Type Inference for Nakano's Modality

Abstract

Around fifteen years ago, Nakano introduced an extension to a system of recursive types for the Lambda Calculus consisting of a unary type constructor, or *modality*, guarding the occurrences of recursive references. This modest extension afforded a powerful result: the guarantee of a particular kind of termination (specifically, head normalisation). That is, programs typeable in this system are guaranteed to produce output.

Since then, much research has been done to understand the semantics of this modality, and utilise it in more sophisticated type systems. Notable contributions to this effort include: the work of Birkedal et al. and Benton and Krishnaswami, who study semantic models induced by the modality; research by Appel at al. and Pottier who employ the modality in typed intermediate representations of programs; and Atkey and McBride's work on co-programming for infinite data.

While some of this work explicitly addresses the possibility of type *checking* (e.g., that of Pottier), to the best of our knowledge type *inference* is still an unsolved problem. The lack of a type inference procedure presents a significant barrier to the wider adoption of Nakano's technique in practice. In this paper, we describe a (conservative) extension to Nakano's system which allows us to develop a unification-based algorithm for deciding whether a term is typeable by constructing a suitably general type.

Categories and Subject Descriptors D.3.3 [*Programming Language*]: Language Constructs and Features—Recursion; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Logics of Programs, Mechanical Verification; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Lambda Calculus and Related Systems

Keywords lambda calculus, approximation modality, type inference, insertion variables

1. Introduction

Type systems provide an abstract and compositional way to reason about the behaviour of programs [26]. Usually, one requires that a type system is at least *sound*, and thus captures a notion of partial correctness for programs. Some type systems (referred to as *normalising*) are able to go further and guarantee total correctness: not only will the program not result in an error, but it will also eventually produce a value, i.e. terminate. These stronger guarantees come at a price, however—such type systems are typically either too weak, not being able to type enough programs (e.g. simply typed lambda calculus [9]), or too powerful in that typeability is undecidable (e.g. system F [14, 28], intersection types [30]).

The question of how to type recursively defined programs is also inextricably tied up with the trade-off between partial and total correctness. Approaches to typing recursion may take one of two forms. On the one hand, recursive definitions can be allowed at the type level [7]; on the other, constructs can be added to the language (along with appropriate typing rules) whose computational behaviour implements the desired recursion scheme(s). In either case, permitting general recursive references prevents the system from differentiating between partial and total correctness of programs—there must be non-terminating programs which are well-typed. Fundamentally, this has to with the fact that such type systems, when viewed as logics under the Curry-Howard propositions-as-types correspondence [16], are *inconsistent*.

In many cases, we are happy to settle for partial correctness in return for the ability to write typeable recursive programs in a natural way, however this is not an entirely satisfactory solution. Conversely, it *is* possible to obtain a normalising system with typeable recursion by placing some kind of restriction on the occurrence of recursive references [21], or on the structure of programs [13, 20]. However this is not fully satisfactory either as it either forces the programmer to write code in an unnatural style, or fails to be able to express useful idioms. Mendler's well-known restriction to positive occurrences of recursive type references only [21], for example, is incompatible with the type schemes for binary methods in object-orientation, and cannot deal with fixed point combinators.

In order to incorporate a simpler and more flexible form of recursive references into a consistent theory of propositions-as-types, Nakano introduced a unary type constructor ($\bullet$), along with the modest restriction (called *properness*) that each recursive reference appear under the scope of this constructor [25]. Thus, the type $\mu X.(X \to A)$ is not well-formed, whereas $\mu X.(\bullet X \to A)$ is. When interpreted as a logical entity via the Curry-Howard isomorphism this type constructor corresponds to a modality, and its nature as such is elucidated by Nakano using a Kripke-style semantics. The semantics of Nakano's modality and systems incorporating it have since been studied in more detail. Birkedal et al., for instance, have studied the relationship of the modality to step-indexed models of logical relations, and the topos of trees [5]. Krishnaswami and Benton [18] have also considered the role of the modality in context of reactive programming, and its underlying model of ultrametric spaces.

From a computational perspective, the consistency of Nakano's system as a logic corresponds to the property that typeable programs *converge*; that is, they will eventually produce a(n at least partial) value. Combined with the fact that the system does not require any syntactic restriction on terms, or on the position of recursive references in types, the utility and applicability of Nakano's approach is immediately obvious. This has not gone unnoticed by the research community. Appel and colleagues [2] were the first to use Nakano's modality in a system for typed low level and intermediate languages. This has been followed up by Pottier [27]. Atkey and McBride [3] and Møgelberg [23] have incorporated Nakano's

ideas into type systems for productive co-programming (i.e. programming with streams, and other infinite objects).

Despite the wide variety of systems that have been developed, so far none of them have addressed the question of automatically inferring types for programs. Pottier discusses decidability of type *checking* for his system, and it seems likely that the other systems admit the same result. Thus, programs in these systems can be verified correct provided the programmer gives type annotations. Notwithstanding, the type *inference* paradigm offers numerous benefits: it frees the programmer from the burden of having to annotate programs; it can potentially provide the programmer with more detailed information regarding errors; and it is able to aid in efficient generation of optimised code during compilation. Systems employing type inference, such as ML, Ocaml, and Haskell, are well established and make use of Hindley-Milner style type inference algorithms [10, 15, 22]. These are based on the notion of *type unification* [29] and have the advantage of being intuitive and easy to understand and implement.

We should also mention that, more or less concurrently with the developments in Nakano-style systems, alternative frameworks of type-based terminating systems have been developed, see e.g. [4, 31]. Nakano's approach has connections with these frameworks on a fundamental level, but it has clear advantages over these systems: its definition is concise and it is more intuitive. Indeed, Møgelberg [23] comments that (Nakano-style) guarded recursion can be seen as a 'lightweight' version of sized types [4].

The aim of our work is to come to unification-based solution to the type inference problem for Nakano's system—so to develop an algorithm in the Hindley-Milner (HM) style (although, at this stage, with let-polymorphism). Extending the basic notion of unification to recursively defined types is easy: when trying to unify a type variable φ with a type τ in which it occurs (an 'occurs check'), instead of failing one constructs a substitution of the variable φ for an appropriately constructed recursive type solving the equation $\varphi = \tau$. The situation is more complicated in the guarded Nakano setting, since we have to ensure that these types satisfy the properness restriction. We also find that the process of unification is also more complicated since a term has, in general, a *family* of Nakano types rather than a single (principal) type as in systems of unrestricted recursive types [7].

We propose an extension of Nakano's system which allows us to represent the family of types possessed by a term, by explicitly marking in types where the $\bullet$ modality may occur. We call the elements comprising this extension *insertion variables*, since they allow modalities to be inserted into types. They have connections with, and were in part inspired by, Kfoury and Wells' expansion variables for intersection types [17]. The notion of unification and type inference that we develop for this extended system constructs and preserves this generality. Our extension is both conservative over, and complete with respect to, Nakano's original system. Thus our approach *is* a type inference procedure for Nakano's system.

Contributions To summarise, the contributions of this paper are: (i) a procedure for soundly unifying recursive types with the $\bullet$ modality modulo subtyping; (ii) the concept of insertion variables as a mechanism for indicating where in types the $\bullet$ modality may appear; and (iii) the first type inference procedure for a system with the Nakano modality. Our approach is based on unification, which we believe makes it intuitive and simple to understand.

We hope the approach that we demonstrate in this paper will serve as a proof-of-concept and facilitate the development of type inference for other, more sophisticated systems with Nakano-style modalities. This would, in turn, bring the obvious benefits of Nakano-style program verification closer to practical use.

$$\tau \leq \top \qquad \tau \leq \bullet\tau \qquad \bullet(\sigma \to \tau) \simeq \bullet\sigma \to \bullet\tau$$

$$\frac{\sigma \leq \tau}{\bullet\sigma \leq \bullet\tau} \qquad \frac{\sigma' \leq \sigma \quad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'}$$

Figure 1. Subtyping for Nakano's System

Paper Outline The rest of the paper is organised as follows. In Section 2 we review Nakano's original system. In Section 3 we highlight some of the subtleties involved in type inference by considering a specific example: that of Curry's fixed point combinator. We then describe how we extend Nakano's system by adding insertion variables in Section 4. Section 5 details our unification procedure, and we show how this enables us to infer types in Section 6. Finally, in Section 7 we conclude and consider directions for future work.

2. Nakano's System

Nakano defined four variant type systems employing the $\bullet$ modality [25]. In this section we recall one of them in particular, S-$\lambda\bullet\mu^{+}$, since it is that system which we build on in our work. A similar summary of this system is given in [27].

Definition 2.1 (Terms)**.** *The terms of Nakano's system are those of the λ-calculus:*

$$M, N ::= x \mid \lambda x.M \mid MN$$

We write $M \to N$ for the standard notion of (multistep) β-reduction on terms.

Types are (possibly infinite) tree structures, constructed using the standard function (binary) type constructor $\to$ and the (unary) $\bullet$ modality.

Definition 2.2 (Types)**.** *Types are defined co-inductively as follows:*

$$\tau, \sigma ::= \varphi \mid \bullet\tau \mid \sigma \to \tau$$

where φ ranges over a countably infinite set of type variables. We use $\mathsf{Vars}(\tau)$ to denote the set of type variables occurring in τ, and we write $\bullet^n\tau$ to denote the type $\underbrace{\bullet \ldots \bullet}_{n \text{ times}}\tau$.

This definition allows for types to have arbitrary infinite structure, however we will restrict our attention to those types with finite or *regular* infinite structure. Such types can be finitely represented using recursive type definitions. Types must additionally adhere to the following well-formedness condition.

Definition 2.3 (Properness)**.** *We say that a type τ is* proper *iff every infinite path through τ passes a $\bullet$ constructor infinitely often.*

When considering types using recursive definitions, this corresponds to the restriction mentioned in the introduction that every recursive type reference must occur under the scope of the $\bullet$ modality. The following property of types is also needed.

Definition 2.4 (Finiteness)**.** *We say that a type τ is* tail finite *(or simply* finite*) exactly when every infinite path through τ enters the domain (i.e. left-hand side) of a $\to$ constructor.*

We can consider non-finite types to be equivalent to a universal (top) type, and so from now on we will assume the existence of such a (unique) type, denoted by $\top$. As shown by Nakano, it is decidable if a type is finite or not [25]. Finite types are those tree structures having a right-most leaf (i.e. type variable), which we will denote using $\mathsf{Tail}(\tau)$.

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash M : \top}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

$$\frac{\bullet\,\Gamma \vdash M : \bullet\,\tau}{\Gamma \vdash M : \tau} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \tau}\ (\sigma \leq \tau)$$

Figure 2. Type Assignment for Nakano's System

The subtyping relation is defined as the largest relation on types satisfying the rules in Figure 1. We write $\sigma \simeq \tau$ to mean that both $\sigma \leq \tau$ and $\tau \leq \sigma$. For standard recursive types (i.e. without Nakano's modality) it has been shown that it is decidable whether two types are in the subtype relation [1, 6, 8, 11, 12], and these results can straightforwardly be extended to Nakano types (see [27]). For space reasons we elide the details here and refer the reader to the literature. We develop our unification procedure by extending these same techniques, however, and so the presentation in Section 5 should afford a flavour of how they work.

Types are assigned to terms using the rules given in Figure 2. As usual, type environments Γ map term variables to types, and the typing judgement $\Gamma \vdash M : \tau$ says that the term M can be assigned the type τ using the type environment Γ. We write $\vdash M : \tau$ when τ can be assigned to M using the empty typing environment. $\Gamma, x : \sigma$ stands for the type environment where $(\Gamma, x : \sigma)(y) = \sigma$ if $x = y$ and $(\Gamma, x : \sigma)(y) = \Gamma(y)$ otherwise. $\bullet\,\Gamma$ denotes the typing environment defined by $(\bullet\,\Gamma)(x) = \bullet\,\tau$ if and only if $\Gamma(x) = \tau$.

The type system satisfies *subject reduction*:

Lemma 2.5 ([25, Prop. 2])**.** *If $\Gamma \vdash M : \tau$ and $M \to N$, then $\Gamma \vdash N : \tau$.*

More importantly, however, the system has a *convergence* property. We say that a term has a head normal form if it reduces to (or already is) a term of the form $\lambda x_1 \ldots x_n.xM_1 \ldots M_n$. Convergence, then, is the following:

Theorem 2.6 (Convergence [25, Thm. 2])**.** *If $\Gamma \vdash M : \tau$ with τ finite, then M has a head normal form.*

The notion of head normal form formalises the intuitive concept of 'output' for a program, and so this result is the essence of the productivity guarantee given by the Nakano modality.

