Type-Based Termination Checking in Agda

Master's thesis in Computer science and engineering

Kanstantsin Nisht

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Master's thesis 2024

Type-Based Termination Checking in Agda

Kanstantsin Nisht

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Type-Based Termination Checking in Agda
Kanstantsin Nisht

Supervisor: Andreas Abel, Department of Computer Science and Engineering
Examiner: Thierry Coquand, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in $\LaTeX$
Gothenburg, Sweden 2024

Type-Based Termination Checking in Agda
Kanstantsin Nisht
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

We present a description of a type-based termination checker for the dependently-typed language Agda.

Our termination checker uses System $F_\omega^{\text{cop, SCT}}$ as the semantical foundation – a variant of strongly-normalizing higher-order polymorphic lambda calculus with pattern and copattern matching. In this work, we provide a proof of strong normalization for System $F_\omega^{\text{cop, SCT}}$, which is based on Girard-Tait reducibility candidates in combination with the Size-Change Principle applied to well-founded sized types.

We also provide an algorithm of size annotation inference for terms written in System F_ω, and prove its soundness. The algorithm has linear time complexity depending on the size of the syntax, which makes it admissible for practical implementation.

This work also discusses our implementation of the outlined algorithm for Agda, and we show that the proposed termination checker features acceptable performance and significant increase in expressivity of termination checking for proof assistants.

Keywords: Termination, type theory, proof assistants, sized types, size-change principle.

Contents

1
Introduction

Termination checking holds significant importance within the field of programming language theory. At its core, it involves determining whether a function halts its execution for all possible input values. Notably, this problem has been formally proven to be undecidable by Turing [Turing et al., 1936].

Nonetheless, there exists a clear motivation to address this challenge to some extent. Specifically, it is often feasible to generate a termination certificate—an evidence that attests to the termination of a function. It is important to note, however, that it is not feasible to provide termination certificates for all computable functions.

One particularly valuable application of termination checking emerges within the domain of *dependently typed languages*. These languages extend lambda calculus by allowing types to depend on terms, thereby significantly enhancing expressiveness. However, the type-checking process is tightly coupled with evaluation – if a type uses a function, how can we know that a function is safe to evaluate? In other words, we need to know that a function is terminating in order to use it in the type signature. Having non-terminating functions would make the problem of type checking a dependently typed term undecidable, which is a major obstacle for implementing these languages as computer programs.

Another usage of termination checking in dependently typed languages lies in the theoretical aspect. Defining recursive functions is highly beneficial as it enables the principle of mathematical induction, which is foundational in mathematical reasoning. The most basic form of induction, known as primitive recursion, involves equipping each datatype in the theory with a higher-order function, which is called *induction*. This function allows for the definition of any term dependent on an element of the datatype through the use of special term formers—constructors. For instance, consider the induction corresponding to natural numbers: $ind : \forall A.\ A \to (\mathbb{N} \to A \to A) \to \mathbb{N} \to A$. It enables the construction of a term of type A based on an arbitrary natural number, given the values of A at zero and the successor—defined by the standard constructors of natural numbers in Peano arithmetic.

The issue here is that the use of induction functions is quite restrictive and inconvenient. In this context, most dependently typed languages employ an approach borrowed from functional programming, known as *pattern matching*. The essence of this approach lies in the direct definition of a function's behavior on different constructors of the provided parameters, with recursion utilized to reference the outcomes of computation on these parameters. The advantage of this method is that a function can depend on several arguments simulataneously, which is difficult to achieve with primitive recursion principle. However, a drawback arises in that recursion is represented simply as a plain function application, leaving it to an external checking process to determine if the call is safe.

It is important to ensure the safety of the recursive calls, as unrestricted recursion can quickly render the logic of a dependently typed language trivial. Consider a term $f : \mathbb{N} := f$: this term passes the type-checking, as it simply calls itself, but it is non-terminating (i.e., it can be unfolded infinite number of times). Further, by definition it can represent any type, including the empty one. In particular it means that every type in the theory is inhabited, which is the definition of triviality of a type theory.

In most dependently typed languages there is a special component, which is called *termination checker*. The aim of this project is to investigate the utilization of type information for the purpose of termination checking.

1.1 Existing Approaches

Currently, termination checkers are based on a simple principle: each recursive call must involve a structurally smaller argument compared to the original parameter of a function. The expected way to obtain structurally-smaller terms is the use of pattern-matching. Indeed, pattern-matching on natural numbers brings smaller numbers in the scope, and making recursive calls with these numbers should be safe.

As you might guess, the termination checker is inherently incomplete since it aims to solve an undecidable problem. This implies that there is always room for improvement in the algorithm of termination checking, although it may come at the cost of performance. For example, one particularly powerful method [Jones et al., 2001] has a very high computational complexity.

The concept of type-based termination is not novel either. The approach here, known as sized types, involves annotating types with their respective sizes. The theoretical framework surrounding sized types is considerably more intricate than that of structural recursion [Barthe et al., 2006], suggesting the potential for discovering a theory better suited for termination checking. Additionally, a common finding in works within this field is the necessity for explicit size annotation of terms by the user.

Furthermore, an endeavor to implement type-based termination in Rocq [Chan et al., 2023] concludes that type-based termination checking based on sized types is not feasible due to performance considerations.

1.2 Contribution

In this work, we introduce an approach to type-based termination that integrates sized types [Abel and Pientka, 2016], higher-order polymorphic lambda-calculus [Barendregt, 1991], and the size-change termination principle [Jones et al., 2001]. We present a formal system that accommodates definitions by both pattern- and copattern-matching, incorporates size annotations, and demonstrates the property of strong normalization—i.e., the absence of infinite reduction sequences. Additionally, we offer an algorithm for inferring size annotations for this system, enabling the development of an implicit type-based termination checker. As a practical outcome, we implement the proposed termination checker for a dependently typed language, Agda.

The theoretical solution we propose is novel in the sense that it combines sized types and size-change termination. Furthermore, our work contributes by developing an algorithm for inferring size annotations for the well-founded flavor of sized types, a task that has not been previously attempted. Finally, our implementation for Agda demonstrates the potential for practical usage of sized typing, challenging previous claims that such usage was infeasible [Chan et al., 2023].

1.3 Structure

In chapter 2 we give a general overview of the parts of type theory that are relevant to this thesis. The reader interested in semantical part of this thesis is welcome to read section 2.2 to get prepared for the language of formal systems we use later. The reader interested in practical parts is welcome to review section 2.4 and section 2.5 where we briefly describe Agda – the target language of practical implementation of this work, – and the existing approaches to termination checking in this language.

In chapter 3, we describe the intuition behind the proposed approaches. If the reader has to choose only one chapter to look at, we highly advise choosing chapter 3. The reason is that there we try to convince the audience of our ideas without diving into technical details.

In chapter 4, we describe the syntax of System $F_\omega^{\text{cop, SCT}}$, the sized higher-order polymorphic lambda-calculus which serves as the target system of our proposed termination checker. Additionally, we provide a technical overview of the size-change termination principle [Jones et al., 2001] in section 4.4, which serves as a powerful tool in our inference algorithms.

In chapter 5, we provide a proof of strong normalization for System $F_\omega^{\text{cop, SCT}}$. This chapter contains the main theoretical novelty of this work, namely, the application of the size-change principle to sized typing.

In chapter 6, we describe the termination checking for System F_ω and prove its soundness. This chapter contains a theoretical description of the algorithm for termination checking, and therefore can serve as a reference for practical implementation.

In chapter 7 we provide the architectural details and overview the challenges we met during the implementation process of the proposed termination checker.

In chapter 7, we provide architectural details and overview the challenges we encountered during the implementation process of the proposed termination checker.

In chapter 9, we offer a more precise overview of the work presented in this thesis and discuss its impact on the future of type theory.

2
Background

This section aims to briefly introduce the reader to the main objects studied in this work, namely, lambda-calculus, higher-order polymorphic lambda-calculus, dependently-typed systems, Agda, and the existing methods for proving termination in Agda: the current termination checker and sized types.

2.1 Lambda Calculus

Lambda calculus serves as a formalism for describing computations as a mathematical object. It is inherently simple, characterized by a grammar of terms comprised of three constructs, with the first two termed *abstraction* and *application*.

$$T ::= \lambda x.\, T \mid T\ T \mid x$$

Additionally, lambda calculus entails a single rewrite rule known as *β-reduction*, where $T[x := T']$ represents the process of substituting the term T' for x within T:

$$(\lambda x.\, T_1)\ T_2 \Rightarrow T_1[x := T_2]$$

This formalism is highly expressive, with a conjecture suggesting that any intuitively computable function can be expressed using lambda calculus [Kleene, 1943].

The power of lambda calculus also enables the expression of non-terminating computations. For instance, the term $(\lambda x.; x; x); (\lambda x.; x; x)$ can undergo infinite reduction. However, for certain applications, the presence of infinite reduction sequences is undesirable. The property of absence of infinite reduction sequences in the calculus is referred to as *strong normalization*, and a calculus possessing this property is termed *strongly normalizing*.

One of the popular approaches to rule out infinitely reducible constructs is to annotate terms with *types*. The most basic extension of lambda calculus with types is called *simply typed lambda calculus*, where types are formed from $A ::= A \mid A \to A$. The grammar for terms remains the same, but typically, attention is restricted to a certain set of *well-typed* terms. Well-typedness is typically expressed as a ternary relation $\boxed{\Gamma \vdash t : A}$, indicating that a term t has type A within context Γ. This relation is built using the following rules:

$$\frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\ s : B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\ t : A \to B} \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$

There are known results that simply-typed lambda calculus is strongly normalizing [Joachimski and Matthes, 2003].

2.2 System F_ω

Simply-typed lambda calculus is indeed not very expressive. One motivation for lambda calculus is to serve as a semantical counterpart of programming languages, which are typically more complex and feature operations beyond abstraction and application. One notable extension is *polymorphism*—the ability to define a function that operates simultaneously on multiple types. Polymorphic lambda calculus is known as *System F* [Reynolds, 1974] [Girard, 1972].

Let's delve a little deeper and define *higher-order* polymorphic lambda calculus, denoted as System F_ω [Barendregt, 1991]. Its main distinction from System F is the presence of type operators. An interesting property of System $F_\omega$is that on the type-level, it resembles simply-typed lambda calculus with a single *kind* $*$, which serves as the base type in simply-typed lambda calculus. On the term level, there exists a *type lambda* Λ, which allows for explicit quantification over types within terms.

Kinds ι in System $F_\omega$are generated by the following grammar:

$$\iota ::= * \mid \iota \to \iota$$

Types in System $F_\omega$are generated by the following grammar:

$$A ::= X \mid A \to A \mid \forall X : \iota.\ A \mid \lambda X : \iota.\ A \mid A\ A$$

We are now ready to present the rules of *well-formedness for types*, denoted as $\boxed{\Delta \vdash_F A : \iota}$. Here, Δ is a type context, A is a type, and ι is a kind. The relation $=_\beta$ represents *beta-convertibility*, which holds if one term is β-reducible to another.

$$\frac{(X : \iota) \in \Delta}{\Delta \vdash_F X : \iota} \qquad \frac{\Delta \vdash_F A : \iota_1 \to \iota_2 \qquad \Delta \vdash_F B : \iota_1}{\Delta \vdash_F A\ B : \iota_2} \qquad \frac{\Delta, X : \iota_1 \vdash_F A : \iota_2}{\Delta \vdash_F \lambda X : \iota_1.\ A : \iota_1 \to \iota_2}$$

$$\frac{\Delta, X : \iota \vdash_F A : *}{\Delta \vdash_F \forall X : \iota.\ A : *} \qquad \frac{\Delta \vdash_F A : * \qquad \Delta \vdash_F B : *}{\Delta \vdash_F A \to B : *}$$

Now we can present the rules of well-typedness of terms in System F_ω.

$$\frac{\Delta; \Gamma \vdash_F r : A \to B \qquad \Delta; \Gamma \vdash_F s : A}{\Delta; \Gamma \vdash_F r\ s : B} \qquad \frac{\Delta; \Gamma \vdash_F r : \forall A : \iota.\ B \qquad \Delta \vdash_F G : \iota}{\Delta; \Gamma \vdash_F r\ G : B[A := G]}$$

$$\frac{\Delta; \Gamma \vdash_F t : A \qquad A =_\beta A'}{\Delta; \Gamma \vdash_F t : A'}$$

$$\frac{x : A \in \Gamma}{\Delta; \Gamma \vdash_F x : A} \qquad \frac{\Delta; \Gamma, x : A \vdash_F r : B \qquad \Delta \vdash A : *}{\Delta; \Gamma \vdash_F \lambda x : A.\ r : A \to B} \qquad \frac{\Delta, A : \iota; \Gamma \vdash_F r : B}{\Delta; \Gamma \vdash_F \Lambda A.\ r : \forall A : \iota.\ B}$$

One notable feature of System $F_\omega$is its allowance for the introduction of *fixpoint operators* $\mu, \nu : (* \to *) \to *$ (representing the least and greatest fixpoints, respectively). These type operators are utilized to introduce inductive and coinductive types. For instance, the type of natural numbers $\mathbb{N}$ can be expressed as $\mu(\lambda X.\ 1 + X)$, where $+ : * \to * \to *$ denotes a sum type operator.

2.3 Dependent Types

One particularly powerful extension of lambda calculus is dependent types. Essentially, dependently typed languages permit the usage of terms within types, thereby blurring the distinction between these two realms. We present the definition of a dependently typed system from [Barendregt, 1991]. The grammar of terms and types is unified:

$$A, B, C ::= x \mid \lambda x : A.\ B \mid A\ B \mid \Pi x : A.\ B \mid U$$

Here, Π serves as a generalization of $\to$ from System F_ω, and U is a generalization of a kind, referred to as *universe.* In this exposition, we consider the set of universes to be $*, \Box$. With this foundation, we can now present the rules of well-typedness $\Gamma \vdash_D t : A$ for a dependent type system:

$$\frac{}{\Gamma \vdash_D * : \Box} \qquad \frac{\Gamma \vdash_D A : U \qquad x : A \in \Gamma}{\Gamma \vdash_D x : A} \qquad \frac{\Gamma \vdash_D A : B \qquad \Gamma \vdash_D C : U}{\Gamma, x : C \vdash_D A : B}$$

$$\frac{\Gamma \vdash_D A : U_1 \qquad \Gamma, x : A \vdash_D B : U_2 \qquad (U_1, U_2, U_3) \in \{(*, \Box, \Box), (\Box, \Box, \Box), (\Box, *, *), (*, *, *)\}}{\Gamma \vdash_D \Pi x : A.\ B : U_3}$$

$$\frac{\Gamma, x : A \vdash_D t : C \qquad \Gamma \vdash_D \Pi x : A.\ C : U}{\Gamma \vdash_D \lambda x : A.\ t : \Pi x : A.\ C} \qquad \frac{\Gamma \vdash_D f : \Pi x : A.\ B \qquad \Gamma \vdash_D a : A}{\Gamma \vdash_D \lambda f a : B[x := a]}$$

$$\frac{\Gamma \vdash_D t : A \qquad \Gamma \vdash_D A' : U \qquad A =_\beta A'}{\Gamma \vdash_D t : B'}$$

Dependent types offer significant expressive power, rendering them suitable for describing and proving mathematical theorems. Internally, dependent types are grounded in intuitionistic type theory, enabling the definition of a decidable type-checking algorithm for them. This, in turn, makes them well-suited as a type system for programming languages, with Agda being one prominent example.

2.4 Agda

Agda [Norell, 2007] is a programming language featuring dependent types. Its syntax draws inspiration from Haskell [Marlow et al., 2010], thus having its roots in lambda calculus.

Agda naturally incorporates System $F_\omega$as part of its type system, enabling the expression of the following definition (where Set is analogous to $*$):

```
f : (F : Set → Set) → (A : Set) → (G : Set → A) → A
f F A G = G (F A)
```

In Agda, it is possible to define inductive data types by specifying their *constructors*. These data types can also be parameterized by types and values, enabling the creation of polymorphic definitions. Here, we present examples of natural numbers $\mu(\lambda X.\ 1 + X)$ and lists $\lambda A : *.\ \mu(\lambda X : *.\ 1 + (A \times X))$ in Agda:

```
data Nat : Set where
  zero : Nat
  suc : Nat → Nat

data List (A : Set) : Set where
  nil  : List A
  cons : A → List A → List A
```

In Agda, defining functions involves *pattern matching*: the user specifies a set of clauses, akin to rewrite rules. When a function is invoked with arguments matching the left-hand side of any clause, the function can be substituted with the right-hand side of that clause, with bound variables in the pattern substituted for corresponding parts of the arguments. The utilization of induction principles for data types is facilitated through recursive calls.

```
add : Nat → Nat → Nat
add zero y = y
add (suc x) y = suc (add x y)
```

In contrast to induction principles, which focus on finitely-generated data, Agda also supports coinductive data types, which are inherently infinite. For instance, the type of streams $\lambda A : *.\ \nu(\lambda X : *.\ A \times X)$ in Agda can be represented as a *coinductive record*:

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A

open Stream
```

The preferred method for working with coinductive data involves the use of coinductive functions, which are defined using *copattern matching*. This approach is dual to pattern matching in that it decomposes the output rather than the input. The concept is that a function returning a coinductive type defines its behavior on the fields of that type. The function then reduces only when a field is applied to it, which is necessary to avoid infinite unfolding. Here, we present an infinite stream of zeros.

```
zeros : Stream Nat
zeros .head = zero
zeros .tail = zeros
```

The key observation here is that zeros does not unfold by itself. Since our terms are finite, they contain only a finite number of applications of .tail, which means that zeros can only be unfolded a finite number of times. This approach allows for the retention of strong normalization in the presence of infinite data structures such as coinductive records.

2.5 Termination Checker

In section 2.4, we presented a definition of a function add using pattern matching, where recursive calls to this function occur in the right-hand side of clauses. This definition is accepted by Agda because the termination checker can verify its safety. The logic here is evident: the calls to add are made on structurally smaller data, so if we provide a finitely constructed argument (and natural numbers are finite), the function will eventually terminate.

However, the capabilities of the termination checker in Agda extend beyond the simple requirement that an argument to a function should be strictly smaller. Consider the following definition of the *Ackermann function*:

```
ack : Nat → Nat → Nat
ack zero n          = suc n
ack (suc m) zero = ack m (suc zero)
ack (suc m) (suc n) = ack m (ack (suc m) n)
```

Here, we observe that the inner call to ack occurs on an argument that is *not* structurally smaller than the first argument. However, this definition is still accepted by Agda. The reason for this is that in this call, the first argument remains the same as the parameter of the enclosing function, while the second argument is smaller. Agda recognizes that with each call, either the first argument decreases or remains the same, while the second argument decreases. By applying lexicographical induction, Agda concludes that the function terminates.

The termination checker in Agda is grounded in the Size-Change Principle [Jones et al., 2001], which is a potent method for determining whether a set of mutually-recursive functions terminates based on the relation between arguments and parameters. If a set of these relations satisfies certain computable criteria, then the function is deemed terminating by Agda. This criterion encompasses any form of lexicographical induction.

While the Size-Change Principle applied in Agda is a direct interpretation of the classical algorithm [Jones et al., 2001], it's worth noting that there exists a generalized framework for the application of this method in dependent type theory [Wahlstedt, 2007].

Coinductive functions also need to satisfy a certain variant of a termination criterion. Here, it is referred to as *guardedness* [Coquand, 1994]. The concept is that recursive usages of coinductive functions should be enclosed within a coinductive constructor. This condition is automatically ensured for definitions by copattern matching. However, an additional requirement is that the coinductive function should not be wrapped in anything else. For example, the following definition *does not* pass the guardedness check:

```
wrong-zeros : Stream Nat
wrong-zeros .head = zero
wrong-zeros .tail = wrong-zeros .tail
```

Here, wrong-zeros.tail is actually a syntactic sugar for the function tail applied to wrong-zeros. It is apparent that wrong-zeros.tail can be infinitely unfolded. Therefore, to ensure strong normalization, Agda prohibits such functions.

2.6 Sized Types

The concept of using types to ensure termination is not novel. One approach to type-based termination checking in functional languages is *sized types*. This idea originated for Haskell-like languages [Hughes et al., 1996] and initially utilized domain theory [Scott, 1976] to establish soundness.

The further theoretical evolution of sized types unfolds as follows:

- Barthe [Barthe et al., 2005] pioneered sized typing for System F and devised an inference algorithm for it. Subsequently, this work was generalized to the Calculus of Constructions [Barthe et al., 2006]. Ultimately, this effort culminated in an implementation of an implicit termination checker based on sized types for Rocq [Chan and Bowman, 2019], although this attempt was later revealed to be unsuccessful [Chan et al., 2023].

- In parallel, Abel [Abel, 2006] explored the application of sized types for System F with equirecursive data types. One notable result of this theoretical development [Abel and Pientka, 2016] demonstrates that the requirements on signatures with size type annotations can be lifted while still preserving the proof of strong normalization.

A practical implementation of sized types was carried out in Agda. Their utilization enables the proof of termination for non-trivial recursive functions. For instance, consider the division of x by $y + 1$. Typically, this function cannot be verified by the syntactic termination checker, as it employs a non-structural recursion principle. However, with the aid of sized types, termination can be established as follows:

```
{-# OPTIONS --sized-types #-}
open import Agda.Builtin.Size

data Nat : Size → Set where
  zero : {i : Size} → Nat i
  suc : {i : Size} → Nat i → Nat (↑ i)

minus : {i : Size} → Nat i → Nat ∞ → Nat i
minus zero x = zero
minus (suc x) zero = (suc x)
minus (suc x) (suc y) = minus x y

div : {i : Size} → Nat i → Nat ∞ → Nat i
div zero x = zero
div (suc x) y = suc (div (minus x y) y)
```

One notable characteristic here is that sized types are *infective*: if certain code employs sized types, then any code that interacts with it must also acknowledge sized types. This precludes the compartmentalization of type-based termination techniques, particularly when sized types do not seamlessly integrate with other features of Agda.

3 Main Idea

In this chapter, we delve into the conceptual underpinnings of our proposal for the type-based termination checker and delineate an approach to its implementation. The discussion presented here represents a further elaboration of the ideas elucidated in [Abel and Pientka, 2016], which forms the technical bedrock of this project.

The term *type-based termination* inherently suggests that termination information should be conveyed at the type level. This stands in contrast to a traditional termination checker, which infers termination information based on terms. Consider a function $f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, which may be recursive. We can annotate each input with a size, yielding $f : (i : \text{Size}) \to (j : \text{Size}) \to \mathbb{N}^i \to \mathbb{N}^j \to \mathbb{N}$. The process of annotation is straightforward: since we can pattern-match on parameters to obtain something structurally smaller than the parameters themselves, we should annotate all parameters with associated size information. Moreover, since a function can be invoked with arbitrary arguments, the size information labels are independent.

3.1 Induction

Let's illustrate this concept with an example involving a finitely-branching tree:

```
data RoseTree' (A : Set) : Set where
  rose : A → List (RoseTree' A) → RoseTree' A
```

One intuitive principle regarding inductive data types is that they are finite; thus, subterms of a larger element of an inductive data type can be regarded as smaller than the element itself. This observation suggests that we can apply the following size annotations to this definition:

```
data RoseTree (i : Size) (A : Set) : Set where
  rose : {j : Size< i} → A → List (RoseTree j A) → RoseTree i A
```

This implies that any occurrence of a sub-rose-tree within a larger rose-tree will be smaller than the entire rose-tree. Furthermore, it underscores the notion that the utilization of a constructor yields a larger data type.

Before moving forward, it's necessary to define a function *map* that applies a given function to every element of a list. An important observation is that this function is polymorphic, meaning it cannot alter the internal structure of the list. This property, known as *parametricity* [Wadler, 1989], holds true in Agda.

```
map : {A B : Set} → (A → B) → List A → List B
map f nil = nil
map f (cons x xs) = cons (f x) (map f xs)
```

This function terminates because it calls itself on a structurally smaller argument, thereby obviating the need for size annotations.

Now we are ready to define a function *mapRose*, which applies a provided function to every element of the list. This function, in contrast to *map*, is not structurally recursive, and we need our size annotation to show its termination.

```
mapRose : {i : Size} → {A B : Set} → (A → B) → RoseTree i A → RoseTree ∞ B
mapRose f (rose {j} x rest) = rose {∞} (f x) (map (mapRose {j} f) rest)
```

The key insight here is that mapRose is invoked with a smaller size j, which is lesser than i. This reduction in one of the arguments serves as evidence of the function's termination.

The challenge now lies in achieving this without explicit insertion of sizes. To address this, let's focus on the function map, which is polymorphic. Its first parameter is a function $A \to B$, which in our case is RoseTree $\{k\}$ $A \to$ RoseTree $\{l\}$ B. Additionally, we observe that the second parameter of map is List (RoseTree $\{k\}A$), which is analogous to the first argument of a function. Now, considering that the second argument is List (RoseTree $\{j\}$ A), we can infer that $k = j$, indicating that mapRose f is invoked on structurally smaller rose-trees.

It's crucial that the function is defined by pattern-matching. With type-based termination enabled, Agda can monitor the "sizes" of pattern-matched data in the left-hand side of clauses, thereby treating polymorphic functions as size-invariant.

Our algorithm of type-based termination treats all type variables as carrying the same size. For instance, the following function g, while strongly normalizing, would fail the termination check.

