Acta Cybernetica — *online–first paper version* — pages 1–19.

Type Inference of Simple Recursive Functions in Scala

Gergely Nagy[a], Gabor Olah[a], and Zoltan Porkolab[a]

Abstract

Scala is a well-established multi-paradigm programming language known for its terseness that includes advanced type inference features. Unfortunately this type inferring algorithm does not support typing of recursive functions. This is both against the original design philosophies of Scala and puts an unnecessary burden on the programmer. In this paper we propose a method to compute the return types for simple recursive functions in Scala. We make a heuristic assumption on the return type based on the non-recursive execution branches and provide a proof of the correctness of this method. We implemented our method as an extension prototype in the Scala compiler and used it to successfully test our method on various examples. The algorithm does not have a significant effect on the compilation speed. The compiler extension prototype is available for further tests.

Keywords: Scala, type inference, recursion

1 Introduction

Scala is a well-established programming language providing both object-oriented and functional programming language elements. As a consequence, the language syntax needs to reflect both paradigms that results in a high level of expressiveness. Most of the new language properties are targeting the extensibility, safety and flexibility of the language. Examples for such features include advanced pattern matching, lambda expressions, by-name parameter passing and case classes. Improving the readability of the source code was a primary goal of language design; including the ability to avoid unnecessary boilerplate, repetitive code elements.

Terseness is important not only to make software code cleaner, but also to accentuate key parts of the solution expressed by the existing elements. Furthermore, the more automatically computed information the compiler can provide, the less possibly erroneous code snippets the developer writes.

[a]Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary. E-mail: {njeasus, olikas, gsd}@caesar.elte.hu. ORCID: https://orcid.org/{0000-0002-0736-8903, 0000-0001-5804-6448, 0000-0001-6819-0224}.

DOI: 10.14232/actacyb.282762

The Scala programming language is well-known for its terseness that includes advanced type inference features. These range from automatic deduction of variable types based on their initializations, to infer type parameters of generic functions based on call-side information.

Function return types are inferred for most trivial cases as well: the type of the lastly evaluated expression provides the return type of the function. In cases when multiple return statements are present the least upper bound (LUB) type of these will be used.

Since the subtyping relation ($<:$) is reflexive ($A <: A$), transitive ($A <: B \wedge B <: C \Rightarrow A <: C$) and antisymmetric ($A <: B \wedge B <: A \Rightarrow A = B$), it defines a *partial ordering* on the set of the types and thus this set has the *least upper bound* property. Scala supports all three kinds of variance, so it is up to the programmer to define how the type arguments affect the subtyping in case of generics.

However, if any of the lastly evaluated expressions in a function include a reference to the containing function, this algorithm cannot provide a meaningful result. This causes recursive functions without explicitly provided return types (such as on Figure }1) to fail the compilation process.

```
1    def factorial(n: Int) = n match {
2        case 0 => 1
3        case _ => n * fact(n−1)
4    }
```

Figure 1: A recursive function with no defined return type.

For any developer it is obvious that the `factorial` function will return an `Int`. Such similar simple recursive functions (see formal definition on Def. 3.1) occur frequently in most functional codebases. Not inferring their return type is both against the original design philosophies of Scala and puts an unnecessary burden on the programmer even in these simple cases.

```
1    abstract class Base
2    class Derived1 extends Base
3    class Derived2 extends Base
4    class Derived3 extends Base
5
6    def lousyType(d: Base, m: Double) = d match {
7        case _: Derived1 if m > 3 => new Derived3
8        case x: Derived2 if m < 2 => lousyType(x, m)
9    }
```

Figure 2: A recursive function with return types of a hierarchy.

Consider the example on Figure 2 that does not compile under current Scala type inference rules. In such situations the programmer may choose an unnecessarily wide type, such as the top type `Any`, corrupting the highly praised type system of Scala.

In this paper we propose a method to compute the return types for simple recursive functions. Similarly to the intentions of the developer, our heuristic assumption on the return type is based on the non-recursive execution branches. Assuming that the recursive functions will always end up in a non-recursive execution branch, we argue that the LUB of these branches provides a sufficient return type for the function. If this assumption is not met, our method reports the same error as the current Scala compiler.

To create a prototype implementation we have extended the `typer` not to immediately fail on recursive functions but use the proposed method to calculate the missing type. The extension does not have a significant effect on the compilation speed. We implemented our method as an extension prototype for the Scala compiler version 2.12.4 and used it to successfully test our method on various examples. The compiler extension prototype is publicly available for further tests.

This paper is structured as follows. In Section 2 we further evaluate the problem space with more examples and real-world issues. We provide the theoretical foundations of our mechanism in Section 3. We overview our results in Section 4 while also describing implementation details. Related works in Section 5 discusses similar problems in C++. Our paper concludes in Section 6.

