Type Inference and Reconstruction for First Order Dependent Types

Neal Nelson
B.A., Mathematics, Washington State University, 1970
M.S., Computer Science, Washington State University, 1976

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

January 1995

The dissertation "Type Inference and Reconstruction for First Order Dependent Types" by Neal Nelson has been examined and approved by the following Examination Committee:

James Hook, Ph. D.
Assistant Professor of Computer Science
Thesis Research Adviser

Richard Kieburtz, Ph. D.
Professor of Computer Science

David Maier, Ph. D.
Professor of Computer Science

Jonathan P. Seldin, Ph. D.
Adjunct Associate Professor of Mathematics
Concordia University
Montreal, Quebec
Canada

Dedication

To Mary Alice and Richard
and to Sherri
for Ricki, Aaron, and Isabel

Acknowledgements

I owe the completion of this dissertation to Jim Hook, who first led me out of the woods onto a research trail, became my mentor throughout, and who most of all taught me how to see more clearly. My deepest thanks to Dick Kieburtz who helped me get started and who has long been an inspiration and a source of great insight. Over the years Dick has helped me refine my intuition and sharpen my critical eye. I am thankful for Dave Maier's dedicated attention to teaching and to making difficult and technical material accessible. Dave has taught me the value of simplicity and the beauty of examples. For the many others here at the Oregon Graduate Institute with whom I have shared these amazing years, thank you.

I owe a special thanks to my friends and colleagues in the Mathematics Department at Reed for introducing me to the breadth and deeper thoughtfulness of a liberal arts education and, moreover, for showing me so much more of the wonderful world of mathematics. I am thankful as well to my friends and colleagues at the Evergreen State College who have supported me through the final years of this work and with whom I have had a most remarkable and rewarding opportunity to continue learning.

To Jonathan Seldin, thank you for making the history of mathematics alive and real for me, and for the inspiration you have given me in doing mathematics as part of a history and culture of ideas.

To my parents I am grateful for many things far beyond what I have accomplished here, but I hope my thanks here after all this work will symbolize my thanks for so much else.

I am most of all grateful to Sherri Shulman for many stimulating and enlightening discussions about this research as well as just about everything else in life, but more,

for teaching me a deeper sense of understanding and thoughtfulness and for sharing a rewarding life with three great kids.

Contents

List of Tables

List of Figures

Abstract

Type Inference and Reconstruction
for
First Order Dependent Types

Neal Nelson, Ph.D.
Oregon Graduate Institute of Science & Technology, 1995

Supervising Professor: James Hook, Ph. D.

This thesis extends a simple ML-style type system by adding type reconstruction for first-order dependent product types. Our study concentrates upon a simple type system we call $\mathcal{T}^\pi$ that includes primitive recursive sequences of types from which we construct product types dependent on natural numbers. The $\mathcal{T}^\pi$ type system is a representation of an earlier system by Tait [Tai65] and Martin-Löf [Mar72a] called $\mathcal{T}^\omega$, postulating infinite sequences of terms and types but having unspecified coding for these sequences. We give a detailed characterization the $\mathcal{T}^\pi$ type system and show that it is possible to type a sensible primitive recursive function that is not typable in ML. The first part of the thesis presents the $\mathcal{T}^\pi$ typing rules, a semantic model with soundness and completeness theorems, and examples of how to type representations of tuples and projections as products dependent on the tuple width. The second part of the thesis proves the existence of $\mathcal{T}^\pi$ principal types and presents a method for type reconstruction that extends ML-style type reconstruction to dependent types.

Chapter 1

Introduction

1.1 Dependent Types

Experience with the ML type inference system has shown that it is possible to have both the advantages of a strongly typed programming language and the convenience of programming without explicit types [Pau91]. The heart of the technique is the ML type inference algorithm that examines programs written without types, and determines the types of the terms in the program from the context of their use. Thus, for example, a program fragment consisting of a function f applied to the two arguments $(4, 3)$ would imply the type of f is a function of two integer arguments. A later context might indicate that f returns an integer argument. When we collect all the information about all the terms and subterms of a program we can either successfully report a type assignment or else indicate failure to type. We call this process *type reconstruction*, and show in this thesis how ML-style type reconstruction can be extended to cover dependent types such as might be used for variable-length arrays, or variable-dimension matrices.

Type reconstruction in ML assumes that programs are typed in a well specified type system. For example, the simplest type system might consist of just function types and natural numbers.

Nat	is the type of natural numbers
$\sigma \to \tau$	is the type of functions from σ to τ if σ, τ are types

Notice that such a type system supports *higher order functions*, that is, functions having other functions as arguments or results. For example, a function f might have type $(Nat \to Nat) \to Nat$, accepting arguments of function type $Nat \to Nat$. An ML-style type

system generalizes the simple type system above with *type variables* A, B, et cetera, so that types can be schemas representing families of types of the same structure or *shape*. The type reconstruction process described above determines the shape of a type. Type variables are place-holders standing for any well formed type (including other types with variables). Thus the type system becomes

$$\begin{array}{ll} Nat & \text{is a type} \\ A & \text{is a type where } A \text{ is a type variable ranging over all types} \\ \sigma \to \tau & \text{is a type if } \sigma, \tau \text{ are types} \end{array} \tag{1.1}$$

The assignment of types to program fragments is specified by a set of rules called the *type inference system*. For the simple type system above, we have the following rules for assigning types to the terms of the lambda calculus, where the symbol : is read "has type". The resulting set of rules defines the *well typed* terms in the *simply typed lambda calculus*.

$$\begin{array}{rcll} x & : & \tau & \text{for } x \text{ a variable of type } \tau \quad \textit{Variable} \\ \lambda x.\delta & : & \sigma \to \tau & \text{if } \delta : \tau \text{ assuming } x : \sigma \quad \to \textit{intro} \\ \delta\ \delta' & : & \tau & \text{if } \delta : \sigma \to \tau, \text{ and } \delta' : \sigma \quad \to \textit{elim} \end{array} \tag{1.2}$$

In working with rules for typing, we follow the convention of specifying the rules in the form of fractions with the premises of the rule above the line and the conclusion of the rule below the line. If the rule has no premise then we call that rule an *assumption* and leave out the line. For rules like $\to$*intro* with premises that depend on existing assumptions, we place the required assumption in brackets just above the premise. Following these conventions for the simply typed lambda calculus, the rules for types 1.1 are expressed as in Figure 1.1 and the typing rules for terms 1.2 above are expressed as in Figure 1.2.

Using these rules we can express the derivation of a well typed term according to the method of *natural deduction*. (See the book by Girard et al [GLT89], for an introduction to the method of natural deduction.) A derivation is represented as a tree with partial type assignments as nodes. Assumption assignments are placed at the leaves and the conclusion type assignment at the root with the leaves written above and the root at the bottom. Natural deduction trees are easily grasped by example. The following tree represents the derivation of the type assignment $\lambda x.\lambda y.xy : (A \to B) \to (A \to B)$.

$$Nat : Type \qquad \textit{Natural numbers}$$

$$A : Type \qquad \textit{Type variables}$$

$$\frac{\sigma : Type \qquad \tau : Type}{\sigma \to \tau : Type} \qquad \textit{Arrow}$$

Figure 1.1: Rules for Simple Lambda Calculus Types

$$x : \tau \qquad \textit{Variable}$$

$$\frac{\begin{array}{c}[x : \sigma] \\ \delta : \tau\end{array}}{\lambda x.\delta : \sigma \to \tau} \qquad \to \textit{intro}$$

$$\frac{\delta : \sigma \to \tau \qquad \delta' : \sigma}{\delta\ \delta' : \tau} \qquad \to \textit{elim}$$

Figure 1.2: Typing Rules for Simple Lambda Calculus Terms

$$\dfrac{\dfrac{\dfrac{[x : A \to B]^2 \quad [y : A]^1}{xy : B}\ \to \textit{elim}}{\lambda y.xy : A \to B}\ \to \textit{intro}^1}{\lambda x.\lambda y.xy : (A \to B) \to (A \to B)}\ \to \textit{intro}^2$$

Each derivation step corresponds to the use of an inference rule whose premise matches the conclusion of the previous derivation step. The rules used are listed to the right of each step in the derivation tree. Whenever the $\to$*intro* rule is used, one or more corresponding assumptions at leaves of the tree are *discharged.* A bracket with a superscript is placed around each discharged assumption along with a corresponding superscript on the rule that actually discharged them. The root of any derivation tree is a valid type assignment provided all the undischarged assumptions are valid type assignments. If all assumptions have been discharged, as in the example above, then the final type assignment is valid under no assumptions. A finite set Γ of all undischarged assumptions in a derivation is

referred to as the *context* of the final type assignment.[1]

Given a specification of well typed terms by an inference system, the type reconstruction algorithm is expected to locate a valid type assignment for an untyped or *raw* term. Thus the type reconstruction algorithm may be seen as the search for a derivation of a well typed term in the type inference system. Raw terms in the simply typed lambda calculus belong to the untyped lambda calculus described thoroughly by Barendregt [Bar84].

The computational model of the untyped (and untyped) lambda calculus is function application. Function application in the lambda calculus is expressed by a rewriting system based primarily on a single equality rule among terms known as the β rule. The β rule imitates the action of function application by performing the associated substitution. (See [Bar84].) If a β rule applies to a term then we say the term has *computational content* and can be *reduced* by the β rule to a simpler term. When a term can no longer be reduced it is considered to be in *normal form* and represents a final computed value.

The derivation of a well typed term is largely independent of the computational content of that term. The only expectation is the *subject reduction* property discussed later: if there is a derivation for a term, then there is a derivation resulting in the same final type for any reduction of that term. In systems of natural deduction, computations by β reductions correspond to an operation on derivations called *cut elimination.* See Girard, Taylor and Lafont [GLT89] for a discussion of cut elimination in natural deduction systems.

The presense of type variables in the type system gives *type schemes* representing infinite families of types. If we assume that a type scheme is itself a type, then we can think of the family of types associated with a type scheme as all those types that can be obtained by *replacing* the type variables in the scheme by any other type (possible with variables). If a type τ is obtained from another type σ by such a replacement, then we say that σ *subsumes* τ, denoted $\sigma \geq \tau$, or that τ is an *instance* of σ. This ordering of types is called the *subsumption* order and forms the basis of the categorization of types for terms.

Let us examine the type scheme of the following *map* function that takes as arguments

[1] Typically contexts are ordered lists, but ours are sets ordered by type dependencies.

a function f, and a list $[x_0, \ldots, x_n]$ and applies the function to all elements of the list.

$$map\ f\ [x_0, \ldots, x_n] = [f(x_0), \ldots, f(x_n)]$$

If we let A and B be variable types and *A-List* and *B-List* be lists of elements of variable types A and B respectively, then the *map* function has the following type scheme. Note that the *map* function is higher order because it is a function that takes another function as an argument.

$$map : (A \rightarrow B) \rightarrow (A\text{-}List \rightarrow B\text{-}List)$$

The *map* function may take on a variety of instances of its type scheme as a type.

Working with type schemes implies that program terms may have multiple types, a situation that is problematic in programming. Roger Hindley [Hin69] showed that every well typed term in the simply typed lambda calculus introduced above has a unique representative type known as the *principal type scheme.* Not all type systems have the property of principality. Hindley's proof was constructive, showing the existence of a principal type scheme for a term by finding one. The principal type scheme for a term δ is the one that subsumes all other legal type schemes for the term δ. We will retain this important property for the type system of this thesis.

Finding principal type schemes for programs is the essence of ML-style type reconstruction. It was Robin Milner [Mil78] who translated Hindley's constructive proof of the existence of principal type schemes for the simply typed lambda calculus into an algorithm for finding type schemes for ML, a simple programming language variation of the simply typed lambda calculus. Milner introduced *polymorphic types* as encapsulations of type schemes by universally quantifying over the type variables. Later, his student Luis Damas [DM82] devised type inference rules to succinctly characterize the polymorphic type system for ML. Both Milner and Damas gave a type inference algorithm (what we call a type reconstruction algorithm) and Damas proved the soundness and completeness of the type inference algorithm with respect to his type inference system. We follow Damas' approach and use inference rules as a non-deterministic specification of our type system. We use the term *type reconstruction* to emphasize that the process of reconstructing types from raw terms is based on the inference rules.

ML-style type systems have a collection of reconstructible types that encompass most of the standard programming language data structures [Pau91]. The general purpose of this thesis is to show that it is possible to extend the ML-style type system and reconstruction algorithm to support the typing of variable-length arrays and other related types that depend on computed values. The types we use are *first-order dependent products*, introduced by Martin-Löf as a *Cartesian product of a family of types* [Mar75] in an article on intuitionistic type theory. The fundamental assumption that makes dependent type reconstruction possible is that we can only express terminating programs. We will simplify our study by specifically investigating first-order product types dependent on natural numbers.

The motivation of this research is to address the long-standing programming problem of coding and assigning a type to functions such as a matrix transpose, whose operation and type depend on the dimensions of the matrix arguments. What are the types of the arguments and results? Common practice assigns them some maximum dimension array, disregarding that the actual argument and result may be a quite different dimension. On the other hand, our system will assign first-order dependent types to the matrices, where the matrix dimensions will now be part of the type. For example, we would write the type

$$\mathit{transpose} : \Pi n.\Pi m.\mathit{Array}(n,m) \rightarrow \mathit{Array}(m,n)$$

indicating that the function accepts three arguments, including two dimension parameters n and m plus an n by m array, and produces as a result an m by n array.

In this approach to typing where types are dependent on values, we have made the significant assumption that all programs terminate, or at least all program fragments to which we apply our typing system. It is this key termination assumption that opens the possibility of much stronger type systems. We call our type system $\mathcal{T}^{\pi}$, where the π suggests products.

The $\mathcal{T}^{\pi}$ type system can be viewed as a fragment of Martin-Löf's type theory, but examined from the point of view of type reconstruction and a Curry philosophy where types serve to group together untyped terms. Thus, where Martin-Löf's type theory is monomorphic, $\mathcal{T}^{\pi}$ has type variables permitting type schemes, which open the system for

type reconstruction. On the other hand, for simplicity we stay with monomorphic type schemas as in Hindley [Hin69] rather than polymorphic types as in Damas [DM82]. The extension of $\mathcal{T}^\pi$ to polymorphic types is possible in the same way that Damas extended Milner's work, though we do not pursue this extension. For an enlightening discussion of Curry's philosophy see Hindley and Seldin [HS86]. For a discussion of an alternative Church philosophy, where terms are always introduced with their types, see the discussion in Harper and Mitchell on the type structure of standard ML [HM93]. For a comparison of the Church and Curry philosophies see Pierce, et al [PDM89].

Our study actually begins with an examination in the next section of an early and relatively obscure type system of Martin-Löf [Mar72a] originally introduced by Tait [Tai65] and known as the $\mathcal{T}^\infty$ system.[2] The $\mathcal{T}^\infty$ system was the first to deal with types for possibly infinite indexed sequences of terms. In their studies of $\mathcal{T}^\infty$, Martin-Löf and Tait were interested in the effect of certain transfinite rule schemas on the normalization properties of derivations. They proved that it was possible to introduce infinite sequences of terms and types dependent on natural numbers and still retain strong normalization properties. Tait and Martin-Löf were not concerned with how infinite sequences of terms and types were *coded*, and it is our focus on *representable* sequences and products that distinguishes $\mathcal{T}^\pi$ from $\mathcal{T}^\infty$. We will introduce the concept of *recursively based sequences* to characterize the representation property of our product types and the underlying sequences of types. Recursively based sequences are sequences that can be specified with finite or well founded recursive formulas.

In the next section we introduce the main concepts of the theisis, exploring them in the simplified form of the $\mathcal{T}^\infty$ system, abstracted away from the *representations* of infinite sequences that we later require. Following the Introduction, Chapter 2 defines the $\mathcal{T}^\pi$ type inference system and then gives a model for it along with soundness and completeness theorems. We also elaborate three complete examples, including a function that is not typable in ML. Chapter 3 gives a constructive proof of the existence of principal type schemes in $\mathcal{T}^\pi$ and then presents the type reconstruction algorithm. Chapter 4 wraps

[2]The author thanks Jon Seldin for pointing out that the type system in this thesis is in part a rediscovery of the earlier type system $\mathcal{T}^\infty$.

up the results, offering conclusions, comparisons with other work, and future research.

1.2 Dependent Type Inference and Reconstruction

Martin-Löf in 1972 [Mar72a] presented Tait's system $\mathcal{T}^\infty$ of infinite terms as a system of natural deduction. In this general framework we present the basic concepts of dependent type inference and type reconstruction.

The $\mathcal{T}^\pi$ system realizes all infinite sequences as primitive recursive formulas, whereas the $\mathcal{T}^\infty$ system *presumes* that we have certain infinite rules for dealing with sequences of types and terms, but these rules are not specified. Both systems use infinite sequences as the basis of constructing first order product types dependent on natural numbers. Examining type inference and type reconstruction in $\mathcal{T}^\infty$ gives us an abstract view of the the type system without the details required to handle particular finite codings of sequences.

The rules for $\mathcal{T}^\infty$ types, shown in Figure 1.3, are those of a simple type system with an added product type over infinite sequences of types. We further generalize the system with type variables and thus type schemas, as in the previous section. The symbol ω mentioned in the product rule stands for the natural numbers.

$$Nat : Type \qquad \textit{Natural numbers}$$

$$A : Type \quad \text{for } A \text{ a type variable} \qquad \textit{Type variables}$$

$$\frac{\sigma : Type \qquad \tau : Type}{\sigma \to \tau : Type} \qquad \textit{Arrow}(\to)$$

$$\frac{\tau_0 : Type \quad \tau_1 : Type \quad \cdots}{\Pi(\tau_i) : Type} \quad \text{for } i \in \omega \qquad \textit{Product}(\Pi)$$

Figure 1.3: Rules for $\mathcal{T}^\infty$ Types

The type inference rules for terms in $\mathcal{T}^\infty$, shown in Figure 1.4, are the ones for simple types given in the introductory section plus product introduction and elimination. In our formulation of the $\to$ *intro* rule we use the notation $[x : \sigma]$ to indicate that the premise

$\delta : \tau$ is derived *assuming* the variable x in δ has type σ, i.e., $x : \sigma$, and that the $\rightarrow$ *intro* rule *discharges* this assumption as discussed in the previous section for the simply typed lambda calculus.

$$x : \tau \qquad \textit{Var}$$

$$\frac{\begin{array}{c}[x:\sigma]\\ \delta : \tau\end{array}}{\lambda x.\delta : \sigma \rightarrow \tau} \qquad \rightarrow \textit{intro}$$

$$\frac{\delta : \sigma \rightarrow \tau \qquad \delta' : \sigma}{\delta\ \delta' : \tau} \qquad \rightarrow \textit{elim}$$

$$\frac{\delta_0 : \tau_0 \quad \delta_1 : \tau_1 \quad \cdots}{(\delta_0, \delta_1, \ldots) : \Pi(\tau_i)} \qquad \Pi\textit{-intro}$$

$$\frac{\delta : \Pi(\tau_i) \qquad k \in \omega}{\delta_k : \tau_k} \qquad \Pi\textit{-elim}$$

Figure 1.4: Typing Rules for $\mathcal{T}^\infty$ Terms

The contraction rules[3] for $\mathcal{T}^\infty$, shown in Equations 1.3, are η and β reduction as in the simply typed lambda calculus plus a projection equation π. The full equational theory of the simply typed lambda calculus and the $\mathcal{T}^\infty$ system both involve other equations, for example, there is an α-equality rule for variable name changes and the $\mathcal{T}^\infty$ system has a *distribution*-rule not in the simply typed lambda calculus. (See Tait's presentation [Tai65].) However, mostly these other equations are not of concern to us in this thesis about typing.

$$\begin{array}{rcll} (\lambda x.\delta\ x) & = & \delta & (\eta) \\ (\lambda x.\delta)\ \delta' & = & \delta[x := \delta'] & (\beta) \\ (\delta_0, \delta_1, \ldots)\ k & = & \delta_k & (\pi) \end{array} \tag{1.3}$$

As in Section one, consider a natural-deduction style of representing $\mathcal{T}^\infty$ derivations

[3]Equality rules between terms are often referred to as *conversion* rules. A conversion rule is considered a *reduction* rule or a *contraction* rule if the equation is intended to be used primarily from left to right, that is, substituting the right side for the left side, and the right side is simpler by some measure than the left side.

as trees of formulas having assumptions at the leaves, conclusions at the nodes, and nodes labeled by the typing rule used. The derivation of the typing $\lambda x.(\delta_0, \delta_1, \ldots) : \sigma \rightarrow \Pi(\tau_i)$ can be represented by the following natural deduction tree.

$$\dfrac{\dfrac{\begin{array}{cccc}[\{x:\sigma\}]^1 & [\{x:\sigma\}]^1 & \ldots \\ \vdots & \vdots & \ldots \\ \delta_0 : \tau_0 & \delta_1 : \tau_1 & \ldots\end{array}}{(\delta_0, \delta_1, \ldots) : \Pi(\tau_i)}\ \Pi\text{-}intro}{\lambda x.(\delta_0, \delta_1, \ldots) : \sigma \rightarrow \Pi(\tau_i)}\ \rightarrow intro^1$$

As in the simply typed lambda calculus, discharges only occur at the use of the $\rightarrow$ *intro* rule.

Let us investigate the character of the $\mathcal{T}^\infty$ system by studying the typing of a program that cannot be typed in the simply typed lambda calculus. The following tautology checking function from the SASL language manual [Tur76] will not type in ML. However, as shown by Nordström, Petersson, and Smith [NPS90], the *taut* function can be typed with dependent product types.[4]

$$\begin{aligned} taut\ 0\ f &= f \\ taut\ (succ\ k)\ f &= (taut\ k\ (f\ true))\ \&\ (taut\ k\ (f\ false)) \end{aligned}$$

The principle difficulty in reconstructing a type for *taut* is that the type of the second argument f changes depending on the value of the first argument, a natural number k. The ML type synthesizer will fail to unify the type *Bool*, a single type variable for the type of f in the zero case, with the type $Bool \rightarrow X$, the type of f in the non-zero case.

To type *taut*, postulate a type sequence $\{Bool_0, (Bool \rightarrow Bool)_1, (Bool \rightarrow Bool \rightarrow Bool)_2, ...\}$ for f and informally represent this sequence by the expression $(Bool \rightarrow)^n Bool$. Now the type of *taut* can be informally expressed as a *Nat*-dependent product, where the superscript n can take on any natural number k.

$$\Pi n.((Bool \rightarrow)^n Bool) \rightarrow Bool \tag{1.4}$$

The term *taut* 0 would have the type $Bool \rightarrow Bool$ and the term *taut* (*succ* k) would

[4]The term *succ* k stands for the successor of k, that is, $k + 1$.

have the type $(Bool \to (Bool \to)^k Bool) \to Bool$. Combining the typings for these two equations gives the type of *taut* for all natural numbers k expressed uniformly by 1.4.