```
app : {A B : Set} → (A → A) → (A → B) → A → B
app i f x = f (i x)

g : Nat → Nat
g zero = zero
g (suc zero) = zero
g (suc (suc n)) = app suc g n
```

The issue arises because suc is expected to be size-preserving, which it is not. In the definition of app, we could apply i an arbitrary number of times, leading to g becoming non-terminating.

Our analysis extends to a set of mutually-recursive functions:

```
h : Nat → Nat
i : Nat → Nat

h zero = zero
h (suc n) = app (λ x → x) i n
i zero = zero
i (suc n) = app (λ x → x) h n
```

The essence of our inference lies in treating sizes as additional parameters of mutually recursive functions. We then apply the size-change principle [Jones et al., 2001] to determine whether a set of calls is safe. This allows us to generate a termination certificate inferred from these synthetic size parameters.

3.1.1 Size preservation

Sometimes, polymorphic functions alone are insufficient to address the problem. For instance, the division function mentioned earlier necessitates sized types to prove its termination to Agda.

```
minus : Nat → Nat → Nat
minus zero x = zero
minus (suc x) zero = (suc x)
minus (suc x) (suc y) = minus x y

div : Nat → Nat → Nat
div zero x = zero
div (suc x) y = suc (div (minus x y) y)
```

We observe that *minus* is not polymorphic, indicating that it cannot be covered by the process described above.

To handle such functions, the type-based termination checker is extended with a ”size preservation analysis”, a process that involves understanding the dependencies in data types within function signatures. Examining the function *minus*, we notice that its result is never larger than the first argument. Given that we understand how all clauses behave regarding size information, this dependency between the result and the first argument becomes inferable. Consequently, we learn that *minus* x y is not larger than x, which, in turn, demonstrates that *div* is called on a smaller argument than (*suc* x). Thus, we can conclude that *div* is terminating.

It is crucial to grasp the process of selecting data types for size preservation. The intuition here lies in distinguishing between users' *input* and *output*. For instance, the function can be decomposed into input and output by separating its domain and codomain. However, our separation goes a bit further: we categorize the data types in the signature into *positive* and *negative* positions. For example, consider the following signature:

```
record _×_ (A B : Set) : Set where
  field
    fst : A
    snd : B

r : (Nat₁ → Nat₂) → Nat₃ → Nat₄ × Nat₅
```

Here, Nat_2 and Nat_3 occur *negatively*, and they are under the control of the user. Hence, the type-based termination checker assigns different and independent sizes to them.

On the contrary, Nat_1, Nat_4, and Nat_5 occur *positively*, and the signature can be size-preserving precisely in these arguments. For example, Nat_1 is not under the control of the user – the arguments to it are supplied within r, and the argument may always be that for Nat_3. In this case, Nat_1 would have the same size as Nat_3.

3.2 Coinduction

The idea above also useful for checking the productivity of coinductive function.

We shall remind the definition of infinite streams in Agda:

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A

open Stream
```

First, we need to explain a shift in intuition for sized types when applied to coinductive definitions. Normally, sized types represent a "size" of a term. This is a valid intuition for inductive data types, since they are finite and can be assigned an ordinal representing the level of nestedness. For example, a term for the natural number 3, which is represented in Agda as `(suc (suc zero))`, can be assigned a size of 3. However, this intuition does not work well when infinite data structures are involved. How can the size of an infinite stream be anything other than ∞?

Following [Abel and Pientka, 2016], we propose to think about the size of streams as their "depth", meaning a stream of size 3 represents a stream from which we are allowed to take three elements. In this sense, streams are contravariant in their size; that is, we can take 3 elements from an infinitely deep stream but cannot do so for a stream with depth 2. Note that the variance here is opposite to that of inductive data types.

With this in mind, we can provide a definition of sized streams:

```
record Stream' (i : Size) (A : Set) : Set where
  coinductive
```

```
    field
      head : A
      tail : {j : Size< i} → Stream' j A

  open Stream'
```

The rationale behind this is that the operation *tail* enables us to obtain a stream of smaller depth from a stream of greater depth. Similar to inductive data types, the process of annotation here is straightforward.

Now, let's provide an example of a simple coinductive function, namely, an infinite stream of zeros with inserted sizes:

```
  zeros' : {i : Size} → Stream' i Nat
  zeros' .head = zero
  zeros' .tail {j} = zeros' {j}
```

Opposing functions with inductive types, where we assign size annotations to parameters, here we annotate *the returned type*, because that is what we define by copattern matching.

Now, the process of copattern matching with *tail* results in a new size variable in the scope, namely j. The type of the right-hand side of the clause in this case is `Stream` j $\mathbb{N}$, so we need to provide a *deeper* stream than `Stream` j $\mathbb{N}$. In other words, we need such `Stream` k $\mathbb{N}$ where $k \geq j$. However, we do not want an arbitrarily big stream, since we need a proof that we are defining a deep stream in terms of shallower streams. This leads us to a conclusion that a suitable assignment would be $k := j$.

As another example, this function would not pass the termination check:

```
  wrong-zeros' : {i : Size} → Stream' i Nat
  wrong-zeros' .head = zero
  wrong-zeros' .tail {j} = wrong-zeros' .tail
```

The reason is that the invocation of *.tail* on the right-hand side requires some size variable k_1, and the recursive invocation of $wrong - zeros'$ also requires a size k_2 where $k_2 > k_1$. If k_1 is assigned to j (the smallest size in the scope), then k_2 can be assigned to i at best, and as a result, there is no proof that $wrong - zeros$ is defined in terms of shallow streams.

3.2.1 Size preservation

Indeed, the significance of size preservation in coinductive functions surpasses that in inductive ones. As demonstrated earlier, we attributed an independent size variable to the return type of *zeros*. This choice isn't arbitrary; it stems from the contravariance of coinductive types.

In practice, we annotate all positive occurrences of coinductive types with separate size variables, while negative occurrences can retain size preservation. This strategy accounts for the fact that users have control over the output of coinductive functions. Since the input is inherently infinite and externally supplied, careful analysis ensures that the function doesn't consume more than it produces, a crucial aspect for guaranteeing productivity.

With the aid of coinductive size preservation, Agda accepts functions such as the following:

```
zipWith : {A B C : Set} → (A → B → C) → Stream A → Stream B → Stream C
zipWith f s1 s2 .head = f (s1 .head) (s2 .head)
zipWith f s1 s2 .tail = zipWith f (s1 .tail) (s2 .tail)

fib : Stream Nat
fib .head = zero
fib .tail .head = suc zero
fib .tail .tail = zipWith add fib (fib .tail)
```

The reason why fib passes the termination check is that zipWith preserves size in both its stream parameters. Agda understands that zipWith does not consume more than it produces, and annotates both arguments with the same size as the output. Later, in the context of fib, this information helps Agda recognize that zipWith does not call .tail on the provided fib arguments in a way that would violate termination, allowing fib to be considered terminating overall.

4

Syntax

In this chapter, we provide a syntactic overview of System $F_\omega^{\text{cop, SCT}}$, an extension of System F_ω^{cop} [Abel and Pientka, 2016] incorporating the size-change termination principle [Jones et al., 2001]. It is worth noting that the system bears similarities to Agda, supporting the definition of datatypes, mutual recursion, pattern matching, and copattern matching.

We will only elucidate the rationale behind specific syntactic constructs. For a more comprehensive understanding of the design decisions for System F_ω^{cop}, we direct the reader to [Abel and Pientka, 2016].

The distinctions between System $F_\omega^{\text{cop, SCT}}$ and System F_ω^{cop} will be emphasized in light gray .

The primary distinction lies in the absence of measures in System $F_\omega^{\text{cop, SCT}}$, unlike in System F_ω^{cop} where they are used to facilitate lexicographic induction on tuples of ordinals in the semantic interpretation. Instead, our system relies on an external requirement, which is considered more robust. Additionally, we expand the structure on size expressions to a bounded meet-semilattice, a necessity for the inference algorithm discussed later. Further rationale for introducing the semilattice structure is elaborated in section 6.3.

4.1 Kinds and Sizes

Let's begin our overview of the syntax of System $F_\omega^{\text{cop, SCT}}$ with the system of kinds. The grammar for kinds is represented as follows:

$$\kappa ::= * \mid {<}a \mid \pi\kappa \to \kappa$$

Note the non-standard kind $< a$, which is not present in traditional System F_ω. This special kind allows us to emulate judgments about sizes without the use of dependent types. We should also remark the usage of *polarity* π, which refers to covariant ($\pi = +$), contravariant ($\pi = -$), constant ($\pi = \top$), and mixed ($\pi = \circ$) variances.

The grammatical structure for kinds in our syntax is presented in Table 1.

SizeVar	$\ni i,\ j$		Size variable
SizeMin	$\ni a^\wedge,\ b^\wedge$	$::= (i+n) \wedge (j+m) \mid a^\wedge \wedge (j+n)$	Minimum of size variables
SizeExp	$\ni a,\ b$	$::= i+n \mid \infty + n \mid a^\wedge$	Size expression ($n \geq 0$)
Pol	$\ni \pi$	$::= \circ \mid + \mid - \mid \top$	Polarity/variance
SizeCtx	$\ni \Psi$	$::= \cdot \mid \Psi, i : \pi(< a)$	Size variable context
SKind	$\ni \iota, \iota'$	$::= * \mid o \mid \iota \rightarrow \iota'$	Simple kind
SCtx	$\ni \lvert\Delta\rvert$	$::= \cdot \mid \lvert\Delta\rvert, X : \iota$	Simple kinding context
Kind	$\ni \kappa, \kappa'$	$::= * \mid {<}a \mid \pi\kappa \rightarrow \kappa'$	Kind with variance
TyCtx	$\ni \Delta$	$::= \cdot \mid \Delta, X : \pi\kappa$	Type variable context

Table 1: Grammar description for kinds and sizes

4.1.1 Polarities

Our motivation for introducing polarities is explained by the fact that our types are parameterized by ordinal-like objects (the sizes), hence we need to introduce subtyping. To interpret the ordering on type operators, we need to know whether they are monotone, antitone, or constant in their arguments, so they are obliged to have polarity annotations. A thorough discussion of higher-order polarized subtyping can be found in [Abel, 2006].

Polarities form a lattice, where $\circ < + < \top$ and $\circ < - < \top$. The polarities can be composed and inverted, which is useful during the kind-checking procedure.

Formally, $\boxed{\pi < \pi'}$ represents the rule for the lattice of polarities. We can think of $\circ$ as representing a loss of information, and $\top$ as representing an "unused" polarity.

$$\frac{}{\pi \leq \pi} \qquad \frac{}{\circ \leq \pi} \qquad \frac{}{\pi \leq \top}$$

$\boxed{\pi\pi'}$ represents the rule for the composition (commutative) of polarities.

$$\top\pi = \top \qquad \circ\pi = \circ\ (\pi \neq \top) \qquad +\pi = \pi \qquad -- = +$$

$\boxed{\pi^{-1}\pi}$ represents the rule for inverse composition of polarities.

$$\top^{-1}\pi = \circ \qquad \circ^{-1}\circ = \circ \qquad \circ^{-1}\pi = \top\ (\pi \neq \circ) \qquad +^{-1}\pi = \pi \qquad -^{-1}\pi = -\pi$$

$\boxed{\pi\Delta}$ and $\boxed{\pi^{-1}\Delta}$ extend the rules of composition to the typing context.

$$\pi\cdot = \cdot \qquad \pi(\Delta, X : \pi'\kappa) = (\pi\Delta), X : (\pi\pi')\kappa$$
$$\pi^{-1}\cdot = \cdot \qquad \pi^{-1}(\Delta, X : \pi'\kappa) = (\pi^{-1}\Delta), X : (\pi^{-1}\pi')\kappa$$

4.1.2 Sizes

Our system features a *bounded meet-semilattice* of sizes, which are located on the kind level. Intuitively, the sizes can be thought of as ordinals, but there is no requirement on the ordinal structure of them besides having an order, infinity, and a minimum. In contrast to System F_ω^{cop}, we are extending the grammar of size expressions with the meet operation ($\wedge$), which will be useful in the inference algorithm later. One important observation about $\wedge$ is that we allow only minima of incremented size variables. Having ∞ as a permitted element of a $\wedge$-sequence would complicate the rules.

Now we shall explain the basic judgements about sizes.

First, we need to extend the domain of size increment $a + m$:

$$(\infty + m) = \infty + m \qquad (i + n) + m = i + (n + m)$$

$$(a^\wedge \wedge (i + n)) + m = (a^\wedge + m) \wedge (i + (n + m))$$

The judgment $\boxed{\Psi \vdash i < a}$ concerns the well-formedness of a bounded size. It signifies that the size is accurately represented in the size context, as depicted in the following rule. The polarity restriction indicates that we can only refer to sizes that are currently covariant.

$$\frac{(i : \pi(< a)) \in \Psi}{\Psi \vdash i < a}\ \pi \leq +$$

The judgment $\boxed{\Psi \vdash a}$ concerns the well-formedness of a size expression in a size context.

$$\frac{}{\Psi \vdash \infty + n} \qquad \frac{\Psi \vdash i < a}{\Psi \vdash i + n} \qquad \frac{\Psi \vdash i < a \qquad \Psi \vdash j < b}{\Psi \vdash (i + n) \wedge (j + m)} \qquad \frac{\Psi \vdash a^\wedge \qquad \Psi \vdash i < a}{\Psi \vdash a^\wedge \wedge (i + n)}$$

The judgment $\boxed{\vdash \Psi}$ concerns the well-formedness of the entire size context. The polarity inversion indicates that contravariant sizes can be correctly included in the size context, but they cannot be used until the polarity becomes $+$ or $\circ$.

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \Psi \qquad \circ^{-1}\Psi \vdash a}{\Psi \vdash i : \pi(< a)}$$

The judgment $\boxed{\Psi \vdash \vec{a} \leftleftarrows \Psi'}$ pertains to the well-formedness of a size substitution. The intuition behind this rule is that it involves constructing a sequence of sizes from the context Ψ that satisfies a "blueprint" specified by the context Ψ'.

$$\frac{}{\Psi \vdash \cdot \leftleftarrows \cdot} \qquad \frac{\Psi \vdash \vec{a} \leftleftarrows \Psi' \qquad \Psi \vdash a < b[\vec{a}/\hat{\Psi}']}{\Psi \vdash \vec{a}\ a \leftleftarrows \Psi', i : \pi(< b)}$$

The judgment $\boxed{\Psi \vdash a < b}$ pertains to strict size comparison. These rules establish a strict order relation on size expressions. Additionally, our system includes rules for defining the minima of size variables:

$$\frac{n < m}{\Psi \vdash \infty + n < \infty + m} \qquad \frac{n < m \qquad \Psi \vdash i < a}{\Psi \vdash i + n < i + m} \qquad \frac{\Psi \vdash i < \infty}{\Psi \vdash i + n < \infty + m}$$

$$\frac{\Psi \vdash i < \infty + m}{\Psi \vdash i + n < \infty + (m + n)} \qquad \frac{\Psi \vdash a + n \leq b}{\Psi, i : \pi(< a), \Psi' \vdash i + n < b}\ \pi \leq +$$

$$\frac{\Psi \vdash (i + n) < a \text{ for some } (i + n) \in a^\wedge}{\Psi \vdash a^\wedge < a} \qquad \frac{\Psi \vdash a < (i + n) \text{ for all } (i + n) \in a^\wedge}{\Psi \vdash a < a^\wedge}$$

$$\frac{\Psi \vdash a^\wedge \leq b^\wedge \qquad \Psi \vdash a^\wedge < (j+m)}{\Psi \vdash a^\wedge < b^\wedge \wedge (j+m)} \qquad \frac{\Psi \vdash a^\wedge < b^\wedge \qquad \Psi \vdash a^\wedge \leq (j+m)}{\Psi \vdash a^\wedge < b^\wedge \wedge (j+m)}$$

$\boxed{\Psi \vdash a \leq b}$ is a judgement about nonstrict size comparison:

$$\frac{\Psi \vdash a < b + 1}{\Psi \vdash a \leq b}$$

The judgment $\boxed{\Psi \vdash \exists \Psi'}$ indicates that Ψ' consistently extends Ψ. This judgment's significance is discussed in section 3.6 of [Abel and Pientka, 2016]. Essentially, it asserts that for all size valuations η of Ψ, there exists a valuation for $\eta(\Psi')$. This condition may not always hold; for instance, if $\Psi \equiv i \leq \infty$, a valid valuation could be $\eta(i) := 0$, but $\Psi' \equiv i \leq \infty, j \leq i$ would lack a consistent valuation.

We also define $\boxed{a^\uparrow}$, referred to as *bound normalization*, by the following rules. Bound normalization implies that when working with an infinite size, arbitrary increments of it lose significance.

$$(\infty + n)^\uparrow = \infty + 1 \qquad (i+n)^\uparrow = i + n \qquad (a^\wedge \wedge (i+n))^\uparrow = a^\wedge \wedge (i+n)$$

4.1.3 Kinding

In this section, we present the rules relevant to the subkinding system and the well-formedness of kinds.

The judgment $\boxed{\Psi \vdash \kappa}$ indicates the well-formedness of a kind. It's crucial to note that sizes can only be used in positive positions or in mixed positions.

$$\frac{}{\Psi \vdash *} \qquad \frac{\Psi \vdash a}{\Psi \vdash < a} \qquad \frac{-\Psi \vdash \kappa \qquad \Psi \vdash \kappa'}{\Psi \vdash \pi\kappa \to \kappa'}$$

The judgment $\boxed{\Psi \vdash \kappa \leq \kappa'}$ pertains to subkinding. This is essential due to the presence of an order on sizes, and ultimately, the aim is to utilize the rule of subsumption. The reversal of polarity comparison in the third rule reflects that the arrow kind operator is contravariant in its arguments.

$$\frac{}{\Psi \vdash * \leq *} \qquad \frac{\Psi \vdash a \leq b}{\Psi \vdash (< a) \leq (< b)} \qquad \frac{\pi' \leq \pi \qquad -\Psi \vdash \kappa_1' \leq \kappa_1 \qquad \Psi \vdash \kappa_2 \leq \kappa_2'}{\Psi \vdash \pi\kappa_1 \to \kappa_2 \leq \pi'\kappa_1' \to \kappa_2'}$$

Here we present the rule of parameterized comparison for sizes and kinds: $\boxed{\Psi \vdash O \leq^\pi O' \text{ for } O ::= a \mid \kappa}$. This rule is generalized over different grammatical entities, thus representing a slight abuse of notation.

$$\frac{\Psi \vdash O \leq O' \qquad \Psi \vdash O' \leq O}{\Psi \vdash O \leq^\circ O'} \qquad \frac{\Psi \vdash O \leq O'}{\Psi \vdash O \leq^+ O'} \qquad \frac{\Psi \vdash O' \leq O}{\Psi \vdash O' \leq^- O} \qquad \frac{}{\Psi \vdash O \leq^\top O'}$$

We also define the rule $\boxed{\Delta \vdash \exists \Delta'}$, which extends the rule $\Psi \vdash \exists \Psi'$ to general kinding contexts.

4.2 Types

In this section, we present the available type constructors in our language, along with the rules of kind checking for them.

The syntax of type-related entities in our system is presented in Table 2. Notably, we do not have syntactic categories for measured types and constrained types, which distinguishes our system from System F_ω^{cop}.

TyVar	$\ni X, Y, Z, i, j$		Type and size variables
TyAtom	$\ni K$	$::= a \mid X \mid 1 \mid \times \mid \to \mid \forall_\kappa \mid \exists_\kappa$	Type operators
Type	$\ni F, F', A, B, C$	$::= K \mid \lambda X : \iota.\ F \mid F\ F' \mid \mu^a S \mid \nu^a S$	Type-level expressions
Var	$\ni x, y, z$		Term variable
Ctx	$\ni \Gamma$	$::= \cdot \mid \Gamma, x : A$	Term variable context
Cons	$\ni c$		Constructor of datatype
Proj	$\ni d$		Field of record
Datatype	$\ni S$	$::= \langle c_1 : F_1; \ldots; c_n : F_n \rangle$	Datatype definition
Record	$\ni R$	$::= \{d_1 : F_1; \ldots; d_n : F_n\}$	Record definition

Table 2: Grammar description for type constructors

4.2.1 Kind checking

We'll start by introducing the rules with the judgment $\boxed{\Delta \vdash A \rightrightarrows \kappa}$, which describes the rules for inferring a kind for a type. This inference relation is defined simultaneously with the checking relation, as is typical with bidirectionally checked systems.

$$\frac{}{\Delta \vdash 1 \rightrightarrows *} \qquad \frac{}{\Delta \vdash \times \rightrightarrows +* \to +* \to *} \qquad \frac{}{\Delta \vdash \to \rightrightarrows -* \to +* \to *}$$

$$\frac{\Delta \vdash a}{\Delta \vdash a \rightrightarrows <(a+1)} \qquad \frac{(X : \pi\kappa) \in \Delta}{\Delta \vdash X \rightrightarrows \kappa}\ \pi \leq + \qquad \frac{\Delta \vdash F \rightrightarrows \pi\kappa \to \kappa' \qquad \pi^{-1}\Delta \vdash G \leftleftarrows \kappa}{\Delta \vdash F\ G \rightrightarrows \kappa'}$$

$$\frac{-\Delta \vdash \kappa}{\Delta \vdash \forall_\kappa \rightrightarrows +(\circ\kappa \to *) \to *} \qquad \frac{\Delta \vdash \kappa}{\Delta \vdash \exists_\kappa \rightrightarrows +(\circ\kappa \to *) \to *}$$

$$\frac{\Delta \vdash a \qquad \Delta \vdash S \leftleftarrows \circ* \to *}{\Delta \vdash \mu^a S \rightrightarrows *} \qquad \frac{-\Delta \vdash a \qquad \Delta \vdash R \leftleftarrows \circ* \to *}{\Delta \vdash \nu^a R \rightrightarrows *}$$

$\boxed{\Delta \vdash F \leftleftarrows \kappa}$ is a judgment about checking the kind for a type. It's important to highlight that we don't allow explicit usage of sizes as parameters of the type operators; rather, they should be standalone.

$$\frac{\Delta \vdash F \rightrightarrows \kappa \qquad \Delta \vdash \kappa \leq \kappa'}{\Delta \vdash F \leftleftarrows \kappa'} \qquad \frac{\circ^{-1}\Delta \vdash \iota \qquad \Delta, X : \pi\kappa \vdash F \leftleftarrows \kappa'}{\Delta \vdash \lambda X : \iota.\ F \leftleftarrows \pi\kappa \to \kappa'}$$

$$\frac{\Delta \vdash S_c \leftleftarrows \kappa \text{ for all } c \in S}{\Delta \vdash S \leftleftarrows \kappa} \qquad \frac{\Delta \vdash R_d \leftleftarrows \kappa \text{ for all } d \in R}{\Delta \vdash R \leftleftarrows \kappa}$$

$\boxed{\Delta \vdash \Delta'}$ is a relation for the well-formedness of a kinding context.

$$\frac{}{\Delta \vdash \cdot} \qquad \frac{\circ^{-1}\Delta \vdash \kappa \qquad \Delta, X : \pi\kappa \vdash \Delta'}{\Delta \vdash X : \pi\kappa, \Delta'}$$

Given a well-formed kinding context, we can also define the well-formedness of a variable context $\boxed{\Delta \vdash \Gamma}$:

$$\frac{}{\Delta \vdash \cdot} \qquad \frac{\Delta \vdash \Gamma \qquad \Delta \vdash A}{\Delta \vdash \Gamma, x : A}$$

4.2.2 Subtyping

Our system features a subtyping relation because we need to compare types with different sizes. We shall note that the fixpoint operator ν is contravariant in its size annotation, whereas μ is, on the other hand, covariant.

The structural recursive function F @ G represents a normalizing application, as we are only interested in β-redexes of the types [Watkins et al., 2003].

The rules for subtyping are also defined in a bidirectional checking and inference style, as they mirror the rules of well-formedness for types.