3. Motivating Insertion Variables

As stated in the introduction, our aim is to develop a HM style algorithm for inference of Nakano types. In so doing, we discover that an extension is necessary in order to keep track of vital information characterising the *family* of types that may be assigned to terms. To do this, we propose insertion variables. These mark the locations within types where occurrences of $\bullet$ may soundly be introduced. In this respect, they are very similar to the expansion variables of Kfoury and Wells [17], which serve to mark the locations in types where intersections may be introduced. Note that we will assume some familiarity on the part of the reader with the standard procedure for type inference in the simply typed lambda calculus.

In this section, we motivate our introduction of insertion variables by considering how one might go about trying to use the HM approach to infer a type for a particular term: Curry's fixed point combinator. We will see that, at a certain point, type inference will 'get stuck' due to the inability to unify two types. The cause of this failure will be the absence of a $\bullet$ at a crucial position in one of the types. We will point out that it is not the case that we cannot use the required type for the term we are building (in fact, we can). It is simply that the unification has been too eager, in the sense that it produces only as many occurrences of $\bullet$ as required for a given unification (sub)problem even it is possible to produce more. It is here that insertion variables come to the rescue, since they leave open the possibility to add more occurrences of $\bullet$ as and when they are needed.

Consider Curry's fixed point combinator:

$$\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Using a compositional approach to type inference, we start by building types for the smaller components of terms before then using these to construct the types for the larger components of which they are a part. The basic building block of $\mathbf{Y}$ is the self-application xx. After generating fresh typings $\langle\, \{x{:}\varphi_1\}, \varphi_1 \,\rangle$ and $\langle\, \{x{:}\varphi_2\}, \varphi_2 \,\rangle$ for each occurrence of the term variable x, we first unify the type φ_1 with $\varphi_2 \to \varphi_3$ (φ_3 fresh) so that we may type the application. We must then unify the two resulting type environments $\{x{:}\varphi_2 \to \varphi_3\}$ and $\{x{:}\varphi_2\}$. Since φ_2 occurs in the type $\varphi_2 \to \varphi_3$, we must construct a recursive type as a solution and since we are in Nakano's system this type must be proper. We should therefore construct the substitution $[\varphi_2 \mapsto \mu.\bullet\,\mathbf{0} \to \varphi_3]$. Applying this substitution to the two type environments gives us $\{x{:}(\mu.\bullet\,\mathbf{0} \to \varphi_3) \to \varphi_3\}$ and $\{x{:}\mu.\bullet\,\mathbf{0} \to \varphi_3\}$. The latter environment is the more specific one[1] and so we must use that for typing the application. Therefore, we have the following typing for xx: $\langle\, \{x{:}\mu.\bullet\,\mathbf{0} \to \varphi_3, \varphi_3 \,\rangle$. Eliding the fine-grained steps, which the reader may check for themselves, this leads to $\langle\, \{f{:}\varphi_3 \to \varphi_4\}, (\mu.\bullet\,\mathbf{0} \to \varphi_3) \to \varphi_4 \,\rangle$ as the typing for $\lambda x.f(xx)$.

We may now proceed to try and infer a typing for the (self) application $(\lambda x.f(xx))(\lambda x.f(xx))$. Notice that the inference procedure will produce two α-equivalent but distinct (i.e. using disjoint sets of type variables) typings for each occurrence of the subterm $\lambda x.f(xx)$: $\langle\, \{f{:}\varphi_3 \to \varphi_4\}, (\mu.\bullet\,\mathbf{0} \to \varphi_3) \to \varphi_4 \,\rangle$ and $\langle\, \{f{:}\varphi_7 \to \varphi_8\}, (\mu.\bullet\,\mathbf{0} \to \varphi_7) \to \varphi_8 \,\rangle$. Type inference continues by trying to solve the following unification problem, in order to be able to type the application (where φ_9 is fresh):

$$\mathsf{Unify?} \quad (\mu.\bullet\,\mathbf{0} \to \varphi_3) \to \varphi_4\ ,\ ((\mu.\bullet\,\mathbf{0} \to \varphi_7) \to \varphi_8) \to \varphi_9$$

The first step is to try and unify the domains of the arrow types. However, since we are dealing with a system that incorporates subtyping, we must unify domains of function types *contra-variantly*, in line with the definition of subtyping for function types:

$$\mathsf{Unify?} \quad (\mu.\bullet\,\mathbf{0} \to \varphi_7) \to \varphi_8\ ,\ \mu.\bullet\,\mathbf{0} \to \varphi_3$$

which we can try and do by unfolding the right-hand definition:

$$\mathsf{Unify?} \quad (\mu.\bullet\,\mathbf{0} \to \varphi_7) \to \varphi_8\ ,\ (\bullet\,\mu.\bullet\,\mathbf{0} \to \varphi_3) \to \varphi_3$$

Again, we proceed by trying to (contra-variantly) unify the domains of the types:

$$\mathsf{Unify?} \quad \bullet\,\mu.\bullet\,\mathbf{0} \to \varphi_3\ ,\ \mu.\bullet\,\mathbf{0} \to \varphi_7$$

However, now it is clear that we have a problem: due to the occurrence of $\bullet$ preceeding the left-hand recursive type, unification must fail. There is no substitution we can apply that will unify these types modulo the subtyping relation.

What is the cause of this failing? And indeed it *is* a failing, since there does exist a type for the fixed point combinator in Nakano's system [24]. We point to the inadequacy of the typing that we originally inferred for the subterm $\lambda x.f(xx)$. The problem is that that typing is not the *only* one which may be assigned to this term. For example, $\langle\, \{f{:}\bullet\,\varphi_7 \to \varphi_8\}, (\bullet\,\mu.\bullet\,\mathbf{0} \to \varphi_7) \to \varphi_8 \,\rangle$ is also a valid typing. In fact, this is exactly the typing that we need to

[1] Notice that the following subtyping relationship holds: $\mu.\bullet\,\mathbf{0} \to \varphi_3 \simeq (\bullet\,\mu.\bullet\,\mathbf{0} \to \varphi_3) \to \varphi_3 < (\mu.\bullet\,\mathbf{0} \to \varphi_3) \to \varphi_3$

use for the right-hand occurrence of the term $\lambda x.f(xx)$. If we rerun the unification procedure using this new type, the attempt will succeed:

$1 : \mathsf{Unify}?\ (\mu.\bullet\,\mathbf{0} \to \varphi_3) \to \varphi_4\ ,\ ((\bullet\,\mu.\bullet\,\mathbf{0} \to \varphi_7) \to \varphi_8) \to \varphi_9$

$2 : \mathsf{Unify}?\ (\bullet\,\mu.\bullet\,\mathbf{0} \to \varphi_7) \to \varphi_8\ ,\ \mu.\bullet\,\mathbf{0} \to \varphi_3$

$3 : \mathsf{Unify}?\ (\bullet\,\mu.\bullet\,\mathbf{0} \to \varphi_7) \to \varphi_8\ ,\ (\bullet\,\mu.\bullet\,\mathbf{0} \to \varphi_3) \to \varphi_3$

$4 : \mathsf{Unify}?\ \bullet\,\mu.\bullet\,\mathbf{0} \to \varphi_3\ ,\ \bullet\,\mu.\bullet\,\mathbf{0} \to \varphi_7$

$\vdots$

Given this solution to the unification problem, we can proceed straightforwardly to infer $\langle\, \emptyset, (\bullet\,\varphi_9 \to \varphi_9) \to \varphi_9 \,\rangle$ as a typing for $\mathbf{Y}$, which is the expected one.

Notice that there is no way of transforming (via substitutions and weakenings) the originally inferred typing for $\lambda x.f(xx)$ into the alternative one we have given. Something extra is required. We propose that this something extra is the notion of *insertion variable*. Consider if we had been able to infer the following typing for $\lambda x.f(xx)$: $\langle\, \{f{:}\iota\,\varphi_3 \to \varphi_4\}, (\iota\,\mu.\bullet\,\mathbf{0} \to \varphi_3) \to \varphi_4 \,\rangle$. The intention behind the ι entity is that it marks the place where we can insert a $\bullet$, exactly as required.

In the remainder of this paper we will describe how we extend Nakano's system with such an entity, and how it enables us to develop a procedure for inferring widely applicable and general typings for terms using Nakano types.

4. Extending the Type System

We now come to the point where we can begin describing our technical contribution. In this section, we will define our extension of Nakano's original system S-$\lambda\bullet\mu^+$, which consists of adding insertion variables into the language of types, and extending subtyping and type assignment accordingly. Our insertion variables are inspired by the *expansion* variables of Kfoury and Wells [17].

We first extend the definition of types with a case for constructing types using insertion variables.

Definition 4.1 (Types with Insertion Variables)**.** *Types are defined co-inductively as follows:*

$$\tau, \sigma ::= \varphi \mid \bullet\,\tau \mid \iota\,\tau \mid \sigma \to \tau$$

where φ and ι range over countably infinite sets of type and insertion variables respectively. We will write $\vec{\iota}_n\,\tau$ to denote the type $\iota_1 \ldots \iota_n\,\tau$ and for a sequence of insertion variables $\vec{\iota}_n = \iota_1 \ldots \iota_n$ we write $\iota \in \vec{\iota}_n$ where there is some $1 \leq k \leq n$ such that $\iota = \iota_k$, and write $\iota \notin \vec{\iota}_n$ when there is no such k. We will write ϵ for the sequence $\vec{\iota}_n$ when $n = 0$.

The definitions of proper and finite types transfer unchanged from Nakano's original system (see Section 2). Again, we only consider finitely representable types. The subtyping relation for the extended notion of types is defined as for the original system, as the largest relation on types satisfying the rules in Figure 1 and also the additional rules given in Figure 3. The first three of the added rules are direct analogues for insertion variables of the corresponding rules for $\bullet$. The last two are more interesting, and say that we can distribute and factorise insertion variables across occurrences of $\to$ (as well as $\bullet$) in the same way that we can for the $\bullet$ modality.

Extending the definition of type assignment is even more straightforward, and needs only the following single extra rule, again a direct analogue of the corresponding rule in Nakano's original system for $\bullet$.

$$\frac{\iota\,\Gamma \vdash M : \iota\,\tau}{\Gamma \vdash M : \tau}$$

where $\iota\,\Gamma$ denotes the typing environment defined by $(\iota\,\Gamma)(x) = \iota\,\tau$ if and only if $\Gamma(x) = \tau$. Where we want to explicitly refer to type assignment in the extended system in opposition to the original system we will write $\Gamma \vdash_+ M : \tau$, however when there is no ambiguity we will normally use the plain turnstile $\vdash$ for both systems.

$$\tau \leq \iota\,\tau \qquad \frac{\sigma \leq \tau}{\iota\,\sigma \leq \iota\,\tau} \qquad \iota(\sigma \to \tau) \simeq \iota\,\sigma \to \iota\,\tau$$

$$\iota_1\,\iota_2\,\tau \leq \iota_2\,\iota_1\,\tau \qquad \bullet\,\iota\,\tau \simeq \iota\,\bullet\,\tau$$

Figure 3. Additional Rules for Subtyping with Insertion Variables

Since the extended set of types and rules for subtyping and type assignment are strict supersets of those of Nakano's original system, we immediately obtain the corollary that our extension is *conservative* over Nakano's original system. That is to say, every type that we can assign to a term in Nakano's system we can also assign in ours.

Theorem 4.2 (Conservativity)**.** *If $\Gamma \vdash M : \tau$ in Nakano's original system, then $\Gamma \vdash_+ M : \tau$ in our extended system.*

We can also show that our extension is sound with respect to the original system, i.e. whenever we can assign a type to a term in the extended system, then we can assign a type in the original system. However, in order to show this we will first need to define an operation on types: the *insertion* operation. This operation is analogous to the familiar operation of substitution of types for (type) variables. In the case of insertions, we substitute (possibly empty) sequences of insertion variables and the $\bullet$ modality for insertion variables. It is the insertion operation that really characterises the meaning of insertion variables.