2 Motivation

One of the main focuses of software development methodologies and practices nowadays is trying to lift off work and complexity of programmers as much as possible. This is achieved by various methods ranging from tooling and programming methods to language design. This trend is sensible with the spread of multiparadigm and especially functional languages which are usually tightly bound by their type systems. These type systems are mathematically proven to be correct and well-known algorithms were developed that can be used to prove if programs meet these bounds. The algorithms in question are built into compilers, so any programmer can easily check the correctness of their program. Being peer-reviewed and fully proven, one can trust that if the compiler finishes work on a piece of code it meets certain criteria. This should lead to fewer bugs and runtime errors in production software.

Scala is one of the more recent multiparadigm languages, that tries to solve a lot of complex problems before a developer meets them. The presence of this idea can be found in many of Scala's design goals, for example having as clean of a syntax and being as terse as possible [13]. This is achieved by introducing a fairly complex type inference system in the compiler, so programmers do not have to take time and effort to annotate their programs with types that can be deducted by an algorithm.

The type inference algorithm of Scala is far from complete though. It does not support any recursion, let it either be a simple recursive function, a recursive chain or a recursive type declaration. This can be surprising and frustrating to anyone writing Scala code, furthermore it can lead to non-trivial issues in one's code and later, a software product.

In the following we show a few examples where the lack of type inference on recursive functions may lead to possible runtime application errors.

```
1  def map[C, A <: C, B <: C](y: Seq[A], f: A => C) /*: Seq[C]*/ = {
2      y match {
3          case Nil => Seq[B]()
4          case x :: xs => map(xs, f) :+ f(x)
5      }
6  }
```

Figure 3: A recursive map implementation with explicit type annotation.

In our first example we start off with a higher-order function, `map` with our own interpretation. In the version that can be seen on Fig. 3, we leverage generic types as well as functions as first-class entities in Scala. We would always like to get the minimum type of the collection from this function, hence we are providing information on the relationships of the types. Unfortunately the Scala compiler is not much of a help here: it will throw an error when we try to call `map` with the remainder list. This becomes even more annoying when we provide an incorrect return type: the compiler will be able to recognize the error at the place of concatenating the computed element to the list. One can spend minutes on trying to find the type that makes the type system satisfied by recompiling several times, but it would be much more convenient if the compiler were able to find it for us at the very beginning.

```
1  class Level1
2  case object Class1Level1 extends Level1
3  case object Class2Level1 extends Level1
4  class Level2 extends Level1
5  case object Class1Level2 extends Level2
6  case object Class2Level2 extends Level2
```

Figure 4: A multi-level class hierarchy.

Let us consider another recursive method that computes its result that has one of the types of a multi-level class hierarchy, as seen on Fig. 4. An example method using these types can be found on Fig. 5. The Scala compiler will fail to infer the correct return type `Level1`, and will require the developer to define it for the function explicitly. Determining `Level1` as the return type is trivial in this case as we have listed all the types involved near the function definition in one place, but

```
1   def deepRec(n: Int): Level1 = {
2       if (n == 0) {
3           Class1Level2
4       }
5       else if (n == 1 || n == 2 || n == 3) {
6           n match {
7               case 3 => Class2Level2
8               case _ => {
9                   if (n != 1) {
10                      deepRec(n − 1)
11                  } else {
12                      Class2Level1
13                  }
14              }
15          }
16      } else if (n == 4) {
17          Class1Level2
18      } else {
19          Class1Level1
20      }
21  }
```

Figure 5: A recursive function using type hierarchy on Fig. 4. with explicit type annotations.

recognizing it when for example, class definitions are scattered in a fairly large and complicated framework can be challenging and time consuming even for seasoned developers.

Unfortunately, there is a pretty easy shortcut to make compile errors disappear in this case: define the return type as `Any` (the base type of all classes in Scala), making the type system and the compiler temporarily happy. As we all know, marking objects by the widest type is simply neglecting the type system, thus we are not using one of the main services offered by Scala.

3 Theoretical foundations

In this section we present the details of the theoretical background for our solution. We focus only on typing recursive functions, thus we do not detail typing e.g. objects.

Scala is a statically typed language, thus type checks happen at compile time. Type declarations can be omitted in the source in many places, and the compiler runs static type analysis to infer the types of variables, functions and other language elements. Scala compilation is designed as multi-staged process. In the first step

an AST of the program is constructed. Our main focus in this paper, typing, is executed as the third phase. Upon successful type inference, the abstract syntax tree is enriched with type information in this stage. Scala's type system contains such features that are not compatible with the Hindley-Milner type inference algorithm so it relies on one-directional, context-unaware local type inference.