$\boxed{\Delta \vdash F \leq^{\pi} F' \rightrightarrows \kappa}$ where $\pi \neq \top$ is the judgment for inferring the subtyping relation.

$$\frac{\Delta \vdash K \rightrightarrows \kappa}{\Delta \vdash K \leq^{\pi} K \rightrightarrows \kappa} \qquad \frac{\Delta \vdash F \leq^{\pi} F' \rightrightarrows \pi_1\kappa_1 \to \kappa_2 \qquad \pi_1^{-1}\Delta \vdash G \leq^{\pi_1\pi} G' \leftleftarrows \kappa_1}{\Delta \vdash F\ G \leq^{\pi} F'\ G' \rightrightarrows \kappa_2}$$

$$\frac{-\Delta \vdash \kappa \leq^{-\pi} \kappa' \qquad \kappa'' = \max^{-\pi}(\kappa, \kappa')}{\Delta \vdash \forall_{\kappa} \leq^{\pi} \forall_{\kappa'} \rightrightarrows -(\circ\kappa'' \to *) \to *} \qquad \frac{\Delta \vdash \kappa \leq^{\pi} \kappa' \qquad \kappa'' = \max^{\pi}(\kappa, \kappa')}{\Delta \vdash \exists_{\kappa} \leq^{\pi} \exists_{\kappa'} \rightrightarrows +(\circ\kappa'' \to *) \to *}$$

$$\max^{+} = \max^{\circ} = \max$$
$$\max^{-} = \min$$

$$\frac{\Delta \vdash a^{\uparrow} \leq^{\pi} a'^{\uparrow} \qquad \Delta \vdash S \leq^{\pi} S' \leftleftarrows \circ* \to *}{\Delta \vdash \mu^{a} S \leq^{\pi} \mu^{a'} S' \rightrightarrows *}$$

$$\frac{-\Delta \vdash a^{\uparrow} \leq^{-\pi} a'^{\uparrow} \qquad \Delta \vdash R \leq^{\pi} R' \leftleftarrows \circ* \to *}{\Delta \vdash \nu^{a} R \leq^{\pi} \nu^{a'} R' \rightrightarrows *}$$

We shall also define a judgment $\boxed{\Delta \vdash F \leq F' \leftleftarrows \kappa}$.

$$\frac{}{\Delta \vdash F \leq^{\top} F' \leftleftarrows \kappa} \qquad \frac{\Delta \vdash A \leq^{\pi} A' \rightrightarrows *}{\Delta \vdash A \leq^{\pi} A' \leftleftarrows *}$$

$$\frac{\circ^{-1}\Delta \vdash \kappa_1 \qquad \Delta, X : \pi_1\kappa_1 \vdash (F \text{ @ } X) \leq^{\pi} (F' \text{ @ } X) \leftleftarrows \kappa_2}{\Delta \vdash F \leq^{\pi} F' \leftleftarrows \pi_1\kappa_1 \to \kappa_2}$$

$$\frac{\Delta \vdash S_c \leq^{\pi} S'_c \leftleftarrows \kappa \text{ for all } c \in S}{\Delta \vdash S \leq^{\pi} S' \leftleftarrows \kappa} \qquad \frac{\Delta \vdash R_d \leq^{\pi} R'_d \leftleftarrows \kappa \text{ for all } d \in R}{\Delta \vdash R \leq^{\pi} R' \leftleftarrows \kappa}$$

Finally, we define the judgment $\boxed{\Delta \vdash A \leq A'}$, which is an entry point for subtyping.

$$\frac{\Delta \vdash A \leq^{+} A' \rightrightarrows *}{\Delta \vdash A \leq A'}$$

4.3 (Co)patterns

The pattern machinery in System $F_\omega^{\text{cop, SCT}}$ is the same as in System F_ω^{cop}. In this section, we will briefly overview the process of pattern and copattern matching.

The grammar for patterns is presented in Table 3.

Pat	$\ni p$	$::= x \mid () \mid (p_1, p_2) \mid c\ p \mid {}^{Q}p$	Pattern
Copat	$\ni q$	$::= p \mid X \mid .d$	Copattern
PatSp	$\ni \mathbf{q}$	$::= \vec{q}$	Pattern spine

Table 3: Grammar description for patterns

Pattern spine may seem like a new concept, while in fact, it is a way to define pattern and copattern matching simultaneously.

For example, consider the following pattern spine: $\vec{q} \equiv A\ x\ y\ (\texttt{suc}\ z)\ \texttt{.force}\ i\ w$. It matches the function $\forall A.\ \texttt{List}\ A \to \mathbb{N} \to \mathbb{N} \to \texttt{Wrapper}^{\infty}$, where $\texttt{Wrapper}^{i} \equiv \{\texttt{force} : \lambda X : *.\ \mathbb{N} \to \mathbb{N}\}$. The result of spine matching yields a gathered type context $\Delta \equiv \{A : *, i : <\infty\}$ and term context $\Gamma \equiv \{x : \texttt{List}\ A, y : \mathbb{N}, z : \mathbb{N}, w : \mathbb{N}\}$.

Formally, the relation $\boxed{\Delta; \Gamma \vdash_{\Delta_0} p \Leftarrow A}$ defines the rules for typing pattern matching. Here, Δ_0 is the type context in which the pattern matching occurs. As a result, new typing context Δ and term context Γ are returned.

$$\frac{}{\cdot; x : A \vdash_{\Delta_0} x \Leftarrow A} \qquad \frac{}{\cdot; \cdot \vdash_{\Delta_0} () \Leftarrow 1} \qquad \frac{\Delta_1; \Gamma_1 \vdash_{\Delta_0} p_1 \Leftarrow A_1 \qquad \Delta_2; \Gamma_2 \vdash_{\Delta_0} p_2 \Leftarrow A_2}{\Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \vdash_{\Delta_0} (p_1, p_2) \Leftarrow A_1 \times A_2}$$

$$\frac{\Delta; \Gamma \vdash_{\Delta_0} p \Leftarrow \exists j < a^{\uparrow}.S_c(\mu^j S)}{\Delta; \Gamma \vdash_{\Delta_0} c\ p \Leftarrow \mu^a S} \qquad \frac{\Delta; \Gamma \vdash_{\Delta_0, X:\kappa} p \Leftarrow F\ @^{\kappa}\ X}{X : \kappa, \Delta; \Gamma \vdash_{\Delta_0} {}^{X}p \Leftarrow \exists_\kappa F}$$

Given the typing process for patterns, it is possible to consider copatterns as well. Here we define the relation $\boxed{\Delta; \Gamma | A \vdash_{\Delta_0} \vec{q} \rightrightarrows C}$, where Δ is the new typing context, Γ is the new term context, A is the currently eliminated type (i.e., the type of function for which the sequence of copatterns is defined), and C is the resulting type of the clause.

$$\frac{}{\cdot; \cdot | C \vdash_{\Delta_0} \cdot \rightrightarrows C} \qquad \frac{\Delta_1; \Gamma_1 \vdash_{\Delta_0} p \Leftarrow A \qquad \Delta_2; \Gamma_2 | B \vdash_{\Delta_0} \vec{q} \rightrightarrows C}{\Delta_1, \Delta_2; \Gamma_1, \Gamma_2 | A \to B \vdash_{\Delta_0} p\ \vec{q} \rightrightarrows C}$$

$$\frac{\Delta; \Gamma | \forall j < a^{\uparrow}.R_d(\nu^j R) \vdash_{\Delta_0} \vec{q} \rightrightarrows C}{\Delta; \Gamma | \nu^a R \vdash_{\Delta_0} .d\ \vec{q} \rightrightarrows C} \qquad \frac{\Delta; \Gamma | F\ @^{\kappa}\ X \vdash_{\Delta_0, X:\kappa} \vec{q} \rightrightarrows C}{X : \kappa, \Delta; \Gamma | \forall_\kappa F \vdash_{\Delta_0} X\ \vec{q} \rightrightarrows C}$$

4.4 Invocation Graphs

Before introducing the rules for type checking of terms, we need to explain the notion of *invocation graphs*, as it is the main tool for ensuring termination in

System $F_\omega^{\text{cop, SCT}}$. This concept is borrowed from [Jones et al., 2001] and generalized to meet our requirements.

Invocation graphs serve as a description of recursive calls within the defined functions. Later, we shall define our termination criteria based on a set of invocation graphs.

An *invocation graph* is a bipartite oriented simple graph, where the edges are unidirectional and labelled with either $<$ or $\leq$. In the following text, we use the following notation:

- G as an invocation graph;
- $V(G)$ as the set of vertices of the graph;
- $L(G)$ and $R(G)$ as the parts of the graph, which can be thought of as being ”left” and ”right”.
- $E(G) \subset L(G) \times R(G) \times \{<, \leq\}$ as the set of edges;

There is an enumeration on the nodes within parts, which implies that two graphs are not considered equal if there is a permutation of nodes that establishes an isomorphism. An example of an invocation graph can be found in Figure 1.

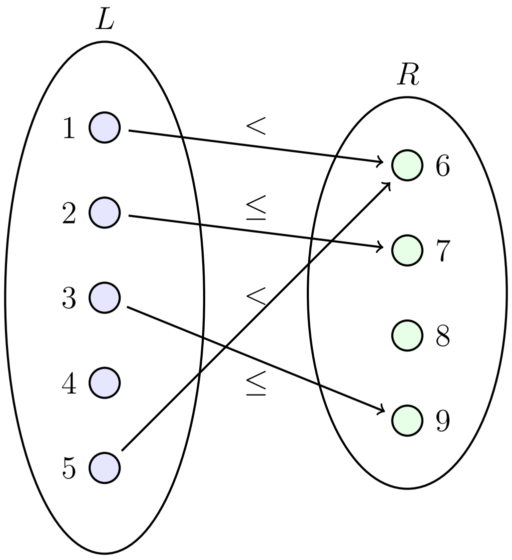

Figure 1: Example of an invocation graph

The invocation graphs can be thought of as a generalization of a relation between two elements from an ordered set to tuples. The following concept captures this intuition. Let $(S, \leq_s)$ be a partially ordered set. We define that two tuples $\vec{s}, \vec{s'} \in S^+$ *conform* to an invocation graph G if $|\vec{s}| = |L(G)|$, $|\vec{s'}| = |R(G)|$, $(a_i, b_j, <) \in E(G)$ implies $s_i < s'_j$, and $(a_i, b_j, \leq) \in E(G)$ implies $s_i \leq s'_j$. We shall write $\vec{s} \prec_G \vec{s'}$ if $\vec{s}$ and $\vec{s'}$ conform to a graph G. This definition is a reformulation of the *safe description* from [Jones et al., 2001].

We define the *composition of invocation graphs* $\boxed{G_1 \circ G_2}$ as an invocation graph

G_3, where $L(G_3) := L(G_1)$, $R(G_3) := R(G_2)$, and $(a, b, r) \in E(G_1)$, $(b, c, r') \in G_2$ implies $(a, c, r'') \in E(G_3)$, where:

- $r'' = <$ if $r = <$ or $r' = <$;
- $r'' = \leq$ otherwise.

A visual example of composition for invocation graphs can be found on Figure 2.

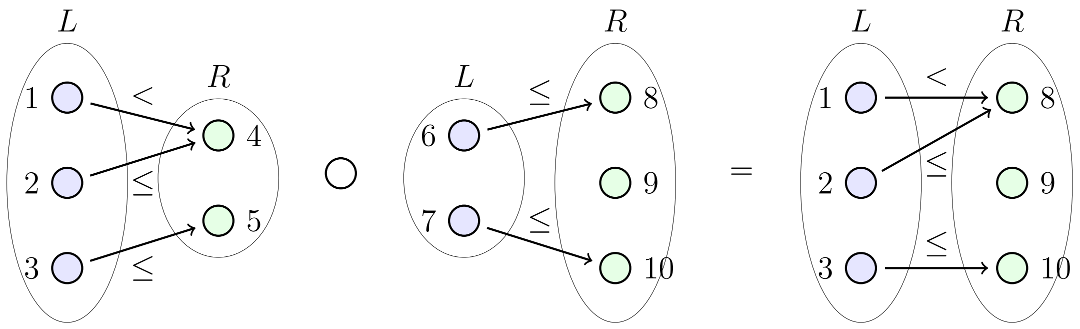

Figure 2: Example of composition of invocation graphs

Now we shall return to our type system. We define $\mathbb{G}(f, g)$ to be a set of invocation graphs indexed by textual symbols f and g, where $G_1 \in \mathbb{G}(f, g)$, $G_2 \in \mathbb{G}(g, h)$ implies $|R(G_1)| = |L(G_2)|$. The intuition behind this definition is that f and g represent the names of some functions, and an invocation graph corresponds to a call to function g within f. The left part of the invocation graph represents some ordered parameters of f, while the right part represents the arguments of g. The restriction on parts' length thus makes sense, since we want g to have the same arity whenever it acts as an enclosing function of a called function.

We are ready to define the rule $\boxed{\vdash \mathbb{G}}$, which is our main termination criterion. Let $M := \{G \mid \exists f.\ \exists g.\ G \in \mathbb{G}(f, g)\}$. Let M' be the closure of M under composition. M' is finite because the number of vertices in the invocation graphs is finite. We say that $\vdash \mathbb{G}$ if $\forall G \in M'.\ G = G \circ G \implies \exists a \in V(G).\ (a, a, <) \in E(G)$. This criterion mirrors the idea from [Jones et al., 2001].

We can informally explain the idea behind $\vdash \mathbb{G}$ as ensuring that in every sequence of recursive calls, something is always decreased. For semantic justification, the reader is welcome to refer to section 5.3.

4.5 Terms

The syntactic entities of terms that are used in the program are presented in Table 4.

Exp	$\ni r, s, t$	$::= u \mid v \mid \lambda\vec{D}$	Term
Intro	$\ni v$	$::= () \mid (t_1, t_2) \mid c\ t \mid {}^G t$	Introduction term
App	$\ni u$	$::= x \mid f \mid r\ e$	Applicative term
Fun	$\ni f, g$		Function name
Elim	$\ni e$	$::= t \mid G \mid .d$	Elimination

Table 4: Grammar description for terms

For the type-checking process, we shall follow the idea of bidirectional type checking [Dunfield and Krishnaswami, 2021]. One important distinction from System F_ω^{cop} is that in our system, the checking is parameterized additionally by the environment of already type-checked functions Σ, the functions in the current mutual-recursive block Ξ, and the implicit size context Ψ_f. To shorten the rules below, we shall abbreviate $\boxed{\Sigma; \Xi; \mathbb{G}; \Psi_f \text{ as } \Omega}$, unfolding where necessary.

$\boxed{\Sigma; \Xi; \mathbb{G}; \Psi_f; \Delta; \Gamma \vdash r \rightrightarrows C}$ is a judgement about expression typing (inference mode). Input: $\vdash \Sigma$, $\vdash \Xi$, $\vdash \mathbb{G}$, $\Delta \vdash \Psi_f$, $\vdash \Delta$, and $\Delta \vdash \Gamma$, along with r. Output: C with $\Delta \vdash C$ or failure.

The following rules closely resemble those of System $F_\omega$and are therefore straightforward. We will not delve into them in detail.

$$\frac{(x : A) \in \Gamma}{\Omega; \Delta; \Gamma \vdash x \rightrightarrows A} \qquad \frac{\Omega; \Delta; \Gamma \vdash r \rightrightarrows \nu^a R}{\Omega; \Delta; \Gamma \vdash r.d \rightrightarrows \forall j{<}a^{\uparrow}.R_d(\nu^j R)}$$

$$\frac{\Omega; \Delta; \Gamma \vdash r \rightrightarrows A \to B \qquad \Omega; \Delta; \Gamma \vdash s \leftleftarrows A}{\Omega; \Delta; \Gamma \vdash r\ s \rightrightarrows B} \qquad \frac{\Omega; \Delta; \Gamma \vdash r \rightrightarrows \forall_\kappa F \qquad \Delta \vdash F' \leftleftarrows \kappa}{\Omega; \Delta; \Gamma \vdash r\ F' \rightrightarrows F\ @^\kappa\ F'}$$

$$\frac{\Delta \vdash A \qquad \Omega; \Delta; \Gamma \vdash t \leftleftarrows A}{\Omega; \Delta; \Gamma \vdash (t : A) \rightrightarrows A}$$

The next two rules deserve a more detailed description, since this is a novelty of our system.

Let's delve deeper into the rule for inferring the type of a function from the global environment Σ. In System $F_\omega^{\text{cop, SCT}}$, Σ contains functions that have already been type-checked, implying that they are semantically safe to use. However, these functions might still embody non-trivial size dependencies within their signatures (see section 6.7). Therefore, we retain the size annotations in their signatures, unlike in [Abel and Pientka, 2016] where they were erased.

$$\frac{(g : \forall\Psi.A) \in \Sigma \qquad \Delta \vdash \vec{a} \leftleftarrows \Psi}{\Sigma; \Xi; \mathbb{G}; \Psi_f; \Delta; \Gamma \vdash g\ \vec{a} \rightrightarrows A[\vec{a}/\hat{\Psi}]}$$

Let's take a closer look at the rule for type-checking a usage of a function g taken from the same mutual block Ξ as f. The crucial aspect of this rule lies in its restriction: it allows only the usage of sizes that are permitted by some size graph $\mathbb{G}(f, g)$.

This restriction is fundamental for establishing the required semantic properties and ensuring the integrity of the program.

$$\frac{(g : \forall\Psi.A) \in \Xi \qquad \Delta \vdash \vec{a} \Leftarrow \Psi \qquad G \in \mathbb{G}(f,g) \qquad \vec{a} \prec_G \Psi_f}{\Sigma; \Xi; \mathbb{G}; \Psi_f; \Delta; \Gamma \vdash g\ \vec{a} \rightrightarrows A[\vec{a}/\hat{\Psi}]}$$

The judgement $\boxed{\Sigma; \Xi; \mathbb{G}; \Psi_f; \Delta; \Gamma \vdash r \Leftarrow C}$ describes expression typing in checking mode. In this mode, we adhere closely to the rules laid out in [Abel and Pientka, 2016], as no new rules are introduced. This mode ensures that expressions are checked against a given type C, with the additional context provided by the function environment Σ, mutual block Ξ, invocation graphs $\mathbb{G}$, implicit size context Ψ_f, type context Δ, and term context Γ.

$$\frac{}{\Omega; \Delta; \Gamma \vdash () \Leftarrow 1} \qquad \frac{\Omega; \Delta; \Gamma \vdash t_1 \Leftarrow A_1 \qquad \Omega; \Delta; \Gamma \vdash t_2 \Leftarrow A_2}{\Omega; \Delta; \Gamma \vdash (t_1, t_2) \Leftarrow A_1 \times A_2}$$

$$\frac{\Omega; \Delta; \Gamma \vdash t \Leftarrow \exists (j < a^{\uparrow}).\ S_c(\mu^j S)}{\Omega; \Delta; \Gamma \vdash c\ t \Leftarrow \mu^a S} \qquad \frac{\Delta \vdash F' \Leftarrow \kappa \qquad \Omega; \Delta; \Gamma \vdash t \Leftarrow F\ @^{\kappa}\ F'}{\Omega; \Delta; \Gamma \vdash {}^{F'}t \Leftarrow \exists_{\kappa} F}$$

$$\frac{\Omega; \Delta; \Gamma \vdash D_k \Leftarrow A \text{ for all } k}{\Omega; \Delta; \Gamma \vdash \lambda\vec{D} \Leftarrow A} \qquad \frac{\Omega; \Delta; \Gamma \vdash r \rightrightarrows A \qquad \Delta \vdash A \leq C}{\Omega; \Delta; \Gamma \vdash r \Leftarrow C}$$

4.6 Declarations

At the top level, programs defined in System $F_{\omega}^{\text{cop, SCT}}$ consist of blocks of definitions. These blocks serve as the glue that combines all previously defined entities, including patterns, terms, and the introduction of invocation graphs.

The syntactic categories for definitions are delineated in Table 12.

DefCl	$\ni D$	$::= \{\mathbf{q} \rightarrow t\}$	Definition clause
Def	$\ni \vec{D}$	$::= \{D_1; \ldots; D_n\}$	Definition clauses
Decl	$\ni \delta$	$::= f : \forall\Psi.\ A = \vec{D}$	Declaration
Block	$\ni \Xi$	$::=$ mutual $\vec{\delta}$	Mutual block
Prg	$\ni P$	$::= \vec{\Xi}; u$	Program
Sig	$\ni \Sigma$	$::= \vec{\delta}$	Signature

Table 5: Grammar description for definitions

We define $\boxed{\Omega; \Delta; \Gamma \vdash D \Leftarrow A}$ as the rule for verifying a clause. It's important to recall that the significance of the condition $\Delta \vdash \exists\Delta'$ is elucidated in section 3.6 of [Abel and Pientka, 2016].

$$\frac{\Delta'; \Gamma' | A \vdash_{\Delta} \vec{q} \rightrightarrows C \qquad \Delta \vdash \exists\Delta' \qquad \Omega; \Delta, \Delta'; \Gamma, \Gamma' \vdash t \Leftarrow C}{\Omega; \Delta; \Gamma \vdash \{\vec{q} \rightarrow t\} \Leftarrow A}$$

The previous rule can be extended to handle a sequence of clauses, representing a case of definition. In our approach, each clause is independent, so we can introduce the rule for verifying such a sequence as $\boxed{\Omega; \Delta; \Gamma \vdash \vec{D} \Leftarrow A}$.

$$\frac{\Omega; \Delta; \Gamma \vdash D_k \Leftarrow A \text{ for all } k}{\Omega; \Delta; \Gamma \vdash \vec{D} \Leftarrow A}$$

Now we are ready to define a rule for type-checking a function symbol. An important aspect here is to remember the size context of the function and use it later to justify the recursive calls within the right-hand side of clauses of f. Since the size variables of f are common to all clauses, we include them in the typing context Δ at this stage. Formally, the rule $\boxed{\Sigma; \Xi; \mathbb{G} \vdash f}$ is defined as follows:

$$\frac{\Sigma; \Xi; \mathbb{G}; \Psi; \Psi; \cdot \vdash \vec{D} \Leftarrow A}{\Sigma; \Xi; \mathbb{G} \vdash f : (\forall \Psi. A) = \vec{D}}$$

Given the definitions provided above, we can introduce the typing rule $\boxed{\Sigma \vdash \Xi}$ for a block. It is important to note that this rule necessitates the existence of a set of graphs seemingly appearing "out of nowhere." Although the user is expected to furnish these graphs, in section 6.5, we offer a method for inferring $\mathbb{G}$ from the information gathered earlier for the term.

$$\frac{\Sigma; \Xi; \mathbb{G} \vdash f : (\forall \Psi. A) = \vec{D} \text{ for all } f \in \Xi \qquad \vdash \mathbb{G}}{\Sigma \vdash \Xi}$$

Similarly, we can define $\boxed{\vdash \Sigma}$ as the typing rule for a set of signatures.

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \Sigma \qquad \Sigma \vdash \Xi}{\vdash \Xi\ \Sigma}$$

Finally, we are prepared to conclude and present a rule for checking the entire program, denoted as $\boxed{\vdash P}$. The program, informally comprising a set of functions defined earlier and a "main" function, lends itself to a straightforward rule.

$$\frac{\vdash \Sigma \qquad \Sigma; \cdot; \cdot; \cdot; \vdash u \rightrightarrows A}{\vdash \Sigma; u}$$

5

Semantics

We are interested in the strong normalization property of the provided system. System $F_\omega^{\text{cop, SCT}}$ is founded on System F_ω^{cop}, for which strong normalization was established in [Abel and Pientka, 2016].

The proof of strong normalization in [Abel and Pientka, 2016] can be decomposed into two parts.

- One part involves proving the strong normalization of the underlying flavor of System F_ω. This is essential to ensure that terms such as $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$ are not admissible in the calculus. The proof of this aspect relies on Girard-Tait reducibility candidates and can be directly applied to our system, as we do not alter the fundamental structure of the calculus itself.
- The other part pertains to justifying recursive calls and definition typing. This component is intricately tied to the syntactical structure of sized types and the conditions governing the use of other functions from Σ and Ξ. Notably, this part underwent modifications in our system. The primary theoretical contribution of this thesis lies in establishing the proof that strong normalization persists in the presence of these changes.

5.1 Reduction Relation

In this section, we will examine the reduction relation of System $F_\omega^{\text{cop, SCT}}$, which remains consistent with that of System F_ω^{cop}. This review is crucial for the proof of strong normalization, as the normalization process is defined with respect to the reduction relation.

The judgement $\boxed{t\ /\ p \searrow \tau; \sigma}$ involves matching a term t against a pattern p and acquiring a type substitution τ and a term substitution σ.

$$\frac{}{t\ /\ x \searrow \cdot; t/x} \qquad \frac{}{()\ /\ () \searrow \cdot; \cdot} \qquad \frac{t\ /\ p \searrow \tau; \sigma}{c\ t\ /\ c\ p \searrow \tau; \sigma} \qquad \frac{t\ /\ p \searrow \tau; \sigma}{{}^{F}\ t\ /\ {}^{X}\ p \searrow F/X, \tau; \sigma}$$

$$\frac{t_1\ /\ p_1 \searrow \tau_1; \sigma_1 \qquad t_2\ /\ p_2 \searrow \tau_2; \sigma_2}{(t_1, t_2)\ /\ (p_1, p_2) \searrow \tau_1, \tau_2; \sigma_1, \sigma_2}$$

Similarly, we define $\boxed{e\ /\ q \searrow \tau; \sigma}$, which is a judgement about matching a copattern:

$$\frac{}{F \;/\; X \searrow F/X; \cdot} \qquad \frac{}{.d \;/\; .d \searrow \cdot; \cdot}$$

And we also extend this to matching a pattern spine $\boxed{\vec{e} \;/\; \vec{q} \searrow \tau; \sigma}$.

$$\frac{}{\cdot \;/\; \cdot \searrow \cdot; \cdot} \qquad \frac{e \;/\; q \searrow \tau; \sigma \qquad \vec{e} \;/\; \vec{q} \searrow \tau'; \sigma'}{e\;\vec{e} \;/\; q\;\vec{q} \searrow \tau, \tau'; \sigma, \sigma'}$$

Given the definition of pattern matching, we are now able to define the rule of weak head reduction $\boxed{t \mapsto t'}$:

$$\frac{\vec{e} \;/\; \vec{q} \searrow \tau; \sigma}{\lambda\{\vec{q} \to t\}\;\vec{e}\;\vec{e'} \mapsto t\;\tau\;\sigma\;\vec{e'}} \qquad \frac{\lambda D_k\;\vec{e} \mapsto t' \text{ for some k}}{\lambda\vec{D}\;\vec{e} \mapsto t'} \qquad \frac{\lambda\vec{D}\;\vec{e} \mapsto t'}{f\;\vec{e} \mapsto t'}\; f \in \Sigma \text{ or } f \in \Xi$$

We can now define a reduction relation on terms $\boxed{t \longrightarrow t'}$:

$$\frac{t \mapsto t'}{t \longrightarrow t'} \qquad \frac{t_1 \longrightarrow t_1'}{(t_1, t_2) \longrightarrow (t_1', t_2)} \qquad \frac{t_2 \longrightarrow t_2'}{(t_1, t_2) \longrightarrow (t_1, t_2')} \qquad \frac{t \longrightarrow t'}{c\;t \longrightarrow c\;t'}$$

$$\frac{t \longrightarrow t'}{{}^{F}t \longrightarrow {}^{F}t'} \qquad \frac{r \longrightarrow r'}{r\;e \longrightarrow r'\;e} \qquad \frac{s \longrightarrow s'}{r\;s \longrightarrow r\;s'}$$

Similarly, we define the rules of reduction on clauses $\boxed{D \longrightarrow D'}$ and definitions $\boxed{\vec{D} \longrightarrow \vec{D'}}$:

$$\frac{\vec{D} \longrightarrow \vec{D'}}{\lambda\vec{D} \longrightarrow \lambda\vec{D'}} \qquad \frac{t \longrightarrow t'}{\{\vec{q} \to t\} \longrightarrow \{\vec{q'} \to t'\}} \qquad \frac{D \longrightarrow D'}{\vec{D}_1, D, \vec{D}_2 \longrightarrow \vec{D}_1, D', \vec{D}_2}$$

It is important to note that the reduction of clause can occurr only if the full pattern spine matches the provided arguments.