Now consider the $\mathcal{T}^\infty$ system as an abstract way to examine type reconstruction for dependent products. Define the *context* Γ of any node $\delta : \tau$ in a derivation as the mapping from variables to types representing the collection of all undischarged assumptions at that node. We write a *sequent* in the form $\Gamma \vdash \delta : \tau$ to indicate that the type assignment $\delta : \tau$ has context Γ. Let *Tvars* be a countable and infinite set of type variables and let *Types* be the set of all types in $\mathcal{T}^\infty$. Define a *type substitution* as a total map Θ : *Tvars* $\to$ *Types* such that $\Theta(X) \neq X$ for finitely many $X \in$ *Tvars*. Extend the type substitution domain to all types, Θ : *Types* $\to$ *Types* in the usual way. (See, for example, the presentation in Snyder and Gallier [SG89].) Let $\Theta\tau$ denote the application of substitution Θ to type τ and let $\Theta_2\Theta_1$ be the composition of substitutions Θ_2 and Θ_1 defined by $(\Theta_2\Theta_1)(x) = \Theta_2(\Theta_1(x))$.[5] The *support* of a substitution Θ, denoted su(Θ), is the set of type variables A for which $\Theta A \neq A$. Define Θ_{id} as the substitution that maps every type to itself. Finally, extend substitutions to apply to contexts, $\Theta(\Gamma\{x : \tau\}) = \Theta\Gamma\{x : \Theta\tau\}$. The notation $\Gamma\{x : \tau\}$ means the function Γ extended with the assignment of type τ to variable x.

We can imagine extending Hindley's results for simple types [Hin69] (see also Damas and Milner [DM82]) to show that the $\mathcal{T}^\infty$ system supports *principal derivations* and *principal type schemes* for terms. To do this we must make sense of infinite compositions of substitutions $(\ldots \Theta_2\Theta_1\Theta_0)$ and make sense of constructing an infinite sequence of principal types $(\tau_0, \tau_1, \tau_2, \ldots)$ for an infinite sequence of terms $(\delta_0, \delta_1, \delta_2, \ldots)$. For now let us suppose such infinite compositions and infinite sequences are somehow sensible and postulate a hypothetical method for type reconstruction in $\mathcal{T}^\infty$ to gain insight into type reconstruction for the $\mathcal{T}^\pi$ system. In the remainder of the thesis we will show how the explicit coding of infinite sequences in $\mathcal{T}^\pi$ makes the reconstruction of principal types both sensible and realizable.

The method of type reconstruction for the simply typed lambda calculus suggests how we might reconstruct $\mathcal{T}^\infty$ types from untyped terms. Reconstruction is based on

[5] We always use application order for composition rather than diagrammatic order.

structural decomposition of a term in a context to produce a principal derivation for the term. The method depends on computing the *most general unifier* of two type schemes using a unification algorithm. Hypothetical methods U and W for unification and type reconstruction of $\mathcal{T}^\infty$ are shown in Figure 1.6 and Figure 1.5. We will never be able to unify pairs of infinite sequences of types in any reasonable amount of time, so the methods for type unification and reconstruction for $\mathcal{T}^\infty$ are simply *illustrative* as mentioned above.

The method W takes a context Γ and a term δ as input and returns, if they exist, a substitution Θ and a principle type τ such that $\delta : \tau$ is derivable in context $\Theta\Gamma$. The unification method U returns a substitution called the *most general unifier* that, when applied to either of the two input types, yields their *highest common instance.*[6] The first three steps of the W algorithm are the same as the reconstruction algorithm for a simple type system. The last two steps handle reconstructing types for sequences.

We can postulate the soundness for $\mathcal{T}^\infty$ type reconstruction assuming the soundness of the U algorithm for $\mathcal{T}^\infty$. We only conjecture a form of completeness for $\mathcal{T}^\infty$ type reconstruction in favor of commencing our investigation of the $\mathcal{T}^\pi$ system.

Theorem 1 (Soundness of W) *If $W(\Gamma, \delta)$ succeeds with result (Θ, τ), then there is a derivation of $\Theta\Gamma \vdash \delta : \tau$ according to the typing rules of $\mathcal{T}^\infty$.*

Proof The proof is by a straightforward induction on the structure of δ following the same method as Damas and Milner [DM82]. ∎

Conjecture 2 (Completeness of W) *Given a context Γ and a term δ, if there is a derivation $\Gamma' \vdash \delta : \tau$ according to the rules of $\mathcal{T}^\infty$ with $\Gamma' \leq \Gamma$, then $W(\Gamma, \delta)$ succeeds with result (Θ, τ) and furthermore, $\Gamma' \leq \Theta\Gamma$ and $\tau \leq \sigma$.*

In the methods W and U, notice that type reconstruction can be determined only if the reconstruction in steps *iv* or *v* of W and the unification of a product in step *iv* of U are not demands for an infinite computation, which, of course, they are. As indicated earlier, the

[6] Hindley's term *highest common instance* is now more commonly called the *most general common instance.*

$W(\Gamma, \delta)$ $= (\Theta, \tau)$ where

(i)	$W(\Gamma, x)$	$=$	(Θ_{id}, τ) if $\{x : \tau\} \in \Gamma$
(ii)	$W(\Gamma, \lambda x.\delta)$	$=$	Let $(\Theta, \tau) = W(\Gamma\{x : A\}, \delta)$ and $x \notin \mathrm{fv}(\tau)$ in $(\Theta, \Theta A \to \tau)$ where A is a new type variable
(iii)	$W(\Gamma, \delta\ \delta')$	$=$	Let $(\Theta, \tau) = W(\Gamma, \delta)$ and $(\Theta', \sigma) = W(\Theta\Gamma, \delta')$ and $\Theta'' = U(\Theta'\tau, \sigma \to B)$ in $(\Theta''\Theta'\Theta, \Theta''B)$ where B is a new type variable
(iv)	$W(\Gamma, (\delta_0, \delta_1, \ldots)\ )$	$=$	Let $(\Theta_0, \tau_0') = W(\Gamma, \delta_0)$ and $(\Theta_{i+1}, \tau_{i+1}') = W(\Theta_i \ldots \Theta_0\Gamma, \delta_{i+1})$ for all i and $\tau_i = (\ldots \Theta_{i+2}\Theta_{i+1})\tau_i'$ for all i in $(\ (\ldots \Theta_1\Theta_0), \Pi(\tau_i)\ )$
(v)	$W(\Gamma, (\delta_0, \delta_1, \ldots)\ n)$	$=$	Let $(\Theta, \tau) = W(\Gamma, (\delta_0, \delta_1, \ldots)\ )$ and $\Theta'' = U(\tau, \Pi(A_i)\ )$ in $(\Theta''\Theta, \Theta''A_n)$ where n is a natural number and where A_i are new type variables for all i

Figure 1.5: Type Reconstruction for $\mathcal{T}^\infty$

$\mathcal{T}^\pi$ system we study in Chapter 2 is primarily distinguished from $\mathcal{T}^\infty$ by its finite coding of sequences of terms, types, rules, and derivations using primitive recursive formulas. This means the infinite step of unifying sequences of types can be done by *unifying primitive recursive formulas for types.* Also, the infinite step of reconstructing principal derivations for product types from sequences of derivations can be done by *constructing the principal derivations for the primitive recursive codings* of sequences of derivations.

Before commencing our study of $\mathcal{T}^\pi$, we briefly investigate how we can actually represent the sequences of terms and types of the $\mathcal{T}^\infty$ system and give the typing rules for assigning types to programs such as *taut*. The representation of terms and types in $\mathcal{T}^\pi$ is a variation of Gödel's theory $\mathcal{T}$ of primitive recursive functionals. The terms (i.e. the programs) of $\mathcal{T}^\pi$ are coded just as in theory $\mathcal{T}$. The type system for $\mathcal{T}^\pi$ is the simple system of finite types in $\mathcal{T}$ extended to allow primitive recursive formulations of type sequences from which *Nat*-dependent product types can be constructed.

$$
\begin{array}{llll}
U(\sigma,\tau) & = \Theta \text{ where} & & \\
(i) & U(\tau,\tau) & = & \Theta_{id} \\
(ii) & U(A,\tau) \text{ or } U(\tau,A) & = & \Theta_{id}\{A := \tau\} \qquad \text{provided } A \notin \mathrm{fv}(\tau) \\
(iii) & U(\sigma \to \tau, \sigma' \to \tau') & = & \text{Let } \Theta = U(\tau,\tau') \\
 & & & \text{in } U(\Theta\sigma, \Theta\sigma') \\
(iv) & U(\Pi(\sigma_i), \Pi(\tau_i)\ ) & = & \text{Let } \Theta_0 = U(\sigma_0,\tau_0) \\
 & & & \text{and } \Theta_{i+1} = U(\Theta_i \ldots \Theta_0\sigma_i, \Theta_i \ldots \Theta_0\tau_i) \text{ for all } i \\
 & & & \text{in } (\ldots \Theta_1\Theta_0)
\end{array}
$$

Figure 1.6: Unification for $\mathcal{T}^\infty$ Types

The programs in $\mathcal{T}^\pi$ (and $\mathcal{T}$) are coded using the typed lambda calculus extended with *numerals* (i.e., with $\mathbf{0}$ and a successor operator $\mathbf{S}$) and a primitive recursion operator $\mathbf{R}$. The recursion operator $\mathbf{R}$ takes three arguments. The first two arguments are the base case and the unfolding case of a recursive expression. The third argument is a numeral that controls the unfolding of the recursive expression during evaluation. The $\mathbf{R}$ operator obeys the following equality rules where the term $\mathbf{S}^k$ is shorthand for the k^{th} numeral.

$$
\begin{aligned}
\mathbf{R}\ \delta'\ \delta\ \mathbf{0} &= \delta' \\
\mathbf{R}\ \delta'\ \delta\ (\mathbf{S}^{k+1}\mathbf{0}) &= \delta\ (\mathbf{S}^k\mathbf{0})\ (\mathbf{R}\ \delta'\ \delta\ (\mathbf{S}^k\mathbf{0}))
\end{aligned}
$$

For a discussion of theory $\mathcal{T}$ see Girard, Lafont, and Taylor [GLT89].

The *taut* program in the $\mathcal{T}$ (or $\mathcal{T}^\pi$) theory is

$$
taut \equiv \mathbf{R}\ (\lambda f.f)\ (\lambda n'.\lambda p.\lambda f.((p\ (f\ true))\ \&\ (p\ (f\ false))))
$$

The *taut* function can be understood, according to the theory $\mathcal{T}$, in terms of the following equations, where δ stands for an arbitrary expression in the language and we do not make use of the bound variable n'. These primitive recursive equations can also be seen as defining a well-founded sequence of terms.

$$
\begin{aligned}
taut\ \mathbf{0} &= \lambda f.f && (= Zero) \\
taut\ (\mathbf{S}\ \delta) &= \lambda f.(((taut\ \delta)(f\ true))\&((taut\ \delta)(f\ false))) && (= Succ)
\end{aligned}
$$

Sequences of types used to build product types are also formulated using the theory $\mathcal{T}$, but at the level of type formulas. We assume primitive type constructors *Nat*, *Bool*, and $\to$, and use the $\mathbf{R}$ combinator to extend simple finite type formulas to primitive recursive

sequences of type formulas. The type of the function argument f of *taut*, expressed informally above as $(Bool \rightarrow)^n Bool$, can now be formulated as[7]

$$Boolfam \quad \equiv \quad \mathbf{R}\ Bool\ (\lambda n'.\lambda X.Bool \rightarrow X)$$

Notice in the definition of *Boolfam* that the subexpression type $Bool \rightarrow X$, which we call the *successor expression*, is abstracted on both X and n', but that the term variable n' never appears in the formula. When the successor expression does not depend on a term variable, as in the *Boolfam* example above, then the type formula satisfies one of two restrictions that define what we call *weak recursion.* (We will see the complete definition of weak recursion in Chapter 2.) Our results for reconstruction apply only to the *weak* $\mathcal{T}^\pi$ system where only weak recursion is allowed in type formulas.

The elements of the type expressed by the *Boolfam* formula above can be understood according to the following type equality rules that elaborate the *sequence* of types comprising *Boolfam.*

$$\begin{array}{lcll} Boolfam\ \mathbf{0} & = & Bool & (= \Pi R\text{-}Zero) \\ Boolfam\ (\mathbf{S}\ \delta) & = & Bool \rightarrow (Boolfam\ \delta) & (= \Pi R\text{-}Succ) \end{array}$$

The type of the *taut* program can now be formally expressed as a *Nat*-dependent *product* using the *Boolfam* type sequence above. The following typing for the *taut* program can be derived using the typing rules presented in Chapter 2, see Section 2.2 for a detailed presentation of this example.[8]

$$\begin{array}{rcll} taut & : & \Pi n.(Boolfam\ n) \rightarrow Bool & \\ & : & \Pi n.(\mathbf{R}\ Bool\ (\lambda n'.\lambda X.Bool \rightarrow X)\ n) \rightarrow Bool & (\equiv Boolfam) \end{array}$$

We have shown the character of the $\mathcal{T}^\pi$ type system by its relationship with the simply typed lambda calculus and the $\mathcal{T}^\infty$ system with dependent products. In studying the simple $\mathcal{T}^\infty$ system, we created a framework for understanding dependent type reconstruction. We postulated the existence of principal types in $\mathcal{T}^\infty$, gave a $\mathcal{T}^\infty$ type reconstruction

[7] We use the symbol $\equiv$ to mean syntactic replacement. This is simply used as a shorthand to make the presentation easier to read.

[8] In our notation, when a term is shown with two type assignments as in the taut example, we are expressing the fact that the two types are equivalent and give the reason in parentheses.

algorithm, and conjectured the soundness and completeness of the algorithm with respect to the $\mathcal{T}^{\infty}$ inference rules. The $\mathcal{T}^{\infty}$ type system, however, is a mathematical abstraction without sufficient basis for computing. We highlighted the coding distinctions required to transform the $\mathcal{T}^{\infty}$ system into the $\mathcal{T}^{\pi}$ type system for computing, namely, the finite representation of *sequences* of types and terms, and the use of numerals as computed representations of natural numbers. The next chapter introduces the $\mathcal{T}^{\pi}$ type system in detail.

Chapter 2

Primitive Recursive Functionals with Dependent Types

2.1 A Dependent Type System for $\mathcal{T}^\pi$

We embark on a detailed description of the $\mathcal{T}^\pi$ type system, concentrating on a subsystem we call the *weak* $\mathcal{T}^\pi$ type system for which we will be able to reconstruct principal types. In our philosophy of typing, not all syntactically well formed term formulas are valid terms; we use types to identify and group the valid term formulas. There is a subtle distinction between this Curry philosophy and a Church philosophy where terms are always introduced with their types. We imagine that we are looking for well typed terms in a larger universe of untyped terms. See the presentations in Hindley and Seldin, Harper and Mitchell, and Pierce et al [HS86, HM93, PDM89] for discussions and comparisons of the Curry versus Church philosophies and explicit versus implicit typing.

In this first section we give rules for types and rules for deriving well typed terms, that is, what we know as the $\mathcal{T}^\pi$ *type inference system.* In the next section we present some example derivations of well typed terms, and in the final section of this chapter we give a simple set model for the $\mathcal{T}^\pi$ type system, showing that typing statements derivable according to the rules correspond to valid set memberships in the semantic domain.

2.1.1 Terms

In our discussion of types and terms, we will use the following notation. Metavariables δ and ν range over terms, and metavariables τ, σ, ρ, and γ range over types. Other Greek

Terms	δ	$::=$	$\mathbf{0}$	*Zero*
		$\mid$	$\mathbf{S}$	*Successor*
		$\mid$	$\mathbf{R}$	*Recursion*
		$\mid$	x	*Variable*
		$\mid$	$\lambda x.\delta$	*Abstraction*
		$\mid$	$\delta\ \delta$	*Application*

Figure 2.1: Syntax of Terms in $\mathcal{T}^\pi$

letters also occasionally stand for types. A subscripted metavariable τ_i ranges over families of types indexed by $i \in \omega$. The countably infinite set $Tvar = \{A, B, \ldots, Z, A_1, B_1, \ldots\}$ of upper case roman letters are type variables. Similarly, the countably infinite set of lower case Roman letters $Var = \{a, b, \ldots, z, a_1, b_1, \ldots\}$ are term variables.

The syntax of terms is given in Figure 2.1. A *term replacement* $\delta[x := \delta']$ denotes the term formula obtained from δ by replacing x everywhere in δ by δ'. Throughout this thesis, we assume the usual convention for avoiding the unsound capture of free variables in replacements by requiring that the free and bound variables of δ and δ' be entirely distinct, renaming them if necessary to avoid conflicts. We call this the *non-interference assumption of replacement.* For a finite set of variables $\{x_i | i \in \omega\}$, a *simultaneous replacement* is denoted $\delta[x_i := \delta_i]$. In a simultaneous replacement all of the replacements operate in parallel on the original δ rather than sequentially where each substitution operates on the result of a previous substitution. The notation $\mathrm{fv}(\delta)$ refers to the set of free variables of a term, that is, those variables x that are not in the scope of some lambda binder λx in δ.

The notation $\mathrm{dom}(\phi)$ refers to the domain of a function ϕ. We write an *extension* of a function ϕ as $\phi\{x := v\}$ denoting the function ϕ' that is everywhere the same as ϕ but extended with the new domain element $x \notin \mathrm{dom}(\phi)$ assigned the value v. The *restriction* of a function ϕ to a subset S of its domain is denoted $\phi|_S$. We let $\phi|_{\mathrm{dom}(\phi)-S}$ be denoted $\phi|_{-S}$. The empty set $\{\}$ notation also represents the empty map.

A *term equality statement* is a formula $\delta' = \delta$. Term equality is also called *term convertibility* and denoted $\overset{\mathrm{cnv}}{=}$ to distinguish it from other equalities. We assume terms obey the usual lambda calculus equality rules (see Hindley and Seldin [HS86]). However, for our purposes we are only interested in the β and η equality rules of the lambda calculus.

The terms of $\mathcal{T}^\pi$ obey the equations given in Figure 2.2. We define *term reduction*, $\xrightarrow{\text{red}}$, as the special case of conversion in which the equations η, β, *R-Zero*, and *R-Succ* are used only left to right. The term equality rules do not require that the terms be well typed.

$$\begin{array}{rcll} \lambda x.\delta\ x & = & \delta & \eta \\ (\lambda x.\delta)\ \delta' & = & \delta[x := \delta'] & \beta \\ \mathbf{R}\ \delta'\ \delta\ \mathbf{0} & = & \delta' & \textit{R-Zero} \\ \mathbf{R}\ \delta'\ \delta\ (\mathbf{S}\ \delta'') & = & \delta\ \delta''\ (\mathbf{R}\ \delta'\ \delta\ \delta'') & \textit{R-Succ} \end{array}$$

Figure 2.2: Term Equality Rules of $\mathcal{T}^\pi$

The set of natural numbers is referred to as ω, the intended interpretation of the type *Nat* of numerals discussed below. The canonical form for a numeral is $\mathbf{S}^k\mathbf{0}$, indicating the k^{th} numeral, where $\mathbf{S}^k$ is shorthand for $\mathbf{S}\ \mathbf{S}\ \ldots\mathbf{S}$ (k times).

2.1.2 Types

The rules for *weak* $\mathcal{T}^\pi$ type formulas are given in Figure 2.3. The weakening of the $\mathcal{T}^\pi$ type system is manifest in the conditions on the *Tseq* rule, which we discuss in detail shortly. Removing the restrictive conditions (b) and (c) on the *Tseq* rule of Figure 2.3 leaves us with the (non-weak) $\mathcal{T}^\pi$ type system. We prove our primary results concerning principal types and type reconstruction for the weak system. The $\mathcal{T}^\pi$ model and the soundness and completeness for the $\mathcal{T}^\pi$ type inference system given in Section 2.3 are shown for the non-weak $\mathcal{T}^\pi$ system.

The rules for types really specify families of types in the following sense. If τ is a type formula with free term variables $\{n_1, \ldots, n_i\}$ then we say τ represents a *family of types with indices* $\{n_1, \ldots, n_i\}$. An *index closed type* is a family of types τ with no indices, that is, the set of free term variables in τ is empty. Notice that in the $\mathcal{T}^\pi$ types we must make explicit distinction between a *Product*, $\Pi n.\tau$, and an *Arrow* type, $Nat \to \tau$, by requiring that index variables n be manifest in the products. (See the product rule in Figure 2.3.)[1]

[1]The distinction we make between $\Pi n.\tau$ and $Nat \to \tau$ is somewhat unusual in that often $Nat \to \tau$ is considered to be just a shorthand for the less discriminating product $\Pi n.\tau$ where n is not in the free variables of τ.

As we will soon see, all $\mathcal{T}^\pi$ families represent *recursively based sequences* of types that correspond to the sequences of types used in $\mathcal{T}^\infty$ to construct products. (See the premises of the *Product* rule of $\mathcal{T}^\infty$ in Figure 1.3 of Section 1.2.)

$$Nat : Type \qquad Num$$

$$X : Type \quad \text{for } X \text{ a type variable} \qquad Tvar$$

$$\frac{\sigma : Type \qquad \tau : Type}{\sigma \to \tau : Type} \qquad Arrow(\to)$$

$$\frac{\tau : Type \qquad \begin{matrix}[n' : Nat] \\ \sigma : Type\end{matrix} \qquad n : Nat}{\mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ n : Type} \quad \text{if} \begin{cases} \text{(a) } X \in \mathrm{fv}(\sigma) \text{ but } \sigma \neq X \\ \text{(b) } n' \notin \mathrm{fv}(\sigma) \text{ and} \\ \text{(c) } n \notin \mathrm{fv}(\sigma, \tau) \text{ and} \end{cases} \qquad Tseq$$

$$\frac{\begin{matrix}[n : Nat] \\ \tau : Type\end{matrix}}{\Pi n.\tau : Type} \quad \text{if } n \in \mathrm{fv}(\tau) \qquad Product(\Pi)$$

Figure 2.3: Rules for Weak $\mathcal{T}^\pi$ Type Families

The notation $\mathrm{fv}(\tau)$ denotes the set of free variables of the type expression τ (either term variables or type variables). We say $\mathrm{fvtm}(\tau)$ to refer specifically to the free term variables, and $\mathrm{fvty}(\tau)$ to refer specifically to the free type variables of τ. Sometimes we say $\mathrm{fv}(\sigma, \tau)$ to mean $\mathrm{fv}(\sigma)$ and $\mathrm{fv}(\tau)$. If, by using the rules for $\mathcal{T}^\pi$ type families of Figure 2.3, we can derive τ as a type under a set of assumptions Γ about free term variables, then we denote this fact by the sequent $\Gamma \vdash \tau : Type$.

A *type replacement* $\tau[X := \sigma]$ denotes the type formula obtained from τ by replacing type variable X everywhere in τ by σ provided the free and bound variables of τ and σ are entirely distinct. We have, as with terms, the non-interference assumption of replacement. A term replacement in a type is denoted $\tau[n := \delta]$ and defined similarly. We do not demand that the term δ is well typed, however, we are most interested in the following closure theorem where only well typed terms participate in replacements.