3.1 Related works on type systems

Most statically typed languages such as C++ or Java require explicit type declarations (for later advances in C++, see Section 5). Other statically typed languages like Haskell or ML use static type inference to calculate types for functions. Some unification-based type inference [11] can be used to calculate types for functions in these languages. Another unification-based type inference is Hindley-Milner method [5], supposing that the return type of a function has a well defined type. Scala on the other hand is less restrictive on return types, branching expressions like the `match` construct allow that the return values on different branches have different types [4]. (E.g. a function can return either an integer or a string. In this case the return type of the function will be the $LUB - type$ of integer and string which is the top type, `Any`.)

Dynamically typed languages like Erlang [2] also allow to return values of different types. These kinds of *polymorphic* return types are extensively used in Erlang. Since types are not first-class citizens, external tools were developed to check for discrepancies in software [6], incorporating a type system called *success typing* [7]. The major difference between success typing and other type inference algorithms is that the aim of success typing is not to prove the type correctness of the program, but rather to discover cases where there would most certainly be a type error at runtime. It uses least upper bound types to enable the constraint solving algorithm to reach a fixed point.

Success typing uses union types to express coupling between types that are not in subtype relation. It is very useful for languages like Erlang where types are not an integral part of the language. It has a major drawback though when both the input parameter and the return type of a function are union types. The connection between the input and output is not expressed by the inferred type, and that decreases the number of discoverable errors. A possible improvement can be the use of conditional types similarly to the work of Aiken et al.[1]. This soft-type system (also using union types) includes conditional types where the constraints between type variables are built into the type. This type is more accurate in the above sense of finding discrepancies, but the size and complexity of inferred types makes it comprehensible for humans. If the human-readable criterion is ignored then the precision of inferred types can be increased without the need to calculate a fixed point [10]. These types are also very complex but can be used for specific tasks, e.g. automatic test data generation.

Success typing inspired us to type recursive functions. Since Scala is statically typed and types are inserted into the AST, using union types is not suitable for our needs. Unions would introduce new types to our program, that we do not intend to

$$
\begin{array}{rcl}
e & ::= & x \mid c(e_1, \ldots, e_n) \mid e_1\, e_2 \mid f \mid \\
 & & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \\
 & & \texttt{letrec}\ x = f\ \texttt{in}\ e \mid \\
 & & \texttt{case}\ e\ \texttt{of} \\
 & & \quad (p_1\ \texttt{if}\ g_1 \Rightarrow b_1); \\
 & & \quad \ldots; \\
 & & \quad (p_n\ \texttt{if}\ g_n \Rightarrow b_n) \\
 & & \texttt{end} \\
f & ::= & \lambda(x) \Rightarrow e \\
p & ::= & x \mid c(p_1, \ldots, p_n) \\
g & ::= & g_1\ \texttt{and}\ g_2 \mid g_1\ \texttt{or}\ g_2 \mid x_1 = x_2 \mid \texttt{true} \mid e\, x
\end{array}
$$

Figure 6: The λ_s language.

do since it would be hidden to the programmer and might cause unforeseen errors. Instead, we use the type hierarchy already present in the language. Scala already has a solid type system for nested classes, abstract types, path dependent types, etc [4, 9]. Since type inferring is solidly working in Scala, we do not want to replace or improve these theories.

3.2 The λ_s Language

We propose a small language and the corresponding calculus to demonstrate the theoretical soundness of our approach to type simple recursive functions. Let us call the language λ_S and be defined in Fig. 6. We would like to emphasize that this small language is not intended to be either generic-purpose or a full representation of Scala, rather to be the minimal language that can help us describe our proposed method formally.

The language contains variables (x) that are immutable. Data constructors (c) can be used to construct any kind of data, including constants, objects, etc. λ_S contains only single-argument function application. We assume that all Scala functions with at least one parameter can be curried, that is, they can be transformed to a function with multiple parameter lists containing only one parameter. It contains the standard polymorphic *let* expression. The recursive *let* expression has only one function component ($x = f$) since in this paper we deal with only self-recursive function. We define a branching expression (`case`). In the head of the case expression, e is matched against the patterns (p) sequentially. The first matched pattern will invoke the evaluation of the body (b) for the pattern. If no patterns match, then an exception is raised. Each branch has a pattern and a guard. A pattern can be a variable or a construct of patterns. Guards, that are always present in the

syntax, can be type check or other value checks. Using `true` as a guard, we can express the the case when we actually do not need any guards.