Remark Since we have defined types co-inductively, functions on types must be defined co-recursively. However, as we consider only regular (in)finite types, it suffices to define these functions inductively over their finite representations (see e.g. [19]). For ease of presentation, we will take this approach in the remainder of the paper. Nakano's original presentation uses the familiar notation of binding recursive type variables (e.g. $\mu X.\bullet\, X \to \tau$), however we switch to a presentation based on de Bruijn indices (i.e. $\mu.\bullet\,\mathbf{0} \to \tau$), ranged over by $\mathbf{n}$. This is to avoid having to deal with alpha-renaming and keeping track of equated variable names when performing unification. Certainly from an implementation point of view, this is desirable.

Before defining insertions, we first define an auxiliary operation on types which inserts a $\bullet$ into types, pushing it down until a terminal or recursive structure is reached.

Definition 4.3 (bPush)**.** *The operation* bPush *on types is defined as follows:*

$$\mathsf{bPush}(\varphi) = \bullet\,\varphi \quad \mathsf{bPush}(\bullet\,\tau) = \bullet\,(\mathsf{bPush}(\tau))$$
$$\mathsf{bPush}(\mathbf{n}) = \bullet\,\mathbf{n} \quad \mathsf{bPush}(\iota\,\tau) = \iota\,(\mathsf{bPush}(\tau))$$
$$\mathsf{bPush}(\sigma \to \tau) = (\mathsf{bPush}(\sigma)) \to (\mathsf{bPush}(\tau))$$
$$\mathsf{bPush}(\mu.\tau) = \bullet\,\mu.\tau$$

This definition can be extended to define $\mathsf{bPush}[n]$ *which inserts n modalities into a type, with* $\mathsf{bPush}[0]$ *equivalent to the identity.*

It is easy to show that $\bullet^n\,\tau \simeq \mathsf{bPush}[n](\tau)$.

Definition 4.4 (Insertions)**.** *An* insertion $[\iota \mapsto \vec{\iota}_m \bullet^n]$ *(where* $n, m \geq 0$*) is an operation on types, defined as follows:*

$$[\iota \mapsto \vec{\iota}_m \bullet^n](\varphi) = \varphi \quad [\iota \mapsto \vec{\iota}_m \bullet^n](\bullet\,\tau) = \bullet\,([\iota \mapsto \vec{\iota}_m \bullet^n](\tau))$$

$$[\iota \mapsto \vec{\iota}_m \bullet^n](\mathbf{n}) = \mathbf{n} \quad [\iota \mapsto \vec{\iota}_m \bullet^n](\mu.\tau) = \mu.([\iota \mapsto \vec{\iota}_m \bullet^n](\tau))$$

$$[\iota \mapsto \vec{\iota}_m \bullet^n](\iota'\,\tau) = \begin{cases} \vec{\iota}_m\,(\mathsf{bPush}[n]([\iota \mapsto \vec{\iota}_m \bullet^n](\tau))) & \\ & \text{if } \iota = \iota' \\ \iota'\,[\iota \mapsto \vec{\iota}_m \bullet^n](\tau) & \text{otherwise} \end{cases}$$

$$[\iota \mapsto \vec{\iota}_m \bullet^n](\sigma \to \tau) = ([\iota \mapsto \vec{\iota}_m \bullet^n](\sigma)) \to ([\iota \mapsto \vec{\iota}_m \bullet^n](\tau))$$

If I_1 *and* I_2 *are two insertions, then so is their composition* $I_2 \circ I_1$*. We extend the operation to type environments by* $I(\Gamma)(x) = I(\tau)$ *if and only if* $\Gamma(x) = \tau$*.*

We give this rather esoteric definition of insertion (using bPush), rather than the obvious straightforward one, in order for insertions to preserve a *canonical* structure of types that we will define in the next section. Working with this canonical representation allows an entirely syntax-directed definition of unification.

Insertion operations are sound with respect to subtyping.

Lemma 4.5. *Let* I *be an insertion; if* $\sigma \leq \tau$ *then* $I(\sigma) \leq I(\tau)$*.*

Proof. By co-induction on the definition of subtyping. □

This leads to the main property that we desire of insertions, that they are sound with respect to type assignment.

Theorem 4.6. *If* $\Gamma \vdash M : \tau$ *then* $I(\Gamma) \vdash M : I(\tau)$*.*

Proof. By straightforward induction on the structure of typing derivations; the case for subtyping follows from Lemma 4.5. □

This result demonstrates that insertion variables truly fulfil the purpose for which they were introduced: that is, they mark the places in types where the $\bullet$ modality may be introduced. The notion of insertion also allows us to show that our extension is sound with respect to Nakano's original system.

Theorem 4.7 (Soundness of the Extended System)**.** *If* $\Gamma \vdash_+ M : \tau$ *then* $\Gamma' \vdash M : \sigma$*, for some* Γ' *and* σ*.*

Proof. Take the insertion I which replaces each insertion variable by the empty sequence. By Theorem 4.6 we have $I(\Gamma) \vdash_+ M : I(\tau)$. Notice that $I(\Gamma)$ and $I(\tau)$ are a type environment and type respectively in the original system, since they do not contain any insertion variables. It is easy to show by induction on (extended) typing derivations that, if $\Gamma \vdash_+ M : \tau$ with Γ and τ a type environment and type in the original system, then $\Gamma \vdash M : \tau$, from which the result follows immediately. □

Since it is easy to see that insertion operations preserve finiteness of types, the corollary of this soundness result is that our extended type system also has the convergence property.

We conclude this section by giving a few type-theoretic results for our system. The first two of these will be used to show soundness of the type inference procedure (Theorem 6.5, Section 6).

Lemma 4.8 (Weakening)**.** *For type environments* Γ *and* Γ'*, write* $\Gamma' \leq \Gamma$ *to mean that for all* x*,* $\Gamma(x) = \sigma \Rightarrow \Gamma'(x) = \sigma'$ *for some* $\sigma' \leq \sigma$*; if* $\Gamma' \leq \Gamma$ *and* $\Gamma \vdash M : \tau$ *then* $\Gamma' \vdash M : \tau$*.*

Proof. By straightforward induction on the structure of derivations. □

Let Γ_1 and Γ_2 be disjoint type environments (i.e. $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$), and write $\Gamma_1 \cup \Gamma_2$ for the type environment with $\mathsf{dom}(\Gamma_1 \cup \Gamma_2) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$ satisfying $\Gamma_1(x) = \tau \Rightarrow (\Gamma_1 \cup \Gamma_2)(x) = \tau$ and $\Gamma_2(x) = \tau \Rightarrow (\Gamma_1 \cup \Gamma_2)(x) = \tau$.

Lemma 4.9 (Degradation)**.** *Let* Γ_1 *and* Γ_2 *be disjoint type environments; if* $\Gamma_1 \cup \Gamma_2 \vdash M : \tau$*, then both* $(\bullet\,\Gamma_1) \cup \Gamma_2 \vdash M : \bullet\,\tau$ *and* $(\iota\,\Gamma_1) \cup \Gamma_2 \vdash M : \iota\,\tau$*.*

Proof. By straightforward induction on typing derivations. □

Lastly, the extended system exhibits a full subject reduction property.

Theorem 4.10 (Subject Reduction)**.** *If* $\Gamma \vdash M : \tau$ *and* $M \to N$*, then* $\Gamma \vdash N : \tau$*.*

Proof. By straightforward induction on the structure of derivations. The proof exactly mirrors that for Nakano's original system. □

5. Unification Modulo Subtyping

This section describes the core mechanism needed by our type inference procedure: the unification of two types. While the formal definition of our unification process is intricate, the main technical difficulty lies in how to show that it is a computable notion. To do this, we build on and extend the techniques of Brandt and Henglein [6] for deciding equality of recursive type definitions. We define unification using a judgement $\mathsf{O} \vdash \sigma \leq \tau$, which says that the operation O unifies the types σ and τ modulo subtyping (i.e. $\mathsf{O}(\sigma) \leq \mathsf{O}(\tau)$), and derive valid judgements via an inference system. Since the inference system we define is *syntax-directed* it naturally leads to the definition of an algorithm, however to show that the algorithm is terminating we need to show that the height of a derivation (if it exists) is bounded.

5.1 A Canonical Form for Types

In order to achieve a syntax-directed unification procedure, we work with a canonical form of types that we now define. Each type has a canonical form to which it is equivalent (in the sense of the equivalence induced by subtyping).

Definition 5.1 (Canonical Types)**.** *Canonical (regular) types are defined by the following grammar:*

κ	$::=$	β	$\mid$ $\kappa_1 \to \kappa_2$		*(canonical types)*
β	$::=$	α	$\mid$ $\iota\,\beta$		*(partially approximative types)*
α	$::=$	ξ	$\mid$ $\bullet\,\alpha$		*(fully approximative types)*
ξ	$::=$	φ	$\mid$ $\mathbf{n}$	$\mid$ $\mu.\kappa$	*(exact types)*

We note that this definition of canonicity is different to the one given by Nakano in [24]. The system described there is F-$\lambda\bullet\mu^+$, and the definition given by Nakano is appropriate to that system. Our definition above is appropriate for our extension of the system S-$\lambda\bullet\mu^+$.

A further advantage of our definition of canonical types is that it affords a clean separation of the *structural* content of a type from its *logical* content. This will allow the unification procedure to treat the two sub-problems of structural unification and checking of logical consistency in an orthogonal manner. The information encapsulating logical consistency is expressed in the $\bullet$ modalities and insertion variables, whereas the structural information is contained in the functional shape of the type, given by the $\to$ and constructor and the μ operator for recursive definition.

5.2 Operations on Types

The unification procedure will return an operation on types that preserves canonicity. To that end, in addition to the operations we defined in the previous section we must define two more. The first will insert insertion variables at appropriate places according to the grammar just defined.

Definition 5.2 (iPush)**.** *The operation* iPush *on types is defined as:*

$$\mathsf{iPush}[\iota](\varphi) = \iota\,\varphi \quad \mathsf{iPush}[\iota](\bullet\,\tau) = \iota \bullet \tau$$
$$\mathsf{iPush}[\iota](\mathbf{n}) = \iota\,\mathbf{n} \quad \mathsf{iPush}[\iota](\iota'\,\tau) = \iota\,\iota'\,\tau$$
$$\mathsf{iPush}[\iota](\sigma \to \tau) = (\mathsf{iPush}[\iota](\sigma)) \to (\mathsf{iPush}[\iota](\tau))$$
$$\mathsf{iPush}[\iota](\mu.\tau) = \iota\,\mu.\tau$$

This definition is extended to sequences of insertion variables by $\mathsf{iPush}[\vec{\iota}_n](\tau) = (\mathsf{iPush}[\iota_1] \circ \ldots \circ \mathsf{iPush}[\iota_n])(\tau)$.

Analogously to the case for bPush (see Definition 4.3), it is easy to show that $\iota_1 \ldots \iota_n\,\tau \simeq \mathsf{iPush}[\vec{\iota}](\tau)$.

The other operation we need is one that substitutes types for type variables. When defining type substitutions, we will need to ensure that the type we substitute is *closed*, in the sense that its recursive definition has no 'free' occurrences of recursive type references $\mathbf{n}$ (i.e. de Bruijn indices). Although we may formally include such open representations in the set of true types by considering 'free' references to simply stand for ordinary type variables, allowing them to take place in substitutions is *unsound* since they may be 'captured' by recursive binders according to the definition of substitution we now give.

Definition 5.3 (Type Substitution)**.** *A (canonicalising) type substitution* $[\varphi \mapsto \kappa]$ *is an operation on types that replaces the type variable* φ *by the (closed) canonical type* κ*, and is defined by:*

$$[\varphi \mapsto \kappa](\mathbf{n}) = \mathbf{n} \qquad [\varphi \mapsto \kappa](\varphi') = \begin{cases} \kappa & \text{if } \varphi = \varphi' \\ \varphi' & \text{otherwise} \end{cases}$$
$$[\varphi \mapsto \kappa](\bullet\,\tau) = \mathsf{bPush}([\varphi \mapsto \kappa](\tau))$$
$$[\varphi \mapsto \kappa](\iota\,\tau) = \mathsf{iPush}[\iota]([\varphi \mapsto \kappa](\tau))$$
$$[\varphi \mapsto \kappa](\mu.\tau) = \mu.([\varphi \mapsto \kappa](\tau))$$
$$[\varphi \mapsto \kappa](\sigma \to \tau) = ([\varphi \mapsto \kappa](\sigma)) \to ([\varphi \mapsto \kappa](\tau))$$

We collect of the operations that we have defined on types into a single definition of *type operation*.