Proposition 3 (Closure of type replacement) *Type replacement is closed in the* $\mathcal{T}^\pi$

type system, that is, if τ and σ are $\mathcal{T}^\pi$ types then $\tau[X := \sigma]$ is a $\mathcal{T}^\pi$ type.

Proof Take a derivation of σ and replace the derivation of the premise X : *Type* by the derivation of τ : *Type*. ∎

We can interpret the family of types $\mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ n$ in the *Tseq* rule as representing the following primitive recursive sequence of types.

$$\begin{array}{lcll} \gamma_0 & = & \mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ \mathbf{0} & \\ & = & \tau & \Pi R\text{-}Zero \\ \gamma_{k+1} & = & \mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)(\mathbf{S}^{k+1}\mathbf{0}) & \\ & = & \sigma[n' := \mathbf{S}^k\mathbf{0}][X := \mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ (\mathbf{S}^k\mathbf{0})] & \Pi R\text{-}Succ \end{array} \tag{2.1}$$

The elements $\gamma_0, \gamma_1, \ldots$ of a recursive type sequence are all distinct, that is, $\gamma_i \neq \gamma_j$ provided $i \neq j$ for all i, j. The condition (a) $X \in \mathrm{fv}(\sigma)$ but $\sigma \neq X$ on the *Tseq* rule guarantees the distinctness of elements of a recursive type sequence.

All $\mathcal{T}^\pi$ types are constructed from either finite types such as A, *Nat*, $\sigma \to \tau$, or recursive type families, such as $\mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ n$. In this way, $\mathcal{T}^\pi$ type families are finite constructions ultimately based on the primitive recursive type sequences. We refer to this property of the $\mathcal{T}^\pi$ type system by saying that all type families are *recursively based.* Type families can be multi-indexed, for example, $\mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ n \to \mathbf{R}\ \tau'\ (\lambda m'.\lambda Y.\sigma')\ m$.

To reach our goal of *reconstructing* recursively based type families, we restrict the kind of recursion allowed.

Definition 4 (Weak $\mathcal{T}^\pi$ types) *A recursive type sequence* $\mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ n$ *is* weak recursive *if and only if*

$$\begin{array}{ll} (a) & X \in \mathrm{fv}(\sigma) \text{ but } \sigma \neq X \\ (b) & n' \notin \mathrm{fv}(\sigma) \\ (c) & n \notin \mathrm{fv}(\sigma, \tau) \end{array}$$

A $\mathcal{T}^\pi$ type γ is said to be weak recursively based, *denoted* $\mathrm{wrb}(\gamma)$, *if and only if every recursive type sequence subexpression is weak recursive.*

The limitation to weak recursively based types is what we mean by the *weak* $\mathcal{T}^\pi$ type system. Weak recursion is specified in the conditions on the *Tseq* typing rule in Figure 2.3.

Condition (a) of Definition 4 guarantees that a sequence of types has distinct elements. Conditions (b) and (c) simplify the kind of type recursion allowed. Condition (b) disallows the dependency of the unfolding expression on the unfolding index. This transforms recursion into *iteration*. Girard, Lafont, and Taylor [GLT89] compare iteration with recursion and show that using iteration alone results in no essential loss in expressive power. Condition (c) enforces a restriction on the nesting of recursion indices. We do not know what restriction this places on expressive power.

Weak recursion is the essential limitation on the type system that we assume to prove the principal types theorem for $\mathcal{T}^\pi$ and to obtain the completeness of the reconstruction algorithm. These restrictions make it possible to *effectively* match primitive recursion schemes as accomplished by the matching algorithm and theorem in Section 3.2. Our restrictions may be too strong, for we have not attempted to prove the matching theorem without the restriction to weak recursion. We leave this investigation for later work.

We will have circumstances in the $\mathcal{T}^\pi$ typing rules for terms where a replacement of a numeral $\mathbf{S}^k\mathbf{0}$ for a free term variable in a type τ will arise. For example, the form $\tau[n := \mathbf{S}^k\mathbf{0}]$ appears in the conclusion of the $\Pi elim$ rule of Figure 2.4 where it is subsequently normalized. These substitution formulas do not yield types, as can be seen by examination of the rules for types in Figure 2.3. We call the term $\mathbf{S}^k\mathbf{0}$ a *projector*. When we substitute projectors in types, the resulting formula contains subexpressions of the form $\mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ (\mathbf{S}^k\mathbf{0})$ that we call *sequence projections*. Let us introduce *extended* weak $\mathcal{T}^\pi$ types, denoted *Type**, that include types resulting from the substitution of projectors. Thus,

$$Type^* = Type \cup \{\tau[n_i := \mathbf{S}^{k_i}\mathbf{0}] \mid \tau \in Type, n_i \in \mathrm{fv}(\tau), \text{ and } i, k_i \in \omega\}$$

An *extended type reduction* of an extended type τ, denoted $\tau \xrightarrow{\text{red}} \sigma$, is the application of one of the sequence projections (Equations 2.1), used left to right, to the extended type τ that results in the extended type σ. An *extended type normalization* of an extended type τ, denoted $\tau\downarrow$, is the result of successively reducing τ until no further reductions apply. There may be many possible ways to reduce a type, with each *reduction path* consisting of a different sequence of one step reductions by one of the reduction rules. A rewrite system is called *normalizing* if there exists a terminating reduction path called a *normalizing path*

for every term. The system is *strongly normalizing* if *every* reduction path terminates. Finally, the system is called *confluent* if every terminating reduction path results in the same normal form.

The following theorems show that extended type normalization always terminates and results in a unique non-extended type in the weak $\mathcal{T}^\pi$ type system.

Theorem 5 (Strong normalization of weak $\mathcal{T}^\pi$ extended types) *For every type ρ in the extended type system, every sequence of reductions for $\rho\downarrow$ terminates.*

Proof Define a complexity measure on weak $\mathcal{T}^\pi$ extended types by assigning ΠR-*Zero* redeces $\mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ \mathbf{0}$ complexity 1 and ΠR-*Succ* redeces $\mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ (\mathbf{S}^{k+1}\mathbf{0})$ complexity $k+2$. There are no other kinds of redeces in weak $\mathcal{T}^\pi$ extended types. Let the complexity of an extended type ρ be the sum of the complexities of all its redeces.

We claim that a reduction of any chosen redex in an extended type ρ strictly decreases its complexity. First note that in weak $\mathcal{T}^\pi$ types, redeces cannot propagate in the subexpression type σ because $n' \notin \mathrm{fv}(\sigma)$. This means the total number of redeces in any reduction step cannot increase. For a ΠR-*Zero* redex $\mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ \mathbf{0}$, the complexity of the result is the complexity of the zero projected type τ, strictly less than the complexity of ρ. For a ΠR-*Succ* redex $\mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ (\mathbf{S}^{k+1}\mathbf{0})$ with complexity $k+2$, we have the same strict decrease in complexity because only one new redex arises and it has complexity $k+1$. Thus every reduction sequence for a weak $\mathcal{T}^\pi$ extended type ρ corresponds to a strictly decreasing sequence of integers. ∎

We know from standard results in term rewriting theory that a strongly normalizing system with no overlapping rules, such as ours, must be confluent. (See Huet [Hue80].) Thus we have a proof of the following theorem.

Theorem 6 (Confluence of weak $\mathcal{T}^\pi$ extended types) *Every extended type τ has a unique normal form $\tau\downarrow$ in the $\mathcal{T}^\pi$ type system.*

For a type τ with $n \in \mathrm{fv}(\tau)$ we call the combined operation $(\tau[n := \mathbf{S}^k\mathbf{0}])\downarrow$ a *normalizing projection* from τ. Normalizing projections show up in the Π*elim* typing rule for $\mathcal{T}^\pi$

terms in the next sub-section. When we project and then normalize like this, the resulting type is unique by the confluence theorem above.

The $\mathcal{T}^\pi$ type formulas are an elaboration of the rules for $\mathcal{T}^\infty$ described earlier in Figure 1.3, Section 1.2. The $\mathcal{T}^\pi$ rules *Tseq* and *Product* correspond to the $\mathcal{T}^\infty$ *Product* rule and explicitly give a recursive *coding* of the type sequences used to construct the $\mathcal{T}^\infty$ products.

2.1.3 Typing Rules

The type system of $\mathcal{T}^\pi$ consists of rules for deriving *judgements* of the form $\Gamma \vdash \delta : \tau$. A *type assignment statement* or just *statement* is a formula $\delta : \tau$ relating the term δ to the type τ.

A statement $\delta : \tau$ is *well typed* or *provable* in context Γ, if the judgement $\Gamma \vdash \delta : \tau$ can be derived according to the typing rules for terms in Figure 2.4. In this case, the term δ is said to be *typable* in context Γ. Judgements are assumed to be derivable and thus signify statements about provability, unless otherwise noted. Such judgements are called *well typings*.

Statements of the form $x : \tau$ are *assumptions*. A *context* is a function $\Gamma : \mathit{Var} \rightarrow \mathit{Types}$ that represents a set of assumptions. A *context extension* is an extension $\Gamma\{x := \tau\}$ which is usually written $\Gamma\{x : \tau\}$ or, if Γ is the empty context, $\{x : \tau\}$. If $\{x : \tau\} \in \Gamma$ then we sometimes say Γ *covers* x, or that x is *covered by* Γ. The union of two contexts, $\Gamma \cup \Gamma'$ is the context Γ extended by every variable assignment in Γ', provided the domains of Γ and Γ' do not intersect.

A *well formed context*, denoted wf(Γ), is defined inductively according to the rules:

(1) wf($\{\}$)

(2) wf($\Gamma\{x : \tau\}$) provided wf(Γ), $\Gamma \vdash \tau : \mathit{Type}$, and fvtm($\tau$) $\subseteq$ dom(Γ)

The next theorem is intended to verify that well formed contexts exactly cover all the free term variables in types, that is, we want each such term variable to be in the domain of the context once, but not more than once. The theorem also insures that there is not circularity in context dependencies.

Theorem 7 (Well formed contexts) *If* $\mathrm{wf}(\Gamma\{x:\tau\})$ *then*

(i) $x \notin \mathrm{dom}(\Gamma)$

(ii) $\mathrm{fvtm}(\tau) \subseteq \mathrm{dom}(\Gamma)$

(iii) $x \notin \mathrm{fv}(\tau)$

Proof Statement (i) follows from the definition of function extension. The proof of (ii) is covered directly the definition of a well formed context, and (iii) Follows from (i) and (ii) directly. ∎

The typing rules for $\mathcal{T}^\pi$ terms are shown in Figure 2.4. For comparison, Figure 2.5 gives the sequent formulation of the typing rules for $\mathcal{T}^\infty$. The rule *Cext* is not necessary in this formulation of the rules, but it is convenient for modularizing proofs in the manner given in the examples (it helps avoid unnecessary context information in sub-proofs). The Π*intro* rule requires a manifest dependency, that is, it can be used only when the $\rightarrow$*intro* rule cannot be used. The Π*intro-seq* typing rule assigns a dependent product to recursive sequences whereas the $\rightarrow$*intro-seq* typing rule assigns an *Arrow*($\rightarrow$) type $Nat \rightarrow \tau$. In the proof of the principal types theorem the $\rightarrow$*intro-seq* typing rule is handled by the unification theorem whereas the Π*intro-seq* rule is handled by the *matching* theorem. (See section 3.2.) The restrictions on the Π*elim* rule mean that dependent types can only be eliminated using closed *Nat* terms.

The following theorems verify some important properties that the type system exhibits. All of the theorems apply to both the weak and non-weak $\mathcal{T}^\pi$ type systems. The first theorem makes sure that the typing rules for terms do not introduce ill-formed contexts.

Theorem 8 (Context preservation) *The typing rules of* $\mathcal{T}^\pi$ *preserve the well formedness of contexts, that is, whenever* $\Gamma \vdash \delta : \tau$ *is derivable according to the* $\mathcal{T}^\pi$ *rules, then* Γ *is a well formed context.*

Proof The proof is by induction over the structure of derivations. The only rules which could result in improperly formed contexts are those that alter the context in their conclusions, namely, the rules *Zero*, *Succ*, *Var*, *Cext*, $\rightarrow$*intro*, Π*intro*, and Π*intro-seq*. In the

$$\Gamma \vdash \mathbf{0} : Nat \qquad \text{if } \mathrm{wf}(\Gamma) \qquad \textit{Zero}$$

$$\Gamma \vdash \mathbf{S} : Nat \to Nat \qquad \text{if } \mathrm{wf}(\Gamma) \qquad \textit{Succ}$$

$$\Gamma\{x:\tau\} \vdash x : \tau \qquad \text{if } \mathrm{wf}(\Gamma\{x:\tau\}) \qquad \textit{Var}$$

$$\frac{\Gamma \vdash \delta : \tau}{\Gamma \cup \Gamma' \vdash \delta : \tau} \qquad \text{if } \mathrm{wf}(\Gamma'), \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \phi \qquad \textit{Cext}$$

$$\frac{\Gamma \vdash \delta' : \tau \qquad \Gamma \vdash \delta : Nat \to \tau \to \tau}{\Gamma \vdash \mathbf{R}\ \delta\ \delta' : Nat \to \tau} \qquad \to \textit{intro-seq}$$

$$\frac{\Gamma\{x:\sigma\} \vdash \delta : \tau}{\Gamma \vdash \lambda x.\delta : \sigma \to \tau} \qquad \text{if } x \notin \mathrm{fv}(\tau) \qquad \to \textit{intro}$$

$$\frac{\Gamma \vdash \delta : \sigma \to \tau \qquad \Gamma \vdash \delta' : \sigma}{\Gamma \vdash \delta\ \delta' : \tau} \qquad \to \textit{elim}$$

$$\frac{\Gamma \vdash \delta' : \rho \qquad \Gamma\{n : Nat\} \vdash \delta : Nat \to \sigma \to \tau}{\Gamma\ \vdash \mathbf{R}\ \delta'\ \delta : \Pi n.\sigma} \quad \text{if} \begin{cases} n \notin \mathrm{fv}(\delta) \\ n \in \mathrm{fv}(\sigma), \mathrm{wrb}(\sigma) \\ (\sigma[n := 0])\downarrow = \rho \\ (\sigma[n := \mathbf{S}\ n])\downarrow = \tau \end{cases} \qquad \Pi\textit{intro-seq}$$

$$\frac{\Gamma\{n : Nat\} \vdash \delta : \tau}{\Gamma \vdash \lambda n.\delta : \Pi n.\tau} \qquad \text{if } n \in \mathrm{fv}(\tau) \qquad \Pi\textit{intro}$$

$$\frac{\Gamma \vdash \delta : \Pi n.\tau \qquad \Gamma \vdash \delta' : Nat}{\Gamma \vdash \delta\ \delta' : (\tau[n := \delta'])\downarrow} \quad \text{if} \begin{cases} \delta'\downarrow = \mathbf{S}^k\mathbf{0} \text{ for } k \in \omega \\ \text{or } \delta' = m \text{ a } Nat \text{ variable} \end{cases} \qquad \Pi\textit{elim}$$

Figure 2.4: Typing Rules for $\mathcal{T}^\pi$ Terms

rules *Zero*, *Succ*, and *Var*, well formedness of the context is a condition on the rule. In the *Cext* rule, Γ is well formed by the induction hypothesis and Γ' is well formed by the condition on the rule, so $(\Gamma \cup \Gamma')$ is well formed because Γ and Γ' do not intersect. Finally, in $\to$*intro*, Π*intro*, and Π*intro-seq* the context in the conclusion is a restriction of the context in the premises so the well formedness of the context in the premises implies the well formedness of the context in the conclusion. ∎

The following free variables theorem insures that all free term variables are covered by the context in derivable typings for $\mathcal{T}^\pi$ terms.

$$\Gamma\{x:\tau\} \vdash x:\tau \qquad \text{if } \mathrm{wf}(\Gamma\{x:\tau\}) \qquad \textit{Var}$$

$$\frac{\Gamma \vdash \delta : \tau}{\Gamma \cup \Gamma' \vdash \delta : \tau} \qquad \text{if } \mathrm{wf}(\Gamma'), \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \phi \qquad \textit{Cext}$$

$$\frac{\Gamma\{x:\sigma\} \vdash \delta : \tau}{\Gamma \vdash \lambda x.\delta : \sigma \to \tau} \qquad \text{if } x \notin \mathrm{fv}(\tau) \qquad \to \textit{intro}$$

$$\frac{\Gamma \vdash \delta : \sigma \to \tau \qquad \Gamma \vdash \delta' : \sigma}{\Gamma \vdash \delta\ \delta' : \tau} \qquad \to \textit{elim}$$

$$\frac{\Gamma \vdash \delta_0 : \tau_0 \quad \Gamma \vdash \delta_1 : \tau_1 \quad \cdots}{\Gamma \vdash (\delta_0, \delta_1, \ldots) : \Pi(\tau_i)} \qquad \text{for } i \in \omega \qquad \Pi \textit{intro}$$

$$\frac{\Gamma \vdash \delta : \Pi(\tau_i) \quad k \in \omega}{\Gamma \vdash \delta_k : \tau_k} \qquad \Pi \textit{elim}$$

Figure 2.5: Typing Rules for $\mathcal{T}^\infty$ Terms in Sequent Formulation

Theorem 9 (Free variables) *Whenever* $\Gamma \vdash \delta : \tau$ *is derivable according to the* $\mathcal{T}^\pi$ *typing rules, then all free term variables in* δ *and* τ *are assigned types in* Γ*, that is,* $\mathrm{fv}(\tau) \subseteq \mathrm{dom}(\Gamma) \cup$ *Tvar and* $\mathrm{fv}(\delta) \subseteq \mathrm{dom}(\Gamma)$.

Proof The proof is by induction on the structure of the derivation of $\Gamma \vdash \delta : \tau$. The *Var* typing rule directly introduces the term variable x, but the context $\Gamma\{x:\tau\}$ assigns x type τ. The $\to$*intro* rule could introduce uncovered free variables in the type σ coming from the context. However, the well formed context theorem guarantees that the free term variables of σ are in the domain of Γ. No other rules can introduce term variables not already covered by the context in the premise of the rule. ∎

We must not have a well typed term whose type is not well formed!

Theorem 10 (Well formed types) *Whenever* $\Gamma \vdash \delta : \tau$ *is derivable according to the* $\mathcal{T}^\pi$ *typing rules, then* τ *is a derivable type in the* $\mathcal{T}^\pi$ *type system.*

Proof The proof is by induction on the structure of the derivation of $\Gamma \vdash \delta : \tau$. The *Zero* and *Succ* rules clearly have resulting types that are in $\mathcal{T}^\pi$ (i.e. derivable by the $\mathcal{T}^\pi$ rules for types). The well formed context condition on the *Var* rule guarantees the

resulting type is in $\mathcal{T}^\pi$. The rules *Cext* and $\rightarrow$*elim* have resulting types in $\mathcal{T}^\pi$ by the induction hypothesis. Each of the rules $\rightarrow$*intro*, $\rightarrow$*intro-seq*, Π*intro*, and Π*intro-seq* have resulting types in $\mathcal{T}^\pi$ by applying a corresponding type rule and appealing to the induction hypothesis. Finally, the Π*elim* rule has a resulting type in $\mathcal{T}^\pi$ by the induction hypothesis and Theorem 6. ∎

Finally, we conjecture the following standard theorem showing the invariance of typing under term reduction. This theorem is not used in this thesis and we claim its proof closely follows similar proofs of subject reduction properties for the simply typed lambda calculus. (See, for example, Barendregt [Bar84].)

Conjecture 11 (Subject reduction) *Let $\Gamma \vdash \delta' : \tau$ be derivable according to the $\mathcal{T}^\pi$ typing rules and suppose $\delta' \stackrel{\text{red}}{\rightarrow} \delta$. Then $\Gamma \vdash \delta : \tau$ is derivable according to the $\mathcal{T}^\pi$ typing rules.*

2.1.4 Strong Normalization of $\mathcal{T}^\pi$ Terms

A $\mathcal{T}^\pi$ term is in normal form if it contains no unreduced η, β, *R-Succ*, or *R-Zero* redeces. In our rewrite system for terms, there can be many possible normalizing reduction paths for a term.

We conjecture that terms in the weak $\mathcal{T}^\pi$ system are strongly normalizing and confluent, that is, every well typed term reduces to a unique normal form. Our conjecture is based on the similarity of this system with Gödel's system T and the system $\mathcal{T}^\infty$, both of which are strongly normalizing and confluent. See Girard about strong normalization of the system T [GLT89], and see Tait [Tai65] and Martin-Löf [Mar72a] about strong normalization of $\mathcal{T}^\infty$.

Conjecture 12 (Strong normalization of weak $\mathcal{T}^\pi$) *Well typed weak $\mathcal{T}^\pi$ terms are strongly normalizing.*

Theorem 13 (Confluence of weak $\mathcal{T}^\pi$) *If terms in the weak $\mathcal{T}^\pi$ system are strongly normalizing, then every well typed term δ in the weak $\mathcal{T}^\pi$ type system has a unique normal form $\delta\downarrow$.*

Proof Again, as with types, we know from standard results in term rewriting theory that a strongly normalizing system with no overlapping rules, such as ours, must be confluent. (See Huet [Hue80].) ∎

2.2 Dependent Typing Examples

We give three examples illustrating formal typing in the $\mathcal{T}^{\pi}$ type system. First we present a detailed example of the typing of the *taut* function introduced in Chapter 1. The *taut* function is an example that is not typable in the ML type system. Secondly, we sketch how to type tupler and projection operations consistently dependent on the size of the tuples. The tupler and projection examples are intended as evidence that the type system of this paper is applicable to typing programming constructs such as arrays and matrices with dependent types. Complete details of the latter two examples are included in Appendix A.1 and Appendix A.2.

A derivation begins with a list of assumptions followed by a numbered sequence of derivation steps. Each step of the derivation is labeled by either (a) the name of a typing rule or (b) the name of a type equality rule (implying a type reduction sequence). The symbol $\equiv$ is used only as a syntactic shorthand to make the derivations more readable. There is no formal logic step associated with a syntactic replacement in a derivation. The typing rules are given in Figure 2.4 Section 2.1. The premises of a derivation step are identified as part of the label; if no premise is listed, then the rule is an axiom or its single premise is the previous step.