Our main focus will be on combining recursive *let* and *case* expressions. Recursive functions can be typed if they have a branch that is not recursive. Having this branch fulfills the termination criteria. If a function does not contain any terminating branches, then the function is divergent, and it cannot be typed.

$$
\begin{array}{rcl}
E & ::= & \texttt{letrec } x = \lambda(a) \Rightarrow \\
 & & \quad \texttt{case } a \texttt{ of} \\
 & & \quad\quad (p_1 \texttt{ if } g_1 \Rightarrow b_1); \\
 & & \quad\quad \ldots; \\
 & & \quad\quad (p_n \texttt{ if } g_n \Rightarrow b_n) \\
 & & \quad \texttt{end in } e
\end{array}
$$

Figure 7: The syntax of simple recursive functions

Definition 3.1 (Simple recursive function)**.** The expression E in Fig. 7 is considered a *simple recursive function*, iff there exists $i \in [1..n]$ that b_i symbolically contains x and there exists $j \in [1..n]$ that b_j does not contain x.

Simple recursive functions do exist and provide an abstract pattern over recursive functions that are not in recursive call chain.

3.3 Derivation rules

We provide type derivation rules for the syntactic constructs of λ_S. We assume that the type of objects, member functions and other language constructs not covered in this paper can be computed.

We present type derivation rules (Fig. 8) in the following form of statements: $\Gamma \vdash e : \tau$, read as "supposing Γ the type of the expression e is τ". Γ is the *context* of mappings from variables to types. The $\cup$ operator is used to denote that a particular mapping is present in the context.

The derivation rules describe a standard way to type our language. A variable can be typed (Rule (Var)), if its type is present in the variable context. The type of a data constructor (Rule (Cons)) is composed of the types of the components. A type is considered a subtype of another type (Rule(Sub) if an expression of the subtype can also be typed to the wider type. A function application can be typed if the argument expression (e_2) is a subtype of the parameter type (τ_1) of arrow type. In our case the arrow type is calculated via the existing type inferring algorithm of Scala. The type of a function (Rule (Fun)) and the let (Rule (Let)) expression follow standard definitions. With the derivation rule of subtyping of functions (Rule (S-Fun)) we would like to express that it is safe to allow a function of one

$$\frac{}{\Gamma \cup x : \tau \vdash x : \tau} \quad \text{(Var)}$$

$$\frac{\Gamma \vdash e_i : \tau_i \qquad (\forall i \in [1..n])}{\Gamma \vdash c(e_1, \ldots, e_n) : c(\tau_1, \ldots, \tau_n)} \quad \text{(Cons)}$$

$$\frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'} \quad \text{(Sub)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau' \quad \tau' <: \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2} \quad \text{(Appl)}$$

$$\frac{\Gamma \cup x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x) \Rightarrow e : \tau_1 \to \tau_2} \quad \text{(Fun)}$$

$$\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1' \to \tau_2 <: \tau_1 \to \tau_2'} \quad \text{(S-Fun)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \cup x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2 : \tau_2} \quad \text{(Let)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_e \\ \Gamma \cup \{x : \tau_x | x \in Var(p_i)\} \vdash \\ \quad b_i : \tau_{b_i}, p_i : \tau_{p_i}, g_i : \tau^{bool}\ (\forall i \in [1..n]) \\ \tau_e <: \bigsqcup_{i=1}^{n} \tau_{p_i}\end{array}}{\Gamma \vdash\ \texttt{case}\ \ : \bigsqcup_{i=1}^{n} \tau_{b_i}} \quad \text{(Case)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_e \\ \Gamma \cup \{x : \tau_x | x \in Var(p_i)\} \vdash \\ \quad b_i : \tau_{b_i}, p_i : \tau_{p_i}, g_i : \tau^{bool}\ (\forall i \in [1..n]) \\ \tau_e <: \bigsqcup_{i=1}^{n} \tau_{p_i}\end{array}}{\Gamma \vdash\ \texttt{case}\ \ : \bigsqcup_{i=1}^{n} FIX\tau_{b_i}} \quad \text{(Letrec)}$$

Figure 8: Derivation rules for λ_S.

type $\tau_1' \to \tau_2$ to be used in a context where another type $\tau_1 \to \tau_2'$ is expected as long as none of the arguments that may be passed to the function in this context will surprise it ($\tau_1 <: \tau_1'$) and none of the results that it returns will surprise the context ($\tau_2 <: \tau_2'$).

To type a `case` expression, first we type the head expression (e, which has the form defined in the `case` branch on Figure 6). For each branch we extend the context with types for free variables of patterns (Var) and calculate types for the patterns and the body of the function. The guards must evaluate to the boolean type. We denote the least upper bound type by the operator $\tau_1 \sqcup \tau_2$. The head of the expression has to be the subtype of the LUB of the types of the patterns (τ_{p_i}). The return type is the LUB of the types of the bodies of the branches.