Definition 5.4 (Type Operations)**.** *A* type operation O *is either a basic operation (i.e. one of* bPush*,* iPush*, an insertion, or a type substitution), or is the composition of two type operations* $\mathsf{O}_1 \circ \mathsf{O}_2$*. Type operations are extended to type environments by* $\mathsf{O}(\Gamma)(x) = \mathsf{O}(\tau)$ *if and only if* $\Gamma(x) = \tau$.

The soundness results that we gave for insertions in Section 4 extend to the larger notion of type operation.

Lemma 5.5 (Soundness of Type Operations)**.** *Let* O *be a type operation; then the following results hold:*

1. *If* $\sigma \leq \tau$*, then* $\mathsf{O}(\sigma) \leq \mathsf{O}(\tau)$.
2. *If* $\Gamma \vdash M : \tau$*, then* $\mathsf{O}(\Gamma) \vdash M : \mathsf{O}(\tau)$.

Proof. As a generalisation of the proof for insertions, the first result follows by co-induction on the definition of subtyping; and the second by induction on the structure of typing derivations. □

5.3 Constructing Recursive Type Solutions

The key task of the unification procedure will be to construct recursively defined types which can be used as substitutions that unify types modulo subtyping. Such types will need to be constructed whenever we encounter a pair of types (σ, τ) such that σ is a type variable that occurs in τ (or vice-versa). In the context of Nakano types, there is the added complication of having to ensure that the type constructed is *proper* (note that it need not be finite). Since we have insertion variables, we are afforded some flexibility: it need not be the case that all occurrences of the type variable fall within the scope of a $\bullet$ in τ, as long as they fall under the scope of one or more insertion variables. In this case, we can construct an insertion that converts the relevant insertion variables to a $\bullet$, and then we can safely 'close' the recursive type by promoting the type variable to a recursive reference.

Definition 5.6 (Raw Type Variables)**.** *We write* $\mathsf{Raw}(\tau)$ *to denote the set of all type variables in* τ *which do not occur under the scope of either an insertion variable or the* $\bullet$ *type constructor.*

We define the notion of *cover set* to be enable us to construct proper recursive type definitions.

Definition 5.7 (Cover Set)**.** *The cover set function* Cov *is defined as follows:*

$$\mathsf{Cov}[\varphi](\varphi') = \mathsf{Cov}[\varphi](\mathbf{n}) = \mathsf{Cov}[\varphi](\bullet\,\tau) = \emptyset$$
$$\mathsf{Cov}[\varphi](\iota\,\tau) = \begin{cases} \{\iota\} & \text{if } \varphi \in \mathsf{Raw}(\tau) \\ \mathsf{Cov}[\varphi](\tau) & \text{otherwise} \end{cases}$$
$$\mathsf{Cov}[\varphi](\sigma \to \tau) = \mathsf{Cov}[\varphi](\sigma) \cup \mathsf{Cov}[\varphi](\tau)$$
$$\mathsf{Cov}[\varphi](\mu.\tau) = \mathsf{Cov}[\varphi](\tau)$$

The cover set $\mathsf{Cov}[\varphi](\tau)$ of a type τ with respect to the type variable φ is the (minimal) set of insertion variables whose conversion to $\bullet$ ensures that the type resulting from promoting φ is *closed* (i.e. a true type).

Proposition 5.8. *Let* $\varphi \in \mathsf{Vars}(\tau) \setminus \mathsf{Raw}(\tau)$ *and* $\mathsf{Cov}[\varphi](\tau) = \{\iota_1, \ldots, \iota_n\}$ *and define the operation* $\mathsf{O} = [\iota_1 \mapsto \iota_1 \bullet] \circ \ldots \circ [\iota_n \mapsto \iota_n \bullet]$*, then* $\mu.([\mathbf{0}/\varphi](\mathsf{O}(\tau)))$ *is closed.*

The last component we must define in the construction of recursive type definitions is the promotion of a type variable to a recursive reference of the type in which it appears. For example, if we wish to construct a recursively defined type that unifies φ with the type $\bullet\,\varphi \to \varphi'$, then we must promote the type variable φ to a recursive reference $\mathbf{0}$ and apply the μ recursion operator over the resulting type to obtain $\mu.\bullet\,\mathbf{0} \to \varphi'$. We then build a substitution that replaces the promoted type variable by the newly constructed recursive type, i.e. $[\varphi \mapsto \mu.\bullet\,\mathbf{0} \to \varphi']$. We will write $\mu.([\mathbf{0}/\varphi](\tau))$ to denote the result of promoting the type variable φ in τ so that it recursively references the type τ in which it appears. Crucially, the following property holds of variable promotion, meaning that such recursive solutions are *sound*:

Proposition 5.9. $\mu.([\mathbf{0}/\varphi](\tau)) \simeq [\varphi \mapsto \mu.([\mathbf{0}/\varphi](\tau))](\tau)$.

5.4 The Unification Procedure

We define the unification procedure itself as an inference system, the derivations of which are proofs of the validity of unification *judgements* of the form $\mathsf{O} \vdash \sigma \leq \tau$. The procedure arises as a proof search algorithm, the deterministic nature of which is given by the fact that the inference system is directed by the syntax of types; thus, at each stage there is only one possible inference rule that will apply.

Definition 5.10 (Unification Inference)**.** *The inference rules for deriving valid unification judgements are given in Figure 4.*

The inference system is extensive and so we do not give an exhaustive explanation of all the rules here. Instead, we will discuss some of the most important and salient aspects. The notation $[\mathbf{0} \mapsto \mu.\tau](\tau)$ which appears in many of the rules is used to indicate the *unfolding* of a recursive type definition (since the concept is standard, we elide a formal definition). Id denotes the identity substitution. Additionally, apart from the (top) rule where we explicitly state we consider the $\top$ type, we assume that all types are not (equivalent to) $\top$.

We note that the rules fall into two categories: logical rules, and structural rules. The logical rules produce insertions, aiming

Top Types (Structural Rule) **Unifyfing Type Variables (Structural Rules)**

$$\frac{}{\mathsf{Id} \vdash \tau \leq \top} \qquad \frac{(\iota \notin \vec{\iota} \text{ and } r \leq s)}{[\iota \mapsto \vec{\iota}\bullet^{s-r}] \vdash \iota \bullet^r \varphi \leq \vec{\iota}\bullet^s \varphi} \qquad \frac{(\iota \notin \vec{\iota} \text{ and } r \leq s)}{[\iota \mapsto \vec{\iota}\bullet^{r-s}] \vdash \vec{\iota}\bullet^r \varphi \leq \iota \bullet^s \varphi} \qquad \frac{(r \leq s)}{\mathsf{Id} \vdash \bullet^r \varphi \leq \bullet^s \varphi}$$

$$\frac{(\varphi \neq \varphi' \text{ and } r \leq s)}{[\varphi \mapsto \bullet^{s-r} \varphi'] \vdash \bullet^r \varphi \leq \bullet^s \varphi'} \qquad \frac{(\varphi \neq \varphi' \text{ and } s < r)}{[\varphi' \mapsto \bullet^{r-s} \varphi] \vdash \bullet^r \varphi \leq \bullet^s \varphi'} \qquad \frac{(s < r)}{[\varphi \mapsto \top] \vdash \bullet^r \varphi \leq \bullet^s \varphi}$$

Unifying Type Variables (Logical Rules) (where $\mathsf{O}_1 = [\iota \mapsto \epsilon]$)

$$\frac{\mathsf{O} \vdash \mathsf{O}'(\vec{\iota}_n \bullet^r \varphi) \leq \mathsf{O}'(\vec{\iota'}_m \bullet^s \varphi')}{\mathsf{O} \circ \mathsf{O}' \vdash \iota \vec{\iota}_n \bullet^r \varphi \leq \iota' \vec{\iota'}_m \bullet^s \varphi'} \begin{pmatrix} \iota \neq \iota' \\ n, m > 0 \\ \mathsf{O}' = [\iota \mapsto \iota'] \end{pmatrix} \qquad \frac{\mathsf{O} \vdash \bullet^r \varphi \leq \bullet^s \varphi' \ (\iota \notin \vec{\iota}, \varphi \neq \varphi')}{\mathsf{O} \circ [\iota \mapsto \vec{\iota}] \vdash \iota \bullet^r \varphi \leq \vec{\iota'} \bullet^s \varphi'} \qquad \frac{\mathsf{O} \vdash \bullet^r \varphi \leq \bullet^s \varphi' \ (\iota \notin \vec{\iota}, \varphi \neq \varphi')}{\mathsf{O} \circ [\iota \mapsto \vec{\iota}] \vdash \vec{\iota}\bullet^r \varphi \leq \iota' \bullet^s \varphi'}$$

$$\frac{\mathsf{O}_2 \vdash \bullet^r \varphi \leq \mathsf{O}_1(\vec{\iota}\bullet^s \varphi')}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \bullet^r \varphi \leq \vec{\iota}\bullet^s \varphi'} \begin{pmatrix} \iota \in \vec{\iota} \text{ or else} \\ \varphi = \varphi', s < r \end{pmatrix} \qquad \frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\vec{\iota}\bullet^r \varphi) \leq \bullet^s \varphi'}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \vec{\iota}\bullet^r \varphi \leq \iota \bullet^s \varphi'} \begin{pmatrix} \iota \in \vec{\iota} \text{ or else} \\ \varphi = \varphi', r \leq s \end{pmatrix} \qquad \frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\vec{\iota}\bullet^r \varphi) \leq \mathsf{O}_1(\kappa_1 \to \kappa_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \vec{\iota}\bullet^r \varphi \leq \kappa_1 \to \kappa_2}$$

$$\frac{\mathsf{O}_2 \vdash \bullet^r \varphi \leq \mathsf{O}_1(\vec{\iota}_m \bullet^s \varphi')}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \bullet^r \varphi \leq \iota \vec{\iota}_m \bullet^s \varphi'} \ (m > 0) \qquad \frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\vec{\iota}_n \bullet^r \varphi) \leq \bullet^s \varphi'}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \vec{\iota}_n \bullet^r \varphi \leq \bullet^s \varphi'} \ (n > 0)$$

Constructing Standard Substitutions (Structural Rules) (where all types are *closed*)

$$\frac{(\varphi \notin \mathsf{Vars}(\kappa_1 \to \kappa_2))}{[\varphi \mapsto \kappa_1 \to \kappa_2] \vdash \varphi \leq \kappa_1 \to \kappa_2} \qquad \frac{(\varphi \notin \mathsf{Vars}(\kappa_1 \to \kappa_2))}{[\varphi \mapsto \kappa_1 \to \kappa_2] \vdash \kappa_1 \to \kappa_2 \leq \vec{\iota}\bullet^s \varphi}$$

$$\frac{(\varphi \notin \mathsf{Vars}(\mu.\kappa), r \leq s)}{[\varphi \mapsto \bullet^{s-r} \mu.\kappa] \vdash \bullet^r \varphi \leq \bullet^s \mu.\kappa} \qquad \frac{(\varphi \notin \mathsf{Vars}(\mu.\kappa), s \leq r)}{[\varphi \mapsto \bullet^{r-s} \mu.\kappa] \vdash \bullet^r \mu.\kappa \leq \bullet^s \varphi} \qquad \frac{(\varphi \notin \mathsf{Vars}(\mu.\kappa), r < s)}{[\varphi \mapsto \mu.\kappa] \vdash \bullet^r \mu.\kappa \leq \bullet^s \varphi}$$