The key step in the typing derivation for *taut* uses the primitive recursion rule Π*intro-seq* of Figure 2.4, Section 2.1, introducing a sequence of terms having a weak recursively based type. The first premise of the Π*intro-seq* rule is the zero or base case and the second premise is the successor or induction step case. The Π*intro-seq* rule is used in the last step of the *taut* derivation to introduce the final **R** combination for the entire program. The derivation of *taut* assumes derivations for *true*, *false*, and &. To guarantee well formed contexts Γ, our derivation also assumes (*Boolfam* n) $\rightarrow$ *Bool* is the well formed dependent product type discussed earlier in Section 1.2. We have assumed the abbreviations *Boolfam*

and *taut* from that section.

$$\begin{aligned} \mathit{Boolfam} &\equiv \mathbf{R}\ \mathit{Bool}\ (\lambda n.\lambda X.\mathit{Bool} \to X) \\ \mathit{taut} &\equiv \mathbf{R}\ (\lambda f.f)(\lambda n'.\lambda p.\lambda f.((p\ (f\ \mathit{true}))\&(p\ (f\ \mathit{false})))) \end{aligned}$$

Theorem

$$\mathit{taut} \ : \ \Pi n.(\mathit{Boolfam}\ n) \to \mathit{Bool}$$

Assumptions

$$\begin{aligned} A_1. &\quad \vdash \mathit{true} : \mathit{Bool} \\ A_2. &\quad \vdash \mathit{false} : \mathit{Bool} \\ A_3. &\quad \vdash \& : \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\ A_4.\{n : \mathit{Nat}\} &\quad \vdash (\mathit{Boolfam}\ n) \to \mathit{Bool} : \mathit{Type} \end{aligned}$$

Proof First derive the base case.

1. $\{f : \mathit{Bool}\} \vdash f \ : \ \mathit{Bool}$ (*Var*)

2. $\vdash \lambda f.f \ : \ \mathit{Bool} \to \mathit{Bool}$ ($\to$ *intro*)

In the successor case, first derive types for f, *true*, *false*, f *true*, and f *false*.

3. $\{n : \mathit{Nat}\}\{n' : \mathit{Nat}\}\{p : (\mathit{Boolfam}\ n) \to \mathit{Bool}\}$
   $\{f : \mathit{Bool} \to (\mathit{Boolfam}\ n)\} \vdash f \ : \ \mathit{Bool} \to (\mathit{Boolfam}\ n)$ (*Var*)

4. $\{n : \mathit{Nat}\}\{n' : \mathit{Nat}\}\{p : (\mathit{Boolfam}\ n) \to \mathit{Bool}\}$
   $\{f : \mathit{Bool} \to (\mathit{Boolfam}\ n)\} \vdash \mathit{true} \ : \ \mathit{Bool}$ (*Cext* A_1)

5. $\{n : \mathit{Nat}\}\{n' : \mathit{Nat}\}\{p : (\mathit{Boolfam}\ n) \to \mathit{Bool}\}$
   $\{f : \mathit{Bool} \to (\mathit{Boolfam}\ n)\} \vdash \mathit{false} \ : \ \mathit{Bool}$ (*Cext* A_2)

6. $\{n : \mathit{Nat}\}\{n' : \mathit{Nat}\}\{p : (\mathit{Boolfam}\ n) \to \mathit{Bool}\}$
   $\{f : \mathit{Bool} \to (\mathit{Boolfam}\ n)\} \vdash f\ \mathit{true} \ : \ \mathit{Boolfam}\ n$ ($\to$ *elim* 3 4)

7. $\{n : \mathit{Nat}\}\{n' : \mathit{Nat}\}\{p : (\mathit{Boolfam}\ n) \to \mathit{Bool}\}$
   $\{f : \mathit{Bool} \to (\mathit{Boolfam}\ n)\} \vdash f\ \mathit{false} \ : \ \mathit{Boolfam}\ n$ ($\to$ *elim* 3 5)

Using an element p belonging to the presumed n^{th} projection of the recursion scheme $(Boolfam\ n) \rightarrow Bool$, derive the terms $p\ (f\ true)$ and $p\ (f\ false)$, and combine them with the & operator.

8. $\{n : Nat\}\{n' : Nat\}\{p : (Boolfam\ n) \rightarrow Bool\}$
   $\{f : Bool \rightarrow (Boolfam\ n)\}$
   $\vdash p \quad : \quad (Boolfam\ n) \rightarrow Bool \qquad (Var)$

9. $\{n : Nat\}\{n' : Nat\}\{p : (Boolfam\ n) \rightarrow Bool\}$
   $\{f : Bool \rightarrow (Boolfam\ n)\}$
   $\vdash p\ (f\ true) \quad : \quad Bool \qquad (\rightarrow elim\ 8\ 6)$

10. $\{n : Nat\}\{n' : Nat\}\{p : (Boolfam\ n) \rightarrow Bool\}$
    $\{f : Bool \rightarrow (Boolfam\ n)\}$
    $\vdash p\ (f\ false) \quad : \quad Bool \qquad (\rightarrow elim\ 8\ 7)$

11. $\{n : Nat\}\{n' : Nat\}\{p : (Boolfam\ n) \rightarrow Bool\}$
    $\{f : Bool \rightarrow (Boolfam\ n)\}$
    $\vdash \& \quad : \quad Bool \rightarrow Bool \rightarrow Bool \qquad (Cext\ A_3)$

12. $\{n : Nat\}\{n' : Nat\}\{p : (Boolfam\ n) \rightarrow Bool\}$
    $\{f : Bool \rightarrow (Boolfam\ n)\}$
    $\vdash (p\ (f\ true))\ \&\ (p\ (f\ false)) \quad : \quad Bool \qquad (\rightarrow elim\ 11\ 9\ 10)$

Finally, abstract on the arguments f, n, p and use the $\Pi intro\text{-}seq$ rule on the zero and

successor case proofs to obtain the final typing of *taut*.

13. $\{n : Nat\}\{n' : Nat\}\{p : (Boolfam\ n) \to Bool\}$
$\vdash \lambda f.((p\ (f\ true))\&\ (p\ (f\ false)))$
$: (Bool \to (Boolfam\ n)) \to Bool$ $(\to intro)$

14. $\{n : Nat\} \vdash \lambda n'.\lambda p.\lambda f.((p\ (f\ true))\&(p\ (f\ false)))$
$: Nat \to ((Boolfam\ n) \to Bool)$
$\to ((Bool \to (Boolfam\ n)) \to Bool)$ $(\to intro, \to intro)$

15. $\vdash \mathbf{R}\ (\lambda f.f)(\lambda n'.\lambda p.\lambda f.((p\ (f\ true))\&(p\ (f\ false))))$
$: (\Pi n.(Boolfam\ n) \to Bool)$ $(\Pi intro\text{-}seq\ 2\ 14)$
$\vdash taut\ : (\Pi n.(Boolfam\ n) \to Bool)$ $(\equiv taut)$
where $(((Boolfam\ n) \to Bool)[n := \mathbf{0}])\downarrow$
$= Bool \to Bool$
and $(((Boolfam\ n) \to Bool)[n := \mathbf{S}\ n])\downarrow$
$= (Bool \to (Boolfam\ n)) \to Bool$

∎

In our second and third examples we sketch how to type tupler and projection operations consistently dependent on the size of the tuples. The complete derivations for these examples are included in Appendices A.1 and A.2.[2] The representation of tuples is similar to the usual lambda calculus formulation of pairs and projections. (See Hindley and Seldin [HS86].) In the untyped lambda calculus an operator *pair* is defined as

$$pair \equiv \lambda x.\lambda y.\lambda z.z\ x\ y$$

The first two arguments are the elements of the pair and the last argument is a projector 1^{st} or 2^{nd}. Projections on pairs can then be defined as

$$1^{st} \equiv \lambda x.\lambda y.x$$
$$2^{nd} \equiv \lambda x.\lambda y.y$$

[2]The examples of tuplers and projections are derived in the non-weak $\mathcal{T}^{\pi}$ type system.

It is easy to show the following expected interaction between *pair* and the projections 1^{st} and 2^{nd}.

$$\begin{aligned} (pair\ a\ b)\ 1^{\text{st}} &= a \\ (pair\ a\ b)\ 2^{\text{nd}} &= b \end{aligned} \tag{2.2}$$

In our calculus we define tuples using a tuple combinator similar to *pair* but having an extra argument expressing the tuple size. Thus, for example, *pair* in our system is defined as

$$pair \equiv tupler\ 1$$

where a 1-tuple has size zero, a 2-tuple has size 1, and so forth. In general the combinator *tupler* has a tuple length as first argument, tuple elements as middle arguments and the last argument a projector like 1^{st} or 2^{nd} of type *Projector*. A *concrete tuple* refers to a tupler applied to a tuple length and all of the tuple elements.

Informally, the family of tupler terms is defined as shown in the following table. The

$tupler\ 0$	$=$	$\lambda a_0.\lambda z.z\ a_0$	1-*tupler*
$tupler\ 1$	$=$	$\lambda a_0.\lambda a_1.\lambda z.z\ a_0\ a_1$	2-*tupler*
	$\vdots$		
$tupler\ k$	$=$	$\lambda a_0 \ldots \lambda a_k.\lambda z.z\ a_0 \ldots a_k$	$(k+1)$-*tupler*

Table 2.1: A Family of Tupler Terms

following type definitions are useful for typing the tupler. We have introduced a special abbreviation to make the *Projector* type easier to read.

$$\begin{aligned} Projector &\equiv \mathbf{R}\ (A \to A)\ (\lambda k.\lambda X.A \to X) \\ &\equiv \Pi n.(A \to)^n (A \to A) \end{aligned}$$

$$Tuple \equiv \Pi n.(Projector\ n \to A)$$

The following typing is derivable using the non-weak $\mathcal{T}^{\pi}$ typing rules. Again we have

introduced a special abbreviation to make the type of the tupler term easier to read.

$$\begin{aligned} \mathit{tupler} &: \Pi n.\mathbf{R}\ (A \to \mathit{Tuple}\ n)\ (\lambda k.\lambda X.A \to X)\ n \\ &: \Pi n.(A \to)^n (A \to \mathit{Tuple}\ n) \end{aligned}$$

Projections work on concrete tuples and are parameterized by the tuple length in the first argument and the projection number in the second argument. As in *tupler*, a tuple length argument of zero means a 1-tuple, a tuple length argument of one means a 2-tuple, and so forth. Similarly, the first projection is indicated by zero, the second projection by one, and so on. The following table shows the family of projection terms. Any attempt to specify a projection number beyond the end of the tuple yields the last element in the tuple, so for each row k the term in the $(k-1)^{\text{th}}$ column is repeated indefinitely thereafter. Notice that row two contains the usual projections for pairs in columns zero and one.

Tuple size	Projection number				
	0	1	2	3	...
(1-*tuple*) 0	$\lambda z.z$	$\lambda z.z$	$\lambda z.z$	$\lambda z.z$	...
(2-*tuple*) 1	$\lambda z.\lambda y.z$	$\lambda y.\lambda z.z$	$\lambda y.\lambda z.z$	$\lambda y.\lambda z.z$	...
(3-*tuple*) 2	$\lambda z.\lambda x.\lambda y.z$	$\lambda x.\lambda z.\lambda y.z$	$\lambda x.\lambda y.\lambda z.z$	$\lambda x.\lambda y.\lambda z.z$	...
(4-*tuple*) 3	$\lambda z.\lambda w.\lambda x.\lambda y.z$	$\lambda w.\lambda z.\lambda x.\lambda y.z$	$\lambda w.\lambda x.\lambda z.\lambda y.z$	$\lambda w.\lambda x.\lambda y.\lambda z.z$	...
⋮	⋮	⋮	⋮	⋮	⋮

Table 2.2: A family of Projection Terms

The following type is derivable for projection in the non-weak $\mathcal{T}^\pi$ system.

$$\mathit{projection} \ : \ \Pi n.(\mathit{Nat} \to \mathit{Projector}\ n)$$

It is easy to calculate that for pairs in $\mathcal{T}^\pi$, the expected interaction between the tupler and the projections holds just as in Equations 2.2 above.

$$\begin{aligned} (\mathit{tupler}\ 1\ a\ b)(\mathit{projection}\ 1\ 0) &= a \\ (\mathit{tupler}\ 1\ a\ b)(\mathit{projection}\ 1\ 1) &= b \end{aligned}$$

2.3 A Term Model Semantics for $\mathcal{T}^\pi$

In this section we give a model for the (non-weak) $\mathcal{T}^\pi$ type system and prove soundness and completeness of type inference. We base the semantics of $\mathcal{T}^\pi$ on what Albert Meyer

has called an environment model of the untyped lambda calculus [Mey82]. We define what it *means* to be a model for $\mathcal{T}^\pi$ in terms of basic elements and behaviors, that is, the essential properties of *all* models of $\mathcal{T}^\pi$, and then prove the soundness of the $\mathcal{T}^\pi$ type inference system of Figure 2.4 Section 2.1. We then construct a term model and use it in the proof of completeness of the $\mathcal{T}^\pi$ type inference system. In our model the semantic domain D is the set of equivalence classes of convertible terms together with the natural numbers ω and the type domain T is a class of sets of terms. Thus we model $\mathcal{T}^\pi$ terms by numbers or functions, and model $\mathcal{T}^\pi$ types by sets of terms along with the set of natural numbers ω.

Our model construction is motived by Roger Hindley's work [Hin83] on a *simple semantics*[3] for the typed lambda calculus based on sets and functions; we give a *simple semantics* for the $\mathcal{T}^\pi$ system. To understand what it means to be a set and function model we give equations that characterize the appropriate behavior of terms as functions, and types as sets. Thus, for example, if we have a typing statement $\delta : \sigma \to \tau$ for a term δ, then we want the semantic requirements to tell us the behaviorial requirements of δ interpreted as a function between sets corresponding to the types σ and τ. Generally, we interpret typing statements $\delta : \tau$ as set memberships, that is, the meaning of the term δ belongs to the set of terms intended as the meaning of the type τ.

A *decent semantics*, according to Hindley [Hin83] is one in which interconvertible terms are given the same interpretation. The rule *Eq* below equates the types of interconvertible terms. As Hindley does, we add this rule to the type system *before* giving a model. This rule is used to obtain completeness of the inference system with respect to the term model, as we will see in the proof of the completeness theorem later in this section.

$$\frac{\Gamma \vdash \delta' : \tau}{\Gamma \vdash \delta : \tau} \qquad \text{if } \delta' \overset{\text{cnv}}{=} \delta \quad Eq$$

Using the rule *Eq* in a decent semantics makes it is possible to assign a type to a term that is not directly typable otherwise. For example, the term $(\lambda z.\lambda y.y)(\mathbf{0}\ \mathbf{S})$ is typable, but only after conversion to the term $\lambda y.y$.[4] Moreover, the *Eq* rule breaks strong

[3]Hindley's use of the word *simple* here is relative to the other semantics presented in his paper.

[4]The stronger *subject reduction* rule of Theorem 11, Section 2.1 will not allow an untypable term to be

normalization. Consider the equality $x = \mathbf{K}x\Omega$ where $\mathbf{K}$ is the combinator $\lambda x.\lambda y.x$ and Ω is the term $(\lambda x.xx)(\lambda x.xx)$.

Other reasonable semantics do not assign the same interpretation to interconvertible terms, for example, Ohori's semantics for ML polymorphism [Oho89]. In that semantics, type information obtained from terms is significant and used to semantically distinguish ML expressions that are otherwise convertible without type information. Thus, the two terms above would be distinct, with one typable and the other not. Ohori's model makes finer distinctions among terms, not lumping together typable and untypable terms in the same semantic set. Harper and Mitchell [HM93] introduce the concept of *coherence* between models and type reconstruction to capture this notion that distinctions made by type reconstruction show up as semantic distinctions. Ohori's model is coherent while ours and Hindley's decent semantics are most likely not coherent because of the *Eq* rule.

Let us return to the general properties required of a model of the $\mathcal{T}^\pi$ system. Let D, the semantic domain, denote a nonempty set that is as yet unspecified except that it includes the natural numbers ω obeying the Peano axioms. Let T, the type domain, denote the set of all subsets of domain elements D.

An *environment* is a mapping ϵ consisting of two type dependent functions, one from term variables to domain elements and the other from type variables to subsets of D. The actual environment function being applied in any situation depends on the kind of the argument.

$$\begin{aligned} \epsilon : \Pi\chi.\text{case } \chi \text{ of} \\ x : \mathit{Vars} \to D \\ X : \mathit{Tvars} \to T \end{aligned}$$

The metavariable χ is intended as a single symbol ranging over both the metavariables x and X. Our use of a type dependent function to express the environment is really just to avoid an overly cumbersome notation. The set of all environments is denoted Env.

A *term interpretation* is a mapping $[\![\ ]\!] : \text{Env} \to \mathit{Terms} \to D$ assigning domain elements to every term relative to an environment. We introduce three semantic operators on terms,

typable. However, we then will not be able to prove completeness using Hindley's method. The *Eq* rule is a more general rule than subject reduction that we cannot prove for our type system. We will not be concerned with the *Eq* rule outside of semantic considerations.

$1^{st} : D \to D$, and $2^{nd} : D \to D$, and $\bullet : D^2 \to D$. The map 1^{st} is intended to project out the base case of a recursive term sequence, and the map 2^{nd} is intended to yield the functional abstraction component of a recursive term sequence, the one that performs the recursive unfolding step.

Figure 2.6 gives the rules for term interpretations required by all models of $\mathcal{T}^\pi$. The rules *i, iv, vii, viii* and *ix* are standard rules for any lambda model. See Hindley and Seldin [HS86] or Meyer [Mey82] for discussions about lambda calculus models. The requirements *ii* and *iii* are simple extensions to the standard lambda calculus model that characterizes the expected behavior of numerals. (See Friedman's model [Fri75].) The remaining two requirements *v* and *vi* are for characterizing the necessary semantic structure of terms representing sequences. These equations define the behavior of the distinguished semantic elements 1^{st} and 2^{nd} that take apart the semantic elements representing sequences.

A *type interpretation* is a map $[\![\ ]\!] : \text{Env} \to \mathit{Types} \to T$ assigning sets of domain elements as the meaning of type formulas. Figure 2.7 gives the semantic requirements for types in all models of $\mathcal{T}^\pi$. The requirements *i, ii,* and *vii* for type interpretations come from Hindley's model of the simply typed lambda calculus [Hin83]. Requirements *v* and *vi* together define the intended meaning of a recursive type sequence as an inductively defined family $S_1, S_2, \ldots$ of sets of domain elements. The semantic rules *iii* and *iv* each split into two parts, a *uniform condition* on membership and a recursive condition on membership. The uniform condition specifies semantic elements that behave like functions and represent the semantics of lambda abstractions $\lambda x.\delta$. The recursive condition specifies semantic elements that behave like sequences representing the sequence terms $\mathbf{R}\ \delta'\ \delta$. The rule *iii* covers functions and sequences that are non-dependent and rule *iv* covers functions and sequences that are dependent on natural numbers.

In both term and type interpretations we use, respectively, the subscript notations $[\![\ ]\!]_\epsilon$ and $[\![\ ]\!]_\epsilon$ to supply the environment ϵ. Sometimes we leave out the environment argument subscript when it is either empty, not needed, or ranges over all possible environments.

An *R-structure* is a tuple $M = (D, \bullet, 1^{st}, 2^{nd}, [\![\ ]\!], [\![\ ]\!])$ obeying the semantic requirements given in Figures 2.6 and 2.7. An R-structure M is a *model* for $\mathcal{T}^\pi$ when the

$$
\begin{array}{lrcl}
(i) & [\![x]\!]_\epsilon & = & \epsilon(x) \\
(ii) & [\![\mathbf{0}]\!]_\epsilon & = & 0 \\
(iii) & (\forall k \in \omega), [\![\mathbf{S}\ ]\!]_\epsilon \bullet k & = & k+1 \\
(iv) & [\![\delta\ \delta']\!]_\epsilon & = & [\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon \\
(v) & 1^{\text{st}}[\![\mathbf{R}\ \ \delta'\ \delta]\!]_\epsilon & = & [\![\delta']\!]_\epsilon \\
(vi) & 2^{\text{nd}}[\![\mathbf{R}\ \ \delta'\ \delta]\!]_\epsilon & = & [\![\delta]\!]_\epsilon \\
(vii) & \text{if } (\forall x \in \text{fv}(\delta)), \epsilon(x) = \epsilon'(x) & & \text{then } [\![\delta]\!]_\epsilon = [\![\delta]\!]_{\epsilon'} \\
(viii) & \text{if } \delta \overset{\text{cnv}}{=} \delta' & & \text{then } [\![\delta]\!]_\epsilon = [\![\delta']\!]_\epsilon \\
(ix) & (\forall d \in D)[\![\lambda x.\delta]\!]_\epsilon \bullet d & = & [\![\delta]\!]_{\epsilon\{x:=d\}}
\end{array}
$$

Figure 2.6: Semantic Requirements for Term Interpretations.

semantic requirements of terms and types hold for all environments ϵ. In Hindley's terminology [Hin83] the model M is a *simple semantics* for the $\mathcal{T}^\pi$ system.

A model M and an environment ϵ *satisfy* a typing statement $\delta : \tau$ if and only if $[\![\delta]\!]_\epsilon \in [\![\tau]\!]_\epsilon$. A model and an environment satisfy a context Γ if and only if they satisfy every statement in the context. A statement $\delta : \tau$ is *valid in context* Γ, denoted $\Gamma \models [\![\delta]\!] \in [\![\tau]\!]$ if and only if Γ is well formed and every model and environment satisfying Γ also satisfies $\delta : \tau$.

The following lemmas are used in the soundness and completeness proofs.

Lemma 14 (Substitution) *Let M be a model for $\mathcal{T}^\pi$ and ϵ an environment. The various forms of substitution have the following relationships with semantic interpretation.*

$$
\begin{aligned}
[\![\delta[x := \delta']]\!]_\epsilon &= [\![\delta]\!]_{\epsilon\{x:=[\![\delta']\!]_\epsilon\}} \\
[\![\tau[X := \tau']]\!]_\epsilon &= [\![\tau]\!]_{\epsilon\{X:=[\![\tau']\!]_\epsilon\}} \\
[\![(\tau[x := \delta'])]\!]_\epsilon &= [\![\tau]\!]_{\epsilon\{x:=[\![\delta']\!]_\epsilon\}}
\end{aligned}
$$

Proof Each of these substitution equations is proved by a routine structural induction on the term or type into which the substitution occurs (e.g. τ or δ). See, for example, Barendregt [Bar84]. ∎

The following lemma connects the set ω in the domain of a model with the semantics of the numeral terms $\mathbf{S}^k\mathbf{0} \in$ *Terms*.

