Tiago Silva Santos

Bachelor in Computer Science and Engineering

Typed Meta-programming with Kind Refinements - Bidirectionally

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Bernardo Toninho, Assistant Professor, NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2021

Abstract

Meta-programming is a programming technique where programs write or manipulate other programs (or themselves) as their data. Meta-programming is useful because, in some cases, it allows programmers to minimize the number of lines of code to express a solution and in turn, reduces development time. However, in general it requires using non-type-safe features such as reflection, resulting in few guarantees of compilation time correctness. So, refinement kinds are a way to have some type-safe meta-programming features. This solution already has a prototype, however the implementation still requires many type annotations and its type-checking mechanism can exhibit unpredictable behaviour, for example, typing errors or unexpected kindings.

In this work, we propose to develop and use of bidirectional type-checking and kind-checking techniques, by combing existing approaches for refinement types and high-rank polymorphism but applied to the kind-checking level. The goals of the solution are a bidirectional kind and type checking algorithm for refinement kinds, interface with bidirectional kind-checking, at the level of type-checking, and applying partial inference techniques of types instantiation.

Keywords: Refinement kinds, Bidirectional typing, High-order polymorphism, Type inference

Resumo

A meta-programação, ou seja, uma técnica de programação onde os programas escrevem ou manipulam outros programas (ou eles próprios) como seus dados, é muito útil porque, em alguns casos, permite aos programadores minimizar o número de linhas de código para expressar uma solução e por sua vez, reduzindo o tempo de desenvolvimento. No entanto, em geral, ele requer o uso de features não type-safe, resultando em poucas garantias de correção em tempo de compilação. Portanto, os refinement kinds são uma forma de ter algumas features de meta-programação type-safe. Esta solução já possui um protótipo, porém a implementação ainda requer muitas anotações de tipo e não é muito "previsível" no comportamento, por exemplo, ocorrendo erros de tipificação ou kindings inesperados.

Portanto, a solução para este problema está no desenvolvimento e uso de técnicas de type-checking e kind-checking bidireccionais, combinando as abordagens existentes para refinement types e higher-rank polymorphism, mas aplicadas ao nível do kind-checking. Os objetivos da solução são o algoritmo de kind e type checking bidireccional para refinement kinds, "interface" com kind-checking bidirecional, ao nível do type-checking, e aplicação de técnicas de inferência parcial de instanciação de tipos.

Palavras-chave: Refinement kinds, Bidirectional typing, High-order polymorphism, Type inference

Contents

List of Figures

Listings

Chapter 1

Introduction

Meta-programming is a programming technique where programs write or manipulate other programs (or themselves) as well as their data. Meta-programming is useful because, in some cases, it allows programmers to minimize the number of lines of code to express a solution and in turn, reduces development time. For example, a function that serializes an object of any type in JSON, for which it is necessary to perform introspection of the object typically using reflection. However, the problem is that to define a function of this kind, current meta-programming techniques require using primitives that are not type-safe, and therefore it is not possible to guarantee (statically) that the function will not crash at runtime or that will produce the result we want.

Given this problem, many works [7, 2] try to statically validate the use of meta-programming, among which the concept of refinement kinds arises. Refinement kinds are a technique of expressing transformations between types in a declarative way, using rich polymorphism combined with logical predicates to verify structural properties of types and, for example, it is possible define a transformation that, given a type of any object, it is possible to produce a new type that includes getters and setters for the fields of that object, to later use these transformations to type (meta) programs.

1.1 Motivation

The idea of refinement kinds is a practical way to make these meta-programming ideas more usable and safe. However, the existing prototype for refinement types suffers from some fundamental limitations:

- Type inference is undecidable (where inference here typically applies to instantiation of polymorphic types) for these more sophisticated type systems, making the prototype require many type and kind annotations, making it difficult to use;

- The typing and kinding algorithm is defined in a best-effort single pass, and therefore has potentially unexpected behavior in terms of the necessary annotations.

1.2 Goals

The main goals of this work are to improve the experience of refinement kinds, taking advantage of existing work that has been done in recent years in terms of typing algorithms called bidirectional for advanced type systems, such as liquid types (types equipped with automatically verified logical assertions about their values) and systems with advanced forms of polymorphism.

The bidirectional typing algorithms are strategies for implementing "advanced" type systems that alleviate the amount of annotations that need to be written in programs, integrating some forms of inference in these advanced contexts, and whose behavior at the level of "errors" is predictable. For example, the works [13, 12] develop bidirectional type inference algorithms for forms of polymorphism for which full inference is undecidable. However, their approach integrates inference (when it is possible) with the use of limited type annotations for the more sophisticated cases. Moreover, these approaches are known to produce good and predictable error behaviours.

For this reason, this work will combine the bidirectional approaches "of polymorphism" with those of "liquid types", but applied at the level of type transformations, therefore at the level of type validation, thus extending and generalizing the prototype of refinement types.

In chapter 2 we explain in more detail all the topics that were mentioned previously and in Chapter 3 the work is described and planned in more detail.