Constructing Recursive Solutions (Structural Rules) (where all types are *closed*)

$$\frac{\begin{pmatrix} \varphi \in \mathsf{Vars}(\kappa_1 \to \kappa_2) \setminus \mathsf{Raw}(\kappa_1 \to \kappa_2) \\ \mathsf{Cov}[\varphi](\kappa_1 \to \kappa_2) = \{\iota_1, \ldots, \iota_n\} \\ \mathsf{O} = [\iota_1 \mapsto \iota_1 \bullet] \circ \ldots \circ [\iota_n \mapsto \iota_n \bullet] \end{pmatrix}}{[\varphi \mapsto \mu.([\mathbf{0}/\varphi](\mathsf{O}(\kappa_1 \to \kappa_2)))] \circ \mathsf{O} \vdash \varphi \leq \kappa_1 \to \kappa_2} \qquad \frac{\begin{pmatrix} \varphi \in \mathsf{Vars}(\kappa_1 \to \kappa_2) \setminus \mathsf{Raw}(\kappa_1 \to \kappa_2) \\ \mathsf{Cov}[\varphi](\kappa_1 \to \kappa_2) = \{\iota_1, \ldots, \iota_n\} \\ \mathsf{O} = [\iota_1 \mapsto \iota_1 \bullet] \circ \ldots \circ [\iota_n \mapsto \iota_n \bullet] \end{pmatrix}}{[\varphi \mapsto \mu.([\mathbf{0}/\varphi](\mathsf{O}(\kappa_1 \to \kappa_2)))] \circ \mathsf{O} \vdash \kappa_1 \to \kappa_2 \leq \vec{\iota'} \bullet^s \varphi}$$

$$\frac{\begin{pmatrix} r \leq s, \varphi \in \mathsf{Vars}(\bullet^{s-r} \mu.\kappa) \setminus \mathsf{Raw}(\bullet^{s-r} \mu.\kappa) \\ \mathsf{Cov}[\varphi](\bullet^{s-r} \mu.\kappa) = \{\iota_1, \ldots, \iota_n\} \\ \mathsf{O} = [\iota_1 \mapsto \iota_1 \bullet] \circ \ldots \circ [\iota_n \mapsto \iota_n \bullet] \end{pmatrix}}{[\varphi \mapsto \mu.([\mathbf{0}/\varphi](\mathsf{O}(\bullet^{s-r} \mu.\kappa)))] \circ \mathsf{O} \vdash \bullet^r \varphi \leq \bullet^s \mu.\kappa} \qquad \frac{\begin{pmatrix} \varphi \in \mathsf{Vars}(\bullet^r \mu.\kappa) \setminus \mathsf{Raw}(\bullet^r \mu.\kappa) \\ \mathsf{Cov}[\varphi](\bullet^r \mu.\kappa) = \{\iota_1, \ldots, \iota_n\} \\ \mathsf{O} = [\iota_1 \mapsto \iota_1 \bullet] \circ \ldots \circ [\iota_n \mapsto \iota_n \bullet] \end{pmatrix}}{[\varphi \mapsto \mu.([\mathbf{0}/\varphi](\mathsf{O}(\bullet^r \mu.\kappa)))] \circ \mathsf{O} \vdash \bullet^r \mu.\kappa \leq \bullet^s \varphi}$$

Unifying Function Types/Recursive References (Structural Rules)

$$\frac{\mathsf{O}_1 \vdash \kappa_1' \leq \kappa_1 \quad \mathsf{O}_2 \vdash \mathsf{O}_1(\kappa_2) \leq \mathsf{O}_1(\kappa_2')}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \kappa_1 \to \kappa_2 \leq \kappa_1' \to \kappa_2'} \qquad \frac{\mathsf{O} \vdash \kappa_1 \leq \kappa_2 \quad (r \leq s)}{\mathsf{O} \vdash \bullet^r \mu.\kappa_1 \leq \bullet^s \mu.\kappa_2} \qquad \frac{(s < r)}{[\mathsf{Tail}(\mu.\kappa_2) \mapsto \top] \vdash \bullet^r \mu.\kappa_1 \leq \bullet^s \mu.\kappa_2} \qquad \frac{(r \leq s)}{\mathsf{Id} \vdash \bullet^r \mathbf{n} \leq \bullet^s \mathbf{n}}$$

$$\frac{\mathsf{O} \vdash \kappa_1 \to \kappa_2 \leq \mathsf{iPush}[\vec{\iota}](\mathsf{bPush}[s]([\mathbf{0} \mapsto \mu.\kappa](\kappa)))}{\mathsf{O} \vdash \kappa_1 \to \kappa_2 \leq \vec{\iota}\bullet^s \mu.\kappa} \qquad \frac{\mathsf{O} \vdash \mathsf{iPush}[\vec{\iota}](\mathsf{bPush}[r]([\mathbf{0} \mapsto \mu.\kappa](\kappa))) \leq \kappa_1 \to \kappa_2}{\mathsf{O} \vdash \vec{\iota}\bullet^r \mu.\kappa \leq \kappa_1 \to \kappa_2}$$

General Logical Rules (where ξ_1 and ξ_2 are not both type variables)

$$\frac{\mathsf{O} \vdash \vec{\iota}_n \alpha_1 \leq \vec{\iota'}_m \alpha_2}{\mathsf{O} \vdash \iota \vec{\iota}_n \alpha_1 \leq \iota \vec{\iota'}_m \alpha_2} \ (n, m > 0) \qquad \frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\vec{\iota}_n \bullet^r \xi_1) \leq \mathsf{O}_1(\vec{\iota'}_m \bullet^s \xi_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \vec{\iota}_n \bullet^r \xi_1 \leq \iota' \vec{\iota'}_m \bullet^s \xi_2} \begin{pmatrix} \iota \neq \iota' \text{ and either } (r \leq s, n > 0) \\ \text{or } (s < r, m > 0) \\ \mathsf{O}_1 = [\iota \mapsto \iota'] \end{pmatrix}$$

$$\frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\xi_1) \leq \mathsf{O}_1(\xi_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \bullet^r \xi_1 \leq \vec{\iota}\bullet^s \xi_2} \begin{pmatrix} \iota \notin \vec{\iota}, r \leq s \\ \mathsf{O}_1 = [\iota \mapsto \vec{\iota}\bullet^{s-r}] \end{pmatrix} \qquad \frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\xi_1) \leq \mathsf{O}_1(\xi_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \vec{\iota}\bullet^r \xi_1 \leq \iota \bullet^s \xi_2} \begin{pmatrix} \iota \notin \vec{\iota}, s < r \\ \mathsf{O}_1 = [\iota \mapsto \vec{\iota}\bullet^{r-s}] \end{pmatrix}$$

$$\frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\bullet^r \xi_1) \leq \mathsf{O}_1(\vec{\iota}\bullet^s \xi_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \bullet^r \xi_1 \leq \vec{\iota}\bullet^s \xi_2} \begin{pmatrix} \iota \in \vec{\iota}, r \leq s \\ \mathsf{O}_1 = [\iota \mapsto \epsilon] \end{pmatrix} \qquad \frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\vec{\iota}\bullet^r \xi_1) \leq \mathsf{O}_1(\bullet^s \xi_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \vec{\iota}\bullet^r \xi_1 \leq \iota \bullet^s \xi_2} \begin{pmatrix} \iota \in \vec{\iota}, s < r \\ \mathsf{O}_1 = [\iota \mapsto \epsilon] \end{pmatrix}$$

$$\frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\vec{\iota}_n \bullet^r \xi_1) \leq \mathsf{O}_1(\bullet^s \xi_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \iota \vec{\iota}_n \bullet^r \xi_1 \leq \bullet^s \xi_2} \begin{pmatrix} n > 0 \text{ or } s < r \\ \mathsf{O}_1 = [\iota \mapsto \epsilon] \end{pmatrix} \qquad \frac{\mathsf{O}_2 \vdash \mathsf{O}_1(\bullet^r \xi_1) \leq \mathsf{O}_1(\vec{\iota}_m \bullet^s \xi_2)}{\mathsf{O}_2 \circ \mathsf{O}_1 \vdash \bullet^r \xi_1 \leq \iota \vec{\iota}_m \bullet^s \xi_2} \begin{pmatrix} m > 0 \text{ or } r \leq s \\ \mathsf{O}_1 = [\iota \mapsto \epsilon] \end{pmatrix}$$

Figure 4. The Rules of the Unification Inference System

$$\dfrac{\dfrac{\dfrac{\vdots}{\mathsf{O}_2^? \vdash \mu.((\bullet\,\mathbf{0} \to \bullet\,\varphi') \to \varphi') \leq \bullet\,\mu.((\bullet\,\mathbf{0} \to \varphi) \to \varphi) \to \varphi} \quad \dfrac{\vdots}{\mathsf{O}_3^? \vdash \mathsf{O}_2^?(\varphi) \leq \mathsf{O}_2^?(\varphi')}}{\mathsf{O}_1^? = \mathsf{O}_3^? \circ \mathsf{O}_2^? \vdash (\bullet\,\mu.((\bullet\,\mathbf{0} \to \varphi) \to \varphi) \to \varphi) \to \varphi \leq \mu.((\bullet\,\mathbf{0} \to \bullet\,\varphi') \to \varphi') \to \varphi'}\ \text{unify components of function types}}{\mathsf{O}_1^? \vdash \mu.((\bullet\,\mathbf{0} \to \varphi) \to \varphi) \leq \mu.((\bullet\,\mathbf{0} \to \bullet\,\varphi') \to \varphi') \to \varphi'}\ \text{unfold left}$$

Figure 5. Initial Steps of a Non-terminating Naïve Proof Search for Unification

to unify the logical structure of the two types, as encoded in the $\bullet$ modalities. Notice how an insertion variable may either be substituted for some sequence of other insertion variables and $\bullet$ modalities, or may be removed via an insertion $[\iota \mapsto \epsilon]$. The latter takes place when we try to unify and insertion variable ι with a sequence of insertion variables in which it occurs; there is no insertion that can solve this – the occurs check moves from type variables to insertion variables. Ultimately, the goal of the logical rules is to unify as much logical information before applying a structural rule.

The structural rules compare the functional shapes of types to make sure that they are compatible. This process involves unfolding recursive definitions at particular points, namely when we are comparing a top-level function type $\sigma \to \tau$ with a top-level recursive definition $\mu.\tau'$. As expected, when comparing two top-level function types, the domains (left-hand sides) of the two types are unified, followed by unification of the ranges (right-hand sides). Since we are unifying modulo subtyping, however, domains of function types are unified *contra-variantly*. The structural base cases of the procedure are when we unify a type variable φ with another type τ. In this case, a substitution of φ for suitable type σ is generated (e.g. when the φ variable occurs in τ, a recursive type is constructed as a solution). At this point, the structural rules also check any logical constraints represented by the $\bullet$ modalities that remain, which essentially amounts to ensuring that the types in the statement $\sigma \leq \tau$ can be made equivalent to $\bullet^r\,\sigma$ and $\bullet^s\,\tau$ where $r \leq s$. If this is not possible, the procedure may produce a substitution that makes the right-hand type equivalent to $\top$.

We point to an important feature of our approach: that of how two top-level recursive definitions $\mu.\tau$ and $\mu.\sigma$ are unified. This is achieved, not by unfolding the two definitions, but by *removing* the μ-binder and unifying the two bodies of the definitions, τ and σ. This approach exactly mirrors that given by Cardone and Coppo for deciding equality between recursive types [8]. The notion of equality that this approach characterises, however, is *weak* (i.e. equality up to finite unfoldings of definitions). Thus, our system may fail to unify some types which *can* be made strongly equivalent. A simple example of this is the problem of unifying $\mu.\bullet\,\mathbf{0} \to \varphi$ with $\mu.\bullet\,\mathbf{0} \to \varphi' \to \varphi'$. We will return to this point in our conclusions.

The unification inference system is *sound*, however, as shown by the following result.

Lemma 5.11 (Soundness of Unification)**.** *If* $\mathsf{O} \vdash \sigma \leq \tau$*, then* O *is an operation and* $\mathsf{O}(\sigma) \leq \mathsf{O}(\tau)$.

Proof. By induction on the structure of the unification inference derivations using the soundness of operations with respect to subtyping (Lemma 5.5). In the base cases where a substitution of type variable for a new recursive type is generated, we use Proposition 5.9. □

Termination

In order to show that the inference system of Figure 4 gives an algorithm, we must show that the proof search procedure terminates. This would not be the case if we naïvely implemented such a procedure since it involves unfolding recursive definitions, thus recursing on larger subproblems. Consider the unification problem shown in Figure 5, for which the first few steps are given. Incidentally, this is another example of two types that *can* be unified such that they are strongly equivalent, but not such that they are weakly so. Notice that after two steps, we are faced with a proof subgoal which has the same structure as the original goal (modulo occurrences of $\bullet$). A naive algorithm would repeat these two steps ad infinitum. In fact, no proof exists of the validity of the desired unification judgement, and this is the source of the non-terminating behaviour.