Lemma 15 (ω-Terms) *Let M be a model for $\mathcal{T}^\pi$ and ϵ an environment. Then for all $k \in \omega$, there exists a term $\mathbf{S}^k\mathbf{0}$ such that $[\![\mathbf{S}^k\mathbf{0}]\!]_\epsilon = k \in \omega$.*

$$
\begin{array}{lll}
(i) & [\![X]\!]_\epsilon & = \epsilon(X) \\
(ii) & [\![Nat]\!]_\epsilon & = \omega \\
(iii) & [\![\sigma \to \tau]\!]_\epsilon & = \{f \in D \mid (\forall d \in [\![\sigma]\!]_\epsilon), f \bullet d \in [\![\tau]\!]_\epsilon\} \qquad \textit{Uniform condition} \\
 & & \bigcup \{h \in D \mid 1^{\text{st}}h \in [\![\tau]\!]_\epsilon \text{ and} \qquad \textit{Recursive condition} \\
 & & (\forall k' \in \omega)(\forall g \in [\![\tau]\!]_\epsilon)(2^{\text{nd}}h) \bullet k' \bullet g \in [\![\tau]\!]_\epsilon \\
 & & \text{where } \sigma = Nat \text{ so } [\![\sigma]\!]_\epsilon = \omega\} \\
(iv) & [\![\Pi n.\tau]\!]_\epsilon & = \{h \in D \mid (\forall k \in \omega), h \bullet k \in [\![\tau]\!]_{\epsilon\{n:=k\}}\} \qquad \textit{Uniform condition} \\
 & & \bigcup \{h \in D \mid (\forall k \in \omega)(1^{\text{st}}h \in [\![\tau]\!]_{\epsilon\{n:=0\}} \qquad \textit{Recursive condition} \\
 & & \text{and } (\forall k' \in \omega)(\forall g \in [\![\tau]\!]_{\epsilon\{n:=k\}}) \\
 & & (2^{\text{nd}}h) \bullet k' \bullet g \in [\![\tau]\!]_{\epsilon\{n:=k+1\}})\}
\end{array}
$$

$$
\begin{array}{ll}
(v) & [\![\mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ n]\!]_{\epsilon\{n:=0\}} \\
 & \quad = [\![\tau]\!]_{\epsilon\{n:=0\}} \\
(vi) & (\forall k \in \omega)\ [\![\mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ n]\!]_{\epsilon\{n:=k+1\}} \\
 & \quad = [\![\sigma]\!]_{\epsilon\{n:=k+1\}\{n':=k\}\{X:=S_k\}} \\
 & \quad \text{where } S_k = [\![\mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ n]\!]_{\epsilon\{n:=k\}} \\
(vii) & \text{if } (\forall \chi \in \text{fv}(\delta)), \epsilon(\chi) = \epsilon'(\chi) \text{ then } [\![\tau]\!]_\epsilon = [\![\tau]\!]_{\epsilon'}
\end{array}
$$

Figure 2.7: Semantic Requirements for Type Interpretations.

Proof The proof is by a simple induction on $k \in \omega$. In the base case $0 \in \omega$ we have the term $[\![\mathbf{0}]\!] = [\![\mathbf{S}^{\,0}\mathbf{0}]\!] = 0 \in \omega$. For the induction step, suppose that for all $k \in \omega$, there exists a term $\mathbf{S}^{\,k}\mathbf{0}$ such that $[\![\mathbf{S}^{\,k}\mathbf{0}]\!] = k \in \omega$. Then for $k+1$, create the term $\mathbf{S}\ (\mathbf{S}^k\mathbf{0}) = \mathbf{S}^{k+1}\mathbf{0}$. The semantics of terms together with the induction hypothesis tells us that $[\![\mathbf{S}^{k+1}\mathbf{0}]\!] = [\![\mathbf{S}\ (\mathbf{S}^k\mathbf{0})]\!] = [\![\mathbf{S}\ ]\!] \bullet [\![\mathbf{S}^k\mathbf{0}]\!] = [\![\mathbf{S}\ ]\!] \bullet k = k+1$. ∎

In Section 2.1.2 we showed that a fundamental property of the $\mathcal{T}^\pi$ type system in comparison with the $\mathcal{T}^\infty$ system is the *coding* of type sequences. We can now see how this is modeled in terms of recursive families of sets. Equation 2.1 in Section 2.1 shows that the coding of a recursive type sequence defines a family of types

$$(\forall k \in \omega) \qquad \gamma_k = \mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ (\mathbf{S}^k\mathbf{0})$$

For a model M and environment ϵ, the *interpretation* of such a recursive sequence is given by the family of sets

$$(\forall k \in \omega) \qquad [\![\mathbf{R}\ \tau\ (\lambda n'.\lambda X.\sigma)\ n]\!]_{\epsilon\{n:=k\}}$$

where these sets are defined recursively, as shown in the interpretation of types, rules (v) and (vi).

Our *actual* model obeying all the semantic rules is a term model having as domain elements the natural numbers plus equivalence classes of terms under conversion. Thus, let $|\delta| = \{\delta' \mid \delta' \stackrel{\text{cnv}}{=} \delta\}$ and define $D = \omega \cup \{|\delta| \mid \delta \in \mathit{Terms}\}$. Define the distinguished domain operators and elements as follows:

$$\begin{aligned} |\delta| \bullet |\delta'| &= |\delta\ \delta'| \\ 1^{\text{st}}|\mathbf{R}\ \ \delta'\ \delta| &= |\delta'| \\ 2^{\text{nd}}|\mathbf{R}\ \ \delta'\ \delta| &= |\delta| \\ (\forall k \in \omega), |\mathbf{S}\ ^k\mathbf{0}| &= k \end{aligned}$$

The term interpretation map is defined by the following rule.

$$\forall x_i \in \text{fv}(\delta), [\![\delta]\!]_\epsilon = |\delta[x_i := \epsilon(x_i)]\ | \tag{2.3}$$

We can assume a simple environment ϵ_0 defined by

$$\begin{aligned} (\forall x \in \mathit{Var}), \epsilon_0(x) &= |x| \\ (\forall X \in \mathit{Tvar}), \epsilon_0(X) &= \{|\delta| \mid \Gamma \vdash \delta : X \text{ for some } \Gamma\} \end{aligned}$$

In this environment, $[\![\delta]\!]_{\epsilon_0} = |\delta|$.

We will prove soundness of inference for all models and environments and then use our *actual* model and simple environment ϵ_0 to show the completeness of inference.[5]

Theorem 16 (Soundness) *If $\Gamma \vdash \delta : \tau$ is a well typing, then $\Gamma \models [\![\delta]\!] \in [\![\tau]\!]$ in the simple semantics for $\mathcal{T}^\pi$.*

Proof The proof is by induction over the structure of the derivation of the typing $\Gamma \vdash \delta : \tau$. According to the Context Preservation Theorem 8, we can assume that Γ is well formed for any derivable typing. We will show that each typing rule in $\mathcal{T}^\pi$ preserves the soundness of inference.

[5]The proof of completeness begins by assuming a validity statement true in all models and environments so we are entitled to choose any particular environment and model to show there is a corresponding well-typing derivation.

Zero. Suppose the last step in the derivation uses the *Zero* rule and the final sequent is $\Gamma \vdash \mathbf{0} : Nat$. Then $\Gamma \models [\![\mathbf{0}]\!]_\epsilon \in [\![Nat]\!]_\epsilon$ because $[\![\mathbf{0}]\!] = 0 \in \omega$ and $\omega = [\![Nat]\!]$.

Succ. Suppose the last step in the derivation uses the *Succ* rule and the final sequent is $\Gamma \vdash \mathbf{S} : Nat \to Nat$. According to the definition $[\![\mathbf{S}]\!]$ we have

$$(\forall k \in \omega)[\![\mathbf{S}]\!]_\epsilon \bullet k = k + 1$$

Because $k + 1 \in [\![Nat]\!]_\epsilon$ and $\omega = [\![Nat]\!]_\epsilon$, we know

$$(\forall k \in [\![Nat]\!]_\epsilon)\ [\![\mathbf{S}]\!]_\epsilon \bullet k \in [\![Nat]\!]_\epsilon$$

This is the definition of $[\![\mathbf{S}]\!]_\epsilon \in [\![Nat \to Nat]\!]_\epsilon$. Therefore,

$$\Gamma \models [\![\mathbf{S}]\!]_\epsilon \in [\![Nat \to Nat]\!]_\epsilon$$

Var. Suppose the last step in the derivation uses the *Var* rule and the final sequent is $\Gamma\{x : \tau\} \vdash x : \tau$ where $\Gamma\{x := \tau\}$ is a well formed context. Then if ϵ is an environment satisfying $\Gamma\{x : \tau\}$ then we know $[\![x]\!]_\epsilon \in [\![\tau]\!]_\epsilon$ and thus $\Gamma\{x : \tau\} \models [\![x]\!]_\epsilon \in [\![\tau]\!]_\epsilon$.

$\to$ *intro-seq.* Suppose the last step in the derivation uses the $\to$ *intro-seq* and the final sequent is $\Gamma \vdash \mathbf{R}\ \delta'\ \delta : Nat \to \tau$. The last step of the derivation must have premises

$$\Gamma \vdash \delta' : \tau$$
$$\Gamma \vdash \delta : Nat \to \tau \to \tau$$

From the induction hypothesis we have

$$\Gamma \models [\![\delta']\!]_\epsilon \in [\![\tau]\!]_\epsilon \tag{2.4}$$
$$\Gamma \models [\![\delta]\!]_\epsilon \in [\![Nat \to \tau \to \tau]\!]_\epsilon \tag{2.5}$$

According to the semantic interpretation of *Nat* and arrow types the validity statement 2.5 means

$$(\forall k' \in \omega)(\forall g \in [\![\tau]\!]_\epsilon)\ \Gamma \models [\![\delta]\!]_\epsilon \bullet k' \bullet g \in [\![\tau]\!]_\epsilon \tag{2.6}$$

Construct the term $\mathbf{R}\ \delta'\ \delta$ so that

$$1^{\text{st}}[\![\mathbf{R}\ \delta'\ \delta]\!]_\epsilon = [\![\delta']\!]_\epsilon$$
$$2^{\text{nd}}[\![\mathbf{R}\ \delta'\ \delta]\!]_\epsilon = [\![\delta]\!]_\epsilon$$

Using the term $\mathbf{R}\ \delta'\ \delta$ and these equations, combine 2.4 and 2.6 to get

$$\begin{array}{rl} & \Gamma \models 1^{\text{st}}[\![\mathbf{R}\ \delta'\ \delta]\!]_\epsilon \in [\![\tau]\!]_\epsilon \\ (\forall k' \in \omega)(\forall g \in [\![\tau]\!]_\epsilon) & \Gamma \models 2^{\text{nd}}[\![\mathbf{R}\ \delta'\ \delta]\!]_\epsilon \bullet k' \bullet g \in [\![\tau]\!]_\epsilon \end{array} \tag{2.7}$$

Finally, the validity statements 2.7 form the definition of our desired conclusion according to the *recursive condition* of membership in arrow types.

$$\Gamma \models [\![\mathbf{R}\ \delta'\ \delta]\!]_\epsilon \in [\![Nat \to \tau]\!]_\epsilon$$

$\to$*intro.* Suppose the last step of the derivation uses the $\to$*intro* rule and the final sequent is $\Gamma \vdash \lambda x.\delta : \sigma \to \tau$. The last step of this derivation must have premise

$$\Gamma\{x : \sigma\} \vdash \delta : \tau \qquad x \notin \text{fv}(\tau)$$

From the induction hypothesis we have

$$\Gamma\{x : \sigma\} \models [\![\delta]\!]_\epsilon \in [\![\tau]\!]_\epsilon \tag{2.8}$$

Now any environment $\epsilon = \epsilon'$ satisfying the left side must have $[\![x]\!]_{\epsilon'} \in [\![\sigma]\!]_{\epsilon'}$, that is, $\epsilon'(x) = d \in [\![\sigma]\!]_{\epsilon'}$ by the definition of interpretation for variables x. Therefore we can let $\epsilon' = \epsilon\{x := d\}$ with $d \in [\![\sigma]\!]_\epsilon$ and restate 2.8 as

$$(\forall d \in [\![\sigma]\!]_\epsilon)\ \Gamma \models [\![\delta]\!]_{\epsilon\{x:=d\}} \in [\![\tau]\!]_\epsilon \tag{2.9}$$

Note that $[\![\tau]\!]_{\epsilon\{x:=d\}} = [\![\tau]\!]_\epsilon$ because $x \notin \text{fv}(\tau)$. Using the semantic rule (ix) for terms $(\forall d \in D)[\![\lambda x.\delta]\!]_\epsilon \bullet d = [\![\delta]\!]_{\epsilon\{x:=d\}}$ statement 2.9 becomes

$$(\forall d \in [\![\sigma]\!]_\epsilon)\ \Gamma \models [\![\lambda x.\delta]\!]_\epsilon \bullet d \in [\![\tau]\!]_\epsilon \tag{2.10}$$

This equation is the semantic definition of arrow membership in $[\![\sigma \to \tau]\!]_\epsilon$ by term $[\![\lambda x.\delta]\!]_\epsilon$, therefore we have our desired conclusion

$$\Gamma \models [\![\lambda x.\delta]\!]_\epsilon \in [\![\sigma \to \tau]\!]_\epsilon$$

$\to$*elim.* Suppose the last step of the derivation uses the $\to$*elim* rule and the final sequent is $\Gamma \vdash \delta\ \delta' : \tau$. The last step of the derivation must have premises

$$\Gamma \vdash \delta : \sigma \to \tau$$

$$\Gamma \vdash \delta' : \sigma$$

From the induction hypothesis

$$\Gamma \models [\![\delta]\!]_\epsilon \in [\![\sigma \to \tau]\!]_\epsilon \tag{2.11}$$

$$\Gamma \models [\![\delta']\!]_\epsilon \in [\![\sigma]\!]_\epsilon \tag{2.12}$$

The semantic interpretation for arrow types gives us two possibilities for interpreting the operator $[\![\delta]\!]$ in the case that $\sigma = Nat$. First suppose that σ is *not Nat* or else if σ is *Nat* then δ is not a recursive term of the form $\mathbf{R}\ \delta'\ \delta$. Then according to the *uniform condition* of arrow membership of $[\![\delta]\!]_\epsilon$ we have

$$(\forall d \in [\![\sigma]\!]_\epsilon)\ \Gamma \models [\![\delta]\!]_\epsilon \bullet d \in [\![\tau]\!]_\epsilon \tag{2.13}$$

For d ranging over $[\![\delta']\!]_\epsilon$ for all terms δ', statement 2.13 can be stated as

$$(\forall \delta')\ \Gamma \models [\![\delta']\!]_\epsilon \in [\![\sigma]\!]_\epsilon \text{ implies } \Gamma \models [\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon \in [\![\tau]\!]_\epsilon \tag{2.14}$$

Using statement 2.12 to discharge the implication 2.14 we get

$$\Gamma \models [\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon \in [\![\tau]\!]_\epsilon$$

Finally, according to the semantic interpretation of term application, we have the equality $[\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon = [\![\delta\ \delta']\!]_\epsilon$, therefore we can assert our desired conclusion.

$$\Gamma \models [\![\delta\ \delta']\!]_\epsilon \in [\![\tau]\!]_\epsilon$$

Now consider the case where σ is *Nat* and δ is a term of the form $\mathbf{R}\ \delta_1\ \delta_2$ so that $1^{\text{st}}[\![\mathbf{R}\ \delta_1\ \delta_2]\!]_\epsilon = [\![\delta_1]\!]_\epsilon$ and $2^{\text{nd}}[\![\mathbf{R}\ \delta_1\ \delta_2]\!]_\epsilon = [\![\delta_2]\!]_\epsilon$. According to the *recursive condition* of arrow membership that applies in this case we have

$$\begin{gathered} \Gamma \models 1^{\text{st}}[\![\delta]\!]_\epsilon \in [\![\tau]\!]_\epsilon \\ (\forall k' \in \omega)(\forall g \in [\![\tau]\!]_\epsilon)\quad \Gamma \models (2^{\text{nd}}[\![\delta]\!]_\epsilon) \bullet k' \bullet g \in [\![\tau]\!]_\epsilon \end{gathered} \tag{2.15}$$

According to the *R-Zero* and *R-Succ* conversion rules on terms, the semantic requirements of the 1^{st} and 2^{nd} operators on terms, and the semantics of term application we have the

equations

$$\begin{aligned}
[\![\delta]\!]_\epsilon \bullet 0 &= [\![\delta\ \mathbf{0}]\!]_\epsilon \\
&= [\![(\mathbf{R}\ \delta_1\ \delta_2)(\mathbf{0})]\!]_\epsilon \\
&= [\![\delta_1]\!]_\epsilon \\
&= 1^{\mathrm{st}}[\![\delta]\!]_\epsilon \\
(\forall k \in \omega)\ [\![\delta]\!]_\epsilon \bullet (k+1) &= [\![\delta]\!]_\epsilon \bullet [\![\mathbf{S}^{k+1}\mathbf{0}]\!]_\epsilon \\
&= [\![\delta\ (\mathbf{S}^{k+1}\mathbf{0})]\!]_\epsilon \\
&= [\![(\mathbf{R}\ \delta_1\ \delta_2)(\mathbf{S}^{k+1}\mathbf{0})]\!]_\epsilon \\
&= [\![\delta_2\ (\mathbf{S}^k\mathbf{0})\ (\delta\ (\mathbf{S}^k\mathbf{0}))]\!]_\epsilon \\
&= [\![\delta_2]\!]_\epsilon \bullet k \bullet ([\![\delta]\!]_\epsilon \bullet k) \\
&= (2^{\mathrm{nd}}[\![\delta]\!]_\epsilon) \bullet k \bullet ([\![\delta]\!]_\epsilon \bullet k)
\end{aligned}$$

Thus, using these equations and assuming $g = [\![\delta]\!]_\epsilon \bullet k$ we can restate 2.15 as

$$\Gamma \models [\![\delta]\!]_\epsilon \bullet 0 \in [\![\tau]\!]_\epsilon$$

$$\begin{aligned}
(\forall k \in \omega)\quad & \Gamma \models [\![\delta]\!]_\epsilon \bullet k \in [\![\tau]\!]_\epsilon \text{ implies} \\
& \Gamma \models [\![\delta]\!]_\epsilon \bullet (k+1) \in [\![\tau]\!]_\epsilon
\end{aligned} \tag{2.16}$$

By induction over $k \in \omega$ and then replacing $k \in \omega$ by $[\![\delta']\!]_\epsilon$ we can transform 2.16 into

$$\begin{aligned}
(\forall \delta')\quad & \Gamma \models [\![\delta']\!]_\epsilon \in [\![Nat]\!]_\epsilon \text{ implies} \\
& \Gamma \models [\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon \in [\![\tau]\!]_\epsilon
\end{aligned} \tag{2.17}$$

We know that $\sigma = Nat$ so using the statement 2.12 to discharge the implication in 2.17 above we can assert

$$\Gamma \models [\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon \in [\![\tau]\!]_\epsilon \tag{2.18}$$

Using the semantics of term application $[\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon = [\![\delta\ \delta']\!]_\epsilon$, we can transform the statement 2.18 into our desired conclusion

$$\Gamma \models [\![\delta\ \delta']\!]_\epsilon \in [\![\tau]\!]_\epsilon$$

$\Pi intro$. Assume the last step of the derivation uses the $\Pi intro$ rule and that the final sequent is $\Gamma \vdash \lambda n.\delta : \Pi n.\tau$ where $n \in \mathrm{fv}(\tau)$. The last step of the derivation must have premise

$$\Gamma\{n : Nat\} \vdash \delta : \tau \qquad n \in \mathrm{fv}(\tau)$$

From the induction hypothesis

$$\Gamma\{n : Nat\} \models [\![\delta]\!]_\epsilon \in [\![\tau]\!]_\epsilon \qquad n \in \mathrm{fv}(\tau) \tag{2.19}$$

Now any environment $\epsilon = \epsilon'$ satisfying the left side must have $[\![n]\!]_{\epsilon'} \in [\![Nat]\!]_{\epsilon'}$, that is, $\epsilon'(x) = d \in [\![Nat]\!]_{\epsilon'}$ with $[\![Nat]\!]_{\epsilon'} = \omega$ by the definition of interpretation for variables n. Therefore we can let $\epsilon' = \epsilon\{n := k\}$ with $d \in \omega$ and restate 2.19 as

$$(\forall k \in \omega)\ \Gamma \models [\![\delta]\!]_{\epsilon\{n:=k\}} \in [\![\tau]\!]_{\epsilon\{n:=k\}} \tag{2.20}$$

This case is very similar to the $\rightarrow$*intro* case. However, here we know that $n \in \mathrm{fv}(\tau)$, so that is why the interpretation $[\![\tau]\!]$ uses the extended environment $\epsilon\{n := [\![\delta']\!]_\epsilon\}$.

Using the semantic rule (ix) for terms, $(\forall d \in D)[\![\lambda x.\delta]\!]_\epsilon \bullet d = [\![\delta]\!]_{\epsilon\{x:=d\}}$, and remembering that $\omega \subseteq D$, the statement 2.20 becomes

$$(\forall k \in \omega)\ \Gamma \models [\![\lambda n.\delta]\!]_\epsilon \bullet k \in [\![\tau]\!]_{\epsilon\{n:=k\}} \tag{2.21}$$

This equation is the semantic definition of product membership in $[\![\Pi n.\tau]\!]_\epsilon$ by term $[\![\lambda n.\delta]\!]_\epsilon$, therefore we have our desired conclusion[6]

$$\Gamma \models [\![\lambda n.\delta]\!]_\epsilon \in [\![\Pi n.\tau]\!]_\epsilon$$

Π*elim*. Suppose the last step of the derivation uses the Π*elim* rule and the last sequent is $\Gamma \vdash \delta\ \delta' : (\tau[n := \delta'])\downarrow$ with $n \in \mathrm{fv}(\tau)$. The last step of the derivation must have premises

$$\Gamma \vdash \delta : \Pi n.\tau$$
$$\Gamma \vdash \delta' : Nat$$

From the induction hypothesis

$$\Gamma \models [\![\delta]\!]_\epsilon \in [\![\Pi n.\tau]\!]_\epsilon \tag{2.22}$$

$$\Gamma \models [\![\delta']\!]_\epsilon \in [\![Nat]\!]_\epsilon \tag{2.23}$$

[6]Note that we do not suppose here that dependent products can take on *recursive elements* of product types according to the *recursive condition* of product membership. Recursive elements of products come from the Π*intro-seq* rule below.

The semantic interpretation of the product type membership stated in validity statement 2.22 above gives us two possibilities for the operator $[\![\delta]\!]_\epsilon \in [\![\Pi n.\tau]\!]_\epsilon$.