5.2 Strong Normalization of System $F_\omega^{\mathbf{cop}}$

In this section, we will provide a brief overview of the proof of strong normalization for System F_ω^{cop} as presented in Section 4 of [Abel and Pientka, 2016]. We won't delve into all the technical details, so readers are encouraged to refer to the original source for a more in-depth understanding. Nonetheless, we find it important to offer insight into *why* the proof is modular, allowing for the omission of certain technicalities.

The core concept of the proof involves the application of Girard-Tait reducibility candidates [Tait, 1967] [Girard, 1971], a standard technique for demonstrating strong normalization in typed lambda calculus. The fundamental idea is to interpret each type as a set of strongly normalizing terms (`SN`), referred to as *reducibility candidates*. The proof systematically follows the typing rules, demonstrating that all typing judgments ensure the type-checked term resides within a specific reducibility candidate.

Normally, it is insufficient to merely stipulate that a reducibility candidate is a set of strongly-normalizing terms. Let $\mathscr{A}$ denote such a set. The conventional requirements typically include:

1. $\mathscr{A} \subseteq \texttt{SN}$: "each term in $\mathscr{A}$ is strongly normalizing";
2. If $t \in \mathscr{A}$ then $\{t' \mid t \longrightarrow t'\} \subseteq \mathscr{A}$: "$\mathscr{A}$ is closed under reduction";

For the next condition, we introduce the set of *neutral terms* $\texttt{NE}$: a term $t \in \texttt{NE}$ if t is a redex, or $t\ \vec{e}$ is not a redex for all eliminations $\vec{e}$. This set comprises "good" terms that already manifest behavior conducive to a reducibility candidate. Specifically, either all their reducts are good, or they cannot reduce at all, and are therefore strongly normalizing.

3. If $t \in \texttt{NE}$ and $\{t' \mid t \longrightarrow t'\} \subseteq \mathscr{A}$, then $t \in \mathscr{A}$: "$\mathscr{A}$ contains a neutral if all its redexes are in $\mathscr{A}$".

In [Abel and Pientka, 2016], they additionally define the relation of *simulation*, denoted as $\boxed{r \rhd \vec{r}}$, with the following definition:

$$\forall \vec{e}.\ \forall t.\ r\ \vec{e} \mapsto t \implies \exists k.\ r_k \vec{e} \mapsto t$$

The definition of simulation enables the handling of the application of functions defined by clauses. Notably, it introduces an additional requirement for reducibility candidates. In the definition below, `Intro` is a set of terms that take the form: $() \mid (t_1, t_2) \mid c\ t \mid {}^G t$.

4. If $t \notin \texttt{Intro}$ and $\{t' \mid t \longrightarrow t'\} \subseteq \mathscr{A}$ and $t \rhd \vec{t}$ where $\vec{t} \subseteq \mathscr{A}$, then $t \in \mathscr{A}$: "$\mathscr{A}$ is closed under simulation".

The remainder of the proof meticulously analyzes all judgment rules, ensuring that well-typed terms belong to the semantical interpretation of their types, which are reducibility candidates.

Assuming that the sizes form a well-founded set, the fixpoint operators are defined by the inflationary and deflationary iteration:

$$\boldsymbol{\mu}^\alpha \mathscr{F} = \overline{\bigcup_{\beta<\alpha} \mathscr{F}(\boldsymbol{\mu}^\beta \mathscr{F})} \qquad \boldsymbol{\nu}^\alpha \mathscr{F} = \bigcap_{\beta<\alpha} \mathscr{F}(\boldsymbol{\nu}^\beta \mathscr{F})$$

Here, the sizes are interpreted as ordinals, with $\boldsymbol{\infty}$ representing the first uncountable ordinal.

Now we shall overview the way of dealing with recursive functions in System F_ω^{cop}. For the rule of expression typing, [Abel and Pientka, 2016] introduces a special premise $\vDash \Sigma$, asserting that all $f : A = \vec{D} \in \Sigma$ have the property $f \in [\![A]\!]$. This premise is justified during the proof of soundness for definition typing and is employed to manage functions from the global signature Σ.

The rule for the usage of a mutual-recursive function is more intricate. Let us state here the typing rule for such a function as defined in [Abel and Pientka, 2016].

$$\frac{(x : \forall \Psi.\mathfrak{c} \implies A) \in \Gamma \qquad \Delta \vdash \vec{a} \leftleftarrows \Psi \qquad \Delta \vdash \mathfrak{c}[\vec{a}/\vec{\hat{\Psi}}]}{\Delta; \Gamma \vdash x\ \vec{a} \rightrightarrows A[\vec{a}/\hat{\Psi}]}$$

For the soundness of this judgment to be established, ensuring the condition that x belongs to $\mathscr{A}$ is imperative. One approach to achieving this is by furnishing the semantical interpretation of the term context, denoted as $[\![\Gamma]\!]$.

According to the rule of definition typing (Section 5.2 of [Abel and Pientka, 2016]), the term context is populated by instances of x that are confirmed to be situated within their reducibility candidate through lexicographical induction on the size measures, which semantically form tuples of ordinals. An opportunity for improvement becomes apparent: if an alternative justification for x being in $\mathscr{A}$ is provided, the overarching assertion of the proof regarding strong normalization would still hold. Subsequent sections will delve into the discussion surrounding our modified justification.

5.3 Semantics of Invocation Graphs

The aim of this section is to establish the semantic property of $\mathbb{G}$. More specifically, we aim to demonstrate that when $\vdash \mathbb{G}$, it implies that $\mathbb{G}$ forms a well-founded relation.

Lemma 1 (Soundness of composition). *Let $(S, \leq_s)$ be a partially ordered set, and $\vec{s_1}, \vec{s_2}, \vec{s_3} \in S^+$. Let G_1 and G_2 be two invocation graphs. If $\vec{s_1} \prec_{G_1} \vec{s_2}$ and $\vec{s_2} \prec_{G_2} \vec{s_3}$, then $\vec{s_1} \prec_{G_1 \circ G_2} \vec{s_3}$.*

Proof. Consider an edge $(a, c, r) \in E(G_1 \circ G_2)$. We have two cases:

1. $r \equiv <$. It means that either $(a, b, <) \in E(G_1)$ or $(b, c, <) \in E(G_2)$, where the other relation is either $\leq$ or $<$. By the transitivity or the partial order $\leq_s$, we conclude that $a <_s c$.
2. $r \equiv \leq$. It means that $(a, b, \leq) \in E(G_1)$ and $(b, c, \leq) \in E(G_2)$, which implies that $a \leq_s c$ by transitivity of $\leq_s$.

□

Corollary (Collapsing of finite chains). *Consider a partially ordered set $(S, \leq_s)$. If we have a sequence of tuples $\vec{s_1}, \vec{s_2}, \ldots, \vec{s_n}$ and a sequence of invocation graphs $G_1, \ldots, G_{n-1}$ where $\vec{s_i} \prec_{G_i} \vec{s_{i+1}}$ then $\vec{s_1} \prec_{G_1 \circ \ldots \circ G_{n-1}} \vec{s_n}$.*

Proof. By induction on n. □

We will now define the semantic interpretation of call graphs for our type system. Named "invocation graph," it indicates the description of invocation of one function within another. Formally, consider a set $T := \texttt{Fun} \times O^+$, where `Fun` is a set of symbols representing function names, and O is a set of ordinals. We define the relation $\texttt{SCT}_{\mathbb{G}} \subset T \times T$, where $((f, \vec{\alpha}), (g, \vec{\beta})) \in \texttt{SCT}_{\mathbb{G}}$ if and only if $\exists G \in \mathbb{G}(f, g).\ \vec{\beta} \prec_G \vec{\alpha}$.

Theorem 1 (Soundness of $\mathbb{G}$). *Assume $\vdash \mathbb{G}$. Then $\texttt{SCT}_{\mathbb{G}}$ is a well-founded relation.*

Proof. The proof here proceeds rather classically: we shall prove that there are no infinite chains in $\mathtt{SCT}_{\mathbb{G}}$.

Consider an infinite chain $(f_1, \vec{\alpha_1}), (f_2, \vec{\alpha_2}), \ldots \in T^*$, where $((f_i, \vec{\alpha_i}), (f_{i+1}, \vec{\alpha_{i+1}})) \in \mathtt{SCT}_{\mathbb{G}}$. By definition of $\mathtt{SCT}_{\mathbb{G}}$, we obtain an infinite sequence of invocation graphs $G_1, G_2, \ldots$ such that $\vec{\alpha_{i+1}} \prec_{G_i} \vec{\alpha_i}$.

We define $P_G := \{(n, n') \mid G = G_n \circ G_{n+1} \circ G_{n+2} \circ \ldots \circ G_{n'}\}$. The set $\{P_G \mid G \in M'\}$ is finite, because the number of all possible graphs in the closure of $\mathbb{G}$ is finite. For two different G, G', the sets P_G and $P_{G'}$ are mutually disjoint, because otherwise it would imply that $G = G'$. Since any two numbers (n_1, n_2) where $n_1 < n_2$ give rise to some invocation graph $G = G_{n_1} \circ \ldots \circ G_{n_2}$, we know that each (n_1, n_2) belong to some P_G. By the Ramsey's theorem, we can conclude that there is a finite set of natural numbers N, such that all tuples $(n, n') \in N \times N$ where $n < n'$ are located in the same P_G. We shall denote such G as G°.

Let $n_1 < n_2 < n_3 \in N$. From the definition of P_{G° it follows that $G^\circ = G_{n_1} \circ \ldots \circ G_{n_2} \circ G_{n_2+1} \circ \ldots \circ G_{n_3} = G^\circ \circ G^\circ$. By the condition $\vdash \mathbb{G}$, we know that there is a vertex a such that $(a, a, <) \in E(G^\circ)$.

Now recall that we have $\vec{\alpha_{i+1}} \prec_{G_i} \vec{\alpha_i}$. If we regard the set N as $n_1, n_2, n_3, \ldots$ then by corollary we know that $\vec{\alpha_{n_2}} \prec_{G^\circ} \vec{\alpha_{n_1}}$, $\vec{\alpha_{n_3}} \prec_{G^\circ} \vec{\alpha_{n_2}}$, ... In particular, it means that there is a position i such that $\alpha_{n_1,i} > \alpha_{n_2,i} > \alpha_{n_3,i} > \ldots$, which is an infinitely decreasing chain of ordinals. Since the set of ordinals is well-founded, this is impossible, which means that our original assumption that there is an infinite chain in $\mathtt{SCT}_{\mathbb{G}}$ is false. $\square$

5.4 Strong Normalization of System $F_\omega^{\text{cop, SCT}}$

In this section, we adapt the proof of strong normalization from System F_ω^{cop} to our new system. As previously mentioned, the focal point of our modification lies in how recursive functions are handled.

5.4.1 Interpretation of Sizes

One modification in System $F_\omega^{\text{cop, SCT}}$ that we will account for is the introduction of meets into the size expressions.

Let $\rho \in [\![|\Delta_0|]\!]$ be a substitution, representing the semantical interpretation of kinding contexts. Following [Abel and Pientka, 2016], we interpret syntactic sizes as ordinals.

$$\begin{array}{rcl}
[\![i + n]\!]_\rho & = & [\![i]\!]_\rho + n \\
[\![\infty + n]\!]_\rho & = & \boldsymbol{\infty} + n \\
[\![(i + n) \wedge (j + m)]\!]_\rho & = & \min([\![i + n]\!]_\rho, [\![j + m]\!]_\rho) \\
[\![a^\wedge \wedge (i + n)]\!]_\rho & = & \min([\![a^\wedge]\!]_\rho, [\![i + n]\!]_\rho)
\end{array}$$

The definition above has direct impact on theorem 16 of [Abel and Pientka, 2016]. We shall state it here:

Theorem 2 (Soundness of kind-level judgements)**.** *Let $\vdash \Psi$ and let $\rho \leq \rho' \in \mathscr{D} := [\![\Psi]\!]$.*

1. *If $\Psi \vdash a$ then $[\![a]\!]_\rho \leq [\![a]\!]'_\rho \in O$;*
2. *If $\Psi \vdash a \leq b$ then $[\![a]\!]_\rho \leq [\![b]\!]'_\rho \in O$;*
3. *If $\Psi \vdash a < b$ then $[\![a]\!]_\rho < [\![b]\!]'_\rho \in O$.*

Proof. By the definition of $a < b$ and $a \leq b$ we see that operation $\wedge$ indeed behaves like a minimum. $\square$

Given the soundness of kind-level judgement, the rest of the proof proceeds as it is defined in [Abel and Pientka, 2016], until it reaches the changed rules of expression and definition typing.

5.4.2 Expression Typing

Similar to System F_ω^{cop}, we also need to introduce the semantical interpretations of the conditions $\vdash \Sigma$ and $\vdash \Xi$.

The intuition behind the semantics of $\vdash \Sigma$ lies in the fact that these are the functions that were type-checked *before* the current mutual block. Consequently, functions from Σ cannot reference anything from the present mutual block. This property enables the use of induction on the size of Σ, allowing us to assume, by induction hypothesis, that all these functions were type-checked earlier. Formally, we introduce the semantic judgment $\boxed{\vDash \Sigma}$ as $(f : \forall\Psi.A = \vec{D}) \in \Sigma, \Psi \vdash \vec{a} \implies f\ \vec{\alpha} \in [\![A]\!]_{[\![\vec{a}]\!]}$. The interpretation of this judgment is that all previously type-checked functions are safe with suitable ordinal vectors.

The interpretation of $\vdash \Xi$ is more complicated. The intuition behind the semantical interpretation of Ξ is that we consider a usage of a mutual-recursive function safe with certain ordinal vector ($\vec{\alpha}$) only if there is a "blueprint" (an invocation graph G) which permits this usage. Formally, we consider a judgement $\boxed{\mathbb{G}, [\![\Psi_f]\!] \vDash \Xi}$ as $\forall(g : \Psi.A = \vec{D}) \in \Xi.\ \forall\vec{\alpha} \in [\![\Psi]\!].\ \exists G \in \mathbb{G}(f, g).\ \vec{\alpha} \prec_G [\![\Psi_f]\!] \implies g\ \vec{\alpha} \in [\![A]\!]_{\vec{\alpha}/\hat{\Psi}}$.

Theorem 3 (Soundness of expression typing in System $F_\omega^{\text{cop, SCT}}$)**.** *Assume $\vDash \Sigma$, $\vdash \Delta$, $\Delta \vdash \Gamma$, $\Delta \vdash C$, $\mathscr{D} := [\![\Delta]\!]$, $\mathscr{E}(\rho) = [\![\Gamma]\!]_\rho$, $\mathscr{C}(\rho) = [\![C]\!]_\rho$, $\vDash \Sigma$, $\mathbb{G}, [\![\Psi_f]\!] \vDash \Xi$, $\vDash \mathbb{G}$*

1. *If $\Sigma; \Xi; \mathbb{G}; \Psi_f; \Delta; \Gamma \vdash r \rightrightarrows C$, then $\mathscr{D}, \mathscr{E} \vdash r \in \mathscr{C}$;*
2. *If $\Sigma; \Xi; \mathbb{G}; \Psi_f; \Delta; \Gamma \vdash r \leftleftarrows C$, then $\mathscr{D}, \mathscr{E} \vdash r \in \mathscr{C}$;*
3. *If $\Sigma; \Xi; \mathbb{G}; \Psi_f; \Delta; \Gamma \vdash \vec{D} \leftleftarrows C$, then $\mathscr{D}, \mathscr{E} \vdash \lambda\vec{D} \in \mathscr{C}$*

Proof. Our proof follows a structure similar to Theorem 35 in [Abel and Pientka, 2016]. As the proof is compositional, proceeding to establish semantical counterparts for all typing rules, we will focus on the modified sections.

Initially, we will address the typing of functions derived from the global environment Σ. Let's revisit the typing judgment:

$$\frac{(g : \forall\Psi.A) \in \Sigma \qquad \Delta \vdash \vec{a} \Leftarrow \Psi}{\Sigma; \Xi; \mathbb{G}; \Psi_f; \Delta; \Gamma \vdash g\ \vec{a} \rightrightarrows A[\vec{a}/\hat{\Psi}]}$$

With the precondition $\models \Sigma$, we can straightforwardly deduce that $g\ \vec{a} \in \mathscr{C}$. The rationale behind this is once again that all functions from Σ cannot reference the current function f.

Handling the case of a locally mutual signature follows a similar approach, given the precondition of the semantical soundness of Ξ. Suppose we are checking the function g, and we have the following derivation:

$$\frac{(g : \forall\Psi.A) \in \Xi \qquad \Delta \vdash \vec{a} \Leftarrow \Psi \qquad G \in \mathbb{G}(f, g) \qquad \vec{a} \prec_G \Psi_f}{\Sigma; \Xi; \mathbb{G}; \Psi_f; \Delta; \Gamma \vdash g\ \vec{a} \rightrightarrows A[\vec{a}/\hat{\Psi}]}$$

By the condition of semantical validity for Ξ, we can directly conclude that $g\ \vec{a} \in \mathscr{C}$. □

5.4.3 Definition Typing

In this section, we will integrate all the components and elucidate why precisely a well-typed mutually-recursive block comprises terminating functions.

We define $\boxed{\Sigma \models \Xi}$ as $\forall(f : (\forall\Psi.\ A) = \hat{\Psi}\vec{D}) \in \Xi.\ f \in [\![\forall\Psi.\ A]\!]$.

Theorem 4 (Soundness of mutual blocks). *If* $\models \Sigma$, $\Sigma \vdash \Xi$, *then* $\Sigma \models \Xi$.

Proof. Since we know that $\Sigma \vdash \Xi$, we can conclude that we also have $\vdash \mathbb{G}$ and $\Sigma; \Xi; \mathbb{G}; \Psi_f; \Psi_f; \cdot \vdash \vec{D} \Leftarrow A$ for each $f \in \Xi$.

By Theorem 1, we know that $\mathbb{G}$ generates a well-founded relation $\mathtt{SCT}_{\mathbb{G}}$. Our proof shall proceed by well-founded induction on $\mathtt{SCT}_{\mathbb{G}}$.

Recall that the carrier set of relation $\mathtt{SCT}_{\mathbb{G}}$ is $T \equiv \mathtt{Fun} \times O^+$. The induction hypothesis is the following: *given a function f and a semantic size vector $\vec{\alpha}$, we have $f\ \vec{\alpha} \in [\![A]\!]_{\vec{\alpha}/\hat{\Psi}}$.*

Now we will prove the step of induction. The induction hypothesis here states that for all functions $g : \forall\Psi'.\ B$ and semantic vectors $\vec{\beta}$ such that $\vec{\beta} \prec_G \vec{\alpha}$ for some $G \in \mathbb{G}(f, g)$, we necessarily have $g\ \vec{\beta} \in [\![B]\!]_{\vec{\beta}/\hat{\Psi}'}$. Note, that this is precisely the definition of $\mathbb{G}, [\![\Psi_f]\!] \models \Xi$, which is a requirement to apply Theorem 3. In particular, it means that we can apply the Theorem 3, which results in $f\ \vec{\alpha} \in [\![A]\!]_{\alpha/\hat{\Psi}}$.

The last part here is the observation that in the definition of $\mathtt{SCT}_{\mathbb{G}}$, the left part of the relation covers the entire space of possible functions and semantic vectors – if there is an absent pair of a function and a semantic vector, then it means that a recursive call would not be permitted with these sizes, which in turn means that the rule of local definition typing would not be applied during the type checking process. This allows to conclude that every definition in a mutual block belongs to the corresponding reducibility candidate, since all possible semantic vectors are covered.

□

Now that we have established the soundness of individual mutual blocks, we can integrate them into the proof of soundness for the global signature.

Lemma 2 (Soundness of global environment)**.** *If* $\vdash \Sigma$*, then* $\vDash \Sigma$*.*

Proof. Since Σ consists of a set of mutual blocks, the proof proceeds by induction on the number of them. The empty set of blocks is semantically valid, and if $\vdash \Sigma$ and $\Sigma \vdash \Xi$, then by the induction hypothesis we have $\vDash \Sigma$, which implies $\Sigma \vDash \Xi$, which in turn implies $\vDash \Xi\Sigma$. □

Finally, we can define what does it mean for a program to be sound. We define $\boxed{\vDash (\Sigma; u)}$ as $u \in \texttt{SN}$, which indicates soundness of the program.

Theorem 5 (Soundness of programs in System $F_\omega^{\text{cop, SCT}}$)**.** *If* $\vdash P$*, then* $\vDash P$*.*

Proof. By Lemma 2, we have $\vDash \Sigma$. Now, since u is well-typed and it uses the definitions in Σ, we conclude that $u \in \texttt{SN}$ by Theorem 3.

□

5.5 Comparison of System F_ω^{cop} and System $F_\omega^{\text{cop, SCT}}$

In this section, we will compare the expressive power of System F_ω^{cop} and System $F_\omega^{\text{cop, SCT}}$. Specifically, we will demonstrate that any terminating set of functions in System F_ω^{cop} can also be accepted by System $F_\omega^{\text{cop, SCT}}$, implying that System $F_\omega^{\text{cop, SCT}}$ is at least as powerful as System F_ω^{cop}.

Consider a mutual block $\Xi = \overline{f : \forall\Psi.\mathfrak{m} \implies A = \vec{D}}$ defined in System F_ω^{cop}. We define an operation of *measure removal* $|\cdot| : \Xi^{\text{cop}} \to \Xi^{\text{cop, sct}}$, which transforms a definition in System F_ω^{cop} to a definition in System $F_\omega^{\text{cop, SCT}}$. It simply removes the measure from a type signature, i.e. $|\overline{f : \forall\Psi.\mathfrak{m} \implies A = \vec{D}}| = \overline{f : \forall\Psi.A = \vec{D}}$. Here we indeed see that this operation transports definitions from System F_ω^{cop} to System $F_\omega^{\text{cop, SCT}}$.

Although we introduced measure removal for Ξ, we will refrain from implementing a similar operation for Σ. The rationale behind this decision is that in System F_ω^{cop}, there already exists a measure erasure operation, applied after the verification of a mutual block. The motivation for this operation is that measures strictly serve as a tool for ensuring termination, and once the block is type-checked, measures become obsolete.

Theorem 6 (Relation of System $F_\omega^{\text{cop, SCT}}$ and System F_ω^{cop})**.** *Assume a mutual block* Ξ *which is well-typed in System* F_ω^{cop} *(i.e.* $\Sigma \vdash \Xi$ *in the sense of System* F_ω^{cop}*). Then it is also the case that* $\Sigma \vdash |\Xi|$ *in the sense of System* $F_\omega^{cop,\ SCT}$*.*

Proof. Given that $\Sigma \vdash \Xi$ in System F_ω^{cop}, we can see that there is a measure $\mathfrak{m}$, such that each $f \in \Xi$ is parameterized by this measure. To prove that $\Sigma \vdash |\Xi|$ in System $F_\omega^{\text{cop, SCT}}$, we need to show that it is possible to construct a set of invocation graphs $\mathbb{G}$ such that $\vdash \mathbb{G}$ and $\Sigma; \Xi; \mathbb{G} \vdash f$ for every $f \in |\Xi|$.

Let us fix any $f \in \Xi$. According to the rules of type-checking for the mutual block in System F_ω^{cop}, all mutually-recursive functions within f are replaced with term variables x with a constrained type. This constrained type allows to use x applied to a tuple of sizes that is lexicographically smaller than the initial measure.

We express the lexicographic ordering as a set of special invocation graphs. Since all mutually-recursive functions share measures of the same size, we can generate a set of bipartite graphs with equal sizes. Let the length of measure as $|\mathfrak{m}|$. Without loss of generality, we assume that the order of variables in $\mathfrak{m}$ coincides with the order of variables in Ψ.