Letrec(Rule (Letrec-C)) is similar to `case`, but it uses the fixed point of the return types of the branches ($FIXf = f(FIXf)$).

The recursive let expression can be typed in our scope only if it consists of a simple recursion function. We provide the constructive algorithm in the following theorem:

Theorem 3.2 (Constructive derivation rule of `letrec`)**.** *Suppose the notation of Fig. 7. Let us denote* $\mathcal{J} := \{i \mid b_i \text{ does not contain } x\}$*. Then the following constructive derivation rule holds:*

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_e \\ \Gamma \cup \{y : \tau_y | y \in Var(p_i)\} \vdash \\ b_i : \tau_{b_i}, p_i : \tau_{p_i}, g_i : \tau^{bool} \ (\forall i \in \mathcal{J}) \\ \Gamma \cup \{y : \tau_y | y \in Var(p_k)\} \cup \{x : \bigsqcup_{i=1}^{n} \tau_{p_i} \to \bigsqcup_{i=1}^{n} \tau_{b_i}\} \vdash \\ b_k : \tau_{b_k}, p_k : \tau_{p_k}, g_k : \tau^{bool} \ (k \in [1..n] \setminus \mathcal{J}) \\ \tau_e <: \bigsqcup_{i=1}^{n} \tau_{p_i}\end{array}}{\Gamma \vdash E : \bigsqcup_{i=1}^{n} \tau_{b_i}} \quad \text{(Letrec-C)}$$

where E is the `letrec` expression defined on Figure 7.

Proof. Let us first divide the case expression into two parts: the ones that do not contain recursive calls and the others that do. For the non-recursive branches we use the regular case typing derivation rule, hence $\Gamma \cup \{y : \tau_y | y \in Var(p_i)\} \vdash$ $b_i : \tau_{b_i}, p_i : \tau_{p_i}, g_i : \tau^{bool} \ \ (\forall i \in \mathcal{J})$ holds. This expression has the type of $\bigsqcup_{\forall i \in \mathcal{J}} \tau_{b_i}$ as per the case derivation rule. Let us later refer to this as the non-recursive type.

For the recursive branches, we have two cases:

1. Tail-recursion, as in $b_k \equiv x\, b_k'$. The expression in this case holds the type of the previously mentioned non-recursive type. We extend the type context

with $x : \tau_{p_k} \to \tau_{b_k}$ where $\tau_{p_k} <: \bigsqcup_{i=1}^{n} \tau_{p_i}$ and $\tau_{b_k} \equiv \bigsqcup_{\forall i \in \mathcal{J}} \tau_{b_i}$, i.e. not changing the non-recursive type.

2. Non-tail recursion. We type the body by applying the intermediate type that has been calculated so far to the recursive expression, then calculate $x : \tau_{p_k} \to \tau_{b_k}$. This will be then added to the type context thus the intermediate type of the expression will be extended by τ_{b_k} to $\bigsqcup_{i \in \mathcal{J}} \tau_{b_i} \sqcup \tau_{b_k}$.

With the above considerations we can type all branches of `letrec`, turning it into a regular `case` expression that has the type of $\bigsqcup_{i=1}^{n} \tau_{b_i}$, resulting in the following type: `letrec` $: \bigsqcup_{i=1}^{n} \tau_{b_i}$.

□

4 Results

In this section we will discuss how the previously described algorithm works in practice. First we apply the algorithm to the two examples shown in section 2 then we will show how this is implemented as an extension in the Scala compiler.

4.1 Typing simple recursive functions

Our first example is an unusual version of the map function on Fig. 9. Its purpose is to return the mapped results in a list of the smallest type possible. As one can see in the listing, the main body of the function is a `match` with two possible `case` branches. This instruction flow is very similar to the extended λ_S language. Each branch has a return type, namely the first one returns a `Seq[B]` and the second one a `Seq[C]`. The type of the second case is defined by the result of the concatenation(`:+`) method. This part can be rewritten in a form of `map(xs, f).:+(f(x))`. If we consider this function call, the appended element is of type `C`, as we have declared `f` to be a function of `A => C`. Since `A` is a subtype of `C`, the result of the map function is a subtype of `C`, making the result of the append call be a type of `Seq[C]`. This leads us to two calculated types for the branches: `Seq[B]` and `Seq[C]`. The typing mechanism we propose would now calculate the least upper bound for these types that would be `Seq[C]`, and typing our special `map` function with `Seq[C]`.