First, according to the *uniform condition* of the interpretation of product membership, operator $[\![\delta]\!]_\epsilon$ must obey

$$(\forall k \in \omega)\ \Gamma \models [\![\delta]\!]_\epsilon \bullet k \in [\![\tau]\!]_{\epsilon\{n:=k\}} \tag{2.24}$$

Let $k = [\![\delta']\!]_\epsilon \in \omega$ and restate 2.24 as the implication

$$(\forall \delta')\ \Gamma \models [\![\delta']\!]_\epsilon \in [\![Nat]\!]_\epsilon \text{ implies } \Gamma \models [\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon \in [\![\tau]\!]_{\epsilon\{n:=[\![\delta']\!]_\epsilon\}} \tag{2.25}$$

Using the statement 2.23 to discharge the implication 2.25 we can assert

$$\Gamma \models [\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon \in [\![\tau]\!]_{\epsilon\{n:=[\![\delta']\!]_\epsilon\}}$$

By the substitution lemma for types, the semantic equivalence of convertible types, and the semantics of term application we have the equalities

$$\begin{aligned} [\![\tau]\!]_{\epsilon\{n:=[\![\delta']\!]_\epsilon\}} &= [\![(\tau[n := \delta'])]\!]_\epsilon \\ &= [\![(\tau[n := \delta'])\downarrow]\!]_\epsilon \\ [\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon &= [\![\delta\ \delta']\!]_\epsilon \end{aligned}$$

Using these equations we can assert our desired conclusion for uniform terms of product types.

$$\Gamma \models [\![\delta\ \delta']\!]_\epsilon \in [\![(\tau[n := \delta'])\downarrow]\!]_\epsilon$$

Now consider the case in which the operator $[\![\delta]\!]_\epsilon$ might be interpreted as a recursive sequence. According to the *recursive condition* of product membership we have

$$\begin{aligned} (\forall k \in \omega)(\quad & \Gamma \models 1^{\text{st}}[\![\delta]\!]_\epsilon \in [\![\tau]\!]_{\epsilon\{n:=0\}} \\ & \text{and} \\ (\forall k' \in \omega)(\forall g \in [\![\tau]\!]_{\epsilon\{n:=k\}}) \quad & \Gamma \models (2^{\text{nd}}[\![\delta]\!]_\epsilon) \bullet k' \bullet g \in [\![\tau]\!]_{\epsilon\{n:=k+1\}}) \end{aligned} \tag{2.26}$$

According to the *R-Zero* and *R-Succ* conversion rules on terms, the semantic requirements of the 1^{st} and 2^{nd} operators on terms, and the semantics of term application we have the

equations

$$\begin{aligned}
[\![\delta]\!]_\epsilon \bullet 0 &= [\![\delta\ \mathbf{0}]\!]_\epsilon \\
&= [\![(\mathbf{R}\ \delta_1\ \delta_2)(\mathbf{0})]\!]_\epsilon \\
&= [\![\delta_1]\!]_\epsilon \\
&= 1^{\mathrm{st}}[\![\delta]\!]_\epsilon \\
(\forall k \in \omega)\ [\![\delta\ ]\!]_\epsilon \bullet (k+1) &= [\![\delta]\!]_\epsilon \bullet [\![\mathbf{S}^{k+1}\mathbf{0}]\!]_\epsilon \\
&= [\![\delta(\mathbf{S}^{k+1}\mathbf{0})]\!]_\epsilon \\
&= [\![(\mathbf{R}\ \delta_1\ \delta_2)(\mathbf{S}^{k+1}\mathbf{0})]\!]_\epsilon \\
&= [\![\delta_2\ (\mathbf{S}^k\mathbf{0})\ (\delta\ (\mathbf{S}^k\mathbf{0}))]\!]_\epsilon \\
&= [\![\delta_2]\!]_\epsilon \bullet k \bullet ([\![\delta]\!]_\epsilon \bullet k) \\
&= (2^{\mathrm{nd}}[\![\delta]\!]_\epsilon) \bullet k \bullet ([\![\delta]\!]_\epsilon \bullet k)
\end{aligned}$$

Using these equations and assuming $g = [\![\delta]\!]_\epsilon \bullet k$ we can restate 2.26 as

$$\begin{aligned}
(\forall k \in \omega)(\quad & \Gamma \models [\![\delta]\!]_\epsilon \bullet 0 \in [\![\tau]\!]_{\epsilon\{n:=0\}} \\
& \text{and} \\
(\forall k' \in \omega) \quad & \Gamma \models [\![\delta]\!]_\epsilon \bullet k' \in [\![\tau]\!]_{\epsilon\{n:=k\}} \text{ implies} \\
& \Gamma \models [\![\delta]\!]_\epsilon \bullet (k'+1) \in [\![\tau]\!]_{\epsilon\{n:=k+1\}})
\end{aligned} \tag{2.27}$$

By induction over $k \in \omega$ and then letting $k = [\![\delta']\!]_\epsilon \in \omega$ we can transform 2.27 into the form

$$\begin{aligned}
(\forall \delta') \quad & \Gamma \models [\![\delta']\!]_\epsilon \in [\![Nat]\!]_\epsilon \text{ implies} \\
& \Gamma \models [\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon \in [\![\tau]\!]_{\epsilon\{n:=[\![\delta']\!]_\epsilon\}}
\end{aligned} \tag{2.28}$$

Using the sequent 2.23 to discharge the implication 2.28 above we can assert

$$\Gamma \models [\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon \in [\![\tau]\!]_{\epsilon\{n:=[\![\delta']\!]_\epsilon\}} \tag{2.29}$$

By the substitution lemma for types, the semantic equivalence of convertible types, and the semantics of term application we get the equalities

$$\begin{aligned}
[\![\tau]\!]_{\epsilon\{n:=[\![\delta']\!]_\epsilon\}} &= [\![(\tau[n := \delta'])]\!]_\epsilon \\
&= [\![(\tau[n := \delta'])\downarrow]\!]_\epsilon \\
[\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon &= [\![\delta\ \delta']\!]_\epsilon
\end{aligned}$$

Finally, using these equations we transform the statement 2.29 into our desired conclusion

$$\Gamma \models [\![\delta\ \delta']\!]_\epsilon \in [\![(\tau[n := \delta'])\downarrow]\!]_\epsilon$$

Π*intro-seq.* Suppose the last step of the derivation uses the Π*intro-seq* rule and the final sequent is $\Gamma \vdash \mathbf{R}\ \delta'\ \delta : \Pi n.\sigma$. The last step of the derivation must satisfy the condition on the Π*intro-seq* rule and have premises

$$\Gamma \vdash \delta' : \rho$$
$$(\forall k \in \omega) \quad \Gamma\{n := k\} \vdash \delta : Nat \to \sigma \to \tau$$

From the induction hypothesis we have, therefore

$$\Gamma \models [\![\delta']\!]_\epsilon \in [\![\rho]\!]_\epsilon \tag{2.30}$$
$$(\forall k \in \omega) \quad \Gamma\{n := k\} \models [\![\delta]\!]_\epsilon \in [\![Nat \to \sigma \to \tau]\!]_\epsilon \tag{2.31}$$
$$\text{where} \quad n \notin \mathrm{fv}(\delta), n \in \mathrm{fv}(\sigma), \mathrm{wrb}(\sigma) \tag{2.32}$$
$$\text{and } (\sigma[n := 0])\downarrow = \rho \tag{2.33}$$
$$\text{and } (\sigma[n := \mathbf{S}\ n])\downarrow = \tau \tag{2.34}$$

From the condition 2.33 above, the semantic equivalence of convertible types, and the substitution lemma we have the equalities

$$\begin{aligned} [\![\rho]\!]_\epsilon &= [\![(\sigma[n := \mathbf{0}])\downarrow]\!]_\epsilon \\ &= [\![(\sigma[n := \mathbf{0}])]\!]_\epsilon \\ &= [\![\sigma]\!]_{\epsilon\{n := [\![\mathbf{0}]\!]_\epsilon\}} \end{aligned}$$

Therefore, using these equations, 2.30 becomes

$$\Gamma \models [\![\delta']\!]_\epsilon \in [\![\sigma]\!]_{\epsilon\{n := [\![\mathbf{0}]\!]_\epsilon\}} \tag{2.35}$$

Transform statement 2.31 by shifting the context extension $\Gamma\{n := k\}$ into an environment extension $\epsilon\{n := k\}$ to get the following equally valid statement. Note that by condition 2.32 $n \notin \mathrm{fv}(\delta)$, so we do not actually need the extension $\{n := k\}$ on ϵ for δ.

$$(\forall k \in \omega)\Gamma \models [\![\delta]\!]_{\epsilon\{n:=k\}} \in [\![Nat \to \sigma \to \tau]\!]_{\epsilon\{n:=k\}} \tag{2.36}$$

Now, we can apply the semantic interpretation of arrow types to statement 2.36 to get the family of interpretations

$$\begin{aligned} (\forall k \in \omega) \quad & [\![Nat \to \sigma \to \tau]\!]_{\epsilon\{n:=k\}} \\ & = \{h \in D \mid (\forall k \in \omega)(\forall g \in [\![\sigma]\!]_{\epsilon\{n:=k\}})h \bullet k \bullet g \in [\![\tau]\!]_{\epsilon\{n:=k\}}\} \end{aligned} \tag{2.37}$$

Using the condition 2.34 above, the semantic equivalence of convertible types, the substitution lemma, and the semantic rules for numeral terms we have the following equations

$$\begin{aligned} [\![\tau]\!]_{\epsilon\{n:=k\}} &= [\![(\sigma[n := \mathbf{S}\ n])\downarrow]\!]_{\epsilon\{n:=k\}} \\ &= [\![(\sigma[n := \mathbf{S}\ n])]\!]_{\epsilon\{n:=k\}} \\ &= [\![\sigma]\!]_{\epsilon\{n:=k+1\}} \end{aligned} \tag{2.38}$$

Use the definitions 2.37, let semantic element $h = [\![\delta]\!]_\epsilon$, and use the equations 2.38 to transform validity statement 2.36 into the statement

$$(\forall k \in \omega)(\ (\forall k' \in \omega)(\forall g \in [\![\sigma]\!]_{\epsilon\{n:=k\}})\ \Gamma \models [\![\delta]\!]_\epsilon \bullet k' \bullet g \in [\![\sigma]\!]_{\epsilon\{n:=k+1\}}) \tag{2.39}$$

Construct the term $\mathbf{R}\ \delta'\ \delta$ so that

$$\begin{aligned} 1^{\text{st}}[\![\mathbf{R}\ \delta'\ \delta]\!]_\epsilon &= [\![\delta']\!]_\epsilon \\ 2^{\text{nd}}[\![\mathbf{R}\ \delta'\ \delta]\!]_\epsilon &= [\![\delta]\!]_\epsilon \end{aligned}$$

Using the term $\mathbf{R}\ \delta'\ \delta$ with these equalities and combining the base and successor validity statements 2.35 and 2.39 we get

$$\begin{aligned} (\forall k \in \omega)(\quad & \Gamma \models 1^{\text{st}}[\![\mathbf{R}\ \delta'\ \delta]\!]_\epsilon \in [\![\sigma]\!]_{\epsilon\{n:=[\mathbf{0}]_\epsilon\}} \\ & \text{and} \\ (\forall k' \in \omega)(\forall g \in [\![\sigma]\!]_{\epsilon\{n:=k\}}) \quad & \Gamma \models 2^{\text{nd}}[\![\mathbf{R}\ \delta'\ \delta]\!]_\epsilon \bullet k' \bullet g \in [\![\sigma]\!]_{\epsilon\{n:=k+1\}} \end{aligned} \tag{2.40}$$

Finally, the 2.40 is the definition of our desired conclusion according to the recursive condition of membership in product types, therefore

$$\Gamma \models [\![\mathbf{R}\ \delta'\ \delta]\!]_\epsilon \in [\![\Pi n.\sigma]\!]_\epsilon$$

We have covered all of the cases and thus by induction over the structure of derivations we have proved the soundness of the inference system with respect to the simple semantics of $\mathcal{T}^\pi$. ∎

Theorem 17 (Completeness) *If* $\Gamma \models [\![\delta]\!] \in [\![\tau]\!]$ *in the simple semantics of* $\mathcal{T}^\pi$ *then there exists a well typing* $\Gamma \vdash \delta : \tau$.

Proof The proof is by induction over the structure of the representative type τ in validity statements $\Gamma \models [\![\delta]\!]_\epsilon \in [\![\tau]\!]_\epsilon$. Since the validity statement $\Gamma \models [\![\delta]\!]_\epsilon \in [\![\tau]\!]_\epsilon$ is true for all models, we may certainly assume any particular model to construct our derivation. Thus, we use our *actual* equivalence class model along with the environment $\epsilon = \epsilon_0$ to construct a derivation for an arbitrarily given validity statement. For every structural case of τ we must show

$$(\forall \delta)\ \Gamma \models [\![\delta]\!]_\epsilon \in [\![\tau]\!]_\epsilon \text{ implies } \Gamma \vdash \delta : \tau$$

Natural Numbers $[\![Nat]\!]$. Suppose

$$\Gamma \models [\![\delta]\!]_\epsilon \in [\![Nat]\!]_\epsilon$$

According to the semantic requirements for the type *Nat* we know $\Gamma \models [\![\delta]\!]_\epsilon \in \omega$ and according to the ω-Terms lemma there must be a term $\mathbf{S}^k\mathbf{0}$ such that $[\![\mathbf{S}^k\mathbf{0}]\!] = [\![\delta]\!]_\epsilon \in \omega$. By the definition of our representative *actual* model, $[\![Nat]\!]$ is an equivalence class of terms under convertibility, therefore $[\![\delta]\!] = [\![\mathbf{S}^k\mathbf{0}]\!]$ implies δ and $\mathbf{S}^k\mathbf{0}$ must be convertible. Finally, we know $\Gamma \vdash \mathbf{S}^k\mathbf{0} : Nat$, therefore by the *Eq* rule we have $\Gamma \vdash \delta : Nat$, our desired conclusion for this case.

Type Variables $[\![X]\!]$. Suppose $\Gamma \models [\![\delta]\!]_\epsilon \in [\![X]\!]_\epsilon$. By the the definition of our chosen environment $\epsilon = \epsilon_0$ we have

$$[\![X]\!]_\epsilon = \epsilon(X) = \{[\![\delta]\!]_\epsilon \in D \mid \Gamma \vdash \delta : X \text{ for some } \Gamma\}$$

Therefore we can directly conclude $\Gamma \vdash \delta : X$.

Arrow Types $[\![\sigma \to \tau]\!]$. Suppose $\Gamma \models [\![\delta]\!]_\epsilon \in [\![\sigma \to \tau]\!]_\epsilon$. There are two cases to consider depending on whether the semantic element $[\![\delta]\!]$ is a member of the semantic domain $[\![\sigma \to \tau]\!]$ according to the *uniform condition* or otherwise according to the *recursive condition* of the semantic interpretation of arrow types. First consider membership according to the uniform condition. In this case the interpretation of arrow types gives

$$\begin{aligned} (\forall \delta')\quad \Gamma \models\ & [\![\delta']\!]_\epsilon \in [\![\sigma]\!]_\epsilon \text{ implies} \\ \Gamma \models\ & [\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon \in [\![\tau]\!]_\epsilon \end{aligned} \tag{2.41}$$

We know $[\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon = [\![\delta\ \delta']\!]_\epsilon$ by definition of the $\bullet$ operator therefore using the induction hypothesis we can say

$$(\forall \delta') \quad \Gamma \vdash \delta' : \sigma \text{ implies } \Gamma \vdash \delta\ \delta' : \tau \tag{2.42}$$

Now construct the context $\Gamma\{x : \sigma\}$ and the derivation

$$\Gamma\{x : \sigma\} \vdash x : \sigma \tag{2.43}$$

Using the sequent 2.43 with $\delta' = x$ to discharge the implication 2.42 we get

$$\Gamma\{x : \sigma\} \vdash \delta\ x : \tau$$

Now by abstraction on x we get

$$\Gamma \quad \vdash \lambda x.\delta\ x : \sigma \to \tau$$

Finally by η-equality with the *Eq* rule we get the desired conclusion

$$\Gamma \quad \vdash \delta : \sigma \to \tau$$

Now for the second case consider $\Gamma \models [\![\delta]\!]_\epsilon \in [\![Nat \to \tau]\!]_\epsilon$ according to the *recursive condition* of the semantics of arrow types, that is,

$$\Gamma \models 1^{\text{st}}[\![\delta]\!]_\epsilon \in [\![\tau]\!]_\epsilon$$

$$\begin{aligned} (\forall k' \in \omega)(\forall \delta'') \quad & \Gamma \models [\![\delta'']\!]_\epsilon \in [\![\tau]\!]_\epsilon \text{ implies} \\ & \Gamma \models (2^{\text{nd}}[\![\delta]\!]_\epsilon) \bullet k' \bullet [\![\delta'']\!]_\epsilon \in [\![\tau]\!]_\epsilon \end{aligned} \tag{2.44}$$

Replace the quantification over all $k' \in \omega$ by a quantification over all terms δ' that denote a natural number $k' \in \omega$. According to the semantic interpretation of sequence terms we know $[\![\delta]\!]$ must have a form to which the operators 1^{st} and 2^{nd} can apply. Thus, assume $[\![\delta]\!] = [\![\mathbf{R}\ \delta_1\ \delta_2]\!]$ where $[\![\delta_1]\!] = 1^{\text{st}}[\![\delta]\!]$ and $[\![\delta_2]\!] = 2^{\text{nd}}[\![\delta]\!]$. With these assumptions 2.44 becomes

$$\Gamma \models [\![\delta_1]\!]_\epsilon \in [\![\tau]\!]_\epsilon$$

$$\begin{aligned} (\forall \delta')(\forall \delta'') \quad & \Gamma \models [\![\delta']\!]_\epsilon \in [\![Nat]\!]_\epsilon \text{ implies} \\ & \Gamma \models [\![\delta'']\!]_\epsilon \in [\![\tau]\!]_\epsilon \text{ implies} \\ & \Gamma \models [\![\delta_2]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon \bullet [\![\delta'']\!]_\epsilon \in [\![\tau]\!]_\epsilon \end{aligned} \tag{2.45}$$

Now invoke the induction hypothesis to get

$$\Gamma \vdash \delta_1 : \tau \tag{2.46}$$

$$\begin{aligned}(\forall\delta')(\forall\delta'') \quad & \Gamma \vdash \delta' : Nat \text{ implies} \\ & \Gamma \vdash \delta'' : \tau \text{ implies} \\ & \Gamma \vdash \delta_2\ \delta'\ \delta'' : \tau\end{aligned} \tag{2.47}$$

Create context $\Gamma\{k' : Nat\}\{p : \tau\}$ and derive

$$\Gamma\{k' : Nat\}\{p : \tau\} \vdash k' : Nat \tag{2.48}$$

$$\Gamma\{k' : Nat\}\{p : \tau\} \vdash p : \tau \tag{2.49}$$

Use 2.48 with variable k' for δ' and 2.49 with variable p for δ'' to discharge the implications of 2.47 to obtain

$$\Gamma\{k' : Nat\}\{p : \tau\} \vdash \delta_2\ k'\ p : \tau$$

Use abstraction to get

$$\Gamma \vdash \lambda k'.\lambda p.\delta_2\ k'\ p : Nat \to \tau \to \tau$$

Now use η-conversion twice with the *Eq* rule to get

$$\Gamma \vdash \delta_2 : Nat \to \tau \to \tau \tag{2.50}$$

Finally, we can use 2.46 and 2.50 with the ($\to$*intro-seq*) rule to get

$$\Gamma \vdash \mathbf{R}\ \delta_1\ \delta_2 : Nat \to \tau$$

which, according to our earlier assumption about the form of δ, gives us our desired result

$$\Gamma \vdash \delta : Nat \to \tau$$

Product Types $[\![\Pi n.\tau]\!]$. Suppose $\Gamma \models [\![\delta]\!]_\epsilon \in [\![\Pi n.\tau]\!]_\epsilon$. There are two cases to consider depending on whether the semantic element $[\![\delta]\!]_\epsilon$ is a member of the product $[\![\Pi n.\tau]\!]_\epsilon$ according to the *uniform condition* or otherwise according to the *recursive condition.* First, according to the *uniform condition* of the semantic requirements for product types and the choice of $[\![\delta']\!]_\epsilon$ for k we have the implication

$$\begin{aligned}(\forall\delta') \quad \Gamma \models \quad & [\![\delta']\!]_\epsilon && \in [\![Nat]\!]_\epsilon \text{ implies} \\ \Gamma \models \quad & [\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon && \in [\![\tau]\!]_{\epsilon\{n:=[\![\delta']\!]_\epsilon\}}\end{aligned} \tag{2.51}$$

Using the substitution lemma, the semantic equivalence of convertible types, the interpretation of the bullet operator we have the equalities

$$\begin{aligned} [\![\tau]\!]_{\epsilon\{n:=[\![\delta']\!]_\epsilon\}} &= [\![(\tau[n := \delta'])\downarrow]\!]_\epsilon \\ &= [\![(\tau[n := \delta'])]\!]_\epsilon \\ [\![\delta]\!]_\epsilon \bullet [\![\delta']\!]_\epsilon &= [\![\delta\ \delta']\!]_\epsilon \end{aligned}$$

With these equalities 2.51 becomes

$$\begin{aligned} (\forall \delta') \quad \Gamma \models\ & [\![\delta']\!]_\epsilon \quad \in [\![Nat]\!]_\epsilon \text{ implies} \\ \Gamma \models\ & [\![\delta\ \delta']\!]_\epsilon \quad \in [\![(\tau[n := \delta'])\downarrow]\!]_\epsilon \end{aligned} \tag{2.52}$$

Using the induction hypothesis we get

$$\begin{aligned} (\forall \delta') \quad & \Gamma \vdash \delta' : Nat \text{ implies} \\ & \Gamma \vdash \delta\ \delta' : (\tau[n := \delta'])\downarrow \end{aligned} \tag{2.53}$$

Now construct the context $\Gamma\{n : Nat\}$ and the derivation

$$\Gamma\{n : Nat\} \vdash n : Nat \tag{2.54}$$

Using the sequent 2.54 to discharge the implication 2.53 we get

$$\Gamma\{n : Nat\} \vdash \delta\ n : (\tau[n := n])\downarrow$$

Finally, we know that $n \in \mathrm{fv}(\tau)$ so by the dependent abstraction rule on n and then the η-equality $\lambda n.\delta n = \delta$ with the *Eq* rule we get our desired result

$$\begin{aligned} \Gamma \quad & \vdash \lambda n.\delta\ n : \Pi n.\tau \\ \Gamma \quad & \vdash \delta : \Pi n.\tau \end{aligned}$$

For the second case consider $\Gamma \models [\![\delta]\!]_\epsilon \in [\![\Pi n.\tau]\!]_\epsilon$ according to the *recursive condition* of the semantics of product types, that is, consider

$$\begin{aligned} (\forall k \in \omega)(\quad & \Gamma \models 1^{\mathrm{st}}[\![\delta]\!]_\epsilon \in [\![\tau]\!]_{\epsilon\{n:=0\}} \\ & \text{and} \\ (\forall k' \in \omega)(\forall \delta'') \quad & \Gamma \models [\![\delta'']\!]_\epsilon \in [\![\tau]\!]_{\epsilon\{n:=k\}} \text{ implies} \\ & \Gamma \models (2^{\mathrm{nd}}[\![\delta]\!]_\epsilon) \bullet k' \bullet [\![\delta'']\!]_\epsilon \in [\![\tau]\!]_{\epsilon\{n:=k+1\}}) \end{aligned} \tag{2.55}$$