Chapter 2

Background and Related Work

This chapter introduces the concepts that form the basis of the work and the key systems of related work. Specifically, we introduce:

1. **Polymorphism** - a language mechanism that allows a single part of the code to be used with different types;
2. **Higher-kind polymorphism** - where we introduced the term *kinding*;
3. **Refinement types** - a system that uses logical assertions to enrich the type system;
4. **Refinement kinds** - a generalization of the idea of refinements to kinds;
5. **Bidirectional typing** - a type checking technique that combines type inference with traditional type-checking;
6. **Bidirectional typing for refinement types** and **Bidirectional typing for high-order polymorphism** - Bidirectional techniques for refinement types and high-order polymorphism.

2.1 Polymorphism

Type systems that allow code to be assigned with multiple types are known as polymorphic systems. The term *polymorphism* [9, Chapter 22] refers to a range of language mechanisms that allow a single part of a program to be used with different types in different contexts. The first form of polymorphism on a language was known as *let-polymorphism* and it's called that because it arises from problems with using *let* expressions. This feature was introduced in the original dialect of ML (Milner, 1978) and has been incorporated in

a number of successful language designs, because it forms the basis of powerful generic libraries of commonly used structures (lists, arrays, trees, hash tables, etc.).

The following Listing 2.1 (in Haskell) shows a generic list data type, with two constructors, and two polymorphic functions over lists:

Listing 2.1: Example about generic list data type

```
1 data List a = Nil | Cons a (List a)
2
3 length :: List a -> Integer
4 length Nil = 0
5 length (Cons x xs) = 1 + length xs
6
7 map :: (a -> b) -> List a -> List b
8 map f Nil = Nil; map f (Cons x xs) = Cons (f x) (map f xs)
```

These functions are polymorphic because can accepts any type of list in their arguments. Function **length** computes the size of a list, if the list is empty (Nil case) then returns zero, and if the list a cell linking x with list xs, then returns one plus the length of list xs. In function **map**, given a function f between any two types a and b and a list l of a's, map f l produces a list of b's by applying f to each element of l.

In modern languages, polymorphism is available in a variety of forms:

- Parametric polymorphism: which allows a single piece of code to be typed "generically," using type variables placeholders for actual types, and then instantiated with particular types as needed, like the example above. The forms available are impredicative or first-class polymorphism, which is the more powerful, and the most common form, ML-style or let-polymorphism.

- Ad-hoc polymorphism allows a polymorphic value to exhibit different behaviors when "viewed" at different types. The most common form is overloading, which is the ability to create multiple functions of the same name with different implementations.

Parametric polymorphism os generally studied using the *System F*, which we present in the following section.

2.1.1 System F

This system [9, Chapter 23] has been used extensively as a research vehicle for foundational work on polymorphism and as the basis for numerous programming language designs. It is also sometimes called the *second-order lambda-calculus*, because it corresponds, via the *Curry-Howard correspondence*, to second-order intuitionistic logic, which allows quantification over predicates (types).

The definition of System F is an extension of the simply typed lambda-calculus, with the addition of rules for abstraction and application of types, which are represented by the grey background, on Figure 2.1.

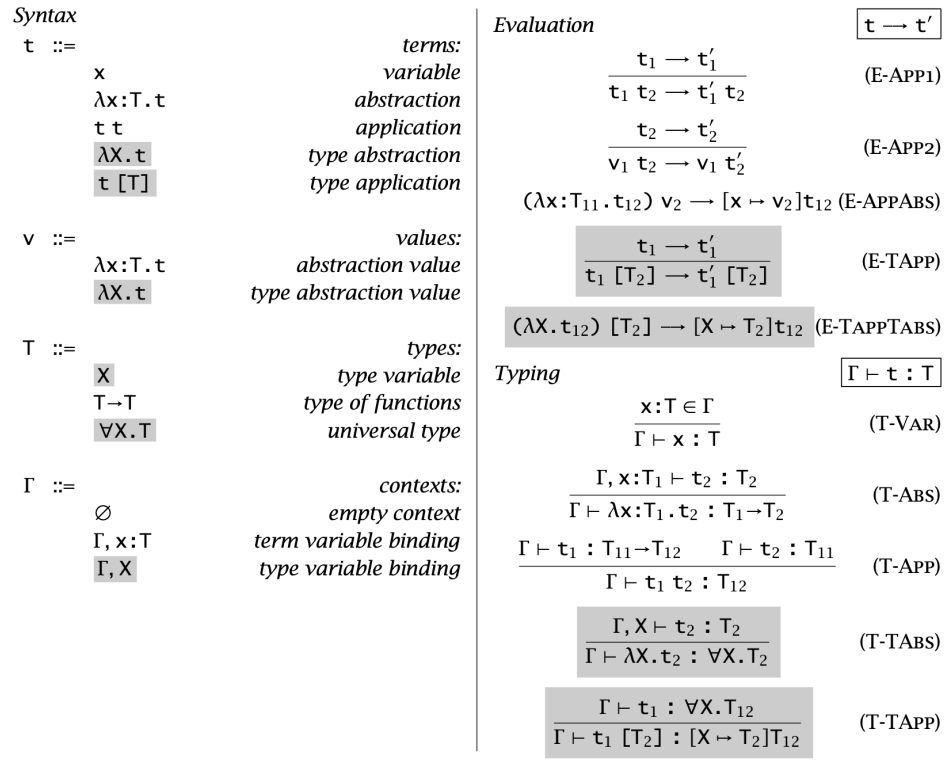

Figure 2.1: Polymorphic lambda-calculus (System F) [9, Chapter 23.3]