We generate a set of invocation graphs G_i. Assume $\vec{a} = L(G_i)$ and $\vec{b} = R(G_i)$. Then $E(G) = \{(a_j, b_j, \leq) \mid j < i\} \cup \{(a_i, b_i, <)\}$. We construct $\mathbb{G}(f, g) := \{G_1, \ldots, G_{|\mathfrak{m}|}$ for every $f, g \in \Xi$. This process is illustrated on Figure 3.

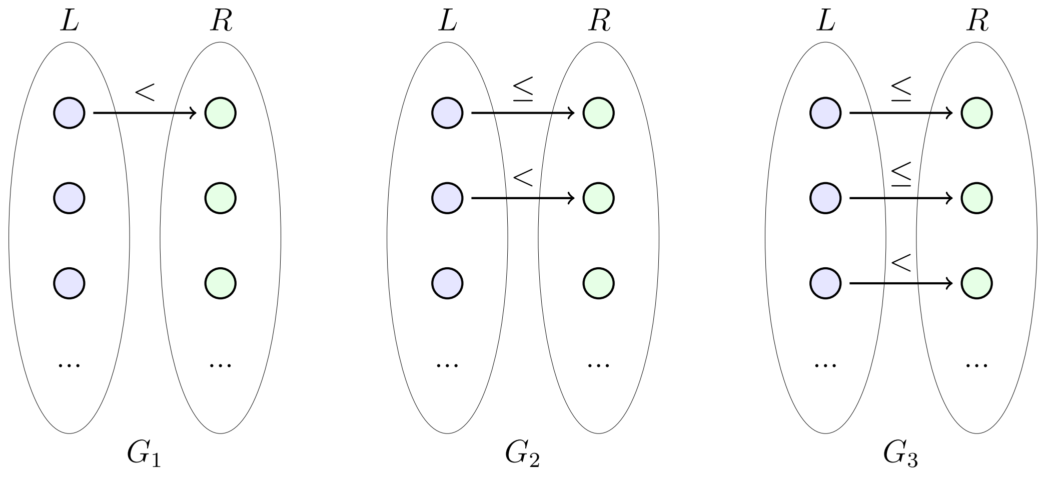

Figure 3: Example of process for generation of graphs

Size tuple $\vec{a_1}$ is lexicographically smaller than $\vec{a_2}$ if there is i such that $a_{1,1} = a_{2,1}, \ldots, a_{1,i-1} = a_{2,i-1}$ and $a_{1,i} < a_{2,i}$. Note that this is precisely expressed by the $\vec{a_2} \prec_{G_i} \vec{a_1}$. Similarly, if $\vec{a_2} \prec_{G_i} \vec{a_1}$, then $\vec{a_1}$ is lexicographically smaller than $\vec{a_2}$ by the same argument.

The only thing left to prove is $\vdash \mathbb{G}$. Our graphs have a very similar structure: there is always an edge between opposite nodes in their parts. Given this fact, we can conclude that in the composition this edge also exists.

Now consider $G \in M'$ such that $G = G \circ G$. Since G is composed of some graphs in M, we know that it also preserves some of first opposing edges. Since the number of nodes in the graph is finite, we can consider a first edge with the label $<$. If it exists, then G satisfies our criterion. If it does not exist, then it means that G is a product of invocation graphs where all edges are $\leq$, which is impossible, because by procedure there must be one edge that is $<$. $\square$

Now we know that our method is at least as expressive as System F_ω^{cop}. Even more, we know that it is *automatically* has this expressive power, as we do not require to express measures explicitly. We conjecture that our method is not strictly more expressive than System F_ω^{cop}, following the ideas in [Ben-Amram, 2002].

6
Inference

The system introduced in the preceding chapters represents a modification of the original System F. As it stands, anyone interested in using it would need to extend their existing theory. This situation is less than ideal, as we aim to provide a means of utilizing our system independently of the original theory. The elimination of the notion of syntactic measures from System F_ω^{cop} is a step in this direction.

Upon closer inspection of the typing rules, it becomes evident that the only places requiring explicit user input are the rule for the application of a size-quantified function and the signature of functions. Indeed, the remaining rules follow a straightforward path: the judgment $\Delta \vdash \exists \Delta'$ is computable, and pattern, copattern, and spine matching proceed through well-defined algorithms.

The language we are developing a termination checker for is not a classical System F, but rather a System F with patterns and copatterns, denoted as F_ω^{c}. For the sake of formality, we will present the language here. However, the rules themselves are not particularly interesting; the main motivation is to eliminate all mentions of size information from the syntax. The complete list of rules is presented in Appendix A.

Before delving into the rules, we need to introduce a meta-rule $\boxed{i \text{ fresh}}$, stating that i is a size variable with a unique name.

6.1 Function Signature

In our approach, we choose to confine ourselves to functions in which all size parameters are independent. In general, it can be challenging to determine the exact restrictions on size dependencies that the user desires. Therefore, we opt for the most general size ascription possible—where we do not assert dependencies on sizes. Additionally, we decide to forgo higher-order size quantification, meaning our types are effectively typed as $\forall_\iota K$ and $\exists_\iota K$ instead of $\forall_\kappa K$ and $\exists_\kappa K$.

Throughout this and the following sections, we will illustrate the process of inference using a function for addition of two numbers. Starting from here, we use $\mathbb{N}$ as a notation for $\mu\langle Z : \lambda X : *.\ 1; S : \lambda X.\ X\rangle$ and $\mathbb{N}^i$ for the sized version of this fixpoint. We can see that the constructors of $\mathbb{N}$ are $Z : 1$ and $S : \mathbb{N}$, following Peano arithmetic.

The function for addition, denoted as `add`, typically has the type $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$.

According to the procedure outlined in the beginning of this section, our desired annotated type is $\forall i : {<}\infty.\ \forall j : {<}\infty.\ \mathbb{N}^i \to \mathbb{N}^j \to \mathbb{N}^\infty$.

The encoding process is defined as follows. Initially, we take as input a well-formed type in terms of System F_ω. By well-formedness, we mean that it checks against the kind $*$. We then scan the type and generate a new free size variable each time we encounter an inductive definition. However, there is a subtlety: as explained in chapter 3, we aim to annotate with sizes only those fixpoints that are under the control of the user. During the encoding process, we track polarities and add new size variables only for negative μ-fixpoints and positive ν-fixpoints.

We formally define a rule $\boxed{\pi; \Delta \vdash_{\mathrm{SCT}} A \rightrightarrows \Psi; A'}$ that collects the size context of a term, suitable for processing later.

- Input: $\Delta \vdash_{F_\omega} A \leftleftarrows \iota$, target polarity π (the processing usually starts with polarity $+$), context of free type variables Δ (here we only track variables, since we are not interested in polarity annotations – their well-formedness should be covered during kind-checking of A in System F_ω). We also implicitly accept sets of polarities π^* and $\overline{\pi^*}$ – these sets indicate *when* we should add a new size variable. For top-level function encoding we shall use $\pi^* = \{-\}$ and $\overline{\pi^*} = \{+\}$.

- Output: a set of size generated fresh size variables Ψ and a size-annotated type A, which is suitable for System $F_\omega^{\text{cop, SCT}}$.

$$\frac{}{\pi; \Delta \vdash_{\mathrm{SCT}} 1 \rightrightarrows \cdot; 1} \qquad \frac{}{\pi; \Delta \vdash_{\mathrm{SCT}} \times \rightrightarrows \cdot; \times} \qquad \frac{}{\pi; \Delta \vdash_{\mathrm{SCT}} \to \rightrightarrows \cdot; \to}$$

$$\frac{X : \pi'\iota \in \Delta}{\pi; \Delta \vdash_{\mathrm{SCT}} X \rightrightarrows \cdot; X}$$

$$\frac{\Delta \vdash F \rightrightarrows \pi'\iota \to \iota' \qquad \pi; \Delta \vdash_{\mathrm{SCT}} F \rightrightarrows \Psi_1; F' \qquad \pi'\pi; \Delta \vdash_{\mathrm{SCT}} G \rightrightarrows \Psi_2; G'}{\pi; \Delta \vdash_{\mathrm{SCT}} F\ G \rightrightarrows \Psi_1, \Psi_2; F'\ G'}$$

$$\frac{}{\pi; \Delta \vdash_{\mathrm{SCT}} \forall_\iota \rightrightarrows \cdot; \forall_\iota} \qquad \frac{}{\pi; \Delta \vdash_{\mathrm{SCT}} \exists_\iota \rightrightarrows \cdot; \exists_\iota} \qquad \frac{\pi; \Delta, X : \circ\iota \vdash_{\mathrm{SCT}} F \rightrightarrows \Psi; F'}{\pi; \Delta \vdash_{\mathrm{SCT}} \lambda X : \iota.\ F \rightrightarrows \Psi; F'}$$

$$\frac{i \text{ fresh} \qquad \pi; \Delta \vdash_{\mathrm{SCT}} S \rightrightarrows \Psi; S'}{\pi; \Delta \vdash_{\mathrm{SCT}} \mu S \rightrightarrows (i < \infty), \Psi; \mu^i S'}\ \pi \in \pi^* \qquad \frac{\pi; \Delta \vdash_{\mathrm{SCT}} S \rightrightarrows \Psi; S'}{\pi; \Delta \vdash_{\mathrm{SCT}} \mu S \rightrightarrows \Psi; \mu^\infty S'}\ \pi \notin \pi^*$$

$$\frac{i \text{ fresh} \qquad \pi; \Delta \vdash_{\mathrm{SCT}} R \rightrightarrows \Psi; R'}{\pi; \nu R \vdash_{\mathrm{SCT}} (i < \infty), \Psi \rightrightarrows \nu^i R';}\ \pi \in \overline{\pi^*} \qquad \frac{\pi; \Delta \vdash_{\mathrm{SCT}} R \rightrightarrows \Psi; R'}{\pi; \Delta \vdash_{\mathrm{SCT}} \nu R \rightrightarrows \Psi; \nu^\infty R'}\ \pi \notin \overline{\pi^*}$$

$$\frac{\pi; \Delta \vdash_{\mathrm{SCT}} S_{c_i} \rightrightarrows \Psi_i; S'_{c_i} \text{ for all } c_1, \ldots, c_n \in S}{\pi; \Delta \vdash_{\mathrm{SCT}} S \rightrightarrows \Psi_1, \ldots, \Psi_n; \langle S'_{c_1}, \ldots, S'_{c_n} \rangle}$$

$$\frac{\pi; \Delta \vdash_{\mathrm{SCT}} R_{d_i} \rightrightarrows \Psi_i; R'_{d_i} \text{ for all } d_1, \ldots, d_n \in R}{\pi; \Delta \vdash_{\mathrm{SCT}} R \rightrightarrows Psi_1, \ldots, \Psi_n; \{R'_{c_1}, \ldots, R'_{d_n}\}}$$

As the reader can notice, the rules above mirror the rule $\Delta \vdash A \rightrightarrows \kappa$. This observation can be captured in the following theorem.

Theorem 7 (Well-formedness of type encoding)**.** *If* $\pi; \Delta \vdash_{SCT} A \rightrightarrows \Psi; A'$*, then* $\Delta \vdash \forall \Psi. A'$.

Proof. The proof proceeds by induction on type derivations for $\pi; \Delta \vdash_{\text{SCT}} A \rightrightarrows \Psi; A'$, where the induction hypothesis is $\Delta, \Psi \vdash A'$. The only rules that are changed from System $F_\omega^{\text{cop, SCT}}$ are the ones for encoding fixpoints. There we create only unbounded size variables, which means that the type context Ψ, Δ will always be well-formed. Also, since the generated size variables immediately appear in Ψ, we can conclude that the induction claim holds. $\square$

6.2 (Co)pattern Matching

In this section, we explain the process of pattern and copattern matching. The core idea is that when a function is defined by pattern-matching, it populates a set of *rigid* variables, which later must serve as valid size arguments in the bodies of clauses. Conceptually, pattern-matching introduces smaller terms into the scope, and copattern matching requires us to define a coinductive term of smaller depth.

Continuing with our example of `add`, given our fixed encoding of patterns and copatterns, the insertion of synthetic sizes is straightforward: any decomposition of a pattern introduces a new size variable. For instance, the insertion of rigid variables is accomplished as follows:

$$\begin{array}{rllllllll}
\texttt{add:} & \forall i : <\infty.\ \forall j : <\infty.\ \mathbb{N}^i \to \mathbb{N}^j \to \mathbb{N} \\
= & \{ & i & j & (Z\ {}^{i_1}\ ()) & y & \to & y \\
 & ; & i & j & (S\ {}^{i_2}\ x) & y & \to & S\ (\texttt{add}\ x\ y) \\
 & \}
\end{array}$$

Here, $\{i, j, i_1\}$ are the rigid variables for the first clause, and $\{i, j, i_2\}$ are rigid variables for the second clause.

We shall now present $\boxed{\Delta_0 \vdash_{\text{SCT}} p : A \rightrightarrows \Gamma; \Delta; \Psi; p'}$, which is the algorithm of gathering rigid variables.

- Input: $\Delta; \Gamma \vdash_{F_\omega \Delta_0} p \Leftarrow A$ as a term in System F_ω.
- Output: the gathered term context Γ, the gathered type context Δ, the set of rigid variables Ψ, size-annotated pattern p'.

$$\frac{}{\Delta_0 \vdash_{\text{SCT}} x : A \rightrightarrows x : A; \cdot; \cdot; x} \qquad \frac{}{\Delta_0 \vdash_{\text{SCT}} () : 1 \rightrightarrows \cdot; \cdot; \cdot; ()}$$

$$\frac{\Delta_0 \vdash_{\text{SCT}} p_1 : A_1 \rightrightarrows \Gamma_1; \Delta_1; \Psi_1; p_1' \qquad \Delta_0 \vdash_{\text{SCT}} p_2 : A_2 \rightrightarrows \Gamma_2; \Delta_2; \Psi_1; p_2'}{\Delta_0 \vdash_{\text{SCT}} (p_1, p_2) : A_1 \times A_2 \rightrightarrows \Gamma_1, \Gamma_2; \Delta_1, \Delta_2; \Psi_1, \Psi_2; (p_1', p_2')}$$

$$\frac{j \text{ fresh} \qquad \Delta_0 \vdash_{\text{SCT}} p : S_c\ (\mu^j S) \rightrightarrows \Gamma; \Delta; \Psi; p'}{\Delta_0 \vdash_{\text{SCT}} c\ p : \mu^a S \rightrightarrows \Gamma; (j : <a), \Delta; (j : <a), \Psi; c\ ({}^{j<a} p')}$$

$$\frac{\Delta_0, X : \iota \vdash_{\text{SCT}} p : F \mathbin{@^\iota} X \rightrightarrows \Gamma; \Delta; \Psi; p'}{\Delta_0 \vdash_{\text{SCT}} {}^X p : \exists_\iota F \rightrightarrows \Gamma; X, \Delta; \Psi; {}^X p'}$$

Once more, the generation of rigid variables follows a pattern similar to the rules for pattern-matching in System $F_\omega^{\text{cop, SCT}}$, which we can encapsulate in the following theorem.

Theorem 8 (Well-formedness of pattern matching). *If* $\Delta_0 \vdash_{SCT} p : A \rightrightarrows \Gamma; \Delta; p'$; *then* $\Delta; \Gamma \vdash_{\Delta_0} p' \Leftarrow A$.

Proof. Here we repeat the process of pattern-matching in System $F_\omega^{\text{cop, SCT}}$, so the proof proceeds by induction on the structure of p. The only rule that is changed is the rule of matching a constructor, but here we making a well-formed pattern p' by augmenting it with a fresh size. □

Similarly, we can define $\boxed{\Delta_0 \vdash_{\text{SCT}} \vec{q} : A \rightrightarrows C; \Gamma; \Delta; \Psi; \vec{q'}}$ – the rule of gathering rigid variables in a pattern spine.

- Input: a list of pure copatterns $\vec{q}$ of System F_ω, a size-encoded type A, a typing context Δ_0. Here we also require that $\Delta; \Gamma | A' \vdash_{F_\omega \Delta_0} \vec{q} \rightrightarrows C'$ for A' such that $+; \Delta_0 \vdash_{\text{SCT}} A' \rightrightarrows A; \_$.
- Output: a sized type of the clause body C, gathered term context Γ, gathered type context Δ, gathered set of rigid variables Ψ, and the size-annotated pattern spine $\vec{q'}$.

$$\frac{}{\Delta_0 \vdash_{\text{SCT}} \cdot : A \rightrightarrows A; \cdot; \cdot; \cdot; \cdot}$$

$$\frac{\Delta_0 \vdash_{\text{SCT}} p : A \rightrightarrows \Gamma_1; \Delta_1; \Psi_1; p' \qquad \Delta_0 \vdash_{\text{SCT}} \vec{q} : B \rightrightarrows C; \Gamma_2; \Delta_2; \Psi_2; \vec{q'}}{\Delta_0 \vdash_{\text{SCT}} p\ \vec{q} : A \to B \rightrightarrows C; \Gamma_1, \Gamma_2; \Delta_1, \Delta_2; \Psi_1, \Psi_2; p'\ \vec{q'}}$$

$$\frac{j \text{ fresh} \qquad \Delta_0 \vdash_{\text{SCT}} \vec{q} : R_d(\nu^j R) \rightrightarrows A; \Gamma; \Delta; \Psi; \vec{q'}}{\Delta_0 \vdash_{\text{SCT}} .d\ \vec{q} : \nu^a R \rightrightarrows A; \Gamma; (j : <a), \Delta; (j : <a), \Psi; .d\ j\ \vec{q'}}$$

$$\frac{\Delta_0, X : \iota \vdash_{\text{SCT}} \vec{q} : F \mathbin{@^\iota} X \rightrightarrows A; \Gamma; \Delta; \Psi; \vec{q'}}{\Delta_0 \vdash_{\text{SCT}} X\ \vec{q} : \forall_\iota F \rightrightarrows A; \Gamma; \Delta; \Psi; X\ \vec{q'}}$$

Theorem 9 (Well-formedness of copattern matching). *If* $\Delta_0 \vdash_{SCT} \vec{q} : A \rightrightarrows C; \Gamma; \Delta; \vec{q'}$; *then* $\Delta; \Gamma \mid A \vdash_{\Delta_0} \vec{q'} \rightrightarrows C$.

Proof. Again, the proof proceeds by induction on $\vec{q}$. Since the process of rigid variable generation for copatterns repeats the process of copattern matching in System $F_\omega^{\text{cop, SCT}}$, the claim follows directly from the rules. □

6.3 Call-Site Inference

This is the most crucial part of the inference process, as it is used to generate the certificate of termination. Recall that we utilize applications of sizes only in places where we use a function from the global or local signature or in the use of constructors.

For the purpose of inference, we will introduce the set of *flexible variables*, later denoted as $\Phi \subset \texttt{SizeVar} \times \pi$. Each flexible variable comes with a specific polarity $\pi \neq \top$. They are generated at the locations of constructors, function call sites, and the introductions of fixpoints.

The primary objective of call-site inference is the process of collecting *constraints* $\mathbb{C} \subset \texttt{SizeExp} \times \texttt{SizeExp} \times \{<, \leq\}$, which is a set of possible relations between variables. The domain of relationships here is the size expression, as sometimes we want to assert that a flexible variable depends on ∞ or attains a minimum.

For example, the function `add` the insertion process has the following representation:

$$\begin{array}{rlllllll}
\texttt{add:} & \multicolumn{7}{l}{\forall i : {<}\infty.\ \forall j : {<}\infty.\ \mathbb{N}^i \to \mathbb{N}^j \to \mathbb{N}} \\
= & \{ & i & j & (Z^{\ i_1}\,()) & y & \to & y \\
 & ; & i & j & (S^{\ i_2}\,x) & y & \to & S\ \boxed{k_1}\ (\texttt{add}\ \boxed{k_2}\ \boxed{k_3}\ x\ y) \\
 & \} & & & & & &
\end{array}$$

In this example, we have $\Phi \equiv \{(k_1, \circ), (k_2, \circ), (k_3, \circ)\}$. The set of gathered constraints should reflect the expected relations in the definition. Following this logic, we have $i_2 \leq k_2$ and $j \leq k_3$ (because k_1 and k_2 act as the sizes of parameters for `add`, and any argument should be smaller than them, due to the covariance of inductive types). k_1 is irrelevant here: the size of the output for `add` is ∞, so a constraint $\infty < k_1$ indicates that k_1 must be assigned to infinity. Returning back to the arguments, we can easily respect the constraints by the assignment $k_2 := i_2$, $k_3 := j$. From this assignment, we can see that `add` is used with a size smaller than i, hence it is terminating.

The algorithm also collects the size variables generated for non-recursive constructors (like Z in $\mathbb{N}$). This set is denoted $\mathbb{T}$, where the name originates from the fact that the corresponding constructor has polarity $\top$ in its size parameter.

Consider the following example:

$$\begin{array}{rlllll}
\texttt{f:} & \multicolumn{5}{l}{\forall i : {<}\infty.\ \mathbb{N}^i \to \mathbb{N}^\infty} \\
= & \{ & i & (Z^{\ i_1}\,()) & \to & (Z^{\ k_1}()) \\
 & ; & i & (S^{\ i_2}\,x) & \to & \texttt{f}\ k_2\ (Z^{\ k_3}()) \\
 & \} & & & &
\end{array}$$

This function is terminating; however, the set of constraints here is only $k_3 \leq k_2$, which does not allow us to produce a termination criterion. Note that k_3 will be a member of $\mathbb{T}$, so the algorithm will be aware of it. Our design choice is to assign k_3 to the *minimum of all available rigid variables*, so that the relationship between k_3 and i_2 will be more apparent.

Formally, we introduce the rule $\boxed{\Phi; \Gamma; \Delta \vdash_{\text{SCT}} t \rightrightarrows A; \Phi'; \mathbb{C}; \mathbb{T}; t'}$.

- Input: the set of existing flexible variables Φ, term context Γ, type context Δ, System F_ω term t.
- Output: size-annotated term A, a modified set of flexible variables Φ', a set of gathered constraints $\mathbb{C}$, a set of nonrecursive variables $\mathbb{T}$, and a size-annotated term t'.

This rule may seem a bit cumbersome. Essentially, it can be simplified to $\vdash_{\text{SCT}} t \rightrightarrows \mathbb{C}$, where the remaining elements are required for the completeness of the formal description. For example, the pair of Φ and Φ' acts as a state monad for this computation.

The presentation of rules mostly reflects the algorithm of type inference for System $F_\omega^{\text{cop, SCT}}$.

The rules of inference for a variable and an application are quite straightforward, as they do not involve sizes.

$$\frac{(x : A) \in \Gamma}{\Phi; \Gamma; \Delta \vdash_{\text{SCT}} x \rightrightarrows A; \Phi; \cdot; \cdot; x}$$

$$\frac{\Phi; \Gamma; \Delta \vdash_{\text{SCT}} r \rightrightarrows A \to B; \Phi_1; \mathbb{C}_1; \mathbb{T}_1; r' \qquad \Phi; \Gamma; \Delta | A \vdash_{\text{SCT}} s \rightrightarrows \Phi_2; \mathbb{C}_2; \mathbb{T}_2; s'}{\Phi; \Gamma; \Delta \vdash_{\text{SCT}} r\ s \rightrightarrows B; \Phi_1, \Phi_2; \mathbb{C}_1, \mathbb{C}_2; \mathbb{T}_1, \mathbb{T}_2; r'\ s'}$$

The rule for the application of a destructor is sized, meaning it requires remembering that the size of the destructed term is less than the size of the destructed record. Therefore, we add new constraints. Since the new size variable corresponds to a coinductive type, we associate polarity $-$ with it.

$$\frac{j \text{ fresh} \qquad \Phi; \Gamma; \Delta \vdash_{\text{SCT}} r \rightrightarrows \nu^a R; \Phi'; \mathbb{C}; \mathbb{T}; r' \qquad \mathbb{T}' \equiv \mathbb{T} \cup \{j, \text{ if } \nu^j R \text{ is unused in } R_d\}}{\Phi; \Gamma; \Delta \vdash_{\text{SCT}} r.d \rightrightarrows R_d(\nu^j R); (j, -), \Phi'; (j < a), \mathbb{C}; \mathbb{T}'; r'.d\ j}$$

The next couple of rules involve interaction with a type from System F_ω. To successfully use them in System F_ω^{cop}, we need to convert them to the sized version first. Since these types will be used arbitrarily, we cannot make any assumptions about the polarity of the fixpoints in the type. Therefore, we assume mixed polarity on them. We also associate a mixed polarity with the fresh size variables that occur in the encoded type. Note that during the encoding process, we generate constraints of the form $i < \infty$ (i.e., the left-hand side is always an expression).

$$\frac{\circ; \Delta \vdash_{\text{SCT}} G \rightrightarrows \Psi; G' \text{ for } \pi^* = \{\circ\}, \overline{\pi^*} = \{\circ\} \qquad \Phi; \Gamma; \Delta \vdash_{\text{SCT}} r \rightrightarrows \forall_\iota F; \Phi'; \mathbb{C}; \mathbb{T}; r'}{\Phi; \Gamma; \Delta \vdash_{\text{SCT}} r\ G \rightrightarrows F \mathbin{@^\iota} G'; \{(j, \circ) \mid (j \leq \_) \in \Psi\}, \Phi'; \mathbb{C}; \mathbb{T}; r'\ G'}$$

$$\frac{\circ; \Delta \vdash_{\text{SCT}} A \rightrightarrows \Psi; A' \text{ for } \pi^* = \{\circ\}, \overline{\pi^*} = \{\circ\} \qquad \Phi; \Gamma; \Delta | A' \vdash_{\text{SCT}} t \rightrightarrows \Phi'; \mathbb{C}; \mathbb{T}; t'}{\Phi; \Gamma; \Delta \vdash_{\text{SCT}} (t : A) \rightrightarrows A'; \{(j, \circ) \mid (j \leq \_) \in \Psi\}, \Phi'; \mathbb{C}; \mathbb{T}; (t' : A')}$$

Finally, we explain the rule for inserting flexible variables for a call to another function. According to the typing rules, for a function with type $\forall\Psi.\ A$, we need to completely eliminate the prepended size context Ψ. We achieve this by inserting an appropriate number of size variables. Since the function type is encoded according to the algorithm in section 6.1, we know that some of the sizes correspond to the least fixpoints, and some of them are for the greatest fixpoints. We denote $\widehat{\Psi_+}$ and $\widehat{\Psi_-}$ correspondingly for these parts of the size context. We can use this information to associate a more precise polarity to generated flexible variables, leading to a simpler graph of constraints.