To obtain a terminating algorithm, we show that when a derivation proving the validity of a unification judgement exists, its height has a well-defined bound. This allows a decreasing measure to be incorporated into the proof search algorithm, thus ensuring termination. The fact that the height of derivations is bounded further implies that the proof search algorithm is complete with respect to the inference system. Our technique is a direct extension of that used by Brandt and Henglein [6], and later by Cardone and Coppo [8]. Fundamentally, it is based upon the fact that a regular infinite tree (type) only has a finite number of distinct subtrees, and thus the amount of information contained in a recursive type definition is *finite*. Equivalently, we can observe that when unfolding recursive types we will only even encounter a finite (and bounded) number of subcomponents. This finite set is encapsulated in the notion of *subterm closure* in [6, 8].

We must be careful when extending the notion of subterm closure to Nakano types, however. In general, the subterm closure of a (canonical) Nakano type is *not* finite, since $\bullet$ modalities accumulate as we unfold the recursive definitions. Luckily, though, we may ignore the logical information encoded by the modalities: the structure of a proof in the unification inference system is dictated only by the *functional shape* of types, which *is* characterised by the (finite) subterm closure.

Definition 5.12 (Structural Closure)**.** *1. We define the* structural representative $\mathsf{Struct}(\tau)$ *of a (Nakano) type* τ *by erasing all insertion variables and occurrences of the* $\bullet$ *modality. Recursive definitions of structural representatives can be obtained from recursive definitions of types as follows:*

$$\mathsf{Struct}(\varphi) = \varphi \quad \mathsf{Struct}(\mathbf{n}) = \mathbf{n}$$
$$\mathsf{Struct}(\bullet\,\tau) = \mathsf{Struct}(\iota\,\tau) = \mathsf{Struct}(\tau)$$
$$\mathsf{Struct}(\sigma \to \tau) = \mathsf{Struct}(\sigma) \to \mathsf{Struct}(\tau)$$
$$\mathsf{Struct}(\mu.\tau) = \mu.(\mathsf{Struct}(\tau))$$

2. The structural closure *of a recursive type definition is given by:*

$$\mathcal{SC}(\varphi) = \{\varphi\} \quad \mathcal{SC}(\mathbf{n}) = \{\mathbf{n}\}$$
$$\mathcal{SC}(\bullet\,\tau) = \mathcal{SC}(\iota\,\tau) = \mathcal{SC}(\tau)$$
$$\mathcal{SC}(\sigma \to \tau) = \{\mathsf{Struct}(\sigma \to \tau)\} \cup \mathcal{SC}(\sigma) \cup \mathcal{SC}(\tau)$$
$$\mathcal{SC}(\mu.\tau) = \{\mathsf{Struct}(\mu.\tau)\} \cup \mathcal{SC}(\tau) \cup \mathcal{SC}([\mathbf{0} \mapsto \mu.\tau](\tau))$$

3. The definition of structural closure is extended to sets of types T *by* $\mathcal{SC}(T) = \cup_{\tau \in T}(\mathcal{SC}(\tau))$.

It is straightforward to show that the structural closure $\mathcal{SC}(\tau)$ of τ is equal to the subterm closure (as defined in [6, 8]) of $\mathsf{Struct}(\tau)$. Thence it follows that $\mathcal{SC}(\tau)$ (and thus also $\mathcal{SC}(T)$) is finite.

We must take further care however. Our termination argument will hinge on the fact that (the structural representative of) every type occurring in a unification derivation belongs to a finitely bounded set. Since the unification procedure applies operations to types as it goes, it is not the case that every such entity will be in the structural (i.e. subterm) closure of the types in the original goal. The subterm closure suffices for deciding equality between recursive types, but for unification we must find another set. Fortunately such a set *does* exist, and we call this set the *unification* closure.

We first define a set that combines the structural closure of all the recursive types that may be generated from a given type by the unification procedure.

Definition 5.13 (Recursion Complete Structural Closure)**.** *The* recursion complete *structural closure of a type is defined by:*

$$\mathcal{SC}^*_\mu(\tau) = \mathcal{SC}(\tau) \cup \bigcup_{\substack{\sigma \in \mathcal{SC}(\tau) \\ \sigma \text{ closed}}} \left(\bigcup_{\varphi \in \mathsf{Vars}(\sigma)} \mathcal{SC}^*_\mu(\mu.([\mathbf{0}/\varphi](\sigma))) \right)$$

This definition is extended to sets of types T by $\mathcal{SC}^*_\mu(T) = \cup_{\tau \in T}(\mathcal{SC}^*_\mu(\tau))$.

Using this, we can then define the unification closure of a set of types, which takes into account all of the types which may possibly be generated during unification of the types in that set.

Definition 5.14 (Unification Closure)**.** *The* unification closure *of a set of types T is defined by:*

$$\mathcal{UC}(T) = \mathcal{SC}^*_\mu(T) \cup \bigcup_{\varphi \in \mathsf{Vars}(T)} \left(\bigcup_{\substack{\tau \in \mathcal{SC}^*_\mu(T) \\ \varphi \notin \mathsf{Vars}(\tau)}} \mathcal{UC}([\varphi \mapsto \tau](T)) \right)$$

where $\mathsf{Vars}(T) = \cup_{\tau \in T}\mathsf{Vars}(\tau)$ and $[\varphi \mapsto \tau](T)$ denotes the set obtained by applying the substitution $[\varphi \mapsto \tau]$ to each type in T.

We are interested in two key properties of the unification closure: that it is finite, and that it contains all of the structural representative of types that occur in judgements in a derivation of the unification inference system.

Lemma 5.15 (Finiteness and Adequacy of Unification Closure)**.**

1. *Let T be a finite set of types, then $\mathcal{UC}(T)$ is finite.*
2. *Let $\mathcal{D}$ be a derivation of the judgement $\mathsf{O} \vdash \sigma \leq \tau$, then all statements $\sigma' \leq \tau'$ occurring in $\mathcal{D}$ are such that both $\mathsf{Struct}(\sigma')$ and $\mathsf{Struct}(\tau')$ are in the set $\mathcal{UC}(\{\sigma, \tau\})$.*

Proof. 1. We show the property for each closure construction in turn. The finiteness of $\mathcal{SC}^*_\mu(\tau)$ follows from the finiteness of $\mathcal{SC}(\tau)$ by induction on the number of distinct type variables in τ, since the number of distinct type variables in $\mu.[\mathbf{0} \mapsto \mu.\varphi](\tau)$ is strictly less than in τ. Finiteness of $\mathcal{SC}^*_\mu(T)$ for finite sets T then follows easily by induction on the size of T. Finally, finiteness of $\mathcal{UC}$ follows from the finiteness of $\mathcal{SC}^*_\mu(T)$ by induction on the number of distinct type variables in T

2. By straightforward induction on the structure of the derivation. □

These properties allow us to show that the height of any derivation of a unification judgement $\mathsf{O} \vdash \sigma \leq \tau$ is finitely bounded, and thus termination of the unification procedure.

Theorem 5.16. *Let $\mathcal{D}$ be a derivation of the judgement $\mathsf{O} \vdash \sigma \leq \tau$, then the height of $\mathcal{D}$ is no greater than $|\mathcal{UC}(\{\sigma, \tau\})|^2$.*

Proof. We define the height of $\mathcal{D}$ as the maximum number of structural rules along any path in $\mathcal{D}$. The proof then proceeds by contradiction. Assume $\mathcal{D}$ has a height $h > |\mathcal{UC}(\{\sigma, \tau\})|^2$. Then there exist derivations $\mathcal{D}_1, \ldots, \mathcal{D}_h$ such that each $\mathcal{D}_i$ is a subderivation of $\mathcal{D}$, and for each $1 \leq i \neq j \leq h$, the heights of derivations $\mathcal{D}_i$ and $\mathcal{D}_j$ are different, specifically $h_i = h_{i+1} + 1$ where h_i and h_{i+1} are the heights of $\mathcal{D}_i$ and $\mathcal{D}_{i+1}$ respectively, for each $1 \leq i < h$. Also, there is a set of pairs of types $\{(\sigma_1, \tau_1), \ldots, (\sigma_h, \tau_h)\}$ which are the types in the concluding judgements of each of the derivations $\mathcal{D}_1, \ldots, \mathcal{D}_h$.

Since the structure of derivations is syntax-directed, we can show by straightforward induction on derivations that if σ, σ', τ and τ' are types such that $\mathsf{Struct}(\sigma) = \mathsf{Struct}(\sigma')$ and $\mathsf{Struct}(\tau) = \mathsf{Struct}(\tau')$, and $\mathcal{D}$ and $\mathcal{D}'$ are derivations of $\mathsf{O} \vdash \sigma \leq \tau$ and $\mathsf{O}' \vdash \sigma' \leq \tau'$ respectively, then the heights of $\mathcal{D}$ and $\mathcal{D}'$ must be equal.

From Lemma 5.15 we know that both $\mathsf{Struct}(\sigma_i)$ and $\mathsf{Struct}(\tau_i)$ are in $\mathcal{UC}(\{\sigma, \tau\})$ for every $1 \leq i \leq h$. Since the number of distinct pairs (σ', τ') such that both $\mathsf{Struct}(\sigma')$ and $\mathsf{Struct}(\tau')$ are in $\mathcal{UC}(\{\sigma, \tau\})$ is $|\mathcal{UC}(\{\sigma, \tau\})|^2 < h$, it must be that there are two distinct $j, k \leq h$ such that $\mathsf{Struct}(\sigma_j) = \mathsf{Struct}(\sigma_k)$ and $\mathsf{Struct}(\tau_j) = \mathsf{Struct}(\tau_k)$. Thus, by the auxiliary lemma stated above, it must be that the heights of $\mathcal{D}_j$ and $\mathcal{D}_k$ are the same. However, this contradicts our earlier deduction that their heights are different. Therefore, our original assumption must have been false and the height of $\mathcal{D}$ cannot exceed $|\mathcal{UC}(\{\sigma, \tau\})|^2$. □

This argument is analogous to the one used in [8].

6. The Type Inference Procedure

Having defined a type unification procedure, we can now present our type inference procedure Type. As expected, it takes a term M, and returns a pair consisting of a type environment Γ and a type τ such that τ can be assigned to M using the environment Γ. Its definition is almost identical to the standard type inference procedure for simply typed lambda calculus, the only difference being a subtlety in the case for typing an application in which we must introduce insertion variables. This case of course also makes use of the unification procedure, which is used to both a) unify the type of the operator with a type constructed using that of the operand, and b) unify the type environments inferred for each of the components of the application. Before defining the type inference procedure itself, we therefore extend the notion of unification to type environments. We will also switch from expressing unifiability of types using judgements to writing $\mathsf{Unify}^\mu_\leq(\sigma, \tau) = \mathsf{O}$, in order to stress that unification is a *procedure*.

Definition 6.1 (Unification of Type Environments)**.** *The unification procedure is extended to type environments as follows, where we write $\emptyset$ for the environment that is undefined on all variables, and $\Gamma \setminus x$ for the environment that is defined exactly like Γ, except on x where it is undefined:*

$$\mathsf{Unify}^\mu_\leq(\emptyset, \Gamma) = \mathsf{Id}$$

$$\mathsf{Unify}^\mu_\leq((\Gamma, x{:}\sigma), (\Gamma', x{:}\tau)) = \mathsf{O}_2 \circ \mathsf{O}_1 \quad \textit{if } \mathsf{Unify}^\mu_\leq(\sigma, \tau) = \mathsf{O}_1 \textit{ and } \mathsf{Unify}^\mu_\leq(\Gamma \setminus x, \Gamma' \setminus x) = \mathsf{O}_2$$

$$\mathsf{Unify}^\mu_\leq((\Gamma, x{:}\sigma), (\Gamma', x{:}\tau)) = \mathsf{O}_2 \circ \mathsf{O}_1$$

$$\textit{if } \mathsf{Unify}^{\mu}_{\leq}(\sigma,\tau) \textit{ fails}$$
$$\textit{and } \mathsf{Unify}^{\mu}_{\leq}(\tau,\sigma) = \mathsf{O}_1$$
$$\textit{and } \mathsf{Unify}^{\mu}_{\leq}(\Gamma \setminus x, \Gamma' \setminus x) = \mathsf{O}_2$$

$$\mathsf{Unify}^{\mu}_{\leq}((\Gamma, x{:}\sigma), \Gamma') = \mathsf{Unify}^{\mu}_{\leq}(\Gamma \setminus x, \Gamma')$$
$$\textit{if } x \notin \mathsf{dom}(\Gamma')$$

Since unification (when it succeeds) does not necessarily make types equivalent, there is a small subtlety involved in combining two environments that have been unified. When doing so, we will need to pick out the more specific type for each term variable whenever there is a choice; if the two types do happen to be equivalent, we may pick the 'simplest' one.