The next example is more involved. First we start off with declaring a multi-level type hierarchy as seen on Fig. 4. This hierarchy declares 3 levels with leaf nodes being `case class`es. The recursive function using these classes is defined on Fig. 10. The control flow graph created by the predicates in the function has several branches, unlike the straight tree of the `match` and `cases` in the first example. This will cause no problems to our typing algorithm, as it can be applied to all leaf branches, and then going upwards in the tree using the previously calculated types. This calculation starts with the `if-else` on line 9. The first branch contains a recursive call, so we cannot type this branch. We need to start with the `else`

```
1   def map[C, A <: C, B <: C](y: Seq[A], f: A => C) = {
2       y match {
3           case Nil => Seq[B]()
4           case x :: xs => map(xs, f) :+ f(x)
5       }
6   }
```

Figure 9: A recursive map implementation with type inference.

branch first, making the calculated type `Class2Level1`. The `case` branch on line 8 then would be typed the same. We have a trivially typed `case` on line 7, with `Class2Level2`. The least upper bound for these types is `Level2`, so the `else` branch will be typed `Level2`. We have arrived at the top-most level of predicates, that has types of `Class1Level2`, `Level2`, `Class1Level2` and `Class1Level1`, in respective order. The least upper bound defined by the hierarchy is therefore `Level1`. This will be the final type of the `deepRec` function.

```
1   def deepRec(n: Int) = {
2       if (n == 0) {
3           Class1Level2
4       } else if (n == 1 || n == 2 || n == 3) {
5           n match {
6               case 3 => Class2Level2
7               case _ => {
8                   if (n != 1) {
9                       deepRec(n − 1)
10                  } else {
11                      Class2Level1
12                  }
13              }
14          }
15      } else if (n == 4) {
16          Class1Level2
17      } else {
18          Class1Level1
19      }
20  }
```

Figure 10: A recursive function using type hierarchy on Fig. 4. with type inference.

The λ language was only defined for single-parameter functions. The reason we can still use it to calculate types for these functions is that by currying multi-parameter functions, we can always transform them to a chain of function appli-

cations having only one parameter. Since Scala supports currying by default – by denoting a function call with the _ (underscore symbol) –, we assume that all functions in question can be transformed to this kind.

4.2 Typing recursive functions with multiple branches

In the previous examples we discussed functions with a single recursive branch. In the following we show that similar solution exists for simple functions with multiple recursive branches.

```
1   class A {
2       def toC: C = ...
3       def toD: D = ...
4   }
5
6   class B extends A
7   class C extends A
8   class D extends A
9
10  def multi(n: Int) = n match {
11      case 0 => new B
12      case n > 0 => foo(n − 1).toC
13      case n < 0 => foo(n + 1).toD
14  }
```

Figure 11: A function with multiple recursive branches.

In the example shown on Fig. 11 there are two recursive branches resulting in two different but related types. Our algorithm finds the non-recursive branch on line 11 and calculates type `B`. Typing the second branch will use this information as the return type of the `foo` call. Using `B` as a placeholder type, the default type inference algorithm of Scala will calculate type `C` as the return type of the branch on line 12. Similarly, type `D` will be calculated for the branch on line 13. Finally, the LUB of types `B`, `C` and `D` will be determined as `A`. Therefore, the return type of function `multi` will be `A`. This result complies with the expected result type of our algorithm and meets the intention of the developer.

4.3 Extending the Scala compiler to handle simple recursive functions

The fundamental design and structure of the Scala compiler makes it an excellent candidate to be extended. The features of the compiler that makes this possible are high separation of compiler phases, the support for macros and fully independent compiler plugins and the compiler being an open source project[14]. The phases of

the compiler start by parsing the source files and generating an AST; later phases transform this AST. The current version of compiler is written in Scala, using Scala objects to describe the nodes of the AST. The compiler also acts as a library to analyze and compile Scala source code, providing a programmatic API that can be accessed from applications.

As the first step, we had to find a way to circumvent the type error generated for recursive functions. In the default version of the compiler, when the typer starts calculating the type of a (recursive) function and it meets an entity that has been defined previously, but the type of it is yet to be determined, it will throw a `CyclicReference` exception that is collected and handled by the generic error handling infrastructure of the compiler. This does not stop the typing phase from continuing with typing other entities. We leverage this property, as it collects all the erroneous recursive calls, but types all other to-us trivial cases.

Our main approach of extending the compiler has focused on creating a separate codebase, as modifying the main branch directly was found to be too time consuming and difficult. Fortunately, we were helped in this effort by the various API calls provided by the package `scala.tools.nsc._` (`nsc` stands for New Scala Compiler).