Replace the outer quantification over all $k \in \omega$ by a quantification over all terms δ' that denote a natural number $k \in \omega$. Also we can make use of the equalities $0 = [\![\mathbf{0}]\!]$ and $[\![\delta']\!] + 1 = [\![\mathbf{S}\ \delta']\!]$ that follow from the semantic rules for terms. Next replace the inner quantification over all $k' \in \omega$ by a quantification over all terms δ''' that denote a natural number $k' \in \omega$. The statements 2.55 then become

$$\begin{array}{lll} & & \Gamma \models 1^{\mathrm{st}}[\![\delta]\!]_\epsilon \in [\![\tau]\!]_{\epsilon\{n:=0\}} \\ & & \text{and} \\ (\forall\delta') & (\forall\delta''')(\forall\delta'') & \Gamma \models [\![\delta''']\!]_\epsilon \in [\![Nat]\!]_\epsilon \text{ implies} \\ & & \Gamma \models [\![\delta'']\!]_\epsilon \in [\![\tau]\!]_{\epsilon\{n:=[\![\delta']\!]_\epsilon\}} \text{ implies} \\ & & \Gamma \models (2^{\mathrm{nd}}[\![\delta]\!]_\epsilon) \bullet [\![\delta''']\!]_\epsilon \bullet [\![\delta'']\!]_\epsilon \in [\![\tau]\!]_{\epsilon\{n:=[\![\mathbf{S}\ \delta']\!]_\epsilon\}} \end{array} \tag{2.56}$$

Using the substitution lemma and the semantic equivalence of convertible types, we have the following equalities

$$\begin{array}{ll} [\![\tau]\!]_{\epsilon\{n:=[\![\mathbf{0}]\!]\}} & = [\![(\tau[n := 0])]\!]_\epsilon \\ & = [\![(\tau[n := 0])\!\downarrow]\!]_\epsilon \\ [\![\tau]\!]_{\epsilon\{n:=[\![\delta']\!]\}} & = [\![(\tau[n := \delta'])]\!]_\epsilon \\ & = [\![(\tau[n := \delta'])\!\downarrow]\!]_\epsilon \\ [\![\tau]\!]_{\epsilon\{n:=[\![\mathbf{S}\ \delta']\!]\}} & = [\![(\tau[n := \mathbf{S}\ \delta'])]\!]_\epsilon \\ & = [\![(\tau[n := \mathbf{S}\ \delta'])\!\downarrow]\!]_\epsilon \end{array} \tag{2.57}$$

According to the semantic requirements for sequence terms, we know $[\![\delta]\!]$ must have a form to which the operators 1^{st} and 2^{nd} can apply. Thus, assume $[\![\delta]\!] = [\![\mathbf{R}\ \delta_1\ \delta_2]\!]$ where $[\![\delta_1]\!] = 1^{\mathrm{st}}[\![\delta]\!]$ and $[\![\delta_2]\!] = 2^{\mathrm{nd}}[\![\delta]\!]$. With this assumption and the equations 2.57, the statements 2.56 become

$$\Gamma \models [\![\delta_1]\!]_\epsilon \in [\![(\tau[n := \mathbf{0}])\!\downarrow]\!]_\epsilon$$

$$\begin{array}{lll} (\forall\delta') & (\forall\delta''')(\forall\delta'') & \Gamma \models [\![\delta''']\!]_\epsilon \in [\![Nat]\!]_\epsilon \text{ implies} \\ & & \Gamma \models [\![\delta'']\!]_\epsilon \in [\![(\tau[n := \delta'])\!\downarrow]\!]_\epsilon \text{ implies} \\ & & \Gamma \models [\![\delta_2]\!]_\epsilon \bullet [\![\delta''']\!]_\epsilon \bullet [\![\delta'']\!]_\epsilon \in [\![(\tau[n := \mathbf{S}\ \delta'])\!\downarrow]\!]_\epsilon \end{array} \tag{2.58}$$

Now invoke the induction hypothesis to get

$$\Gamma \vdash \delta_1 : (\tau[n := \mathbf{0}])\!\downarrow \tag{2.59}$$

$$(\forall\delta')(\forall\delta''')(\forall\delta'') \quad \Gamma \vdash \delta''' : Nat \text{ implies}$$
$$\Gamma \vdash \delta'' : (\tau[n := \delta'])\downarrow \text{ implies} \tag{2.60}$$
$$\Gamma \vdash \delta_2\ \delta'''\ \delta'' : (\tau[n := \mathbf{S}\ \delta'])\downarrow$$

Create context $\Gamma\{n : Nat\}\{k' : Nat\}\{p : \tau\}$ and derive

$$\Gamma\{n : Nat\}\{k' : Nat\}\{p : \tau\} \vdash k' : Nat \tag{2.61}$$

$$\Gamma\{n : Nat\}\{k' : Nat\}\{p : \tau\} \vdash p : \tau \tag{2.62}$$

Use 2.61 and 2.62 with variable n for δ', variable k' for δ''', and variable p for δ'' to discharge the implications of 2.60 to obtain

$$\Gamma\{n : Nat\}\{k' : Nat\}\{p : \tau\} \vdash \delta_2\ k'\ p : (\tau[n := \mathbf{S}\ n])\downarrow$$

Use abstraction to get

$$\Gamma\{n : Nat\} \vdash \lambda k'.\lambda p.\delta_2\ k'\ p : Nat \to \tau \to (\tau[n := \mathbf{S}\ n])\downarrow$$

Now use η-conversion twice with the *Eq* rule to get

$$\Gamma\{n : Nat\} \vdash \delta_2 : Nat \to \tau \to (\tau[n := \mathbf{S}\ n])\downarrow \tag{2.63}$$

Finally, we know $n \in \mathrm{fv}(\tau), n \in \mathrm{fv}((\tau[n := \mathbf{S}\ n])\downarrow)$, and $n \notin \mathrm{fv}((\tau[n := \mathbf{0}])\downarrow)$, so we can use 2.59 and 2.63 with the (Π*intro-seq*) rule to get

$$\Gamma \vdash \mathbf{R}\ \delta_1\ \delta_2 : \Pi n.\tau$$

which, according to our earlier assumption about the form of δ, gives us our desired result

$$\Gamma \vdash \delta : \Pi n.\tau$$

Type sequence $[\![\mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ n]\!]$. In this case we have a family of types indexed by n. Assume that we have a family of domain elements indexed by the same n, say $\{[\![\delta]\!]_{\epsilon\{n:=k\}} \mid k \in \omega\}$, and that we have the following family of semantic memberships

$$(\forall k \in \omega)\ \Gamma \models [\![\delta]\!]_{\epsilon\{n:=k\}} \in [\![\mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ n]\!]_{\epsilon\{n:=k\}} \tag{2.64}$$

We are obliged to show that there is a corresponding family of typing judgements

$$(\forall k \in \omega)\ \Gamma \vdash \delta[n := \mathbf{S}^k\mathbf{0}] : \mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ (\mathbf{S}^k\mathbf{0})$$

According to the semantic interpretation of recursive type sequences, the statement 2.64 represents the following family of semantic memberships. By the definition of recursive type sequences we know we are restricted here to weak recursion.

$$\begin{aligned} &\Gamma \models [\![\delta]\!]_{\epsilon\{n:=0\}} \in [\![\tau]\!]_{\epsilon\{n:=0\}} \\ &(\forall k \in \omega)\quad \Gamma \models [\![\delta]\!]_{\epsilon\{n:=k+1\}} \in [\![\sigma]\!]_{\epsilon\{n:=k+1\}\{n':=k\}\{X:=S_k\}} \\ &\text{where}\qquad S_k = [\![\mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ n]\!]_{\epsilon\{n:=k\}} \end{aligned} \tag{2.65}$$

By the ω-Terms lemma we can rewrite 2.65 as

$$\begin{aligned} &\Gamma \models [\![\delta]\!]_{\epsilon\{n:=[\mathbf{0}]\}} \in [\![\tau]\!]_{\epsilon\{n:=[\mathbf{0}]\}} \\ &(\forall k \in \omega)\quad \Gamma \models [\![\delta]\!]_{\epsilon\{n:=[\mathbf{S}^{k+1}\mathbf{0}]\}} \in [\![\sigma]\!]_{\epsilon\{n:=[\mathbf{S}^{k+1}\mathbf{0}]\}\{n':=[\mathbf{S}^k\mathbf{0}]\}\{X:=S_k\}} \\ &\text{where}\qquad S_k = [\![\mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ n]\!]_{\epsilon\{n:=[\mathbf{S}^k\mathbf{0}]\}} \end{aligned} \tag{2.66}$$

Using the substitution lemma on terms and types and the definition of S_k we have the following equations

$$\begin{aligned} [\![\delta]\!]_{\epsilon\{n:=[\mathbf{0}]\}} &= [\![\delta[n := \mathbf{0}]]\!]_\epsilon \\ [\![\delta]\!]_{\epsilon\{n:=[\mathbf{S}^k\mathbf{0}]\}} &= [\![\delta[n := \mathbf{S}^k\mathbf{0}]]\!]_\epsilon \\ [\![\delta]\!]_{\epsilon\{n:=[\mathbf{S}^{k+1}\mathbf{0}]\}} &= [\![\delta[n := \mathbf{S}^{k+1}\mathbf{0}]]\!]_\epsilon \end{aligned} \tag{2.67}$$

$$\begin{aligned} &[\![\tau]\!]_{\epsilon\{n:=[\mathbf{0}]\}} = [\![(\tau[n := \mathbf{0}])]\!]_\epsilon \\ &[\![\sigma]\!]_{\epsilon\{n:=[\mathbf{S}^{k+1}\mathbf{0}]\}\{n':=[\mathbf{S}^k\mathbf{0}]\}\{X:=[\![\mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ n]\!]_{\epsilon\{n:=[\mathbf{S}^k\mathbf{0}]\}}\}} \\ &= [\![(\sigma[n := \mathbf{S}^{k+1}\mathbf{0}][n' := \mathbf{S}^k\mathbf{0}][X := \mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ (\mathbf{S}^k\mathbf{0})])\downarrow]\!]_\epsilon \end{aligned}$$

Using the equations 2.67 on 2.66 we get

$$\begin{aligned} &\Gamma \models [\![\delta[n := \mathbf{0}]]\!]_\epsilon \in [\![\tau[n := \mathbf{0}]]\!]_\epsilon \\ &(\forall k \in \omega)\quad \Gamma \models [\![\delta[n := \mathbf{S}^{k+1}\mathbf{0}]]\!]_\epsilon \\ &\qquad\qquad \in [\![\sigma[n := \mathbf{S}^{k+1}\mathbf{0}][n' := \mathbf{S}^k\mathbf{0}][X := \mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ (\mathbf{S}^k\mathbf{0})]]\!]_\epsilon \end{aligned} \tag{2.68}$$

By the induction hypothesis we have

$$\Gamma \vdash \delta[n := \mathbf{0}] : \tau[n := \mathbf{0}]$$

$$(\forall k \in \omega) \quad \Gamma \vdash \delta[n := \mathbf{S}^{k+1}\mathbf{0}] : \sigma[n := \mathbf{S}^{k+1}\mathbf{0}][n' := \mathbf{S}^{k}\mathbf{0}][X := \mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ (\mathbf{S}^{k}\mathbf{0})] \tag{2.69}$$

Using the ΠR-*Zero* and ΠR-*Succ* conversion rules for sequence types and factoring replacements, we get the following equalities. Note that for type $\mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ \mathbf{0}$, it does not matter what the successor type is, so we choose σ. Similarly, for type $\mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ (\mathbf{S}^{k+1}\mathbf{0})$, it does not matter what the zero type is, so we choose τ.

$$\begin{aligned} \tau[n := \mathbf{0}] &= \mathbf{R}\ \tau[n := \mathbf{0}]\ \lambda n'.\lambda X.\sigma[n := \mathbf{0}]\ \mathbf{0} \\ &= \mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ n\ [n := \mathbf{0}] \end{aligned}$$

$$\begin{aligned} &\sigma[n := \mathbf{S}^{k+1}\mathbf{0}][n' := \mathbf{S}^{k}\mathbf{0}][X := \mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ \mathbf{S}^{k}\mathbf{0}] \\ &\quad = \mathbf{R}\ \tau[n := \mathbf{S}^{k+1}\mathbf{0}]\ \lambda n'.\lambda X.\sigma[n := \mathbf{S}^{k+1}\mathbf{0}]\ (\mathbf{S}^{k+1}\mathbf{0}) \\ &\quad = \mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ n\ [n := \mathbf{S}^{k+1}\mathbf{0}] \end{aligned} \tag{2.70}$$

Using the equations 2.70 on the 2.69 we get

$$\begin{aligned} &\Gamma \vdash \delta[n := \mathbf{0}] : \mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ n\ [n := \mathbf{0}] \\ (\forall k \in \omega) \quad &\Gamma \vdash \delta[n := \mathbf{S}^{k+1}\mathbf{0}] : \mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ n\ [n := \mathbf{S}^{k+1}\mathbf{0}] \end{aligned}$$

Finally, combining these two schemes and substituting for n gives us our desired conclusion

$$(\forall k \in \omega) \quad \Gamma \vdash \delta[n := \mathbf{S}^{k}\mathbf{0}] : \mathbf{R}\ \tau\ \lambda n'.\lambda X.\sigma\ (\mathbf{S}^{k}\mathbf{0})$$

This completes all the cases for the induction over types and thus completes our proof of completeness. ∎

We have presented a dependent type inference system called $\mathcal{T}^{\pi}$ and shown it to be sound and complete with respect to a simple term model. Now we are ready to show the existence of principal types for $\mathcal{T}^{\pi}$ and develop a type reconstruction algorithm.

Chapter 3

Principal Types and Dependent Type Reconstruction

3.1 Type Subsumption and Unification

To prove the principal type theorem in the next section, we must establish a subsumption order of types with a unification theorem in a way analogous to the principal type theorem for the simply typed lambda calculus. (See Hindley [Hin69].)

In this section we define the subsumption order on $\mathcal{T}^\pi$ types and give a unification algorithm. The unification algorithm together with the matching algorithm introduced in the next section make it possible to prove the existence of principal types and to reconstruct $\mathcal{T}^\pi$ principal types for terms. The results of this section apply equally to both weak $\mathcal{T}^\pi$ and non-weak $\mathcal{T}^\pi$ types because, as we shall see, unification itself has no affect on the weak-recursion properties of $\mathcal{T}^\pi$ types.

A *type substitution* is a total mapping from type variables to types, $\theta : \mathit{Tvar} \rightarrow \mathit{Types}$, such that $\theta(X) \neq X$ for only finitely many $X \in \mathit{Tvar}$. Let the identity substitution be denoted θ_{id} and let the *support* of a substitution θ be the finite, non-identity portion of the mapping, $\mathrm{su}(\theta) = \{X \in \mathrm{dom}(\theta) \mid \theta(X) \neq X\}$. We define the *introduced variables* of a substitution θ as $\mathrm{in}(\theta) = \bigcup_{X \in \mathrm{dom}(\theta)} \mathrm{fv}(\theta(X))$. Often we refer to the substitution $\theta_{id}\{X := \tau\}$ simply by $\{X := \tau\}$. Two substitutions θ_1 and θ_2 are *equal* if and only if their supports are equal and $\forall X \in \mathrm{su}(\theta_1), \theta_1(X) = \theta_2(X)$. Every substitution $\theta : \mathit{Tvars} \rightarrow \mathit{Types}$ can be *extended* to a map from types to types, $\hat{\theta} : \mathit{Types} \rightarrow \mathit{Types}$, using standard methods. (See, for example, Snyder and Gallier [SG89].) We normally assume the appropriate

domain of the substitution map, either *Tvars* or *Types*, from the context of any discussion. The composition of substitutions θ and θ' is denoted $\theta' \circ \theta$, or simply $\theta'\theta$ by juxtaposition, and defined $(\theta' \circ \theta)(\tau) \stackrel{\text{def}}{=} \theta'(\theta(\tau))$.[1] If we are to apply substitution θ to type τ, we always assume $\text{in}(\theta) \cap \text{bv}(\tau) = \{\}$. If $\text{in}(\theta) \cap \text{bv}(\tau) \neq \{\}$, then we rename the bound variables of τ to avoid the conflict and insure our assumption always holds. We call this requirement the *non-interference assumption of substitution*, analogous to the non-interference assumption of replacement.

In the $\mathcal{T}^\pi$ type system, we say a type σ is *more general than* or *subsumes* a type τ by substitution θ, denoted $\sigma \geq_\theta \tau$, if and only if $\theta(\sigma) = \tau$. We also say τ *is an instance of* σ by θ. The relations $\geq_\theta$ together induce what we call the *subsumption order on types.* Using the subsumption order on types, we can also order substitutions. A substitution θ is more general than or subsumes a substitution θ', denoted $\theta \geq \theta'$, if and only if there exists a substitution θ'' such that $\theta'' \circ \theta = \theta'$.

In this section we introduce an additional angle bracket notation, $\langle$ and $\rangle$, for tuples. The angle brackets should help in reading the more detailed formulas appearing in the following sections.

Definition 18 (Unifiers) *A substitution θ is a* unifier *of a pair of types $\langle \sigma, \tau \rangle$ if and only if $\theta(\sigma) = \theta(\tau)$. If such a unifier exists, σ and τ are said to have a* common instance, *namely, the type $\theta(\sigma) = \theta(\tau)$. A substitution θ is a unifier for a finite set of pairs of types if and only if it is a unifier for every pair in the set.*

We use the notation $U(\langle \sigma, \tau \rangle)$ to refer to the set of unifiers for the pair $\langle \sigma, \tau \rangle$. Similarly, if D is a finite set of pairs of types, then $U(D)$ refers to the set of unifiers of D. For our purposes, pairs are strictly ordered, so we distinguish between the pair $\langle \sigma, \tau \rangle$ and the pair $\langle \tau, \sigma \rangle$.

Definition 19 (Most General Unifiers) *A substitution θ is a* most general unifier (MGU) *of a finite set $D = \{\langle \sigma_0, \tau_0 \rangle, \ldots, \langle \sigma_n, \tau_n \rangle\}$ of pairs of types if and only if*

1. *θ unifies every pair $\langle \sigma_i, \tau_i \rangle$ in D*

[1] Note that composition is always given in *applicative* order, even when using the juxtaposition notation.

2. *The support of θ is restricted to the free variables of D, that is,* $\mathrm{su}(\theta) \subseteq \mathrm{fv}(D)$

3. *For any other unifier θ' of D, $\theta \geq \theta'$.*

An MGU of a single pair of types $\langle \sigma, \tau \rangle$ is simply the MGU of the singleton set $\{\langle \sigma, \tau \rangle\}$.

We say a common instance of a pair of types $\langle \sigma, \tau \rangle$ is a *most general common instance* if it subsumes every other common instance of σ and τ. It is easy to show that every most general unifier determines a most general common instance. It is also easy to see that most general unifiers and most general common instances, if they exist, are unique up to variable name changes.

We present the process of finding unifiers for types as a sequence of transformations on a system of pairs of types using a method described by Wayne Snyder and Jean Gallier [SG89]. Snyder and Gallier credit Martelli and Montanari and originally Herbrand for discovery of the method of transformations. We found this method the most suitable for extending to the process of finding matchers in the next section.

Let a *disagreement set* be a finite set of ordered pairs of types $\{\langle \sigma_1, \tau_1 \rangle, \ldots, \langle \sigma_n, \tau_n \rangle\}$. We say a disagreement set D is a *system* of pairs of types that we wish to *solve* and find a unifier for, suggesting that the method comes from the long history of solving systems of equations with unknowns. With this notion of equation solving in mind, we will sometimes use the letters S and S' as well as D and D' to represent a disagreement set for systems we wish to solve. A pair of types $\langle X, \tau \rangle$ in a system D is in *solved form* if X is a type variable that does not appear anywhere else in D, in particular $X \notin \mathrm{fv}(\tau)$. A system is in solved form if all its pairs are in solved form. A disagreement set in solved form is essentially an MGU, as the following lemma adapted from Snyder and Gallier shows. Note that according to our definition of a solved form, $X_i \neq X_j$ for $i \neq j$.

Lemma 20 (Unifiers for Solved Systems) *If $D = \{\langle X_1, \tau_1 \rangle, \ldots, \langle X_n, \tau_n \rangle\}$ is a solved system, then the substitution $\theta = \theta_{id}\{X_1 := \tau_1\} \cdots \{X_n := \tau_n\}$ constructed from D is an MGU of D.*

Proof This proof is adapted from Snyder and Gallier [SG89]. Suppose θ' is a unifier for D. Then for every $\langle X_i, \tau_i \rangle, \theta'(X_i) = \theta'(\tau_i)$. But also $\tau_i = \theta(X_i)$ by the definition of θ. Thus $\theta'(X_i) = \theta'(\tau_i) = \theta'(\theta(X_i))$. But this means $\theta' \leq \theta$ so θ is an MGU of D. ∎

The unification algorithm for $\mathcal{T}^\pi$ types is specified in terms of the simple set of non-deterministic rules in Figure 3.1. These rules transform disagreement sets into solved form. For a disagreement set D and a substitution θ, we say $\theta = \text{Unify}(D)$ if and only if there exists some finite sequence of transformations $D \Rightarrow \ldots \Rightarrow D'$ using the transformation rules such that D' is in solved form, and θ is the MGU of the solved system D' constructed according to Lemma 20 above. If no such sequence of transformation of D into solved form exists, then the algorithm fails.

Unify works primarily over *type* variables, yet we want to deal with term variables properly as well. In the transformation rules, note that the product types and the recursive types are unified modulo bound term variable names, but that the free projection variable n of recursive type sequences in (iv) must match exactly.

Identity
(i) $\quad \{\langle \tau, \tau \rangle\} \cup D \;\Rightarrow\; D$

Decomposition
(ii) $\quad \{\langle \sigma_1 \to \tau_1, \sigma_2 \to \tau_2 \rangle\} \cup D$
$\qquad\qquad \Rightarrow \quad \{\langle \sigma_1, \sigma_2 \rangle, \langle \tau_1, \tau_2 \rangle\} \cup D$

(iii) $\quad \{\langle \Pi n.\sigma, \Pi m.\tau \rangle\} \cup D$
$\qquad\qquad \Rightarrow \quad \{\langle \sigma, \tau[m := n] \rangle\} \cup D$

(iv) $\quad \{\langle \mathbf{R}\ \tau_1\ (\lambda n_1'.\lambda X_1.\sigma_1)\ n, \mathbf{R}\ \tau_2\ (\lambda n_2'.\lambda X_2.\sigma_2)\ n \rangle\} \cup D$
$\qquad\qquad \Rightarrow \quad \{\langle \tau_1, \tau_2 \rangle, \langle \sigma_1, \sigma_2[n_2' := n_1'][X_2 := X_1] \rangle\} \cup D$

Variable
(v) $\quad \{\langle X, \tau \rangle\} \cup D$ or $\{\langle \tau, X \rangle\} \cup D$ and $X \notin \text{fv}(\tau)$
$\qquad\qquad \Rightarrow \quad \{\langle X, \tau \rangle\} \cup \theta D$ where $\theta = (\theta_{id}\{X := \tau\})$

Figure 3.1: Transformation Rules for the Unification of $\mathcal{T}^\pi$ Types

The following lemma adapted from Snyder and Gallier is used in the soundness and completeness proofs for the unification algorithm.