Definition 6.2 (Combining Environments)**.** *Let the* size *of a type $|\tau|$ be some suitable measure of its complexity (e.g. the maximum number of nested μ-binders). We define a* combining *operation $\uplus$ on type environments by $(\Gamma_1 \uplus \Gamma_2)(x) = \tau$ if and only if:*

1. $\Gamma_1(x) = \tau$ & $x \notin \mathsf{dom}(\Gamma_2)$*; or*
2. $\Gamma_2(x) = \tau$ & $x \notin \mathsf{dom}(\Gamma_1)$*; or*
3. $\Gamma_1(x) = \tau$ & $\Gamma_2(x) = \sigma$ & $\tau \leq \sigma$ & $\sigma \not\leq \tau$*; or*
4. $\Gamma_2(x) = \tau$ & $\Gamma_1(x) = \sigma$ & $\tau \leq \sigma$ & $\sigma \not\leq \tau$*; or*
5. $\Gamma_1(x) = \tau$ & $\Gamma_2(x) = \sigma$ & $\tau \simeq \sigma$ & $|\tau| \leq |\sigma|$*; or*
6. $\Gamma_2(x) = \tau$ & $\Gamma_1(x) = \sigma$ & $\tau \simeq \sigma$ & $|\tau| < |\sigma|$*;*

The following property, that combining unified environments produces a more specific environment, is needed to show that the type inference procedure we define below is sound.

Lemma 6.3. *Let Γ_1 and Γ_2 be type environments such that $\mathsf{Unify}^{\mu}_{\leq}(\Gamma_1, \Gamma_2) = \mathsf{O}$, for some operation O; then it holds that both $(\mathsf{O}(\Gamma_1) \uplus \mathsf{O}(\Gamma_2)) \leq \Gamma_1$ and $(\mathsf{O}(\Gamma_1) \uplus \mathsf{O}(\Gamma_2)) \leq \Gamma_2$.*

We can now define the type inference procedure.

Definition 6.4 (Type Inference Procedure)**.** *The procedure* Type *is defined as follows:*

$$\mathsf{Type}(x) = \langle\, \{x{:}\varphi\}, \varphi \,\rangle \qquad \textit{where } \varphi \textit{ fresh}$$

$$\mathsf{Type}(\lambda x.M) = \begin{cases} \langle\, \Gamma \setminus x, \sigma \to \tau \,\rangle \\ \quad \textit{if } \mathsf{Type}(M) = \langle\, (\Gamma, x{:}\sigma), \tau \,\rangle \\ \langle\, \Gamma, \top \to \tau \,\rangle \\ \quad \textit{if } \mathsf{Type}(M) = \langle\, \Gamma, \tau \,\rangle \textit{ and } x \notin \mathsf{dom}(\Gamma) \end{cases}$$

$$\mathsf{Type}(MN) = \langle\, \mathsf{iPush}[\iota_2](\Gamma_1') \uplus \Gamma_2',\ \mathsf{iPush}[\iota_2](\mathsf{O}(\varphi)) \,\rangle$$

$$\begin{array}{l} \textit{if } \mathsf{Type}(M) = \langle\, \Gamma_1, \sigma \,\rangle \\ \quad \mathsf{Type}(N) = \langle\, \Gamma_2, \tau \,\rangle \\ \quad \mathsf{Unify}^{\mu}_{\leq}(\sigma, \mathsf{iPush}[\iota_1](\tau) \to \varphi) = \mathsf{O}_1 \\ \quad \mathsf{Unify}^{\mu}_{\leq}(\mathsf{O}_1(\Gamma_1), \mathsf{O}_1(\Gamma_2)) = \mathsf{O}_2 \\ \textit{where } \varphi, \iota_1, \iota_2 \textit{ fresh} \\ \quad \mathsf{O} = \mathsf{O}_2 \circ \mathsf{O}_1 \\ \quad \Gamma_1' = \{x{:}\sigma \mid x \in \Gamma_1 \,\&\, (\mathsf{O}(\Gamma_1) \uplus \mathsf{O}(\Gamma_2))(x) = \sigma\} \\ \quad \Gamma_2' = \{y{:}\tau \mid y \notin \Gamma_1 \,\&\, (\mathsf{O}(\Gamma_1) \uplus \mathsf{O}(\Gamma_2))(y) = \tau\} \end{array}$$

The main result of this section is that our type inference procedure is *sound*.

Theorem 6.5 (Soundness of Type Inference)**.** *If $\mathsf{Type}(M) = \langle\, \Gamma, \tau \,\rangle$ then $\Gamma \vdash M : \tau$.*

Proof. By induction on the structure of terms. The cases when the term is a variable or an abstraction are trivial. We discuss the case for an application. By the inductive hypothesis, we have typeability of the subcomponents, i.e. $\Gamma_1 \vdash M : \sigma$ and $\Gamma_2 \vdash N : \tau$. The soundness of unification (Lemma 5.11) gives $\mathsf{O}_1(\sigma) \leq \mathsf{O}_1(\mathsf{iPush}[\iota_1](\tau) \to \varphi)$. Thus by soundness of operations (Lemma 5.5), subtyping and weakening (Lemmas 6.3 and 4.8) it follows that $\mathsf{O}(\Gamma_1) \uplus \mathsf{O}(\Gamma_2) \vdash M : \mathsf{O}(\mathsf{iPush}[\iota_1](\tau)) \to \mathsf{O}(\varphi)$. Notice that $\tau \leq \iota_1\, \tau \simeq \mathsf{iPush}[\iota_1](\tau)$. Thus by soundness of operations, subtyping and weakening again, we have $\mathsf{O}(\Gamma_1) \uplus \mathsf{O}(\Gamma_2) \vdash N : \mathsf{O}(\mathsf{iPush}[\iota_1](\tau))$. Then by the typing rule for applications it follows that $\mathsf{O}(\Gamma_1) \uplus \mathsf{O}(\Gamma_2) \vdash MN : \mathsf{O}(\varphi)$. Notice that $\mathsf{O}(\Gamma_1) \uplus \mathsf{O}(\Gamma_2) = \Gamma_1' \cup \Gamma_2'$, so by the degradation property (Lemma 4.9) we have $(\iota_2\, \Gamma_1') \cup \Gamma_2' \vdash MN : \iota_2(\mathsf{O}(\varphi))$. Notice also that $\mathsf{iPush}[\iota_2](\Gamma_1') \simeq (\iota_2\, \Gamma_1')$ and $\mathsf{iPush}[\iota_2](\mathsf{O}(\varphi)) \simeq \iota_2(\mathsf{O}(\varphi))$, and so the final result follows again by weakening and subtyping. □

We do not have a completeness result for our procedure (i.e. if a term can be assigned a type, then our procedure also infers one), and we discuss this in more detail below. However, we believe that our algorithm can infer types for a large class of terms, at the same time inferring types which are, in a sense, most general. This we also discuss below.

6.1 Some Examples of Inferred Types

We will now demonstrate how the type inference procedure works by considering some examples. Firstly we will revisit the typing of Curry's fixed point combinator, showing how the type inference procedure defined above produces a typings with insertion variables at the appropriate place as discussed in Section 3. We will then consider the type that the procedure infers for the familiar λ-term $\mathbf{S} = \lambda xyz.xz(yz)$. As we proceed, we will also discuss the generality of the types that are inferred.

Curry's Fixed Point Combinator

To demonstrate how our algorithm infers a type for $\mathbf{Y}$, we will proceed as in Section 3 from the bottom up. The reader is encourage to compare the presentation here with the one given previously. Recall the definition of the fixed point combinator:

$$\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Similar to the standard type inference algorithm for simply typed lambda calculus, our procedure infers vanilla typings (i.e. without any insertion variables) for term variables. Thus we have typings $\langle\, \Gamma_1 = \{x{:}\varphi_1\}, \sigma = \varphi_1 \,\rangle$, $\langle\, \Gamma_2 = \{x{:}\varphi_2\}, \tau = \varphi_2 \,\rangle$ for the two components of the self-application xx. The inference of a type for the application itself, however, proceeds differently. Instead of unifying φ_1 with $\varphi_2 \to \varphi_3$, our algorithm solves the following problem:

$$\mathsf{Unify}^{\mu}_{\leq} \quad \varphi_1\ ,\ \iota_1\, \varphi_2 \to \varphi_3 \quad (\iota_1, \varphi_3 \text{ fresh})$$

producing a straightforward substitution $[\varphi_1 \mapsto \iota_1\, \varphi_2 \to \varphi_3]$. This substitution is then applied to the type environments inferred for the subcomponents, and these are then unified. This results in the following call.

$$\mathsf{Unify}^{\mu}_{\leq} \quad \iota_1\, \varphi_2 \to \varphi_3\ ,\ \varphi_2$$

Note that φ_2 is not under the scope of a $\bullet$ in the left-hand type, but it *is* under the scope of an insertion variable (the relevant cover set is $\{\iota_1\}$). Thus, we produce an insertion $[\iota_1 \mapsto \iota_1 \bullet]$ to ensure that the ensuing recursive type we construct is proper, and generate the substitution $[\varphi_2 \mapsto \mu.\iota_1 \bullet \mathbf{0} \to \varphi_3]$. Applying the substitutions and insertion generated so far to the type environments, we get

$$\mathsf{O}(\Gamma_1) = \{x{:}\iota_1 \bullet \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3) \to \varphi_3\}$$
$$\mathsf{O}(\Gamma_2) = \{x{:}\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)\}$$

where

$$\mathsf{O} = [\varphi_2 \mapsto \mu.\iota_1 \bullet \mathbf{0} \to \varphi_3] \circ [\iota_1 \mapsto \iota_1 \bullet] \circ [\varphi_1 \mapsto \iota_1\, \varphi_2 \to \varphi_3]$$

Notice that the two types for x are *equivalent* ($\simeq$), which was not the case for the type inference procedure without insertion variables. When combining the environments, we can then pick the simpler one, to get $\mathsf{O}(\Gamma_1) \uplus \mathsf{O}(\Gamma_2) = \{x{:}\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)\}$.

We now split the environment into two disjoint environments according to the which variables occurred in each of the two components of the application, and add an extra insertion variable to the types of those variables from the left-hand component. Since, in this case there are no variables from the right-hand component which do not also occur in the left-hand one, our Γ_1' and Γ_2' from the definition of the type inference algorithm (Def. 6.4) are $\{x{:}\mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)\}$ and $\emptyset$ respectively. Thus, we obtain the following typing for xx: $\langle\, \{x{:}\, \iota_2\, \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)\}, \iota_2\, \varphi_3 \,\rangle$.

A similar procedure then takes place to infer a typing for the term $f(xx)$, and the reader can easily verify that the typing generated for $\lambda x.f(xx)$ is the following one:

$$\langle\, f{:}\iota_4\, \iota_3\, \iota_2\, \varphi_3 \to \iota_4\, \varphi_5,\ \iota_2\, \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3) \to \iota_4\, \varphi_5 \,\rangle$$

Notice that both of the typings we gave in Section 3 can be generated from this one by different combinations of insertions. For example, the operation that sends each of $\iota_1, \ldots, \iota_5$ to ϵ will produce the original typing we considered, and the operation that sends $\iota_1, \iota_3, \ldots, \iota_5$ to ϵ and ι_2 to $\bullet$ will yield the alternative typing. We conjecture that *any and all* the typings of this term in Nakano's original system can be generated from the one returned by our algorithm using type operations and weakening. This would mean that our algorithm infers *principal* typings, although since our algorithm only unifies up to weak equality it seems clear that typings would only be principal up to weak equality.