The extended `scala.tools.nsc.typechecker.Analyzer` contains methods overridden that handle control flow ASTs –namely `if`s and `case`s– and method definitions, so we can annotate these methods with our calculated type. We of course use the original typer to first type these entities and only interrupt cases that are of interest to us. Finding the least upper bound of types is another key point in our algorithm, but we were very fortunate in this regard: we use the function `lub` on `scala.tools.nsc.TypeChecker.Typer`.

The last remaining piece to have a working compiler was inserting the newly created typer into the chain of compile phases and invoking it from the first step, the parser. We have achieved this with our own entry point to an application that simply passes a path to the compiler and invokes it. We only use regular console reporting by `scala.tools.nsc.reporters.ConsoleReporter` and global settings by `scala.tools.nsc.{Global, Settings}`.

We have measured how our extension affects compile speeds by calculating the total time spent in the `main` method of our application. The results are shown in Table 1. For simplicity, we have listed measured microseconds with the default version of the compiler –i.e. invoking it without setting the extended typer– and with the extension in place. As it can be seen in change percentage, the extension has no significant impact on performance. Example1 and Example2 refer to the examples seen in the previous subsection, while Multiple methods contains several other test cases.

The prototype can be downloaded and can be used for further tests from [15].

4.4 Restrictions

Scala supports defining recursive types using explicit type annotations. Soundly calculating all recursive types would require extending our inference method with a

Table 1: Compile times (μs) with and without using our extension

Code snippets	Default	Extended	% change
Example1	2440166	2479896	101.62
Example2	3390745	3422435	100.93
Multiple methods	5691124	5712335	100.37

fixed point calculation. Henceforth, our proposed method in its current form does not support recursive types.

We implemented and tested our compiler extension only on Scala compiler version 2.12.4. It is not guaranteed to work with any other version than 2.12.4.

4.5 Future work

When we created the Scala compiler extension to infer types of simple recursive functions, our main intentions were focused on prototyping the theoretical background we have described in this paper, not developing an industry-standard, complete implementation. This leaves great space for future improvements. Firstly, we can provide a better integration to the compiler by disabling `CyclicReference` exceptions for our cases, then properly type AST nodes "in-place". Then we can merge our changes back to the main line of the compiler.

Besides creating a more robust implementation, we also plan to work on extending the theory by finding methods to analyze recursive chains then proving the soundness of these methods. This would require extending our λ-language and also the way it handles types. As another step, we are looking into providing a clean, correct and complete theoretical background and implementation for inferring types of recursive type definitions.

5 Related works: type inference in C++

The C++ programming language is one of the current mainstream general purpose languages [12]. Its popularity is originated to its suitability in almost all application areas from high performance computing and telecommunication to embedded systems. C++ provides language tools for the programmer to implement complex systems from gradually specified and implemented building blocks without compromising run-time efficiency. C++ is often described as a multiparadigm programming language [3], as it has imperative, object-oriented, generic and functional programming features.

C++ is a strongly typed programming language in the sense, that the type of every (sub)expression is determined in compilation time. However, templates use duck-typing,i.e. no constrained generics exist in current C++. There are plans to improve the template mechanism with constraints.

Earlier C++ codebase was known about notoriously long type notations. To unburden programmers' task and make source more readable, the C++11 standard introduced the `auto` keyword as a placeholder for types [16]. Its primary usage is to avoid needlessly verbose type declarations, like those are used with connection in STL algorithms:

```
1    // C++03
2    typename std::vector<T>::iterator i = v.begin();
```

This can be replaced by usage of keyword `auto`. The type of the `i` variable will be inferred from the initialization expression: `v.begin()`

```
1    // C++11
2    auto i = v.begin()
```

The keyword `auto` to replace the return type for functions but only with a new trailing type syntax introduced in C++11:

```
1    template <typename T, typename S>
2    auto max( T a, S b) -> decltype(a+b) // C++11
3    {
4        if ( a > b )
5        return a;
6        else
7        return b;
8    }
```

Notice, that this usage of `auto` syntax does not imply type inference, the return type is explicitly expressed in the trailing syntax. The role of `auto` here is only a placeholder: since the language elements used in the trailing syntax (`a` and `b` parameters in the `decltype` expression are not in scope *before* the function name).

In the C++14 standard, however, there is automatic type inference available for function return types in the most simple cases [8].

```
1    auto f(); // return type is unknown
2    auto f() // return type is int
3    {
4        return 42;
5    }
6    auto f(); // redeclaration
7    int f(); // error, declares a different function
```

A function with auto return type can have multiply return statements. However, there is a strict restriction here, that each return statement should return the same single type, otherwise the compiler reports error. That is different to Scala, where in case of multiple return statements the return type is inferred as the least upper bound of the return types.

