tau_2$ to say, that τ_1 is a subtype of τ_2[6]. Based on this definition we can now define C-LocalApp:

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \forall \alpha_1, \ldots, \alpha_n : \tau_1 \to \tau_r \\ \sigma_\beta = [\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n] \qquad \sigma_\gamma = [\alpha_1 \mapsto \gamma_1, \ldots, \alpha_n \mapsto \gamma_n] \\ \Gamma \vdash t_2 : \tau_2 \qquad\qquad \tau_2 <: \sigma_\beta \tau_1 \\ \forall \gamma_1, \ldots, \gamma_n : (\tau_2 <: \sigma_\gamma \tau_1 \text{ implies } \sigma_\beta \tau_r <: \sigma_\gamma \tau_r)\end{array}}{\Gamma \vdash t_1\ t_2 : \sigma_\beta \tau_r} \quad \text{(C-LocalApp)}$$

This rule picks an argument of proper type τ_2 for the function t_1, which is subject to the following restrictions:

- $\tau_2 <: \sigma_\beta \tau_1$: The argument type must be a subtype of the function's parameter type.
- $\forall \gamma_1, \ldots, \gamma_n : (\tau_2 <: \sigma_\gamma \tau_1 \text{ implies } \sigma_\beta \tau_r <: \sigma_\gamma \tau_r)$: The arguments β_1 to β_n must be chosen, such that any other arguments γ_1 to γ_n satisfying the previous restriction must yield a return type, which is a supertype of the chosen return type. In other words, the argument type we chose must be the most specific, we can find.

This rule can be implemented using a constraint generation algorithm as described in [PT00, section 3.3].

[5] In fact, it is considered good style in the Haskell community to always use explicit type annotations for top-level bindings. The reference implementation Glasgow Haskell even warns on top-level bindings without explicit type annotations.

[6] The complete semantics of subtyping are explained in chapter 15 of [Pie02].

3.3.2 Bidirectional local type inference

Inferred types must be propagated down the syntax tree in order to infer local bindings and formal parameters. For instance, if we know $f\colon (\texttt{int} \to \texttt{int}) \to \texttt{int}$ (a higher-order function taking a function from `int` to `int`), then we should be able to infer $x\colon \texttt{int}$ in the application $f(\lambda x.x + 1)$ by propagating the type $\texttt{int} \to \texttt{int}$. This propagation is formalized in [PT00, section 4].

Together these two techniques form a system for local type inference, which is able to infer local types in many situations. It forms the base of type inference in Scala.

4 About the limits and problems of type inference

Generally, type inference greatly eases a programmer's life, avoiding the tedious repetition of superfluous type annotations. However, nothing comes for free, and there certainly are some drawbacks, in the general principle of type inference as well as in in the current state of the art implementations of type inference. Two of these, which we consider very important, are shortly discussed in this section.

As stated in section 1, type inference gives the freedom to omit type declarations. The advantage is, that the programmer can shift her attention from trying to please the compiler's type-checker to the algorithmic logic. A side-effect of this freedom is however, that it relieves the need to reason about the types used during programming. A programmer may find herself to write programs, that somehow "just work", without a precise idea of the types involved.

Error messages of type inference While this is fine, as long as the program actually works correctly, it gets problematic, if the programmer made a type error. In this case, the programmer's (probably only remote) idea of the types involved starts to diverge from what the compiler actually sees and infers. With explicit type annotations, this error is often caught by a near-by type annotation. Type inference can however proceed quite a while before actually failing to unify some constraints. The compiler's error message will then not point to the actual cause of the type error, but instead to the point, where unification failed, which can be wholly unrelated to the point of the actual error. The error message given by the compiler will consequently fail to precisely locate the cause of the type error, and may be completely misleading to the programmer, especially if she is a novice to the language in question. This is a field of ongoing research, for as of now common Haskell compiler still fail to provide satisfactory type error messages in many cases. An overview about the research on this topic, including some approaches to improve error messages, is given by [H$^+$05].

Complexity of type inference Another problem is the complexity of type inference algorithms, from the point of a programmer using a language. A programmer should have at least a remote idea of where type annotations can be omitted, because the compiler is able to infer them, and how these types annotations are then reconstructed. The latter immediately relates to the topic of error messages discussed before: A solid knowledge of the underlying type inference algorithm can significantly improve the ability to comprehend the compiler's error message. While the standard Hindley-Milner inference as discussed in section 3.2 is relatively easy to understand, local type inference techniques as described in section 3.3 can be somewhat complex, to the point, that it is non-trivial to tell, where the compiler is actually able to infer types. A tutorial of the Scala programming languages states [SH11, section 5]:

> The compiler is not always able to infer types like it does here, and there is unfortunately no simple rule to know exactly when it will be, and when not. [...] As a simple rule, beginner Scala programmers should try to omit type declarations which seem to be easy to deduce from the context, and see if the compiler agrees. After some time, the programmer should get a *good feeling* about when to omit types, and when to specify them explicitly.

Recommendations like this, caused by the high complexity of type inference rules, should be questioned, as they may lead to a kind of trial and error programming style, which can potentially *lower* the quality of the resulting program instead of *increasing* it (what should be the aim of any programming technique).

Despite these problems, type inference is – even in its primitive form as discussed in section 3.1 – a valuable addition to any programming language, because it combines the safety of statically checked type systems with the comfort of dynamically checked languages like Python, which do not annotate identifiers with types at all.

References

[BBJ02] G. Boolos, J.P. Burgess, and R.C. Jeffrey. *Computability and logic.* Cambridge University Press, 2002.

[CGLO06] V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for scala type checking. *Mathematical Foundations of Computer Science 2006*, pages 1–23, 2006.

[H$^+$05] B.J. Heeren et al. Top quality type error messages. *IPA Dissertation Series;*, 2005.

[HHPJW96] C.V. Hall, K. Hammond, S.L. Peyton Jones, and P.L. Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996.

[Int03] International Organization for Standardization, Geneva, Switzerland. *International Standard ISO/IEC 14882:2003: Programming languages: C++*, 2003.

[Mic07] Microsoft Corporation, Redmond. *C# Language Specification*, 3.0 edition, 2007.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[OZZ01] M. Odersky, C. Zenger, and M. Zenger. Colored local type inference. *ACM SIGPLAN Notices*, 36(3):41–53, 2001.

[Pie02] B.C. Pierce. *Types and programming languages.* The MIT Press, 2002.

[Pot01] F. Pottier. Simplifying subtyping constraints: a theory. *Information and Computation*, 170(2):153–183, 2001.

[PT00] B.C. Pierce and D.N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[SH11] M. Schinz and P. Haller. *A Scala tutorial for Java programmers*, 1.3 edition, 2011.