By using this system, we can now use the syntax $\forall X.T$ to represent a *universal type*. Using the example before of lists, now on System F:

$$\text{map} : \forall X\ \forall Y.\ (X \to Y) \to \text{List } X \to \text{List } Y \qquad \text{length} : \forall X.\ \text{List } X \to \text{Int}$$

With system F, we introduce a new form of abstraction called a *type abstraction*, written λX.t, whose parameter is a type, and a new form of application,named *type application*, t [T] , in which the argument is a type expression. When, during evaluation, a type abstraction meets a type application, the pair forms a redex, creating the following rule (where $[X \mapsto T_2]t_{12}$ denotes substituting T_2 for X in t_{12}):

$$(\lambda X.t_{12})\,[T_2] \to [X \mapsto T_2]t_{12}$$

For instance, let's look at the following example:

id = λX. λx:X. x;

When this *polymorphic identity function* is applied to Nat by writing id [Nat], the result [X ↦ Nat] (λx:X.x), i.e., λx:Nat.x, is the identity function on natural numbers.

2.2 Higher-kinded Polymorphism

With the introduction of polymorphism in a language, we add a more flexible, richer and more sophisticated type system. However, if we add another level on our system of types, it rises to a more sophisticated and powerful language. In this section, we will introduce the realm of *kinds* and type-level functions.

First, let's consider the following example [9, Chapter 29]:

Pair Y Z =∀X. (Y→ Z →X)→X;

The program above is an abbreviation of the type ∀X.(Y→Z→X)→X , where the type variables Y and Z are free, or uninstanced. Such abbreviations are often used to make easier to read and write, by writing λx:Pair Nat Bool. x, for instance, instead of the more cumbersome λx:∀X.(Nat→Bool→X)→X. x. . This type of abbreviation is called *parametric abbreviation* and what it does is when we encounter Pair S T, we must substitute the actual types S and T for the parameters Y and Z in its definition. In other words, abbreviations like Pair give us an informal notation for defining functions at the level of types.

These type-level functions are called *type operators*. Abstraction and application at the level of types, along with a precise definition of when two type expressions should be regarded as equivalent and a well-formed, is called *kinding*.

To model and reason about such functions, the first thing that we need is some notation for abstraction and application. Is standard to use the same notation for term-level functions. For example, we write λX.{a:X,b:X} for the function that, given a type T, yields the record type {a:T,b:T} to represent an abstraction. The application of this function to the argument Bool is written (λX.{a:X,b:X}) Bool. By introducing these types of mechanisms at the level of types, give us the possibility of writing the same type in different ways. For example, if Id is an abbreviation for the type operator λX.X, then the expressions

Nat → Bool	Nat → Id Bool	Id Nat → Id Bool
Id Nat → Bool	Id (Nat → Bool)	Id (Id (Id Nat→ Bool))

are all names for the same arrow type. To make this intuition precise, we need a *definitional equivalence* [9, Chapter 29] on types, which captures type-level computation arising from application of type-level functions. The equivalence $S \equiv T$ is internalized in the type-checker through the rule (usually called the type conversion rule):

$$\frac{\Gamma \vdash t : S \quad S \equiv T}{\Gamma \vdash t : T}\,(T-EQ)$$

Type conversion allows for a term of a given type to be assigned any equivalent (according to definitional equivalence) type. Using the previous example, all types are equivalent to Nat → Bool, i.e., the expression Id Nat → Id bool, through type-level computation, results in the Nat → Bool.

In the presence of such type-level functions, it becomes possible to write ill-formed types. For example, applying one proper type to another, as in the type expression (Bool Nat) is ill-formed. To prevent this kind of ill-formed types, a system of kinds can classify type expressions according to their arity.

Kinds are built from a single atomic kind, written * and pronounced "the kind of proper types" and a single constructor ⇒. They include, for example:

$*$	the kind of proper types (like Bool and Bool→Bool)
$* \Rightarrow *$	the kind of type operators (i.e., functions from proper types to proper types)
$* \Rightarrow * \Rightarrow *$	the kind of functions from proper types to type operators (i.e., two-argument operators)
$(* \Rightarrow *) \Rightarrow *$	the kind of functions from type operators to proper types

Kinds, then, are "the types of types." In essence, the system of kinds is a copy of the simply typed lambda-calculus, "one level up." Type expressions with kinds like (*⇒*)⇒* are called higher-order type operators. To simplify the problem of checking the well-kindness of type expressions, we annotate each type-level abstraction with a kind for its bound variable. A variable x is said to be *bound* [9, Chapter 5] when it occurs in the body t of an abstraction, like $\lambda x.t$, and this are used to know the *scopes* of variables. In $(\lambda x.x)\ x$, the first occurence of x is bound and the second is *free*.

With that, the form of the Pair operator, using type-level functions and kinding:

Pair = λA::*. λB::*. ∀X. (A→B→X)→X;

However, since almost all of these annotations will be *, we will continue to write λX.T as an abbreviation for λX::*.T.