Recursion is allowed by the proposed C++14 inference rules in a very restricted way. The recursive return branch should be preceded by at least one non-recursive

return, from which the return type of the function is inferred. Subsequent return statements are checked against this type. Therefore the following code will be accepted by the proposal:

```
1    auto fib(int n)
2    {
3        if ( 0 == n ) return 1;
4        else return n*fib(n-1);
5    }
```

While the variation, where we have changed the recursive and non-recursive branches will be rejected:

```
1    auto fib(int n)
2    {
3        if ( n > 0 ) return n*fib(n-1);
4        else return 1;
5    }
```

The authors have the opinion that these rules are unnecessary restrictive and can be relaxed without compromising compile-time efficiency.

6 Conclusions

In this paper we have analyzed Scala inference rules for function return types. We stated that with certain types of simple recursive functions, automatic calculation of the return type can be done with some effort. Such an additional feature is in parallel with the original design philosophies of Scala that try to lift unnecessary burden off the programmer.

We have proposed a new method to compute the return types for simple recursive functions. We have defined a small language to demonstrate the theoretical soundness of our approach. Our heuristic assumption on the return type is based on the non-recursive execution branches and we have also provided a proof of its correctness. Furthermore, we have assumed that the recursive functions will always end up in a non-recursive execution branch. The least upper bound type of these branches provide sufficient information allows the default Scala type inference algorithm to infer the return type of recursive branches. Finally, by taking the least upper bound type of all branches we can define the return type of the function.

A prototype implementation has been created by extending the Typer not to immediately fail on recursive functions, but use the proposed method to calculate the return type. We have implemented our method as an extension prototype for the Scala compiler v2.12.4 and have used it to successfully test our method on various examples. In case type discrepancies already exist in the program, our compiler extension will report the same error as the current Scala compiler.

The extension is proved to be effective in the sense that it does not significantly affect compilation speed. The compiler extension prototype is publicly available for further tests.

References

[1] Aiken, Alexander, Wimmers, Edward L, and Lakshman, TK. Soft typing with conditional types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, 1994. DOI: 10.1145/174675.177847.

[2] Armstrong, Joe, Virding, Robert, Wikström, Claes, and Williams, Mike. *Concurrent programming in ERLANG*. Prentice Hall, 1993.

[3] Coplien, James O. *Multi-paradigm design for C++*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[4] Cremet, Vincent, Garillot, François, Lenglet, Sergueï, and Odersky, Martin. A core calculus for Scala type checking. In *International Symposium on Mathematical Foundations of Computer Science*, pages 1–23. Springer, 2006. DOI: 10.1007/11821069_1.

[5] Hindley, Roger. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969. DOI: 10.1090/S0002-9947-1969-0253905-6.

[6] Lindahl, Tobias and Sagonas, Konstantinos. Typer: A type annotator of Erlang code. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 17–25, 2005. DOI: 10.1145/1088361.1088366.

[7] Lindahl, Tobias and Sagonas, Konstantinos. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 167–178, 2006. DOI: 10.1145/1140335.1140356.

[8] Merill, J. Return type deduction for normal functions, 2013. Revision 5. N3638, 2013.04.17. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/~2013/n3638.html.

[9] Odersky, Martin, Cremet, Vincent, Röckl, Christine, and Zenger, Matthias. A nominal theory of objects with dependent types. In *European Conference on Object-Oriented Programming*, pages 201–224. Springer, 2003. DOI: 10.1007/978-3-540-45070-2_10.

[10] Oláh, G., Horpácsi, D., Kozsik, T., and Tóth, M. Type interface for core Erlang to support test data generation. *Studia Universitatis Babes-Bolyai Informatica*, LIX(2014/1):201–215, 2014.

[11] Peyton Jones, Simon, Vytiniotis, Dimitrios, Weirich, Stephanie, and Washburn, Geoffrey. Simple unification-based type inference for GADTs. *ACM SIGPLAN Notices*, 41(9):50–61, 2006. DOI: 10.1145/1159803.1159811.

[12] Stroustrup, Bjarne. The C++ programming language (special 3rd edition), 2000.

[13] Venners, Bill and Sommers, Frank. The goals of Scala's design. A conversation with Martin Odersky, Part II. In Artima developer's web site, 2009. `http://www.artima.com/scalazine/articles/goals_of_scala.html`.

[14] The GitHub home of the Scala compiler. `https://github.com/scala/scala`.

[15] The GitHub home of the Scala compiler extension for simple recursive functions. `https://github.com/njeasus/ScalaRecTyper`.

[16] The ISO C++11 standard, ISO/IEC 14882:2011(E) – Information technology – Programming languages – C++, 2011. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=50372`.