At this point we arrive to where we ran into trouble previously; remember that type inference without insertion variables failed because we could not unify the types inferred for the two occurrences of the subterm $\lambda x.f(xx)$. Having inferred a typing with insertion variables, however, the unification succeeds. Taking a fresh instance for the right-hand occurrence of the subterm, we obtain:

$$\langle\, \Gamma_1, \sigma \,\rangle = \langle\, f{:}\iota_4\, \iota_3\, \iota_2\, \varphi_3 \to \iota_4\, \varphi_5,\ \iota_2\, \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3) \to \iota_4\, \varphi_5 \,\rangle$$

$$\langle\, \Gamma_2, \tau \,\rangle = \langle\, f{:}\iota_8\, \iota_7\, \iota_6\, \varphi_8 \to \iota_8\, \varphi_{10},\ \iota_6\, \mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8) \to \iota_8\, \varphi_{10} \,\rangle$$

The unification then proceeds as follows. We show the steps up to the point at which the approach without insertion variables fails. This step is now easily handled because the insertion variable ι_6 prefixing the right-hand recursive type is able to 'consume' the $\bullet$ prefixing the left-hand recursive type:

$$\begin{aligned} &\mathsf{Unify}^{\mu}_{\leq}\ \sigma\ ,\ \mathsf{iPush}[\iota_9](\tau) \to \varphi_{11} \\ =\ &\mathsf{Unify}^{\mu}_{\leq}\ \iota_2\, \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3) \to \iota_4\, \varphi_5,\ (\iota_9\, \iota_6\, \mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8) \to \iota_9\, \iota_8\, \varphi_{10}) \to \varphi_{11} \end{aligned}$$

As before, we contra-variantly unify the domains of the types:

$$\mathsf{Unify}^{\mu}_{\leq}\ \iota_9\, \iota_6\, \mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8) \to \iota_9\, \iota_8\, \varphi_{10},\ \iota_2\, \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)$$

which requires us to unfold the right-hand type:

$$\mathsf{Unify}^{\mu}_{\leq}\ \iota_9\, \iota_6\, \mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8) \to \iota_9\, \iota_8\, \varphi_{10},\ \iota_2\, \iota_1 \bullet \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3) \to \iota_2\, \varphi_3$$

Again, we must contra-variantly unify the domains:

$$\mathsf{Unify}^{\mu}_{\leq}\ \iota_2\, \iota_1 \bullet \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)\ ,\ \iota_9\, \iota_6\, \mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8)$$

We then pair off the insertion variables ι_2 and ι_9 at the head of each type (an instance of a logical rule), and continue:

$$\mathsf{Unify}^{\mu}_{\leq}\ \iota_1 \bullet \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)\ ,\ \iota_6\, \mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8)$$

This is now the point at which type inference without insertion variables gets stuck. Here, however, notice that we can unify the insertion variable ι_6 with the sequence $\iota_1 \bullet$ that prefixes the left-hand type and continue:

$$\mathsf{Unify}^{\mu}_{\leq}\ \mu.(\iota_1 \bullet \mathbf{0} \to \varphi_3)\ ,\ \mu.(\iota_5 \bullet \mathbf{0} \to \varphi_8)$$

We do not show the remaining steps of the unification, however the reader may verify that it succeeds and results in the following type for the $\mathbf{Y}$ combinator:

$$(\iota\, \iota'\, \iota'' \bullet \varphi \to \iota\, \varphi) \to \iota\, \varphi$$

The S Combinator

As a further example. consider the combinator $\mathbf{S} = \lambda xyz.xz(yz)$. The reader may like to verify as an exercise that our algorithm produces the following type for this term:

$$(\iota_5\, \iota_1\, \varphi_1 \to \iota_5\, \iota_4\, \iota_3\, \varphi_2 \to \iota_5\, \varphi_3) \to (\iota_3\, \iota_2\, \varphi_1 \to \iota_3\, \varphi_2) \to \iota_5\, \varphi_1 \to \iota_5\, \varphi_3$$

Notice, again, that by sending each insertion variable $\iota_1, \ldots, \iota_5$ to ϵ we obtain the standard Curry (principal) type for $\mathbf{S}$. We can generate other types for $\mathbf{S}$ by applying simple insertions that send each insertion variable to ϵ or $\bullet$. Each the following types are obtainable in this way:

$$\begin{aligned} &(\bullet\, \varphi_1 \to \varphi_2 \to \varphi_3) \to (\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_3 \\ &(\varphi_1 \to \bullet\, \varphi_2 \to \varphi_3) \to (\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_3 \\ &(\varphi_1 \to \bullet\, \varphi_2 \to \varphi_3) \to (\bullet\, \varphi_1 \to \bullet\, \varphi_2) \to \varphi_1 \to \varphi_3 \\ &(\bullet\, \varphi_1 \to \bullet\, \varphi_2 \to \bullet\, \varphi_3) \to (\varphi_1 \to \varphi_2) \to \bullet\, \varphi_1 \to \bullet\, \varphi_3 \end{aligned}$$

This result further bolsters our principality conjecture. It is easily verified that the four types we have just given, along with the familiar Curry type, are *not* related to one another via subtyping, i.e. they reside in the same generation in the subtyping partial order. Neither can any of them be obtained from the Curry type via substitution. However, they are all obtainable from the type returned by our inference procedure via operations (insertions) on that type!

6.2 Some Comments on Completeness

We have pointed out that because our unification procedure extends the variant of equality-checking technique that tests for weak equality (i.e. up to finite unfoldings) of recursive definitions, rather than strong equality. This means that there are types, that are unifiable under a strong notion of equality, which our procedure cannot unify. We have given examples of two such pairs of types in Section 5.4.

It is unclear to us, at this time, whether and exactly how our technique may be extended to such a strong notion of unification. In the algorithm for checking strong equality of type definitions, proof goals (in the form of statements of equality between function types) are collected as the procedure progresses; then if and when the same proof goal is encountered again, the equality may be assumed as an axiom. Using this approach does not seem possible for unification, which must produce a substitution: since we do not what the correct substitution should be on first encountering the subgoal, what should be returned when the subgoal is encountered for the second time?

Notwithstanding, it remains to be seen whether this is in fact a significant problem for type inference. Is it the case that our algorithm, when trying to type an application, would actually produce two types which could only be unified such that they are strongly equivalent but not weakly so? We have not been able to think of an example demonstrating this up till now—such an example, if it exists, would certainly be interesting to consider.

More of a problem for completeness is the fact that we do not have a rule in our system that unifies a $\bullet$-prefixed type variable with a function type (i.e. $\mathsf{O}_? \vdash \bullet\varphi \leq \sigma \to \tau$). There are simple examples that this is incomplete. Take $\bullet\varphi$ and $\bullet\varphi' \to \bullet\varphi'$, which are clearly unifiable with the substitution $[\varphi \mapsto \varphi' \to \varphi']$. The fact that our procedure does not do this is an artefact of the definition of canonical form that we have used: we must push $\bullet$ modalities innermost. A straightforward example of where this causes type inference to fail is with the term $\mathbf{Y}(\lambda xy.y(xy))$ (also a fixed point combinator). Perhaps this can be overcome by some 'pulling' operation, that can factorise occurrences of $\bullet$ (and insertion variables) out of a function type. This is a question for future research.

7. Conclusions and Future Work

We have presented an extension of Nakano's original system of guarded recursive types that consists of insertion variables, a kind of type constructor. On an operational level, these insertion variables serve to allow occurrences of the $\bullet$ modality to be introduced at specific points within types. We have extended the type assignment system so that types with insertion variables can be assigned to terms. When such a type can be assigned to a term, it is *sound* to introduce occurrences of $\bullet$ at those locations in the type marked by an insertion variable. This means that they can actually represent families of guarded types that may be assigned to term. Thus, insertion variables are *more* than just an operational device; they constitute an added level of abstraction on top of the raw concept of the $\bullet$ modality.

We have also described a method for unifying two types in our extended system, modulo the subtyping relation, and we have used this unification procedure to define a type inference algorithm. In order to develop the unification procedure, we built upon and extended techniques which have previously only been used to decide equality between recursive type definitions. This involved constructing a generalisation of the notion of subterm closure which is appropriate for unification. We showed our type inference algorithm to be sound, and we demonstrated its operation using typical and illuminating examples.

There are many potential avenues for future work. Regarding the system that we have presented, it would be interesting to investigate to what extent principality of typings holds, and whether our inference algorithm does in fact construct them. There is also the question of extending the notion of unification to gain completeness. Beyond that, it would also be elucidating to research the underlying semantic model of insertion variables, and discover whether there really is a deeper connection with expansion variables. In terms of applying our results, the obvious question is whether our techniques can be applied to those systems already defined that make use of Nakano modalities. Beyond that, an application mentioned by Nakano himself is the object-oriented paradigm. Recursive types are the natural descriptions of objects, and so it seems likely that applying guarded recursion in this setting would bring great rewards.

References

[1] R. M. Amadio and L. Cardelli. Subtyping Recursive Types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993.

[2] A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A Very Modal Model of a Modern, Major, General Type System. In *POPL*, pages 109–122, 2007.

[3] R. Atkey and C. McBride. Productive Coprogramming with Guarded Recursion. In *ICFP*, pages 197–208, 2013.

[4] G. Barthe, B. Grégoire, and C. Riba. Type-Based Termination with Sized Products. In *CSL*, pages 493–507, 2008.

[5] L. Birkedal, R. E. Møgelberg, J. Schwinghammer, and K. Støvring. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. *Logical Methods in Computer Science*, 8(4), 2012.

[6] M. Brandt and F. Henglein. Coinductive Axiomatization of Recursive Type Equality and Subtyping. *Fundam. Inform.*, 33(4):309–338, 1998.

[7] F. Cardone and M. Coppo. Type Inference with Recursive Types: Syntax and Semantics. *Information and Computation*, 92(1):48–80, 1991.

[8] F. Cardone and M. Coppo. Decidability Properties of Recursive Types. In *ICTCS*, pages 242–255, 2003.

[9] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[10] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.

[11] J. Endrullis, C. Grabmayer, J. W. Klop, and V. van Oostrom. On Equal μ-terms. *Theor. Comput. Sci.*, 412(28):3175–3202, 2011.

[12] V. Gapeyev, M. Y. Levin, and B. C. Pierce. Recursive Subtyping Revealed. *J. Funct. Program.*, 12(6):511–548, 2002.

[13] E. Giménez. Structural Recursive Definitions in Type Theory. In *ICALP*, pages 397–408, 1998.

[14] J. Y. Girard. *Interprtation fonctionnelle et limination des coupures de l'arithmtique d'ordre suprieur*. PhD thesis, Universit Paris VII, 1972.

[15] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:pp. 29–60, 1969.

[16] W. A. Howard. The Formulas-as-Types Notion of Construction.

[17] A. J. Kfoury and J. B. Wells. Principality and Type Inference for Intersection Types Using Expansion Variables. *Theor. Comput. Sci.*, 311(1-3):1–70, 2004.

[18] N. R. Krishnaswami and N. Benton. Ultrametric Semantics of Reactive Programs. In *LICS*, pages 257–266, 2011.

[19] J. Matthews. Recursive Function Definition over Coinductive Types. In *TPHOLs*, pages 73–90, 1999.

[20] D. A. McAllester and K. Arkoudas. Walther Recursion. In *CADE*, pages 643–657, 1996.

[21] N. Mendler. Recursive Types and Type Constraints in Second Order Lambda Calculus. In *LICS*, pages 30–36. IEEE, 1987.

[22] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[23] R. E. Møgelberg. A Type Theory for Productive Coprogramming Via Guarded Recursion. In *CSL-LICS*, 2014.

[24] H. Nakano. A Modality for Recursion. In *LICS*, pages 255–266, 2000.

[25] H. Nakano. Fixed-Point Logic with the Approximation Modality and its Kripke Completeness. In *TACS*, pages 165–182, 2001.

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

[27] F. Pottier. A Typed Store-passing Translation for General References. In *POPL*, pages 147–158, 2011.

[28] J. C. Reynolds. Towards a theory of type structure. In *Symposium on Programming*, pages 408–423, 1974.

[29] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[30] S. van Bakel. Strict Intersection Types for the Lambda Calculus. *ACM Comput. Surv.*, 43(3):20, 2011.

[31] H. Xi, C. Chen, and G. Chen. Guarded Recursive Datatype Constructors. In *POPL*, pages 224–235, 2003.