The rule here is unified for Σ and Ξ because, from the position of type-checking, they are identical. The termination criterion with invocation graphs in System $F_\omega^{\text{cop, SCT}}$ is important for ensuring the necessary semantical properties of the program, and we shall address it later in Theorem 10.

$$\frac{\vec{a}\text{ fresh} \qquad \vec{b}\text{ fresh} \qquad (g : \forall\Psi.A) \in \Xi \cup \Sigma}{\Phi;\Gamma;\Delta \vdash_{\text{SCT}} g \rightrightarrows A[\vec{a}/\widehat{\Psi_+}, \vec{b}/\widehat{\Psi_-}]; \overrightarrow{(a,+)}, \overrightarrow{(b,-)}, \Phi; \cdot; \cdot; g\ \vec{a}\ \vec{b}}$$

Similarly, we can mirror the judgement rule for checking, $\boxed{\Phi;\Gamma;\Delta|A \vdash_{\text{SCT}} t \rightrightarrows \Phi';\mathbb{C};\mathbb{T};t'}$.

- Input: a set of flexible variable Φ, a term context Γ, a type context Δ, a size-annotated type A, and a System F_ω term t.
- Output: a set of modified flexible variables Φ', a set of generated constraints $\mathbb{C}$, a set of nonrecursive variables $\mathbb{T}$, and a size-annotated term t'.

The rules of checking introduction of a unit, tuple, and generalized lambda are straightforward.

$$\frac{}{\Phi;\Gamma;\Delta|1 \vdash_{\text{SCT}} () \rightrightarrows \Phi;\cdot;\cdot;1} \qquad \frac{\Phi;\Gamma;\Delta|A \vdash_{\text{SCT}} \vec{D} \rightrightarrows \Phi';\mathbb{C};\mathbb{T};\vec{D'}}{\Phi;\Gamma;\Delta|A \vdash_{\text{SCT}} \lambda\vec{D} \rightrightarrows \Phi';\mathbb{C};\mathbb{T};\lambda D'}$$

$$\frac{\Phi;\Gamma;\Delta|A_1 \vdash_{\text{SCT}} t_1 \rightrightarrows \Phi_1;\mathbb{C}_1;\mathbb{T}_1;t_1' \qquad \Phi;\Gamma;\Delta|A_2 \vdash_{\text{SCT}} t_2 \rightrightarrows \Phi_2;\mathbb{C}_2;\mathbb{T}_1;t_2'}{\Phi;\Gamma;\Delta|A_1 \times A_2 \vdash_{\text{SCT}} (t_1,t_2) \rightrightarrows \Phi_1,\Phi_2;\mathbb{C}_1,\mathbb{C}_2;\mathbb{T}_1,\mathbb{T}_2;(t_1',t_2')}$$

The rule for using a constructor, again, requires manipulation with sizes. We introduce a flexible variable with positive polarity and record the constraint that it is smaller than the constructed inductive fixpoint.

$$\frac{j\text{ fresh} \qquad \Phi;\Gamma;\Delta|S_c(\mu^j S) \vdash_{\text{SCT}} t \rightrightarrows \Phi';\mathbb{C};\mathbb{T};t' \qquad \mathbb{T}' \equiv \mathbb{T} \cup \{j \text{ if } \mu^j S \text{ is unused in } S_c\}}{\Phi;\Gamma;\Delta|\mu^a S \vdash_{\text{SCT}} c\ t \rightrightarrows (j,+),\Phi';(j < a),\mathbb{C};\mathbb{T}';c^{\ j}\ t'}$$

The rule for constructing an inhabitant of an existential type, similarly to the rule of inference for $\forall$, requires encoding of the corresponding System F_ω type.

$$\frac{\circ;\Delta \vdash_{\text{SCT}} G \rightrightarrows \Psi;G' \qquad \Phi;\Gamma;\Delta|F\ @^\iota\ G' \vdash_{\text{SCT}} t \rightrightarrows \Phi';\mathbb{C};\mathbb{T};t'}{\Phi;\Gamma;\Delta|\exists_\iota F \vdash_{\text{SCT}} {}^{G}t \rightrightarrows \Phi';\mathbb{C};\mathbb{T};{}^{G'}t'}$$

Following the algorithm of bidirectional type checking, we also present a rule of transition from inference to checking. This rule is the reason for carrying Φ in a state-monad style, as the process of comparison needs access to all available variables.

$$\frac{\Phi;\Gamma;\Delta\vdash_{\mathrm{SCT}} r \rightrightarrows A;\Phi';\mathbb{C}_1;\mathbb{T};r' \qquad \Phi';\Delta;*\vdash_{\mathrm{SCT}} A \leq^{+} C \rightrightarrows \mathbb{C}_2}{\Phi;\Gamma;\Delta|C\vdash_{\mathrm{SCT}} r \rightrightarrows \Phi';\mathbb{C}_1,\mathbb{C}_2;\mathbb{T};r'}$$

Before proceeding to the next section, we need to specify the rule of *trivializing* the context $\boxed{\Phi \vdash^{\downarrow} \Psi_1 \rightrightarrows \Psi_2}$. This rule removes dependencies in Ψ_1 on Φ, resulting in a simpler context Ψ_2. The reason for this operation is to satisfy the context extension check $\Delta \vdash \exists\Delta'$.

$$\frac{}{\Phi \vdash^{\downarrow} \cdot \rightrightarrows \cdot} \qquad \frac{\Psi \vdash^{\downarrow} \Psi_1 \rightrightarrows \Psi_2 \qquad \text{if } (a, \_) \in \Phi \text{ then } b \equiv \infty \text{ else } b \equiv a}{\Phi \vdash^{\downarrow} \Psi_1, (i : {<}a) \rightrightarrows \Psi_2, (i : {<}b)}$$

Concluding the bidirectional checking process, we specify the rules for checking a set of clauses $\boxed{\Phi;\Gamma;\Delta|C\vdash_{\mathrm{SCT}} D \rightrightarrows \Phi';\mathbb{C};\mathbb{T};D'}$ and $\boxed{\Phi;\Gamma;\Delta|C\vdash_{\mathrm{SCT}} \vec{D} \rightrightarrows \Phi';\mathbb{C};\mathbb{T};\vec{D'}}$:

$$\frac{\Phi;\Gamma;\Delta|C\vdash_{\mathrm{SCT}} D_k \rightrightarrows \Phi_k;\mathbb{C}_K;\mathbb{T}_k;D'_k \text{ for all } k}{\Phi;\Gamma;\Delta|C\vdash_{\mathrm{SCT}} \vec{D} \rightrightarrows \bigcup_k \Phi_k;\bigcup_k \mathbb{C}_k;\bigcup_k \mathbb{T}_k;\vec{D'}}$$

$$\frac{\Delta\vdash_{\mathrm{SCT}} \vec{q} : C \rightrightarrows C';\Gamma';\Delta';\Psi;\vec{q'} \qquad \Phi \vdash^{\downarrow} \Psi \rightrightarrows \Psi' \qquad \Phi,\Psi';\Gamma,\Gamma';\Delta,\Delta'|C'\vdash_{\mathrm{SCT}} t \rightrightarrows \Phi';\mathbb{C};\mathbb{T};t'}{\Phi;\Gamma;\Delta|C\vdash_{\mathrm{SCT}} \{\vec{q} \to t\} \rightrightarrows \Phi';\mathbb{C};\mathbb{T};\vec{q'} \to t'}$$

The rule of transition from inference to checking is an important rule, as it adds the largest number of constraints. For example, in our `add` function, the dependency between arguments and parameters of the recursive call was recorded precisely at this moment. Formally, we write $\boxed{\Phi;\Delta;\iota\vdash_{\mathrm{SCT}} A \leq^{\pi} B \rightrightarrows \mathbb{C}}$, where

- Input: a set of variables Φ, type context Δ, expected kind ι, size-annotated types A and B which are compared, and polarity of comparison π.
- Output: a set of generated constraints $\mathbb{C}$.

Note that we are comparing types obtained from System $F_\omega^{\text{cop, SCT}}$. We should note that the rules here are simpler than the rules for System $F_\omega^{\text{cop, SCT}}$, as the kinding system in System F_ω does not feature subtyping, hence we can limit ourselves only to simple kinds.

The rule of comparison for applied type operators is presented here in a generalized way, and it should feature the product of polarities to compare its arguments. For example, if F is $\to$, then it comes with polarity $-$, which essentially means that the function type constructor is contravariant in the first parameter.

$$\frac{\Phi;\Delta;\pi_1\iota_1 \to \iota_2\vdash_{\mathrm{SCT}} F \leq^{\pi} F' \rightrightarrows \mathbb{C}_1 \qquad \Phi;\Delta;\iota_1\vdash_{\mathrm{SCT}} G \leq^{\pi_1\pi} G' \rightrightarrows \mathbb{C}_2}{\Phi;\Delta;\iota_2\vdash_{\mathrm{SCT}} F\ G \leq^{\pi} F'\ G' \rightrightarrows \mathbb{C}_1,\mathbb{C}_2}$$

The next two rules require the most explanation, as they are about the comparison of fixpoints. An intended way to look at this rule is to consider it as an algorithm in the context of all previous development. Thus, in the rule for comparing μ-fixpoints, the only available polarities are either $\circ$ or $+$, which would mean that the provided constraints are either π or $\circ$. This approach reflects the covariance of flexible variables for inductive types. Similarly, the only available polarities for ν-fixpoints are $-$ and $\circ$, which results in constraints stored with $-\pi$ or $\circ$ polarity. The motivation here is the contravariance of coinductive types and our "depth" argument in chapter 3.

Note that due to our algorithms, we never have the meet of size variables as a size expression, which means that the only possible size annotations for fixpoints are size variables or infinity. To provide a rule that covers all these cases, we extend our set of gathered flexible variables with a "fake" infinity variable, which has a corresponding polarity to respect its fixpoint.

$$\frac{(a,\pi_1) \in \Phi \cup \{(\infty,+)\} \qquad (b,\pi_2) \in \Phi \cup \{(\infty,+)\} \qquad \Phi;\Delta;\circ* \to *\vdash_{\mathrm{SCT}} S \leq^{\pi} S' \rightrightarrows \mathbb{C}}{\Phi;\Delta;*\vdash_{\mathrm{SCT}} \mu^{a} S \leq^{\pi} \mu^{b} S' \rightrightarrows a \leq^{\max(\pi_1,\pi_2)\ \pi} b, \mathbb{C}}$$

$$\frac{(a,\pi_1) \in \Phi \cup \{(\infty,-)\} \qquad (b,\pi_2) \in \Phi \cup \{(\infty,-)\} \qquad \Phi;\Delta;\circ* \to *\vdash_{\mathrm{SCT}} R \leq^{\pi} R' \rightrightarrows \mathbb{C}}{\Phi;\Delta;*\vdash_{\mathrm{SCT}} \nu^{a} R \leq^{\pi} \nu^{b} R' \rightrightarrows a \leq^{\max(\pi_1,\pi_2)\pi} b, \mathbb{C}}$$

The rest of the rules are nearly identical translation of the rules for comparison of types in System $F_{\omega}^{\text{cop, SCT}}$.

$$\frac{K \in \Delta}{\Phi;\Delta;\iota\vdash_{\mathrm{SCT}} K \leq^{\pi} K \rightrightarrows \cdot} \qquad \frac{\Phi;\Delta, X : \iota_1;\iota_2\vdash_{\mathrm{SCT}} (F \ @\ X) \leq^{\pi} (F' \ @\ X) \rightrightarrows \mathbb{C}}{\Phi;\Delta;\pi\iota_1 \to \iota_2\vdash_{\mathrm{SCT}} F \leq^{\pi} F' \rightrightarrows \mathbb{C}}$$

$$\frac{\Phi;\Delta;* \to *\vdash_{\mathrm{SCT}} S_c \leq^{\pi} S'_c \rightrightarrows \mathbb{C}_c \text{ for all } c \in S}{\Phi;\Delta;* \to *\vdash_{\mathrm{SCT}} S \leq^{\pi} S' \rightrightarrows \bigcup_{c\in S} \mathbb{C}_c}$$

$$\frac{\Phi;\Delta;* \to *\vdash_{\mathrm{SCT}} R_d \leq^{\pi} R'_d \rightrightarrows \mathbb{C}_d \text{ for all } d \in R}{\Phi;\Delta;* \to *\vdash_{\mathrm{SCT}} R \leq^{\pi} R' \rightrightarrows \bigcup_{d\in R} \mathbb{C}_d}$$

The next lemma states that in the constraint graph, there is no flexible variable that is smaller than a rigid variable, which means that all rigid variables are the "roots" of the constraint graph. This is an important property of our inference algorithm, which would later imply its soundness.

Lemma 3 (Location of rigid variables)**.** *Consider a System F_ω type A and a pattern spine $\{\vec{q} \to t\}$.*

If

- *A' is a size-encoded version of A, i.e., $+;\Delta_0\vdash_{SCT} A \rightrightarrows \Psi_1; A'$;*
- *A'' is obtained after copattern-elimination of A', and Ψ_1, Ψ_2 is a set of rigid variables, i.e., $\Delta_0, \Psi_1\vdash_{SCT}\vec{q} : A' \rightrightarrows A''; \Gamma; \Delta; \Psi_2; \vec{q'}$;*
- *$\mathbb{C}$ is a set of constraints obtained after checking caluse body and Φ is a set of all collected flexible variables during this process, i.e., $\Psi_1, \Psi_2; \Gamma; \Delta|A\vdash_{SCT} t \rightrightarrows \Phi; \mathbb{C}; \mathbb{T}; t'$*

Then there is no $i \in \Phi$ and $j \in \Psi_1, \Psi_2$ such that $(i\ R\ j) \in \mathbb{C}$.

Proof. Recall, that the rigid variables can be split on two groups: inductive and coinductive ones.

- First, we will cover the case of rigid inductive variables.

  After the process of pattern spine matching, the rigid variable may appear either in the type of a bound variable in positive position (because our system forbids non-positive data types), or as a part of codomain in negative position (because the annotation process inserts these variables strictly there).

  Since the type-checking rule for variables is inference, and the whole process starts with checking, there will be eventually a rule of comparing the inferred type of the variable with some checked type. Now, since inductive rigid variable occurs positively, the polarity π in the rule of comparison will be $+$. Since the rigid itself comes with polarity $+$, the stored constraint would be $\leq^{+}$.

  For inductive rigids in the return type, the things are more complicated. We shall again analyze the rules of checking and inference. Since the occurrences here are negative, it allows us to exclude the rule of the constructor from the analysis – after all, a is a positive occurrence. It means that the only place where some constraints on this type of inductive rigids can be imposed is in the transition from inference to checking. However, since the occurrences are negative, this type of inductive rigids appears in the comparison with $\pi = -$. And since it has an associated polarity $+$, the resulting constraint is $\max(\pi_1, +)- \equiv -$, which means that the stored constraint is inverted. The rigid variables come initially on the right-hand side, since they are checked; in the inverted constraint, they will be on the left-hand side, which is what we wanted to show.

- Second, we will cover the case of coinductive rigid variables. This is somewhat simpler since due to our procedure of encoding, coinductive rigids can occur only in positive positions of the eliminated type. Following the argument for inductive rigids, we conclude that the only place where the constraints can occur is the transition from inference to checking, and the polarity of comparison for fixpoints there is $+$. But coinductive rigids have an associated polarity $-$ with them, which means that $\max(\pi_1, -)+ \equiv -$, which means that coinductive rigid will again be on the left-hand side.

$\square$

6.4 Constraint Solving

A set of flexible variables and a set of constraints describe the behavior of data flow in the program. To ensure strong normalization, we need to assign each flexible variable to some rigid, which would imply that the annotated program is an instance of System $F_{\omega}^{\text{cop, SCT}}$.

For a set of rigid variables, we define a *flexible substitution* $\varphi : \Phi \to \{a \mid \Psi \vdash a\}$. Informally, this is a mapping from flexible variables Φ to a set of size expressions that are generated by rigid size context Ψ. We write $\Psi; \mathbb{C} \vdash \varphi$ to indicate that φ is *coherent*, namely $\forall (i\ R\ j) \in \mathbb{C}.\ \Psi \vdash \varphi(i)\ R\ \varphi(j)^{\uparrow}$.

We extend the domain of φ to terms of System $F_\omega^{\text{cop, SCT}}$ functorially, denoting $\varphi(t)$ as a term obtained after the application of φ to all flexible size variables occurring in t.

The next theorem claims that a coherent substitution for the constraints gathered according to section 6.3 leads to a well-typed (hence strongly normalizing) term in System $F_\omega^{\text{cop, SCT}}$.

Theorem 10 (Soundness of coherent substitutions). *Assume a function symbol f, a System F_ω type A, a set of functions Σ, a mutual block Ξ where $f \in \Xi$, a pattern spine $\{\vec{q} \to t\}$, and a set of invocation graphs $\mathbb{G}$.*

If

1. *A' is an encoded version of A, i.e., $+; \Delta_0 \vdash_{SCT} A \rightrightarrows \Psi_1; A'$;*
2. *C is an eliminated type after copattern-matching of q', and Ψ_2 is an obtained set of rigid variables, i.e., $\Delta_0 \vdash_{SCT} \vec{q} : A' \rightrightarrows C; \Gamma; \Delta; \Psi_2; q'$;*
3. *$\mathbb{C}$ is a set of gathered constraints, and Φ is a set of flexible variables, i.e., $\Psi_1, \Psi_2; \Gamma; \Delta | C \vdash_{SCT} t \rightrightarrows \Phi; \mathbb{C}; \mathbb{T}; t'$;*
4. *There is a flexible substitution defined on Φ, i.e., $\varphi : \Phi \setminus \Psi \to \{\ a \mid \Psi_1, \Psi_2 \vdash a\}$;*
5. *The flexible substitution defined previously is coherent with respect to the constraints, i.e., $\Psi; \mathbb{C} \vdash \varphi$;*
6. *For every usage of $g\ \vec{a}$ where $g \in \Xi$ in t', there is $G \in \mathbb{G}(f, g)$ such that $\overrightarrow{\varphi(a)} \prec_G \Psi_1$.*

Then $\Sigma; \Xi; \mathbb{G}; \Psi_1; \Delta; \Gamma \vdash \varphi(t') \Leftarrow C$.

Proof. The algorithm of gathering constraints repeats the type-checking process in System $F_\omega^{\text{cop, SCT}}$. We shall note that the gathered constraints resemble the actual inequalities that are checked during subtyping and constructor/destructor application, so any substitution that respects the constraints will also lead to successful type-checking of System $F_\omega^{\text{cop, SCT}}$. $\square$

We define an operation of *closest next bound* search $\boxed{\Psi \vdash R\ a \rightrightarrows b}$ where $R \in \{<, \leq\}$ that computes the closest size expression that relates to a. For example,

$$(i : {<}\infty), (j : {<}i) \vdash <j \rightrightarrows i$$

But

$$(i : {<}\infty), (j : {<}i) \vdash <i \rightrightarrows \infty$$

Note that it extends for minima:

$$(i : {<}\infty), (j : {<}i), (k : {\leq}\infty), (l : {\leq}\infty), (m : {<}l) \vdash < (j \wedge k \wedge m) \rightrightarrows (i \wedge l)$$

The rule is formally defined by the following universal property: $\Psi \vdash R\ a \rightrightarrows b$ if $\forall c.\ \Psi \vdash a\ R\ c^{\uparrow} \implies \Psi \vdash b \leq c$.

We define an operation of *least upper bound* $\boxed{\Psi \vdash \texttt{LUB}(a_1, a_2) \rightrightarrows b}$ by the following rule: $\Psi \vdash \texttt{LUB}(a_1, a_2) \rightrightarrows b$ if $\forall c.\ \Psi \vdash a_1 \le c,\ \Psi \vdash a_2 \le c \implies \Psi \vdash b \le c$. This binary operation generalizes to n-ary one. For $n = 0$, `LUB` returns ∞.

Now we are ready to present the algorithm of computing the optimal coherent substitution.

- Input: a set of constraints $\mathbb{C}$, a set of non-recursive flexible variables $\mathbb{T}$ and a set of rigid variables Ψ.
- Output: a flexible substitution φ.

Algorithm 1 (Solving of the constraint graph)**.** *The set $\mathbb{C}$ is interpreted as a graph where the variables act as vertices, and relations act as edges.*

1. *Find strongly connected components in the graph $\mathbb{C}$. Proceed in topological order. Let the current component be $\{i_1, i_2, \ldots, i_n\}$;*
2. *Assign ∞ to those components which contain an edge marked with $<$ internally;*
3. *Assign $\bigwedge\limits_{i \in \Psi} i$ to the components where any variable belongs to $\mathbb{T}$;*
4. *Consider a set of size expressions $\{a_1, a_2, \ldots, a_m\}$ such that $(a_k\ R\ i_p) \in \mathbb{C}$ for all $k \in [1 \ldots m]$ and $p \in [1 \ldots n]$. Let $\Psi \vdash R\ \varphi(a_k) \rightrightarrows b_k$. Let $\Psi \vdash$ `LUB`$(b_1, \ldots, b_m) \rightrightarrows c$;*
5. *Assign $\varphi(i_1) = \varphi(i_2) = \ldots = \varphi(i_n) = c$.*

We shall note that Algorithm 1 is well-formed, i.e., φ is called only on those flexible variables that were assigned on the previous stage, due to the topological ordering.

From this moment we shall denote the substitution built by the algorithm as φ^*.

The next theorem shows that Algorithm 1 generates a valid substitution with respect to the constraint graph.

Theorem 11 (Soundness of φ^*)**.** *Assume the conditions of Lemma 3. If φ^* was built for Ψ, $\mathbb{C}$ and $\mathbb{T}$, then $\Psi; \mathbb{C} \vdash \varphi^*$.*

Proof. The substitution respects all lower bounds by construction. We only need to show that there is no upper bound of a flexible variable that may not be respected by the algorithm.

Since there is a topological ordering on variables, this problem may arise only for variables for which there is no upper bound in the condensed graph. The algorithm processes all flexible variables, which means, if the bound is present, then it is a rigid variable or infinity. However, the infinity as an upper bound is respected by any substitution, and rigid variables cannot have lower bounds by Lemma 3. □

Corollary (Correctness of the algorithm)**.** *Algorithm 1 produces a correctly-typed term in System $F_\omega^{cop,\ SCT}$*

Proof. Follows from Theorem 10 and Theorem 11. □

We also would like to comment on the completeness of Algorithm 1. A reasonable formulation of completeness would be the fact that all other coherent substitutions φ do not behave better than φ^*. It is clear from the coherence of φ that it should agree on components, so we can consider only the graph of connected components in our analysis. An important observation is that there may be components without any lower bound, for which φ actually may infer better results than φ^*. However, we conjecture that this situation is unusual: our goal is to construct a certificate of termination, i.e. to show that the "size" of some pattern-matched variable can be tracked up to a recursive call. By Lemma 3, the rigid variables themselves act as components without lower bounds, so it is reasonable to assume that we cannot deduce anything meaningful for components that do not have this evidence of a rigid variable. We consider an option that there may be a better algorithm in the future that solves this problem and possesses completeness.

Summing up, there is no better assignment than what we do in the constructed substitution. It means that our algorithm finds the best possible assignment to compute recursive calls, and it has a complexity of $O(n)$, where n is the number of all size variables, asymptotically equal to the size of the term.

6.5 Termination Checking of Definitions

Here we shall present the complete algorithm of checking a mutual block of functions for termination.

Algorithm 2 (Termination checker). *Assume that we have a mutual block Ξ in System F_ω.*

1. *Iterate over definitions in $(f : A = \vec{D}) \in \Xi$;*
2. *Encode the type of f, i.e. $+; \cdot \vdash_{SCT} A \rightrightarrows \Psi; A'$;*
3. *Iterate over clauses $\{\vec{q} \to t\} \in \vec{D}$;*
4. *Pattern-match the pattern spine $\vec{q}$ to obtain a set of rigid variables, i.e. $\vec{q} \vdash_{SCT} \Psi : A' \rightrightarrows A''; \Gamma; \Delta; \Psi'; \_$. The clause-local set of rigid variables is Ψ, Ψ'.*
5. *Collect constrants from the clause body, i.e. $\Psi, \Psi'; \Gamma; \Psi, \Delta | A'' \vdash_{SCT} t \rightrightarrows \Phi; \mathbb{C}; \mathbb{T}; t'$.*
6. *Compute flexible substitution $\Psi, \Psi'; \mathbb{C} \vdash \varphi^*$.*
7. *Given a size-annotated term $\varphi^*(t)$, compute a set of invocation graphs $\mathbb{G}_{f,D}$ for all usages of $g \in \Xi$ within $\varphi^*(t)$.*
8. *Compute a union of all $\mathbb{G}_f := \bigcup\limits_{D \in \vec{D}} \mathbb{G}_{f,D}$.*
9. *Compute a union of all $\mathbb{G} := \bigcup\limits_{f \in \Xi} \mathbb{G}_f$.*
10. *Accept the mutual-recursive block Ξ as a set of terminating functions if $\vdash \mathbb{G}$.*

Theorem 12 (Soundness of termination checking). *Consider a mutual-recursive block Ξ. If Algorithm 2 accepts it, then Ξ comprises strongly-normalizing functions.*

Proof. Let Ξ be a mutually-recursive block, let $\mathbb{G}$ be a set of invocation graphs computed by the algorithm of termination checking, where $\vdash \mathbb{G}$.