2.3 Refinement Types

The type systems of modern languages are the most widely used method for establishing guarantees about the correct behavior of software. In essence, types allow the programmer to describe legal sets of values for various operations, thereby eliminating, at compile-time, the possibility of a large swathe of unexpected and undesirable run-time errors. Unfortunately, well-typed programs can still produce various runtime errors, such as[6]:

- ***Divisions by zero***: The fact that a divisor is an **int** does not preclude the possibility of a run-time divide-by-zero, or that a given arithmetic operation will over- or underflow;
- ***Buffer under-/overflows***: The fact that an **array** or **string** index is an **int** does not eliminate the possibility of a segmentation fault, or worse, leaking data from an out-of-bounds access;

Given that such errors generally fall out of scope of mainstream type systems, refinement types [6] allow us to enrich a language's type system with predicates that circumscribe the set of values described by the type. For example, while an int can take any integer value, we can write the refined type

```
type nat = int[ v |0 <= v ]
```

that describes only non-negative integers (*i.e*, integers v such that $v \geq 0$). By combining types and predicates, the programmer can write legal inputs and outputs of functions. For example, the author of an array library could specify that

```
val size : (x : array(a)) => nat[ v | v = length ( x ) ]
val get : (x : array(a)) => nat[ v | 0 <= v < length ( x ) ] => a
```

which say that a call on function **size** ensures the returned value equals to the number of elements in the array, and the function **get** requires the array index needs to be an **nat** type and cannot be larger than the size of the array, to guarantee that all array accesses are safe at runtime.

With that, refinements [6] provide a tunable knob whereby developers can inform the type system about what invariants and correctness properties they care about, in example, are important for the particular domain of their code.

2.3.1 An example from Liquid Haskell

Liquid Haskell is [14] a ML-style programming language extended with logical assertions in types. An excerpt of the syntax of [14] is given in Figure 2.2.

Listing 2.2 shows an example with two refinements of Int, specifying the type Pos of numbers that are positive and the type of Nat representing the natural numbers, a function `div` that ensures no division-by-zero errors, and there are two expressions, which the **good** expression is accepted and the **bad** expression doesn't.

Listing 2.2: LiquidHaskell example

```
type Pos = {v:Int | v > 0}
type Nat = {v:Int | v >= 0}

div :: n:Nat -> d:Pos -> {v:Nat | v <= n}

good :: Nat -> Nat -> Int
good x y = let z = y + 1 in x `div` z

bad :: Nat -> Nat -> Int
bad x y = x `div` y
```

$$
\begin{array}{lrcl}
\textbf{\textit{Refinements}} & r & ::= & \ldots \text{varies} \ldots \\
\textbf{\textit{Basic Types}} & b & ::= & \{v{:}\mathtt{Int} \mid r\} \mid \ldots \\
\textbf{\textit{Types}} & \tau & ::= & b \mid x{:}\tau \to \tau \\
\textbf{\textit{Environment}} & \Gamma & ::= & \emptyset \mid x{:}\tau, \Gamma \\
\textbf{\textit{Subtyping}} & & & \Gamma \vdash \tau_1 \preceq \tau_2 \\
\textbf{\textit{Abbreviations}} & & & \\
x{:}\{r\} & & \doteq & x{:}\{x{:}\mathtt{Int} \mid r\} \\
\{x \mid r\} & & \doteq & \{x{:}\mathtt{Int} \mid r\} \\
\{r\} & & \doteq & \{v{:}\mathtt{Int} \mid r\} \\
\{x{:}\{y{:}\mathtt{Int} \mid r_y\} \mid r_x\} & & \doteq & \{x{:}\mathtt{Int} \mid r_x \wedge r_y\,[x/y]\}
\end{array}
$$

Figure 2.2: Syntax for example [14]

In expression **bad**, the refinement type system will check that second parameter **b** has type Pos at call to **div**, where formally, **y** is a subtype of the type of **div**'s second input, via a subtyping query and using the Abbreviation of Figure 2.2 to simplify:

$$x : \{x \geq 0\}, y : \{y \geq 0\} \vdash \{v \leq 0\} \preceq \{v : Int \mid v > 0\}$$

A refinement type system generates a *verification condition* [14] (VC) , a logical formula that stipulates that under the assumptions corresponding to the environment bindings, the refinement in the subtype implies the refinement in the super-type. So the VC for subtyping query above is like:

$$(x \geq 0) \wedge (y \geq 0) \Rightarrow (v \geq 0) \Rightarrow (v > 0)$$

VCs like the one above can be validated by SMT solvers, so in this case, the solver will reject because the invalid meaning in the implications, and so the refinement type

will reject. On the other hand, a refinement type system will accept the **good** expression because of the valid VC:

$$(x \geq 0) \wedge (y \geq 0) \wedge (z = y + 1) \Rightarrow (v = y + 1) \Rightarrow (v > 0)$$

2.4 Refinement Kinds

In our days, automated code generation, domain specific languages, and meta-programming are good tools to make the productivity in the software industry, by making programming easier to a newcomer, and increasing the level of abstraction expressible in languages and tools for program construction. However, meta-programming constructs and idioms generally can't guarantee the safety in a static typing, which becomes especially problematic because is challenging to test the correctness of the meta-program. For that, *refinement kinds* are a good solution to resolve some of the problems mentioned before.