Consider each clause in each definition in Ξ. By Theorem 11 we know that each ϕ^* is coherent with respect to the clause-local constraints, which implies by Theorem 10 and $\vdash \mathbb{G}$ that each definition can be type-checked as a term in System $F_\omega^{\text{cop, SCT}}$. By Theorem 4 we know that the block is strongly normalizing. $\square$

The problem of deciding whether a set of functions accepted by the algorithm above is PSPACE-hard is well-established in the literature [Jones et al., 2001]. The reason for this complexity is step 10, which requires processing a possibly large number of graphs. This is the price for being free from exact ordering on pattern-matching arguments.

However, we should claim that the algorithm for termination checking provided here is quite practical. Although formally the problem is PSPACE-hard, the sets of invocation graphs in practice are often relatively small, which allows using them for mature languages. As of the moment of writing, Agda, Idris, and Arend use this method, and they do not experience major problems with the performance of their termination checker. On the other hand, the rest of the steps in the algorithm are linear in the size of the program, which implies that they are acceptable from a practical point of view.

6.6 Nontriviality

So far, the approach presented here solves a rather artificial problem. Given a program in System F_ω with copatterns, the algorithm allows to infer *some* size arguments depending on the syntax of the program, and then according to a criterion that *depends on the inserted sizes* we make claim that the program terminates or not. It is probable that our process does not allow to prove termination for *any* function, since there is no guarantee that a function would not be rejected by the $\vdash \mathbb{G}$ method. Therefore, in the absence of completeness, we need a baseline, that would indicate that the set of strongly normalizing functions accepted by Algorithm 2 is not empty.

First, we shall show that our algorithm accepts primitive-recursive functions. We need to express the higher-order operator $\rho : \forall A.\ A \to (\mathbb{N} \to A \to A) \to (\mathbb{N} \to A)$, where $\rho\ g\ h\ Z \longrightarrow g$ and $\rho\ g\ h\ (S\ x) \longrightarrow h\ x\ (\rho\ g\ h\ x)$. Given the rewrite rules, the definition is quite straightforward.

$$\begin{array}{rllllll}
\rho : & \forall A.\ A \to (\mathbb{N} \to A \to A) \to (\mathbb{N} \to A) \\
= & \{ \quad A & g & h & Z & \to & g \\
 & ; \quad A & g & h & (S\ x) & \to & h\ x\ (\rho\ g\ h\ x) \\
 & \}
\end{array}$$

Now in order to show the acceptance of this function, we shall apply the algorithm of termination checking to it. First, according to the annotation process, the type of ρ is $\forall (i : {<}\infty).\ \forall A.\ A \to (\mathbb{N}^\infty \to A \to A) \to \mathbb{N}^i \to \mathbb{N}^\infty$. Next, after the patter-matching and insertion of flexible variable, the definition has the following representation:

$$\begin{array}{rllllllll} \rho : & \multicolumn{8}{l}{\forall(i : {<}\infty)\,.\ \forall A.\ A \to (\mathbb{N}^{\infty} \to A \to A) \to (\mathbb{N}^{i} \to A)} \\ = & \{ & i & A & g & h & (Z\ i_1) & \to & g \\ & ; & i & A & g & h & (S\ i_2\ x) & \to & h\ x\ (\rho\ j_1\ g\ h\ x) \\ & \} \end{array}$$

We are interested in the second clause because it contains a recursive call. The rigid context here is $\Psi \equiv (i_2 : {<}i)$, and the set of constraints is $\mathbb{C} \equiv (i_2 \leq j_1)$ with the flexible substitution $\varphi^* := j_1 \mapsto i_2$. Since there is only one size variable in the type of ρ, there is a single invocation graph consisting of two vertices connected by the edge $<$, as shown in Figure 4.

$$\begin{tikzcd} 1 \arrow[r, "<"] & 2 \end{tikzcd}$$

Figure 4: Representation of an invocation graph

This graph is accepted by the criterion $\vdash \mathbb{G}$, hence ρ is accepted by the checker. As a result, we can express all primitive-recursive functions in our calculus.

Similarly, we will show that our termination checker can emulate guardedness condition.

$$\begin{array}{rllll} \texttt{zeros} : & \multicolumn{4}{l}{\texttt{Stream}\ \mathbb{N}} \\ = & \{ & \texttt{.head} & \to & Z \\ & ; & \texttt{.tail} & \to & \texttt{zeros} \\ & \} \end{array}$$

An annotated version of this definition has the following representation:

$$\begin{array}{rllllll} \texttt{zeros} : & \multicolumn{6}{l}{\forall(i : {<}\infty).\ \texttt{Stream}^{\,i}\ \mathbb{N}^{\infty}} \\ = & \{ & i & \texttt{.head} & i_1 & \to & Z \\ & ; & i & \texttt{.tail} & i_2 & \to & \texttt{zeros}\ j_1 \\ & \} \end{array}$$

For the second clause, the set of rigid variables is $\Psi \equiv (i_2 : {<}i)$, and the set of constraints is $\mathbb{C} \equiv (i_2 \leq j_1)$ with the flexible substitution $\varphi^* := \{j_1 \mapsto i_2\}$. Similar to ρ, the resulting invocation graph satisfies the criterion $\vdash \mathbb{G}$, indicating that this definition is accepted.

Describing the class of accepted definitions remains an open question.

6.7 Size Preservation

Another useful application of inference is *size preservation*, which is the ability to infer dependencies between size variables. Consider the following definition:

$$\begin{array}{ll} \texttt{minus:} & \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ = & \{ \quad Z \quad\; Z \quad\; \to \; Z \\ & ; \quad S\,x \;\; Z \quad\; \to \; S\,x \\ & ; \quad S\,x \;\; S\,y \;\; \to \; \texttt{minus}\;x\;y \\ & \} \end{array}$$

This function passes the termination check defined by the application of the algorithm. During the work of the algorithm, the inferred type of a function would be $\mathbb{N}^i \to \mathbb{N}^j \to \mathbb{N}^\infty$, which is reasonable. However, we can do more here: a close look at the behavior of the function `minus` results in the observation that it never returns a value bigger than its first argument. In other words, `minus` is size-preserving in its first argument.

This observation can be dualized to the case of coinduction.

$$\begin{array}{ll} \texttt{id:} & \texttt{Stream}\;\mathbb{N} \to \texttt{Stream}\;\mathbb{N} \\ = & \{ \quad s \;\; \texttt{.head} \;\; \to \;\; s\,\texttt{.head} \\ & ; \quad s \;\; \texttt{.tail} \;\; \to \;\; \texttt{id}\,(s\,\texttt{.tail}) \\ & \} \end{array}$$

Here we can notice that the function produces n elements of the output while it consumes the same n elements of the input. In other words, `id` is size-preserving in its input.

Our key observation here is that we have to keep independent sizes that are controlled by the user, and we are trying to find if the result of the computation can depend on these user-controlled sizes. Formally, positive occurrences of fixpoints may depend on the negative positions of other fixpoints in the type signature.

Our approach to testing size preservation is rather naïve: given a set of constraints, we are attempting to replace every positive size variable with every negative size variable, and then check that the constraints still behave well.

We shall sketch the algorithm using the example of the following function:

$$\begin{array}{ll} \texttt{f:} & \mathbb{N} \to \mathbb{N} \to \mathbb{N} \times \mathbb{N} \\ = & \{ \quad Z \qquad\; y \;\; \to \;\; (Z, S\;y) \\ & ; \quad (S\;x) \;\; y \;\; \to \;\; \texttt{f}\;x\;y \\ & \} \end{array}$$

Our first step is the modification of the signature of the recursive function. Usually, the process starts with encoding the type of function `f` as $\mathbb{N}^i \to \mathbb{N}^j \to \mathbb{N}^\infty \times \mathbb{N}^\infty$.

However, here we relax the restrictions on the positive occurrences: we allow *one* positive occurrence to have a size variable. For this example, we shall try to check whether it is possible to identify this the left part of the codomain product type with i. After this change, `f` would have the type $\mathbb{N}^i \to \mathbb{N}^j \to \mathbb{N}^i \times \mathbb{N}^\infty$.

The next step is the type-checking of the term with the modified signature. We shall refer to the set of gathered constraints as $\mathbb{C}^M$. For example, the annotated definition of f may have the following representation:

$$\begin{array}{lllllll}
\texttt{f:} & \forall(i : <\infty).\ \forall(j : <\infty)\ .\ \mathbb{N}^{i} \to \mathbb{N}^{j} \to \mathbb{N}^{i} \times \mathbb{N}^{\infty} \\
= & \{\ \ i\ \ j & (Z\ i_1\ ) & y & \to & (Z\ l_1\ , S\ l_2\ y) \\
 & ;\ \ i\ \ j & (S\ i_2\ x) & y & \to & \texttt{f}\ l_3\ \ l_4\ x\ y \\
 & \}
\end{array}$$

Where $\mathbb{C} = \{(i_1 \wedge j \leq l_1)\ , (l_1 \leq i), (j < l_2), (l_2 \leq \infty), (i_2 \leq l_3), (l_3 \leq i), (j \leq l_4), (l_4 \leq \infty)\}$. According to the algorithm, we can generate a suitable substitution $\varphi^* \equiv l_1 \mapsto (i_1 \wedge j), l_2 \mapsto \infty, l_3 \mapsto i_2, l_4 \mapsto j$. It's important to note that this substitution is coherent with respect to $\mathbb{C}$, signifying that the signature $\mathbb{N}^i \to \mathbb{N}^j \to \mathbb{N}^i \times \mathbb{N}^\infty$ is valid for `f`.

Now, let's examine what happens if we attempt to identify the second argument of the function $\mathbb{N}^j$ with the right component of the codomain pair, i.e., to have a signature $\mathbb{N}^i \to \mathbb{N}^j \to \mathbb{N}^\infty \times \mathbb{N}^j$. The annotated term appears the same as described above, but the set of constraints would be different. We have $\mathbb{C} = \{(i_1 \wedge j \leq l_1), (l_1 \leq \infty)\ , (j < l_2), (l_2 \leq j)\ , (i_2 \leq l_3), (l_3 \leq \infty)\ , (j \leq l_4), (l_4 \leq j)\ \}$. It's worth noting that due to the constraints $(j < l_2), (l_2 \leq j)$, we cannot find an assignment for l_2 such that the resulting substitution would be coherent. The reason is that `f` does not preserve the size of the second parameter in its second component of the output; hence, we have to leave ∞ there.

A similar issue may arise if we try to identify the second argument $\mathbb{N}^j$ with the left component of the codomain pair, i.e., to have a signature $\mathbb{N}^i \to \mathbb{N}^j \to \mathbb{N}^j \times \mathbb{N}^\infty$. The set of constraints here is $\mathbb{C} = \{(i_1 \wedge j \leq l_1), (l_1 \leq j)\ , (j < l_2), (l_2 \leq \infty)\ , (i_2 \leq l_3), (l_3 \leq j)\ , (j \leq l_4), (l_4 \leq \infty)\ \}$. Due to constraints $(i_2 \leq l_3), (l_3 \leq j)$, we cannot find a suitable expression for l_3 such that any flexible substitution would be coherent. Indeed, the left component of the pair is not related to the second argument, and it is reflected in the constraint graph as an attempt to relate two independent size variables.

We can make an observation that two variables may be identified whenever the constraint graph permits a coherent substitution. More precisely, the erroneous cases shown above violate the result of Lemma 3. This is not surprising: our main precondition is that the input variables remain independent of each other, which is the result of Lemma 3.

We will now formalize the observation made above. A set of constraints $\mathbb{C}$ is *safe* if $\exists\varphi.\ \Psi; \mathbb{C} \vdash \varphi$. This can be checked, for example, by computing φ^* and then verifying whether it is coherent with respect to $\mathbb{C}$. It's important to note that the process of computing φ^* remains well-formed in this case. The original graph permits the computation of φ^*, and the modified graph simply contains more edges, possibly increasing the number of strongly connected components, but not changing the topological ordering.

Size preservation analysis is a promising direction of improvement for a type-based termination checker. We intentionally provide here a simple approach with the possibility of future improvement, both from a quality and performance point of view.

7

Implementation

In this chapter we will reflect on the practical implementation of Algorithm 2.

7.1 Architecture

The actual implementation of the termination checker undergoes refactoring and constant improvements. Therefore, we will overview the design choices made during the development of the algorithm rather than focusing on specific code snippets.

Before delving into the details, we need to explain the architecture of the language for which we are designing our algorithm. Proof assistants are a special kind of programming language, where the syntax serves as the central means of interaction with the user. The syntax is often layered, with the following main components:

1. *Concrete syntax*, which reflects the exact input of the user, and it is intended to be concise and user-readable;
2. *Abstract syntax*, which is obtained after the parsing and desugaring the concrete syntax. Abstract syntax is organased as a set of data structures in the host language, and is more convenient for processing by other syntax-related algorithm.
3. *Internal syntax*, which is obtained after the type-checking the abstract syntax. It may be difficult to comprehend due to extreme verbosity, but it offers the biggest amount of information for the language-targeterd algorithms.

For our project, the principle choice in the implementation is to develop a separate type system for the internal syntax. This approach has numerous advantages:

1. A separate type system is decoupled from the main type system, which allows separate evolution of the underlying theories. It is especially valuable for our project, since we are targeting dependently-typed language with a System F_ω-based type system;
2. The isolation from other language features allows to limit possible side-effects on the main language, and also to make the algorithm easily replaceable;
3. A separate type system makes the proposed termination checker non-infective, which is not the case with the existing implementation of sized types. Non-

infectiveness allows to regulate the portion of definitions that are covered by the proposed termination checker without affecting the entire codebase.

While this approach offers significant power and flexibility, it comes with some disadvantages. Firstly, the complexity of implementation is increased due to the introduction of a new type system. Modelling System F_ω requires the development of custom mechanisms for substitution and pattern-matching, which can be expensive to maintain over time. Additionally, there is a certain level of duplication of concepts, as existing mechanisms may need to be replicated or adapted to fit the new type system.

The algorithm closely follows the process outlined in chapter 6, which involves bidirectional type-checking against the new type system. Encoded types are stored alongside definitions to ensure easy accessibility for further checks and analysis.

Since our language is based on System F_ω, our types include second-order parameterization. To represent this, we've opted for De Bruijn indexes, which allow us to avoid keeping track of the names of second-order variables. On the other hand, first-order size variables are global, as they later participate in the graph processing procedure.

In Agda, the terms of internal syntax are β-normal, meaning that the bidirectional type-checking process can proceed without requiring explicit type ascriptions. This is advantageous for our project, as type ascriptions are often an element of abstract syntax that is not preserved further. Consequently, we need to infer the internal types of expressions and convert them later, which can be avoided by utilizing β-normal terms.

Let's delve into the discussion about the invocation graphs. This component of the type-based checker already exists in the Agda codebase and has proven itself to be quite practical over the years.

Firstly, it's worth noting that bipartite graphs are isomorphic to a compact version of adjacency matrices. The compactness arises from the fact that there cannot be edges between nodes in the same part, resulting in a matrix of size $n_1 \cdot n_2$, whereas a complete adjacency matrix would have size $(n_1 + n_2)^2$.

The elements of the matrix in this case can be taken from a semiring $\{0, 1, 2\}$, where:

- 0 corresponds to an absence of an edge,
- 1 corresponds to $\leq$,
- 2 corresponds to $<$.

The addition operation here is a maximum, and the multiplication is defined as:

- $0 \cdot x = 0$,
- $1 \cdot x = x$,
- $2 \cdot 2 = 2$.

It can be verified that this structure satisfies the definition of a semiring.

Now, we can observe that the composition of invocation graphs corresponds to matrix multiplication. We can interpret $\vdash \mathbb{G}$ as a check for idempotent matrices where there is a 2 on the diagonal in the set of matrices closed under multiplication.

An important practical consideration is that in practice, the matrices contain a lot of 0 entries, so it makes sense to implement them in a sparse manner.

7.2 Alternative Approaches

It was initially proposed to instrument the internal syntax of Agda with sizes during type-checking, aiming to automatically reuse the checking machinery. However, upon further consideration, this approach was found to be less straightforward than anticipated.

Introducing sizes as direct elements of the syntax added complexity to the terms and increased the load on Agda's internal algorithms. Sized types interfered with other features influencing internal syntax, such as experimental irrelevance.

A concrete example highlighting the subtle issues with incorporating first-class sizes into the syntax can be illustrated with the following function:

```
func : (a b : Nat) → Nat
func a b = b
```

If the parameter list annotates Nat with size, it's required that this size is identical for both a and b. This introduces an accidental dependency between parameters, which is undesirable. To work around this, implementing complex branching logic would be necessary, potentially jeopardizing the safety of existing Agda features.

Moreover, incorporating sized types essentially introduces subtyping to Agda's theory of dependent types. Dependent type theory with subtyping is notably more complex and less performant, which could introduce sources of unsoundness in Agda.

Ultimately, we concluded that maintaining this approach would be highly challenging and opted for a separate type system instead.

7.3 Existing Termination Checkers

Algorithm 2 is not *the* termination checker, as various approaches exist for this problem. In the context of dependently-typed languages, common methods include [Abel, 2002] and [Jones et al., 2001]. Here, we'll focus on Agda's existing termination checker, which implements the Size-Change Termination Principle. This checker generates invocation graphs based on syntactic comparisons between the arguments of recursive calls and the parameters of the enclosing function. Inequalities can arise post pattern-matching, where bound variables are considered smaller than the matched parameter. This algorithm is referred to as the syntax-based termination checker.

It might be tempting to claim that our type-based checker is strictly stronger than the syntax-based one. Indeed, the intuition suggests that we also consider bound variables in patterns to have a strictly smaller size than the parameter. However, the existing syntax-based checker is slightly more powerful. Consider the following example:

```
data List (A : Set) : Set where
  nil : List A
  cons : A → List A → List A

f : {A : Set} → List (List A) → List A
f nil = nil
f (cons nil yss) = nil
f (cons (cons x xs) yss) = f (cons xs yss)
```

Note, that (cons *xs* *yss*) is not directly structurally smaller than (cons (cons *x* *xs*) *yss*). Here we experience a *congruence closure* of the syntactic subterm relation. It is relatively easy to implement this extension in the syntax-based checker while it contradicts the philosophy of the type-based termination checker, hence this function will not pass Algorithm 2. Therefore, it makes sence to retain both termination checkers available.

The key observation about our algorithm is that it aims to produce a termination certificate, which is a task very similar to what the current syntax-based checker does. As long as there exists a termination certificate for a function, it does not matter how it was obtained. In Agda, we made a choice to use both termination checkers on every definition. If either of them produces a valid termination certificate, then Agda marks the function as terminating.

Regarding the ordering of invocation of the checkers, we decided to run the type-based checker first. The reasoning here is that for the feature of size preservation, the type-based checker needs to process all defined functions. Therefore, it makes little sense to schedule it after the syntax-based checker. From a performance point of view, there is an observation that the type-based checker is not asymptotically slower than the syntax-based checker. Thus, it would not introduce significant regressions in the type-checking process.

7.4 Performance

As a result of our implementation, we measured the performance of the type-based termination checker on two major libraries: the standard library of Agda and graded modal type theory. All measurements were performed on a MacBook M2 Pro.

The results of measuring on the standard library are displayed in Table 6. In the top part of the table, we show the measurement time for type-checking the standard library without our type-based termination checker. The bottom part shows the time measurement for various parts of the algorithm: ”call-site inference” refers to bidirectional checking of terms and gathering constraints, ”constraint graph processing” refers to the process of finding the flexible substitution φ^*, ”term encoding” refers to the conversion of internal Agda terms to sized types as per section 6.1, and ”matrix solving” corresponds to the application of the size-change termination method to compute $\vdash \mathbb{G}$.

Name of metric	Time
Total (syntax-based)	256,765ms
Termination check (syntax-based)	3,530ms
Total (type-based)	275,005ms
Termination check (type-based)	5,639ms
Call-site inference	2,735ms
Constraint graph processing	1,060ms
Term encoding	1,035ms
Matrix solving	100ms

Table 6: Performance metrics for type checking of the standard library

The results of measurements on graded modal type theory are displayed in Table 7. One remark is that for graded modal type theory, we had to disable size preservation, as it is currently highly unoptimal for large terms. The type-based termination checker still has value even without size preservation.

Name of metric	Time
Total (syntax-based)	1,383,671ms
Termination check (syntax-based)	31,342ms
Total (type-based)	1,430,003ms
Termination check (type-based)	33,408ms
Call-site inference	9,050ms
Constraint graph processing	8,385ms
Term encoding	4,029ms
Matrix solving	12,595ms

Table 7: Performance metrics for type checking of the standard library

One conclusion we can make here is that the type-based checker is not much slower than the default termination checker of Agda. Given that the total time of type checking is magnitudes bigger than the termination check, it makes sense to consider our implementation of the type-based checker enabled by default. One obstacle to it is the implementation of size preservation, which is done very naively at the moment.

8

Related and Future Work

8.1 Related Work

Our work combines the development of termination checkers, higher-order logic and sized types. Therefore, we shall consider our contribution from all these points of view.

Ultimately, a termination checker is an engineering component of contemporary proof assistants. The need for them is motivated by the fact that dependently-typed languages feature the definition of functions by pattern-matching, which leads to well-readable definitions from a practical point of view, but diverges from the underlying theories, where recursion is usually accessed by the use of special functions, known as induction principles.

1. In Arend [Isaev, 2023], as well as in Agda, the termination checker follows the syntax-based algorithm described in [Jones et al., 2001]. However, our algorithm differs in that it can utilize type information, thus allowing definitions that may not be accepted by syntax-based checkers.
2. In Rocq [Bertot and Castéran, 2013], the approach to recursive definitions closely aligns with theory. Every fixpoint type comes with a generated set of functions that represent induction principles for the defined type. Recursive calls are then desugared to the use of these functions. This means that the termination checker in Rocq cannot detect permutations of arguments, which is covered by the Size-Change principle.

   However, it's worth noting that the termination checker in Rocq is able to unfold certain definitions, which increases its strength compared to Agda. Unfolding may lead to performance issues with the checker, which is why it is not allowed in Agda.
3. In Lean [The Lean Development Team, 2023], the termination checker is more heuristic in nature. It attempts to guess the lexicographical order of pattern-matching and then applies a syntax-based criterion to show that calls are made on smaller arguments.

It's worth noting that Lean and Rocq have a way to specify *termination measures*, which are functions from arguments to some well-founded domain that are supposed to be decreasing for each recursive call. This technique significantly extends the number of definitions that can be accepted by the language, but it comes at the cost of additional interaction with the user.

The development of sized types has a rich history, but few of the theoretic models have reached production level. The most mature implementation of sized types is in Agda, where sizes act as an element of syntax. They indeed allow for the proof of termination of complex functions, and they are the recommended approach for coinductive functions. However, the existing sized types are very feature-heavy, which implies numerous problems.

- The theory behind syntax-based sized types in Agda stipulates that not all size-annotated function types would behave well. Abel [Abel, 2006] requires that these types should be semi-continuous, but with the current size algebra of Agda, it leads to contradictions [Vandikas, 2022]. In contrast, our approach is based on a theory that does not impose limitations on annotated types. This is because the fixpoints in our approach are restricted from a polarity point of view, and termination is guarded by a graph-based measure, instead of relying on the notion of "making sense in a whole".
- Sized types are infective, which makes them unusable locally. This is more of an architectural problem that does not exist in our approach. This issue is tightly coupled with the decision to make size annotations implicit for the user. However, we believe that starting from an implicit approach, we would eventually arrive at a non-invasive way to provide size annotations.
- Sized types have too many features. For example, it is possible to express a maximum of two size expressions in Agda with the help of $\sqcup$. While this is useful in theory, it leads to an exponential increase in time for processing the validity of size expressions. Our choice here is to start with a very simple size algebra (and we do not even have an explicit algebra, as there are no operations), and improve it based on our needs.

Regarding the algorithm for inferring size annotations, it's important to mention the development by Barthe for the Calculus of Constructions [Barthe et al., 2006]. Our algorithm addresses a similar goal but operates in $O(n)$ time complexity instead of $O(n^3)$. However, our algorithm does not guarantee completeness.

We also acknowledge the work of Chan et al. [Chan and Bowman, 2019], which implements the same idea of implicit sized typing for Rocq. They build upon the development of Barthe and essentially implement their algorithm. One of the conclusions of their work is that sized types introduce unmanageable performance overhead, making them unfeasible for practical usage [Chan et al., 2023].