Refinement kinds are a natural transposition of refinement types, but in the realm of kinds. They support static type checking of type-level reflection, parametric and ad-hoc polymorphism, which can all be combined to implement interesting meta-programming idioms, rich and flexible kind specifications by means of comprehension principles expressed by *predicates over types* in the kind domains.

A refinement kind [1] is noted by $\{t :: K | \varphi(t)\}$, where K is a base kind, and the logical formula $\varphi(t)$ expresses a constraint on the type t that inhabits K. The kind refinement rule is thus expressed by

$$\frac{\Gamma \models \varphi\{T/t\} \quad \Gamma \vdash T :: K}{\Gamma \vdash T :: \{t :: K | \varphi\}} \, (KREF)$$

where $\Gamma \models \varphi$ denotes entailment in the refinement logic.

2.4.1 Pratical example of Refinement Kinds

To show refinement types in a more practical way, we will first introduce record types. Usually, a record type is represented by a tuple of label-and-type pairs, and in order to support more effective manipulation by type-level functions, our record types are represented by values of a list-like data structure [1] the record type constructors are the type of empty records $\langle\rangle$ and the "cons" cell $\langle L : T\rangle @ R$, which constructs the record type obtained by adding a field declaration $\langle L : T\rangle$ to the record type R.

For example, the record type **Person** can be defined as $\langle name : String\rangle @ \langle age : Int\rangle @ \langle\rangle$, which can be abbreviate by $\langle name$: String; age : Int$\rangle$. The kinding of the $\langle L : T\rangle @ R$ type constructor may be clarified in the following type-level function addFieldType:

$$\text{addFieldType} :: \Pi \text{l::Nm}.\ \Pi \text{t::Type}.\ \Pi \text{r::}\{\text{s::Rec} \mid \text{l} \notin \text{lab(s)}\}.\text{Rec}$$

$$\text{addFieldType} \triangleq \lambda \text{l::Nm}.\lambda \text{t::Type}.\lambda \text{r::}\{\text{s::Rec} \mid \text{l} \notin \text{lab(s)}\}.\langle l : t\rangle @ r$$

The Π represents a *dependent (type-level) function type* [9, Chapter 30] , where a form $\Pi x : K_1.K_2$ are a more precise form of arrow types $K_1 \rightarrow K_2$, where we bind a variable x representing the function's argument so that we can mention it in the result kind K_2. A *dependent kind* can be seen as the family of kinds indexed by types, offering a degree of precision in describing type-level functions that goes far beyond the other typing features we have seen in Section 2.2.

The addFieldType type-level function takes a label l, a type t and any record type r that does not contain label l, and returns the expected extended record type of kind Rec. Notice that the kind of all record types that do not contain label l is represented by the refinement kind {s::Rec | l ∉ lab(s)}.

A more complex example is a generic record type-level map, which uniformly changes the fields of a record type:

$$\text{Map} :: \Pi G{::}(\text{Type} \rightarrow \text{Type}).\ \Pi R{::}\text{Rec}.\ \{\text{r}{::}\text{Rec} \mid \text{lab}(r) = \text{lab}(R)\}$$

$$\begin{aligned}\text{Map} \triangleq \lambda G{::}(\text{Type} \rightarrow \text{Type}).\lambda R{::}\text{Rec}.&\textbf{case } R \textbf{ of}\\ &\langle l : T\rangle @ R' \Rightarrow \langle l : G(T)\rangle @ (\text{Map } G\ R')\\ &\mid \langle\rangle \Rightarrow \langle\rangle\end{aligned}$$

The kind $\{r :: \text{Rec} \mid \text{lab}(r) = \text{lab}(R)\}$ expresses the weakest invariant needed to kind Map, and G represents a function that given a type, produces a type. For example, if we apply Map (λx.String) ⟨a:Int, b:Int⟩, the resulting record is ⟨a:String, b:String⟩.

2.5 Bidirectional typing

Most statically typed programming languages offer some form of *type inference* [10] , allowing programmers to omit type annotations that can be recovered from context. Such a facility can eliminate a great deal of needless verbosity, making programs easier both to read and to write, and making more easier to a new programmer to learn.

By the study carried out by in [10], where they evaluated 160,000 lines of ML-style code, written by different programming teams, they concluded some of the properties that a type inference scheme should have in order to support the ML programming style conveniently:

- To make fine-grained polymorphism tolerable, type arguments in applications of polymorphic functions must usually be inferred. However, it is acceptable to require annotations on the bound variables of top-level function definitions (since these usually provide useful documentation) and local function definitions (since these are relatively rare);

- To make higher-order programming convenient, it is helpful, though not absolutely necessary, to infer the types of parameters to anonymous function definitions;
- To support the manipulation of pure data structures, local bindings should not usually require explicit annotations.

With that, exist two specific type inference techniques that, together, satisfy all three of the requirements listed above, the *simple synthesis of type arguments* (e.g. inference in ML-style languages) and *bidirectional propagation* or *bidirectional typing*. Both of these methods are local, in the sense that type information is propagated only between adjacent nodes in the syntax tree.

As opposed to type inference found in ML-style languages, bidirectional typing is a much more general technique, because it can be applied to more advanced type systems with higher-kinded types, impredicative polymorphism, dependent types, etc. Bidirectional propagation of type information allows the types of parameters of anonymous functions to often be inferred. A key advantage is that the technique scales to systems for which full inference is undecidable, at the cost of needing some type annotations.

2.5.1 Bidirectional simply typed lambda calculus

To explain how bidirectional typing works, an example [3] will be more easy to understand. But first, we need to introduce a non-bidirectional simply typed lambda calculus. The syntax of this type system is expressed (Figure 2.3) and have six rules (Figure 2.4) deriving the judgement $\Gamma \vdash e : A$: a variable rule, a type equality rule, a rule for type annotations, an introduction rule for unit, an introduction rule for $\rightarrow$, that is, a rule to type a function, and an elimination rule for $\rightarrow$, that is, a application of a function.

$$
\begin{array}{lrcl}
\text{Expressions} & e & ::= & x \mid \lambda x.\, e \mid e\, e \mid () \\
\text{Types} & A, B, C & ::= & \mathsf{unit} \mid A \rightarrow A \\
\text{Typing contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

Figure 2.3: Syntax for simply typed lambda-calculus [3]

$\boxed{\Gamma \vdash e : A}$ Under context Γ, expression e has type A

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \text{Var}$$

$$\frac{\Gamma \vdash e : A \qquad A = B}{\Gamma \vdash e : B}\ \text{TypeEq}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash (e : A) : A}\ \text{Anno}$$

$$\frac{}{\Gamma \vdash () : \text{unit}}\ \text{unitI}$$

$$\frac{\Gamma, x : A_1 \vdash e : A_2}{\Gamma \vdash (\lambda x.\, e) : A_1 \to A_2}\ \to\text{I}$$

$$\frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}\ \to\text{E}$$

Figure 2.4: Rules with simply typed lambda-calculus [3]

For →I rule, given an expression e of type A_2 with a free variable x, used according to type A_1, we can type $\lambda x.e$ with $A_1 \to A_2$. For →E rule, given an expression e_1 that turns type A into type B and an expression e_2 that has type A, the application of expression e_1 on e_2 results in a value of type B, and is called elimination rule because of the aplication of arrow type in an expression, eliminating the arrow type.

Bidirectional typing splits the typing judgement into two: A checking judgement where under Γ, expression e checks against type A ($\Gamma \vdash e \Leftarrow A$). And an inference or synthesis judgement where under Γ, expression e synthesizes type A ($\Gamma \vdash e \Rightarrow A$). In both, the context and the expressions will be "inputs", whereas, if we have to do a checking judgment, the type will be "input", because it is necessary to give the type to check against the expression, and in the case of the synthesis, the type will be "output", because as it synthesizes the type, then it is given as output. With that in mind, we produce each bidirectional rule in turn (treating the type equality rule last):

1. The variable rule Var has no typing premise, so our only decision is whether the conclusion should synthesize A or check against A . The information that x has type A is in Γ, so we synthesize A;
2. From the annotation rule Anno, in the premise, we check e against A and, with that, the conclusion synthesizes A;
3. Unit introduction unit checks if () has type unit;
4. In the arrow →I rule, both the premise and the conclusion, we will check against the type;

5. For the arrow →E rule, in the premises, the expression e_1 synthesizes the type $A \to B$ and we check expression e_2 has type A for the application. And so, in the conclusion, we synthesize the type B;
6. Finally, we come to the type equality rule now named Sub. There are two ways of produce this rule, either the conclusion should synthesize and the premise check, or vice-versa. The first option cannot be implemented, because we don't know B, which means we also don't know A for checking. The second option works because we can check e against a known B in the conclusion, and e synthesizes a type A, we verify that $A = B$.

The figure 2.5 show all the rules now with bidirectional typing, to help to understand more easily the changes talked before.

$\boxed{\begin{array}{l}\Gamma \vdash e \Leftarrow A \\ \Gamma \vdash e \Rightarrow A\end{array}}$ Under Γ, expression e checks against type A
Under Γ, expression e synthesizes type A

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \mathsf{Var}\Rightarrow$$

$$\frac{\Gamma \vdash e \Rightarrow A \qquad A = B}{\Gamma \vdash e \Leftarrow B}\ \mathsf{Sub}\Leftarrow$$

$$\frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (e : A) \Rightarrow A}\ \mathsf{Anno}\Rightarrow$$

$$\frac{}{\Gamma \vdash () \Leftarrow \mathsf{unit}}\ \mathsf{unitI}\Leftarrow$$

$$\frac{\Gamma, x : A_1 \vdash e \Leftarrow A_2}{\Gamma \vdash (\lambda x.\, e) \Leftarrow A_1 \to A_2}\ \to\mathsf{I}\Leftarrow$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\, e_2 \Rightarrow B}\ \to\mathsf{E}\Rightarrow$$

Figure 2.5: Rules with bidirectional typing [3]

2.6 Bidirectional typing for refinement types

The work [6] gives a detailed account of refinement type systems and their implementation using bidirectional typing techniques and an SMT solvers to discharge verification conditions (VC).The work explains how to implement refinement types, first in a simply typed lambda-calculus, and throughout the following sections, new features are added to the language or the type system. The work done in this survey is important because bidirectional typing is integrated with VC generation, needed to validate logival refinements, starting with a version of the simply-typed lambda calculus with refinements, and building up to polymorphic refinements, refinement abstraction and even inference of refinments, which makes it a good starting point to study the creation of bidirectional algorithms for refinement kinds, taking advantage of the work done for the type-checker of [6] and lifting that knowledge to elevate to the level of kinds.