We believe that there are two main reasons for the unmanageable performance overhead of sized types, as observed in [Chan and Bowman, 2019]:

- **Complexity of Algorithms**: [Chan and Bowman, 2019] also constructs a graph of size constraints, where the edges can have negative weights. Their primary criterion for rejecting a definition is the presence of loops of negative weight, and they use the Bellman-Ford algorithm to check this condition.

However, the Bellman-Ford algorithm has a time complexity of $O(n^3)$, which, as noted in the Performance section, makes it unusable for large graphs.

- **Size Preservation**: [Chan and Bowman, 2019] employs the same approach as ours for size preservation, attempting to unify each size variable in the codomain with each variable in the domain. However, as previously noted, this approach can lead to considerable performance degradation. Additionally, it seems that in [Chan and Bowman, 2019], size preservation is not disableable, exacerbating the performance issues.

One may be interested whether the approach by [Chan et al., 2023] is more expressive than ours. We can confirm this observation: [Chan et al., 2023] use more expressive size algebra, which allows them to express arbitrary increases of sizes in the signature. Our approach does not allow it: to express increase of size variable i by n, we would need to add n additional size variables in the signature, and form a tower

$$\forall(i : {<}i_1).\ \forall(i_1 : {<}i_2).\ \ldots \forall(i_n : {<}\infty)\ldots.$$

where i_n denotes the n-increased i. We believe that there is rarely a need in complex size increasings, so we do not consider this limitation to be critical for adoption of the type-based termination checker. This observation also provides an intuition for why our constraint graphs can be processed faster than the ones in [Chan et al., 2023].

We are aiming to solve the same problem that is stated in [Chan et al., 2023], but we argue that our experiments are more successful from the practical point of view. The main reason is that we can process the constraints in our simpler size algebra with greater speed, and we can reuse a mature size-change-termination matrix engine of Agda.

8.2 Future Work

Our work opens up several avenues for further development, spanning both engineering and theoretical aspects.

From a semantic perspective, a crucial direction is adapting the underlying theory from System F_ω to handle dependent types effectively. Currently, our implementation cannot fully replace the syntax-based termination checker of Agda, particularly when dealing with dependent pattern matching. Exploring this area further could lead to a more comprehensive termination checker capable of handling a wider range of functions.

Another important avenue is defining the class of functions accepted by the termination checker. Little work has been done in this direction beyond [Ben-Amram, 2002], and further research could provide valuable insights into the characteristics of terminatable functions.

On the algorithmic front, a key question is whether it's possible to devise an algorithm for finding a flexible substitution that guarantees completeness. While the current implementation achieves optimal complexity, complete algorithms may impact performance negatively. It's also essential to investigate the limitations of the current approach and determine whether addressing corner cases is worthwhile.

Moreover, improving the efficiency of size preservation is a significant area for improvement. Finding a more efficient algorithm for detecting size-preserving functions could enhance the overall performance of the termination checker.

From an engineering and architectural standpoint, there is potential in making size annotations more interactive. While our approach is powerful, it also has inherent limitations. For instance, expressing higher-order size preservation may not be inferable by our algorithm. Incorporating user-provided size annotations would necessitate rejecting function applications lacking correct size information, adding complexity to this idea. Further exploration in this area could lead to more flexible and user-friendly size annotation systems.

9 Conclusion

9.1 Results

In this work we described strongly-normalizing polymorphic higher-order lambda calculus and provided an algorithm for termination checking the definitions written in System F_ω.

More precisely, we have the following achievements:

- We extend the work presented in [Abel and Pientka, 2016] by incorporating the size-change termination principle, while retaining the proof of strong normalization. This novel contribution enables the simultaneous utilization of sized types and the size-change termination principle, resulting in an expressive termination checker. Notably, the incorporation of the size-change termination principle allows for the reduction of a significant portion of the additional syntax that was previously required in [Abel and Pientka, 2016].
- We present an inference algorithm for a system akin to [Abel and Pientka, 2016]. The inference process scales linearly with the size of the syntax, rendering it practical for real-world applications.
- We implement the proposed termination checker for Agda and demonstrate its acceptable performance. This result proves that type-based termination checkers are feasible for mature dependently-typed languages. Our work significantly improves the treatment of coinductive definitions.

9.2 Discussion

The work described in this thesis can be thought of as a new approach to sized types. Sized types are a powerful tool aiming to provide termination certificates, but their practical implementation always had various problems when attempted to appear in real-world languages. Partly, it stemmed from complexity of the lambda-calculus with subtyping, and only after significant time there appeared a theory without significant limitations. We managed to improve this theory even further, making it more suitable for practical considerations.

We also would like to note that our algorithm significantly improves the usability of coinductive types. The existing guardedness checker is too weak for non-trivial development, and people usually resort to explicit sized types or guard modalities [Nakano, 2000]. However, as we mentioned earlier, Agda's sized types are infective, and guard modality requires clocks to unlock the interplay of coinductive data with induction. We believe that our approach, perhaps combined with explicit size annotations developed in the future, unlocks a new view on the application of sized types.

Summing up, we are quite optimistic about the future of type-based termination in dependently typed languages. We believe that this work is a step towards performant and powerful implementation of termination checkers.

Bibliography

Andreas Abel. foetus - termination checker for simple functional programs. 2002. URL `https://api.semanticscholar.org/CorpusID:59868470`.

Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.

Andreas Abel and Brigitte Pientka. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming*, 26:e2, 2016. doi: 10.1017/S0956796816000022.

Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991. doi: 10.1017/S0956796800020025.

Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. Practical inference for type-based termination in a polymorphic setting. In *Typed Lambda Calculi and Applications: 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005. Proceedings 7*, pages 71–85. Springer, 2005.

Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. Ciĉ: Type-based termination of recursive definitions in the calculus of inductive constructions. In *Logic Programming and Automated Reasoning*, 2006. URL `https://api.semanticscholar.org/CorpusID:38601946`.

Amir M. Ben-Amram. *General size-change termination and lexicographic descent*, page 3–17. Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3540003266.

Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

Jonathan Chan and William J. Bowman. Practical sized typing for coq. *CoRR*, abs/1912.05601, 2019. URL `http://arxiv.org/abs/1912.05601`.

Jonathan Chan, Yufeng Li, and William J. Bowman. Is sized typing for coq practical? *Journal of Functional Programming*, 33:e1, 2023. doi: 10.1017/S0956796822000120.

Thierry Coquand. Infinite objects in type theory. In *Proceedings of the International Workshop on Types for Proofs and Programs*, TYPES '93, page 62–78, Berlin, Heidelberg, 1994. Springer-Verlag. ISBN 3540580859.

Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), may 2021. ISSN 0360-0300. doi: 10.1145/3450952. URL `https://doi.org/10.1145/3450952`.

Jean-Yves Girard. Une extension de Ľinterpretation de gödel a Ľanalyse, et son application a Ľelimination des coupures dans Ľanalyse et la theorie des types. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92. Elsevier, 1971. doi: https://doi.org/10.1016/S0049-237X(08)70843-7. URL `https://www.sciencedirect.com/science/article/pii/S0049237X08708437`.

Jean-Yves Girard. Interpretation fonctionelle et elimination des coupures dans l'aritmetique d'ordre superieur. 1972. URL `https://api.semanticscholar.org/CorpusID:117631778`.

John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 410–423, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917693. doi: 10.1145/237721.240882. URL `https://doi.org/10.1145/237721.240882`.

Valery Isaev. Arend, December 2023. URL `https://github.com/JetBrains/Arend`.

Felix Joachimski and Ralph Matthes. Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and gödel's t. *Archive for Mathematical Logic*, 42:59–87, 01 2003. doi: 10.1007/s00153-002-0156-9.

Neil D. Jones, Chin Soon Lee, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, page 81–92, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581133367. doi: 10.1145/360204.360210. URL `https://doi.org/10.1145/360204.360210`.

S. C. Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society*, 53(1):41–73, 1943. ISSN 00029947. URL `http://www.jstor.org/stable/1990131`.

Simon Marlow et al. Haskell 2010 language report. *Available online http://www.haskell. org/(May 2011)*, 2010.

Hiroshi Nakano. A modality for recursion. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, LICS '00, page 255, USA, 2000. IEEE Computer Society. ISBN 0769507255.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, pages 408–425, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg. ISBN 978-3-540-37819-8.

Dana Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976. doi: 10.1137/0205037.

W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):198–212, 1967. ISSN 00224812. URL `http://www.jstor.org/stable/2271658`.

The Lean Development Team. The Lean reference manual. `https://lean-lang.org/reference/`, 2023.

Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

Anthony Vandikas. Proof of bottom with sized types, December 2022. URL `https://github.com/agda/agda/issues/6002`.

Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 347–359, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913280. doi: 10.1145/99370.99404. URL `https://doi.org/10.1145/99370.99404`.

David Wahlstedt. *Dependent Type Theory with Parameterized First-Order Data Types and Well-Founded Recursion.* Chalmers Tekniska Hogskola (Sweden), 2007.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework i: Judgments and properties. Technical report, Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie ..., 2003.

A

Appendix

In this part we present the rules for System F_ω with copatterns, which is a base system for which we apply Algorithm 2.

The syntactic entities of this system are presented in Table 8.

Pol	$\ni \pi$	$::= \circ \mid + \mid - \mid \top$	Polarity/variance
Kind	$\ni \iota, \iota'$	$::= * \mid \pi\kappa \to \kappa'$	Kind with variance
TyCtx	$\ni \Delta$	$::= \cdot \mid \Delta, X : \pi\kappa$	Type variable context

Table 8: Grammar description for kinds and sizes

$\boxed{\pi < \pi'}$ represents the rule for the lattice of polarities.

$$\frac{}{\pi \leq \pi} \qquad \frac{}{\circ \leq \pi} \qquad \frac{}{\pi \leq \top}$$

$\boxed{\pi\pi'}$ represents the rule for composition (commutative) of polarities.

$$\top\pi = \top \qquad \circ\pi = \circ\ (\pi \neq \top) \qquad +\pi = \pi \qquad -- = +$$

$\boxed{\pi^{-1}\pi}$ represents the rule for inverse composition of polarities.

$$\top^{-1}\pi = \circ \qquad \circ^{-1}\circ = \circ \qquad \circ^{-1}\pi = \top\ (\pi \neq \circ) \qquad +^{-1}\pi = \pi \qquad -^{-1}\pi = -\pi$$

$\boxed{\pi\Delta}$ and $\boxed{\pi^{-1}\Delta}$ extend the rules of composition to the typing context.

$$\pi\cdot = \cdot \qquad \pi(\Delta, X : \pi'\iota) = (\pi\Delta), X : (\pi\pi')\iota$$

$$\pi^{-1}\cdot = \cdot \qquad \pi^{-1}(\Delta, X : \pi'\iota) = (\pi^{-1}\Delta), X : (\pi^{-1}\pi')\iota$$

$\boxed{\Delta \vdash \iota}$ is a judgement of well-formedness of a kind.

$$\frac{}{\Delta \vdash *} \qquad \frac{-\Delta \vdash \iota \qquad \Delta \vdash \iota'}{\Delta \vdash \pi\iota \to \iota'}$$

The grammar for types and sizes is presented in Table 9

TyVar	$\ni X, Y, Z, i, j$		Type and size variables
TyAtom	$\ni K$	$::= X \mid 1 \mid \times \mid \to \mid \forall_\kappa \mid \exists_\kappa$	Type operators
Type	$\ni F, F', A, B, C$	$::= K \mid \lambda X : \iota.\ F \mid F\ F' \mid \mu S \mid \nu S$	Type-level expressions
Var	$\ni x, y, z$		Term variable
Ctx	$\ni \Gamma$	$::= \cdot \mid \Gamma, x : A$	Term variable context
Cons	$\ni c$		Constructor of datatype
Proj	$\ni d$		Field of record
Datatype	$\ni S$	$::= \langle c_1 : F_1; \ldots; c_n : F_n \rangle$	Datatype definition
Record	$\ni R$	$::= \{d_1 : F_1; \ldots; d_n : F_n\}$	Record definition

Table 9: Grammar description for type constructors

$\boxed{\Delta \vdash_{F_\omega} A \rightrightarrows \iota}$ describes the rules for inference of a kind for a type.

$$\frac{}{\Delta \vdash_{F_\omega} 1 \rightrightarrows *} \qquad \frac{}{\Delta \vdash_{F_\omega} \times \rightrightarrows +* \to +* \to *} \qquad \frac{}{\Delta \vdash_{F_\omega} \to \rightrightarrows -* \to +* \to *}$$

$$\frac{(X : \pi\iota) \in \Delta}{\Delta \vdash_{F_\omega} X \rightrightarrows \iota}\ \pi \leq + \qquad \frac{\Delta \vdash_{F_\omega} F \rightrightarrows \pi\iota \to \iota' \qquad \pi^{-1}\Delta \vdash_{F_\omega} G \leftleftarrows \iota}{\Delta \vdash_{F_\omega} F\ G \rightrightarrows \iota'}$$

$$\frac{-\Delta \vdash_{F_\omega} \iota}{\Delta \vdash_{F_\omega} \forall_\iota \rightrightarrows +(\circ\iota \to *) \to *} \qquad \frac{\Delta \vdash_{F_\omega} \iota}{\Delta \vdash_{F_\omega} \exists_\iota \rightrightarrows +(\circ\iota \to *) \to *}$$

$$\frac{\Delta \vdash_{F_\omega} S \leftleftarrows \circ* \to *}{\Delta \vdash_{F_\omega} \mu S \rightrightarrows *} \qquad \frac{\Delta \vdash_{F_\omega} R \leftleftarrows \circ* \to *}{\Delta \vdash_{F_\omega} \nu R \rightrightarrows *}$$

$\boxed{\Delta \vdash_{F_\omega} F \leftleftarrows \iota}$ is a judgement about the checking of a kind for a type.

$$\frac{\Delta \vdash_{F_\omega} F \rightrightarrows \iota \qquad \Delta \vdash_{F_\omega} \iota \leq \iota'}{\Delta \vdash_{F_\omega} F \leftleftarrows \iota'} \qquad \frac{\circ^{-1}\Delta \vdash_{F_\omega} \iota \qquad \Delta, X : \pi\iota \vdash_{F_\omega} F \leftleftarrows \iota'}{\Delta \vdash_{F_\omega} \lambda X : \iota.\ F \leftleftarrows \pi\iota \to \iota'}$$

$$\frac{\Delta \vdash_{F_\omega} S_c \leftleftarrows \iota \text{ for all } c \in S}{\Delta \vdash_{F_\omega} S \leftleftarrows \iota} \qquad \frac{\Delta \vdash_{F_\omega} R_d \leftleftarrows \iota \text{ for all } d \in R}{\Delta \vdash_{F_\omega} R \leftleftarrows \iota}$$

$\boxed{\Delta \vdash_{F_\omega} \Delta'}$ is a relation for well-formedness of a kinding context.

$$\frac{}{\Delta \vdash_{F_\omega} \cdot} \qquad \frac{\circ^{-1}\Delta \vdash_{F_\omega} \iota \qquad \Delta, X : \pi\iota \vdash_{F_\omega} \Delta'}{\Delta \vdash_{F_\omega} X : \pi\iota, \Delta'}$$

Given a well-formed kinding context, we can also define the well-formedness of variable context $\boxed{\Delta \vdash_{F_\omega} \Gamma}$:

$$\frac{}{\Delta \vdash_{F_\omega} \cdot} \qquad \frac{\Delta \vdash_{F_\omega} \Gamma \qquad \Delta \vdash_{F_\omega} A}{\Delta \vdash_{F_\omega} \Gamma, x : A}$$

The grammar for patterns is presented in Table 10.

Pat	$\ni p$	$::= x \mid () \mid (p_1, p_2) \mid c\ p \mid {}^{Q}p$	Pattern
Copat	$\ni q$	$::= p \mid X \mid .d$	Copattern
PatSp	$\ni \mathbf{q}$	$::= \vec{q}$	Pattern spine

Table 10: Grammar description for patterns

Formally, the relation $\boxed{\Delta; \Gamma \vdash_{F_\omega \Delta_0} p \leftleftarrows A}$ defines the rules of typing for pattern matching.

$$\frac{}{\cdot; x : A \vdash_{F_\omega \Delta_0} x \leftleftarrows A} \qquad \frac{}{\cdot; \cdot \vdash_{F_\omega \Delta_0} () \leftleftarrows 1}$$

$$\frac{\Delta_1; \Gamma_1 \vdash_{F_\omega \Delta_0} p_1 \leftleftarrows A_1 \qquad \Delta_2; \Gamma_2 \vdash_{F_\omega \Delta_0} p_2 \leftleftarrows A_2}{\Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \vdash_{F_\omega \Delta_0} (p_1, p_2) \leftleftarrows A_1 \times A_2}$$

$$\frac{\Delta; \Gamma \vdash_{F_\omega \Delta_0} p \leftleftarrows S_c(\mu S)}{\Delta; \Gamma \vdash_{F_\omega \Delta_0} c\ p \leftleftarrows \mu S} \qquad \frac{\Delta; \Gamma \vdash_{F_\omega \Delta_0, X:\kappa} p \leftleftarrows F\ @^{\kappa}\ X}{X : \kappa, \Delta; \Gamma \vdash_{F_\omega \Delta_0} {}^{X}p \leftleftarrows \exists_\kappa F}$$

Here we define relation $\boxed{\Delta; \Gamma | A \vdash_{F_\omega \Delta_0} \vec{q} \rightrightarrows C}$.

$$\frac{}{\cdot; \cdot | C \vdash_{F_\omega \Delta_0} \cdot \rightrightarrows C} \qquad \frac{\Delta_1; \Gamma_1 \vdash_{F_\omega \Delta_0} p \leftleftarrows A \qquad \Delta_2; \Gamma_2 | B \vdash_{F_\omega \Delta_0} \vec{q} \rightrightarrows C}{\Delta_1, \Delta_2; \Gamma_1, \Gamma_2 | A \to B \vdash_{F_\omega \Delta_0} p\ \vec{q} \rightrightarrows C}$$

$$\frac{\Delta; \Gamma | R_d(\nu R) \vdash_{F_\omega \Delta_0} \vec{q} \rightrightarrows C}{\Delta; \Gamma | \nu R \vdash_{F_\omega \Delta_0} .d\ \vec{q} \rightrightarrows C} \qquad \frac{\Delta; \Gamma | F\ @^{\kappa}\ X \vdash_{F_\omega \Delta_0, X:\kappa} \vec{q} \rightrightarrows C}{X : \kappa, \Delta; \Gamma | \forall_\kappa F \vdash_{F_\omega \Delta_0} X\ \vec{q} \rightrightarrows C}$$

The syntactic entities that are used in the program are the presented in the Table 11.

Exp	$\ni r, s, t$	$::= u \mid v \mid \lambda\vec{D}$	Term
Intro	$\ni v$	$::= () \mid (t_1, t_2) \mid c\ t \mid {}^{G}t$	Introduction term
App	$\ni u$	$::= x \mid f \mid r\ e$	Applicative term
Fun	$\ni f, g$		Function name
Elim	$\ni e$	$::= t \mid G \mid .d$	Elimination

Table 11: Grammar description for terms

$\boxed{\Sigma; \Delta; \Gamma \vdash_{F_\omega} r \rightrightarrows C}$ is a judgement about expression typing (inference mode).

$$\frac{(x : A) \in \Gamma}{\Sigma; \Delta; \Gamma \vdash_{F_\omega} x \rightrightarrows A} \qquad \frac{\Sigma; \Delta; \Gamma \vdash_{F_\omega} r \rightrightarrows \nu R}{\Sigma; \Delta; \Gamma \vdash_{F_\omega} r.d \rightrightarrows R_d(\nu R)}$$

$$\frac{\Sigma; \Delta; \Gamma \vdash_{F_\omega} r \rightrightarrows A \to B \qquad \Sigma; \Delta; \Gamma \vdash_{F_\omega} s \leftleftarrows A}{\Sigma; \Delta; \Gamma \vdash_{F_\omega} r\ s \rightrightarrows B}$$

$$\frac{\Sigma; \Delta; \Gamma \vdash_{F_\omega} r \rightrightarrows \forall_\kappa F \qquad \Delta \vdash_{F_\omega} F' \leftleftarrows \kappa}{\Sigma; \Delta; \Gamma \vdash_{F_\omega} r\ F' \rightrightarrows F\ @^{\kappa}\ F'}$$

$$\frac{\Delta \vdash_{F_\omega} A \qquad \Sigma; \Delta; \Gamma \vdash_{F_\omega} t \leftleftarrows A}{\Sigma; \Delta; \Gamma \vdash_{F_\omega} (t : A) \rightrightarrows A} \qquad \frac{(g : A) \in \Sigma}{\Sigma; \Delta; \Gamma \vdash_{F_\omega} g \rightrightarrows A}$$

$\boxed{\Sigma; \Delta; \Gamma \vdash_{F_\omega} r \Leftarrow C}$ is a judgement describing the expression typing in checking mode.

$$\frac{}{\Sigma; \Delta; \Gamma \vdash_{F_\omega} () \Leftarrow 1} \qquad \frac{\Sigma; \Delta; \Gamma \vdash_{F_\omega} t_1 \Leftarrow A_1 \quad \Sigma; \Delta; \Gamma \vdash_{F_\omega} t_2 \Leftarrow A_2}{\Sigma; \Delta; \Gamma \vdash_{F_\omega} (t_1, t_2) \Leftarrow A_1 \times A_2}$$

$$\frac{\Sigma; \Delta; \Gamma \vdash_{F_\omega} t \Leftarrow S_c(\mu S)}{\Sigma; \Delta; \Gamma \vdash_{F_\omega} c\ t \Leftarrow \mu S} \qquad \frac{\Delta \vdash_{F_\omega} F' \Leftarrow \kappa \quad \Sigma; \Delta; \Gamma \vdash_{F_\omega} t \Leftarrow F\ @^\kappa\ F'}{\Sigma; \Delta; \Gamma \vdash_{F_\omega}{}^{F'} t \Leftarrow \exists_\kappa F}$$

$$\frac{\Sigma; \Delta; \Gamma \vdash_{F_\omega} D_k \Leftarrow A \text{ for all } k}{\Sigma; \Delta; \Gamma \vdash_{F_\omega} \lambda \vec{D} \Leftarrow A} \qquad \frac{\Sigma; \Delta; \Gamma \vdash_{F_\omega} r \rightrightarrows C}{\Sigma; \Delta; \Gamma \vdash_{F_\omega} r \Leftarrow C}$$

The syntactic categories for definitions are presented in Table 12.

DefCl	$\ni D$	$::= \{\mathbf{q} \to t\}$	Definition clause
Def	$\ni \vec{D}$	$::= \{D_1; \ldots; D_n\}$	Definition clauses
Decl	$\ni \delta$	$::= f:\ A = \vec{D}$	Declaration
Block	$\ni \Xi$	$::= \text{mutual } \vec{\delta}$	Mutual block
Prg	$\ni P$	$::= \vec{\Xi}; u$	Program
Sig	$\ni \Sigma$	$::= \vec{\delta}$	Signature

Table 12: Grammar description for definitions

We define $\boxed{\Sigma; \Delta; \Gamma \vdash_{F_\omega} D \Leftarrow A}$ as a rule of checking a clause.

$$\frac{\Delta'; \Gamma' | A \vdash_{F_{\omega} \Delta} \vec{q} \rightrightarrows C \quad \Sigma; \Delta, \Delta'; \Gamma, \Gamma' \vdash_{F_\omega} t \Leftarrow C}{\Sigma; \Delta; \Gamma \vdash_{F_\omega} \{\vec{q} \to t\} \Leftarrow A}$$

We can introduce the rule for checking a sequence of clauses $\boxed{\Sigma; \Delta; \Gamma \vdash_{F_\omega} \vec{D} \Leftarrow A}$

$$\frac{\Sigma; \Delta; \Gamma \vdash_{F_\omega} D_k \Leftarrow A \text{ for all } k}{\Sigma; \Delta; \Gamma \vdash_{F_\omega} \vec{D} \Leftarrow A}$$

Formally, the rule $\boxed{\Sigma; \Xi \vdash_{F_\omega} f}$ of checking a functional symbol is defined in the following way:

$$\frac{\Sigma; \cdot; \cdot \vdash_{F_\omega} \vec{D} \Leftarrow A}{\Sigma; \Xi \vdash_{F_\omega} f : (A) = \vec{D}}$$

Given the definitions above, we can introduce the rule $\boxed{\Sigma \vdash_{F_\omega} \Xi}$ of typing of a block.

$$\frac{\Sigma; \Xi \vdash_{F_\omega} f : A = \vec{D} \text{ for all } f \in \Xi}{\Sigma \vdash_{F_\omega} \Xi}$$

Similarly, we can define $\boxed{\vdash_{F_\omega} \Sigma}$ – the typing of a set of signatures.

$$\frac{}{\vdash_{F_\omega} \cdot} \qquad \frac{\vdash_{F_\omega} \Sigma \quad \Sigma \vdash_{F_\omega} \Xi}{\vdash_{F_\omega} \Xi\ \Sigma}$$

Finally, we are ready to wrap up and provide a rule for checking the whole program $\boxed{\vdash_{F_\omega} P}$.

$$\frac{\vdash_{F_\omega} \Sigma \qquad \Sigma; \cdot; \cdot; \cdot; \vdash_{F_\omega} u \rightrightarrows A}{\vdash_{F_\omega} \Sigma; u}$$