The work [6] is a survey based on Liquid types [11], which is a system that combines the Hindley-Milner type inference with Predicate Abstraction to automatically infer limited forms of dependent types precise enough to prove a variety of safety properties.

2.7 Bidirectional typing for higher-rank polymorphism

In recent years, several works have developed bidirectional typing systems for rich forms of polymorphism such as higher-rank polymorphism [4], existential polymorphism and GADTs [5, 12] and even impredicative polymorphism [13]. This line of work shows that bidirectional typing systems can account for sophisticated forms of polymorphic type instantiation as an algorithmic subtyping problem, combined with forms of constraint solving. Moreover, these works show the flexibility of the bidirectional approach, since [13, 12] integrate ML-style type inference with explicit type annotations (needed for various instances of higher-rank polymorphism).

Our work will adopt these bidirectional typing techniques, using them at the level of kind checking. Notably, the works [13, 12] will be especially useful for the development of our work since they have a greater focus on the inference component.

.

Chapter 3

Proposed work

In this chapter, the work that will be carried out over the next few months will be explained, with a brief and objective explanation of what will be accomplished, how the work will be evaluated and, finally, a Gantt chart with the decomposition of tasks over time.

3.1 Description

The current implementation of refinement kinds (type checker, kind checking and an interpreter), uses the SMT Solver CVC4, however it does not fully use bidirectional typing, and therefore requires a lot of type annotations and has a somewhat unexpected behavior on those which may or may not be omitted. So, the proposed work described in the next paragraphs, will improve the existing implementation through bidirectional typing and kinding techniques.

The main objective of this work is to design and implement the bidirectional kind and type checking algorithm for refinement kinds, which will make the behavior of type-checker and kind-checker more predictable and uniform, which is a key point in this work. The algorithm will be created by combining existing bidirectional approaches to refinement types and higher-rank polymorphism, but applied at the kind-checking level. Note that bidirectional typing for high-order polymorphism [13, 12] can already integrate cleanly with ML-style inference.

Therefore, this part of the work will be divided into two major tasks: The first will be to study bidirectional checking algorithms for refinement types and for higher-rank polymorphism, where it will be necessary to understand the details of how they work and practice in a test environment and identify which variant for higher-rank applies better in this context, where in the work [12] is more focused in type inference. After that, the

previous algorithms will be implemented as kind-checking algorithms, where they will be implemented in the context of refinement kinds, that is, at the "type level".

Then, we will implement an type equality algorithm adapted from bidirectional techniques for dependent types [8], in order to, at the level of type-checking, interface with bidirectional kind-checking. With that, it is necessary, initially, to understand the details of type equality algorithms and to implement the algorithm, as an extension to the equality algorithm in the existing prototype.

Finally, we will consider various partial inference techniques for type and kind instantiation, contingent on the progress of the previous components of the work. These techniques will draw on existing work for implicit type parameters and metavariables, used in dependently-typed languages such as Idris (https://www.idris-lang.org/) and Agda [8].

3.2 Evaluation

Following the implementation of the algorithms mentioned above, we will evaluate the work through an extensive suite of examples with varying degrees of type and kind annotations, determining under which circumstances type and kind annotations can be omitted.

3.3 Gantt chart

In this section, we will present a Gantt chart with the decomposition of tasks over time, where each task represents a phase, and the phases are:

Preliminary work Research on refinement kinds and types, higher-rank polymorphism and bidirectional typing, and writing of the document.

Phase 1 Study the implementions details of bidirectional checking algorithms for refinement types and higher-rank polymorphism;

Phase 2 Implement as kind-checking algorithms;

Phase 3 Study bidirectional type equality algorithms for dependent types;

Phase 4 Implement bidirectional type equality algorithms;

Phase 5 Explore annotation minimization techniques;

Phase 6 Validation and writing of the final document.

And the work plan is:

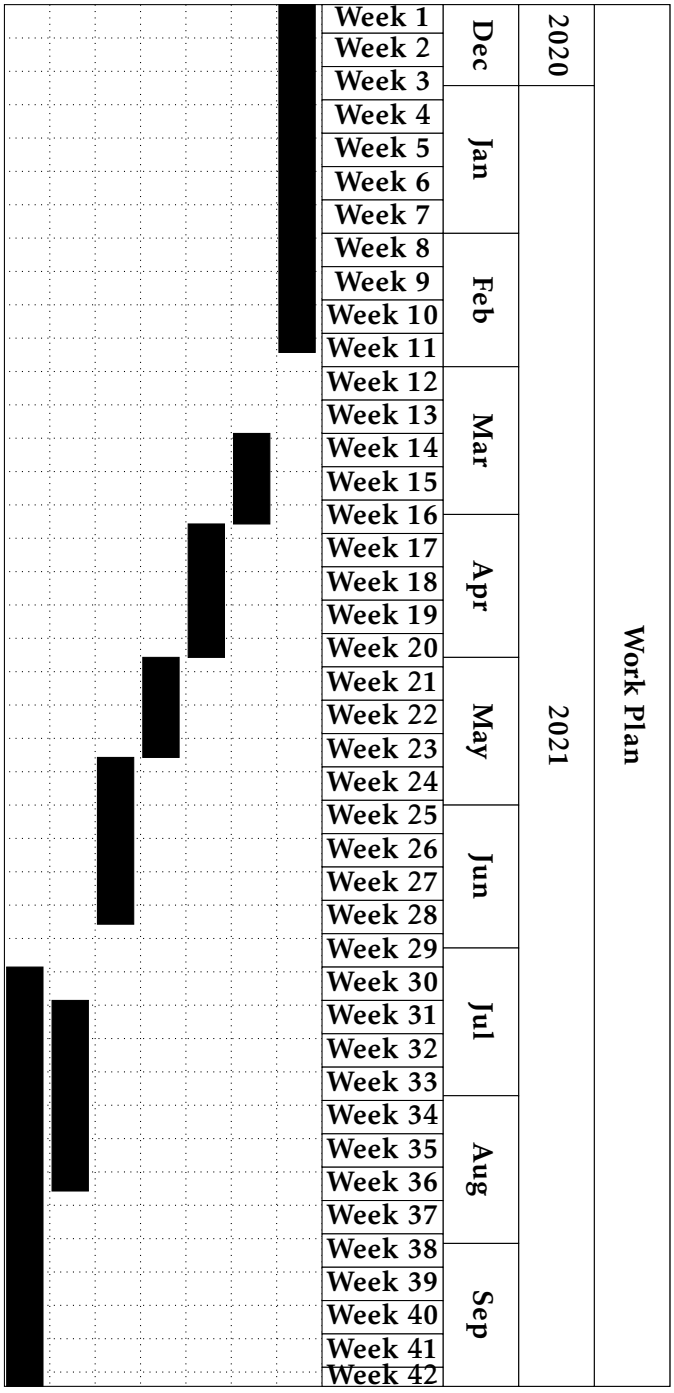

Figure 3.1: Visualization of the allocated timeframes for each phase of development.

Bibliography

[1] L. Caires and B. Toninho. "Refinement kinds: type-safe programming with practical type-level computation". In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 131:1–131:30. DOI: 10.1145/3360557. URL: https://doi.org/10.1145/3360557.

[2] A. Chlipala. "Ur: statically-typed metaprogramming with type-level record computation". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. Ed. by B. G. Zorn and A. Aiken. ACM, 2010, pp. 122–133. DOI: 10.1145/1806596.1806612. URL: https://doi.org/10.1145/1806596.1806612.

[3] J. Dunfield and N. Krishnaswami. "Bidirectional Typing". In: *CoRR* abs/1908.05839 (2019). arXiv: 1908.05839. URL: http://arxiv.org/abs/1908.05839.

[4] J. Dunfield and N. R. Krishnaswami. "Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism". In: *CoRR* abs/1306.6032 (2013). arXiv: 1306.6032. URL: http://arxiv.org/abs/1306.6032.

[5] J. Dunfield and N. R. Krishnaswami. "Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 9:1–9:28. DOI: 10.1145/3290322. URL: https://doi.org/10.1145/3290322.

[6] R. Jhala and N. Vazou. "Refinement Types: A Tutorial". In: *CoRR* abs/2010.07763 (2020). arXiv: 2010.07763. URL: https://arxiv.org/abs/2010.07763.

[7] M. Kazerounian et al. "Type-level computations for Ruby libraries". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by K. S. McKinley and K. Fisher. ACM, 2019, pp. 966–979. DOI: 10.1145/3314221.3314630. URL: https://doi.org/10.1145/3314221.3314630.

[8] U. Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. Goteborg, Sweden: Chalmers University of Technology and Goteborg University, 2007. URL: http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf.

[9] B. C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8.

[10] B. C. Pierce and D. N. Turner. "Local type inference". In: *ACM Trans. Program. Lang. Syst.* 22.1 (2000), pp. 1–44. DOI: 10.1145/345099.345100. URL: https://doi.org/10.1145/345099.345100.

[11] P. M. Rondon, M. Kawaguchi, and R. Jhala. "Liquid types". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Ed. by R. Gupta and S. P. Amarasinghe. ACM, 2008, pp. 159–169. DOI: 10.1145/1375581.1375602. URL: https://doi.org/10.1145/1375581.1375602.

[12] T. Schrijvers et al. "Complete and decidable type inference for GADTs". In: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. Ed. by G. Hutton and A. P. Tolmach. ACM, 2009, pp. 341–352. DOI: 10.1145/1596550.1596599. URL: https://doi.org/10.1145/1596550.1596599.

[13] A. Serrano et al. "A quick look at impredicativity". In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 89:1–89:29. DOI: 10.1145/3408971. URL: https://doi.org/10.1145/3408971.

[14] N. Vazou et al. "Refinement types for Haskell". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. Ed. by J. Jeuring and M. M. T. Chakravarty. ACM, 2014, pp. 269–282. DOI: 10.1145/2628136.2628161. URL: https://doi.org/10.1145/2628136.2628161.