Lemma 21 (Preservation of Unifiers) *If $S \Rightarrow S'$ using any of the transformations of Figure 3.1, then $U(S) = U(S')$.*

Proof First consider the identity and decomposition transformations (i) to (iv) where

neither of the two types we want to unify is a variable. To unify two non-variable types, the head symbols must be the same and then all subterms must unify. But the decomposition transformations do the same thing, for they match the head symbols and then expand the disagreement set of types that must unify by all corresponding pairs of subterms, with appropriate variable renaming. Thus we have for each of the first four cases that

$$\theta' \in U(S) \text{ if and only if } \theta' \in U(S')$$

For case (v) we have $S = \{\langle X, \tau\rangle\} \cup D$ or $S = \{\langle \tau, X\rangle\} \cup D$ and $\theta = \theta_{id}\{X := \tau\}$ by the construction of the algorithm where we know $X \notin \mathrm{fv}(\tau)$. Also we know that when θ' is a unifier of S, then $\theta' X = \theta' \tau$. Because θ is the single assignment $\theta = \theta_{id}\{X := \tau\}$, then θ' and $\theta' \circ \theta$ differ only at X where $\theta(X) = \tau$. But θ' at X and τ are the same, $\theta'(X) = \theta'(\tau)$, therefore $\theta' = \theta' \circ \theta$ at all domain elements. We use this in the third equivalence below. Thus we have

1. $\theta' \in U(S) \quad \Leftrightarrow \quad \theta' \in U(\{\langle X, \tau\rangle\} \cup D)$ or $\theta' \in U(\{\langle \tau, X\rangle\} \cup D)$
2. $\quad \Leftrightarrow \quad \theta' X = \theta' \tau$ and $\theta' \in U(D)$
3. $\quad \Leftrightarrow \quad \theta' X = \theta' \tau$ and $\theta' \circ \theta \in U(D)$
4. $\quad \Leftrightarrow \quad \theta' X = \theta' \tau$ and $\theta' \in U(\theta D)$
5. $\quad \Leftrightarrow \quad \theta' \in U(\{\langle X, \tau\rangle\} \cup \theta D)$
6. $\quad \Leftrightarrow \quad \theta' \in U(S')$

∎

Theorem 22 (Unification Theorem) *Let D be an arbitrary disagreement set of pairs of $\mathcal{T}^\pi$ types.*

1. **Termination** Unify(D) *terminates, either returning successfully with a substitution, or else failing.*
2. **Soundness and Completeness** Unify(D) *successfully returns a substitution θ if and only if θ is the most general unifier of D.*

Proof This proof is adapted from Snyder and Gallier [SG89]. First we show that the algorithm always terminates. Define a complexity measure (a, b) on disagreement sets D of pairs of types where a is the number of occurrences of free variables in D and b is the sum of the sizes of all the types in D. Let the size of a type be the number of nodes in the type formula drawn as type expression trees. The identity and decomposition transformations (i) to (iv) strictly decrease b because at each decomposition a pair of types have their root nodes removed and no new nodes are added. The variable transformation (v) decreases a because it removes one variable from the system. All transformation sequences must terminate because every sequence corresponds to a strictly decreasing sequence in the lexicographic ordering (a, b) of complexity measures on disagreement sets.

To show soundness, suppose $\theta = \text{Unify}(D)$, that is, there is a finite sequence of transformations on disagreement sets $D \Rightarrow \ldots \Rightarrow D'$ such that D' is a solved system with MGU θ constructed according to Lemma 20. We can use induction on k, the length of the transformation sequences, to show that $U(D) = U(D')$ and therefore any unifier of the final solved system D' is also a unifier of the original system D. The proof of the induction is simple: the basis $k = 0$ of the induction is trivial and the step cases of the induction are given by Lemma 21.

Finally, to show completeness we have to show that for every unifier θ' of D, $\theta' \leq \theta$ where $\theta = \text{Unify}(D)$. We know Unify terminates on D yielding substitution θ, so there must exist a finite sequence of transformations $D \Rightarrow \ldots \Rightarrow D'$ with D' in solved form, and with θ the MGU of D' by Lemma 20. By the soundness proof above, we know θ is also a unifier of the original system D. Now let θ' be an arbitrary unifier of D. By the preservation of unifiers, Lemma 21, θ' is also a unifier of D'. But θ is the MGU of D', so $\theta' \leq \theta$ in the solved system D'. But then we also have θ, θ' both unifiers of D with $\theta' \leq \theta$, therefore θ is the MGU of D. ∎

We noted in the beginning of this section that the unification algorithm has no affect on weak-recursion properties. To see this, notice that in in Figure 3.1 the most general unifier under construction is composed of replacements that involve only sub-expression types of the the original system D. Thus if θ is a most general unifier of a system D of

$\mathcal{T}^\pi$ types then D is weak recursively based if and only if θD is weak recursively based. In the next section we will see that, unlike unification, the completeness of matching is restricted to weak recursively based systems of types.

3.2 Matching

The Matching algorithm and theorem in this section are concerned with reconstructing derivation steps based on the Π*intro-seq* typing rule. Given the premises of the Π*intro-seq* rule, the matching algorithm gives a systematic way to find a weak recursively based type satisfying the *conditions* on the Π*intro-seq* rule. The conditions on the Π*intro-seq* rule specify what we call a *weak-recursion scheme.* The matching algorithm, and consequently the reconstruction algorithm given later, only find weak recursively based types, that is, types in the weak $\mathcal{T}^\pi$ type system specified by Definition 4 in Section 2.1. Weak recursion is the essential limitation on the type system that we assume to prove the principal types theorem for $\mathcal{T}^\pi$ and to obtain the completeness of the reconstruction algorithm.

Definition 23 (Weak-recursion schemes) *A triple of types* (α, γ, β) *is an* n-indexed weak-recursion scheme *if and only if*

$$\begin{array}{ll} n \in \mathrm{fv}(\gamma), \mathrm{wrb}(\gamma) & \textit{dependence condition} \\ (\gamma[n := 0])\downarrow = \alpha & \textit{zero condition} \\ (\gamma[n := \mathbf{S}\ n])\downarrow = \beta & \textit{successor condition} \end{array}$$

To give some intuition to the matching of weak-recursion schemes, suppose we are seeking to type a sequence of terms $\delta_0, \ldots, \delta_k, \delta_{k+1} \ldots$ obtained from the Π*intro-seq* rule. From the premises of the Π*intro-seq* rule we know that δ_0 has some type ρ, and that some σ, τ are the types of the terms δ_k, δ_{k+1}, respectively. The matching algorithm will attempt to find a substitution θ such that $\langle \theta\rho, \theta\sigma, \theta\tau \rangle$ becomes a weak-recursion scheme, where the types $(\theta\rho, \theta\sigma, \theta\tau)$ are the types α, γ, β in Definition 23 above. If we let $\gamma_k = (\theta\sigma)[n := k]$ for all k, then, after matching, the sequence $\gamma_0, \ldots, \gamma_k, \ldots$ will be the types for the sequence of terms $\delta_0, \ldots, \delta_k, \ldots$, respectively.

Definition 24 (Weak-recursion matchers) *A substitution θ is an* n-indexed weak-recursion matcher *of a triple of weak recursively based types $\langle \rho, \sigma, \tau \rangle$ if and only if $(\theta\rho, \theta\sigma, \theta\tau)$ is an n-indexed weak-recursion scheme.*

We have *most general matchers* (MGMs), like MGUs, by adopting the last two conditions for most general unifiers given in Definition 19.

The weak-recursion matching algorithm works on a single triple of weak recursively based types $\langle \rho, \sigma, \tau \rangle$, along with a new index variable n, and computes an n-indexed weak-recursion matcher in two basic steps. First, the algorithm *occurs matches* (or *successor matches*) a disagreement set $\{\langle \sigma, \tau \rangle\}$ of the second two types, yielding a *successor matcher* θ_1 satisfying the successor condition of Definition 23. Then, roughly speaking, the algorithm unifies a disagreement set $\{\langle \rho, \theta_1\sigma \rangle\}$ involving the first and second types, yielding a unifier θ_2. The substitution θ_2 is the *zero matcher* satisfying the zero condition of Definition 23.[2] Finally, the successor matcher and the zero matcher substitutions are combined into the desired weak-recursion matcher satisfying both the successor matching condition and the zero matching condition.

To get a preliminary feel for the overall matching process, consider the following matching problem that arises in reconstructing the type for the *taut* example of Section 2.2.[3] The *taut* program is a recursive sequence.

$$taut \equiv \mathbf{R}\ (\lambda f.f)\ (\lambda n'.\lambda p.\lambda f.((p\ (f\ true))\ \&\ (p\ (f\ false))))$$

The base term of the sequence $\lambda f.f$ has a type $N \to N$. The unfolding term

$$\lambda n'.\lambda p.\lambda f.((p\ (f\ true))\ \&\ (p\ (f\ false)))$$

has a type $K \to Bool$ for the variable p representing the k^{th} term of the underlying sequence and a type $(Bool \to K) \to Bool$ for the sub-expression $\lambda f.((p\ (f\ true))\&(p\ (f\ false)))$

[2] More accurately, in the second step we unify $\langle \rho, \theta_1\sigma' \rangle$ where σ' is σ modified by replacements for every *occurs variable* $X_j \in \text{fv}(\sigma)$, specifically, $\sigma' = \sigma[X_j := ((\theta_1 X_j)[n := \mathbf{0}])\downarrow]$. Occurs variables are defined following Definition 25 and the matching algorithm is given in Figure 3.3. The technicality here arises because we are using unification to find the zero matcher rather than defining a separate zero matching algorithm. These details are covered later in this section.

[3] For a complete trace of the matching algorithm of Figure 3.3 on this example see Appendix C.

representing the $(k+1)^{\text{th}}$ term of the sequence. The matching algorithm must find a weak recursion scheme matching the triple of types $(\sigma_0, \sigma_k, \sigma_{k+1}) = (N \to N, K \to \mathit{Bool}, (\mathit{Bool} \to K) \to \mathit{Bool})$.

The algorithm first *successor matches* the types $(\sigma_k, \sigma_{k+1}) = (K \to \mathit{Bool}, (\mathit{Bool} \to K) \to \mathit{Bool})$. To do that, the successor matching unifies these two types up to an occurs failure arising from the attempt to unify K and $\mathit{Bool} \to K$. The resulting successor matcher is $\theta_1 = \{K := \mathbf{R}\ Y\ (\lambda n'.\lambda K.\mathit{Bool} \to K)\ n\}$ for entirely new Y and n. The matching algorithm then *zero matches* by unifying the base type $\sigma_0 = N \to N$ with the zero projection of $\theta_1\sigma_k$. That is, zero matching unifies $N \to N$ and $(\theta_1\sigma_k)\mathbf{0}\downarrow = \theta_1(K \to \mathit{Bool})\mathbf{0}\downarrow = ((\mathbf{R}\ Y\ (\lambda n'.\lambda K.\mathit{Bool} \to K)\ n) \to \mathit{Bool})\mathbf{0}\downarrow = Y \to \mathit{Bool}$. This gives $\{Y := \mathit{Bool}\}\{N := \mathit{Bool}\}$ for the final zero matching substitution.

Combining the zero and successor matching substitutions gives us the final matcher, $\theta_2\theta_1 = \{Y := \mathit{Bool}\}\{N := \mathit{Bool}\}\{K := \mathbf{R}\ Y\ (\lambda n'.\lambda K.\mathit{Bool} \to K)\ n\}$. The final recursion scheme is $\theta_2\theta_1(\sigma_k) = \theta_2\theta_1(K \to \mathit{Bool}) = (\mathbf{R}\ \mathit{Bool}\ (\lambda n'.\lambda K.\mathit{Bool} \to K)\ n) \to \mathit{Bool}$ where the zero and successor projections of the recursion scheme are as required by Definition 23.

$$\begin{aligned}
\theta_2\theta_1(\sigma_k)\mathbf{0}\downarrow &= \mathit{Bool} \to \mathit{Bool} \\
&= \theta_2\theta_1(\sigma_0) \\
\theta_2\theta_1(\sigma_k)(\mathbf{S}\ n)\downarrow &= (\mathit{Bool} \to \mathbf{R}\ \mathit{Bool}\ (\lambda n'.\lambda K.\mathit{Bool} \to K)\ n) \to \mathit{Bool} \\
&= \theta_2\theta_1(\sigma_{k+1})
\end{aligned}$$

Definition 25 (Zero and Successor matchers) *Let ρ, σ, and τ be weak recursively based types and let $n \notin \mathrm{fv}(\rho, \sigma, \tau)$. A substitution θ is called an* n-indexed successor matcher *(SM) for the pair of types $\langle \sigma, \tau \rangle$ if and only if $\theta\sigma$ and $\theta\tau$ satisfy*

$$\begin{aligned}
&n \in \mathrm{fv}(\theta\sigma), \mathrm{wrb}(\theta\sigma) && \textit{dependence condition} \\
&((\theta\sigma)[n := \mathbf{S}\ n])\downarrow = \theta\tau && \textit{successor condition}
\end{aligned} \tag{3.1}$$

A substitution θ is called an n-indexed zero matcher *(ZM) for the pair of types $\langle \rho, \sigma \rangle$ if and only if $\theta\sigma$ and $\theta\rho$ satisfy*

$$\begin{aligned}
&n \in \mathrm{fv}(\theta\sigma), \mathrm{wrb}(\theta\sigma) && \textit{dependence condition} \\
&(\theta\sigma[n := 0])\downarrow = \theta\rho && \textit{zero condition}
\end{aligned} \tag{3.2}$$

A substitution θ is a successor matcher (or zero matcher) for a weak recursively based system D if θ either unifies or successor matches (zero matches) each pair, but does not unify all *pairs in D.*

We call the variable n in Definition 25 the *dependency index* of the zero or successor matcher. Most general successor matchers (MGSMs) and most general zero matchers (MGZMs) are defined by adopting the the last two conditions for most general unifiers given in Definition 19.

In a disagreement set D, a pair $\langle X, \tau \rangle$ with $X \in \mathrm{fv}(\tau)$ is called a *right-occurs pair* and the variable X is called a *right-occurs variable.* Our systems are not symmetric, for we have no corresponding left-occurs pairs; whenever we say *occurs* we mean right occurs. The following definition of occurs-solved systems is a simple extension to solved systems in the previous section on unification. Most of our focus in matching concerns the handling of occurs pairs.

Definition 26 (Occurs solved systems) *An* occurs-solved system *is a finite set of pairs of the form $D = \{\langle X_1, \tau_1 \rangle, \ldots, \langle X_n, \tau_n \rangle\}$ where X_k for each k is a type variable that does not appear free anywhere else in D except that X_k may appear free in τ_k for any occurs pair $\langle X_k, \tau_k \rangle \in D$.*

The following technical lemma contains details of proving the most generality property for successor matchers of solved systems with occurs pairs. The intuition behind this lemma is that every successor matcher θ' for an occurs pair $\langle X, \tau \rangle$ has the form at X

$$\theta' X = \mathbf{R}\ \rho\ (\lambda n'.\lambda X.\sigma)\ n$$

and the most general successor matcher θ of $\langle X, \tau \rangle$ at X has the very similar form at X

$$\theta X = \mathbf{R}\ Y\ (\lambda n'.\lambda X.\tau)\ n$$

where $\sigma \leq \tau$ by $\theta'|_{-\{X\}}$, or equivalently, $\theta'(\lambda n'.\lambda X.\tau) = \lambda n'.\lambda X.\sigma$. We are, in a sense, solving the occurs pair $\langle X, \tau \rangle$ by creating a general sequence form for the substitution. We call the newly introduced dependency index n the *active dependency index* for the occurs pair and we refer to the newly introduced type variable Y as the *zero variable* for the occurs pair.

Lemma 27 (MGSMs for occurs pairs) *Let D be any weak recursively based system of pairs (not necessarily a solved system), let $\langle X, \tau\rangle \in D$ be an occurs pair, and let θ be the substitution $\{X := \mathbf{R}\ Y\ (\lambda n'.\lambda X.\tau)\ n\}$ with Y and n entirely new, $n, n' \notin \mathrm{fv}(\tau)$, and $X \in \mathrm{fv}(\tau)$ but $\tau \neq X$. Then for any successor matcher θ' of $\langle X, \tau\rangle$ with dependency index n we have the equality $\theta'(X) = \theta''(\theta X)$ where $\theta'' = \theta'\{Y := ((\theta' X)[n := \mathbf{0}])\downarrow\}$.*

Proof Let σ and τ be types with $n \in \mathrm{fv}(\sigma)$ and $n, n' \notin \mathrm{fv}(\gamma)$ and $X \in \mathrm{fv}(\gamma)$ but $\gamma \neq X$. The definition of the R combinator and its relationship with the $R - Succ$ rule imply the following crucial property.

$$\begin{gathered}(\sigma[n := \mathbf{S}\ n])\downarrow = \gamma[X := \sigma] \\ \text{if and only if} \\ \sigma = \mathbf{R}\ \rho\ (\lambda n'.\lambda X.\gamma)\ n \\ \text{for some } \rho \text{ with } n \notin \mathrm{fv}(\rho)\end{gathered} \tag{3.3}$$

This justifies the third equation immediately below.

Suppose θ' is a successor matcher for an occurs pair $\langle X, \tau\rangle \in D$. Then

$$\begin{array}{lcll}((\theta' X)[n := \mathbf{S}\ n])\downarrow & = & \theta'\tau & \text{Definition 25} \\ & = & ((\theta'|_{-\{X\}})\tau)[X := \theta' X] & \text{expanding } \theta' \text{ at } X \\ & = & (\mathbf{R}\ \rho\ (\lambda n'.\lambda X.(\theta'|_{-\{X\}})\tau)\ (\mathbf{S}\ n))\downarrow & \text{property 3.3 above} \\ & & \text{for some } \rho \text{ and where} & \\ & & n \notin \mathrm{fv}(\rho, ((\theta'|_{-\{X\}})\tau) & \\ & & \text{and } n' \notin \mathrm{fv}((\theta'|_{-\{X\}})\tau) & \\ & & \text{and } X \in \mathrm{fv}((\theta'|_{-\{X\}})\tau) & \\ & & \text{but } X \neq (\theta'|_{-\{X\}})\tau & \end{array}$$

These equalities imply

$$
\begin{aligned}
(\theta' X) &= \mathbf{R}\ \rho\ \lambda n'.\lambda X.(\theta'|_{-\{X\}})\tau\ n \\
&\quad \text{by the equalities immediately above} \\
&\quad \text{where } \rho = ((\theta' X)[n := \mathbf{0}])\downarrow \\
&= \mathbf{R}\ \rho\ \theta'(\lambda n'.\lambda X.\tau)\ n \\
&\quad \text{by factoring } \theta' \text{ out} \\
&\quad \text{note } X \text{ is now protected by the lambda binder} \\
&= \mathbf{R}\ ((\theta' X)[n := \mathbf{0}])\downarrow\ \theta'(\lambda n'.\lambda X.\tau)\ n \\
&\quad \text{by the } \textit{R-Zero} \text{ rule on the zero component of } \theta'(X), \text{ ie,} \\
&\quad \text{by plugging in } \rho \text{ from two steps above} \\
&= (\theta'\{Y := ((\theta' X)[n := \mathbf{0}])\downarrow\})(\mathbf{R}\ Y\ (\lambda n'.\lambda X.\tau)\ n) \\
&\quad \text{by factoring } \theta' \text{ out} \\
&= (\theta'\{Y := ((\theta' X)[n := \mathbf{0}])\downarrow\})\theta(X) \\
&\quad \text{by the definition of } \theta(X)
\end{aligned}
$$

Therefore, $\theta' = \theta'' \circ \theta$ where $\theta'' = \theta'\{Y := ((\theta' X)[n := \mathbf{0}])\downarrow\}$ ∎

Lemma 28 (MGSMs for occurs-solved systems) *Let* $D = \{\langle X_1, \tau_1\rangle, \ldots, \langle X_n, \tau_n\rangle\}$ *be an occurs-solved system and be weak recursively based. Let* $\theta = \theta_{id}\{X_i := \tau_i\}\{X_j := \mathbf{R}\ Y_j\ \lambda n'.\lambda X_j.\tau_j\ n\}$ *where* $\{X_i := \tau_i\}$ *is an extension for every non-occurs pair* $\langle X_i, \tau_i\rangle \in D$ *and where the* X_j *extensions are for every occurs pair* $\langle X_j, \tau_j\rangle \in D$. *Assume the* Y_j *are entirely new variables and for each* τ_j, *assume* $n, n' \notin \mathrm{fv}(\tau_j)$ *and* $X_j \in \mathrm{fv}(\tau_j)$ *but* $\tau_j \neq X_j$. *Then* θ *is an n-indexed most general successor matcher (MGSM) of D.*

Proof We must show that

1. θ is a successor matcher for D
2. $\mathrm{dom}(\theta) \subseteq \mathrm{fv}(D)$
3. $\theta' \leq \theta$ for every other successor matcher θ' of D

The second condition holds simply because the domain of θ as constructed in this lemma is exactly the X_k type variables of the first components of the pairs in D. Therefore, $\mathrm{dom}(\theta) \subseteq \mathrm{fv}(D)$.

To show that the substitution θ constructed according to the lemma is in fact a successor matcher we have two cases to consider for each pair $\langle X_k, \tau_k \rangle \in D$: (a) non-occurs pairs $\langle X_i, \tau_i \rangle$ with $\mathrm{fv}(\tau_i) \cap \mathrm{dom}(\theta) = \{\}$ and (b) occurs pairs $\langle X_j, \tau_j \rangle$ with $\mathrm{fv}(\tau_j) \cap \mathrm{dom}(\theta) = \{X_j\}$. For the non-occurs pairs $\langle X_i, \tau_i \rangle$ we must show that θ unifies the pair, thus we have

$$\begin{aligned} \theta X_i &= \tau_i && \text{by Lemma construction of } \theta \\ &= \theta\tau_i && \text{because } \mathrm{dom}(\theta) \cap \mathrm{fv}(\tau_i) = \{\} \end{aligned}$$

For occurs pairs $\langle X_j, \tau_j \rangle \in D$ we have to show

$$\begin{aligned} & n \in \mathrm{fv}(\theta X_j), \mathrm{wrb}(\theta X_j) \\ & ((\theta X_j)[n := \mathbf{S}\ n])\downarrow = \theta\tau_j \end{aligned}$$

We know $n \in \mathrm{fv}(\theta X_j)$ by the lemma construction of θ. To see that θX_j is weak recursively based, observe also by the lemma construction of θ that θX_j is weakly recursive and by the lemma assumptions, the constituent types τ_j and Y_j are weak recursively based. Finally, for the occurs pairs $\langle X_j, \tau_j \rangle$ we have the equality

$$\begin{aligned} ((\theta X_j)[n := \mathbf{S}\ n])\downarrow &= \mathbf{R}\ Y_j\ \lambda n'.\lambda X_j.\tau_j\ (\mathbf{S}\ n) && \text{by Lemma construction of } \theta \\ &= \tau_j[X_j := \mathbf{R}\ Y_j\ \lambda n'.\lambda X_j.\tau_j\ n] && \text{by the } \textit{R-Succ} \text{ rule} \\ &= \tau_j[X_j = \theta X_j] && \text{by definition of } \theta \text{ at } X_j \\ &= \theta\tau_j && \text{because} \\ & && \mathrm{dom}(\theta) \cap \mathrm{fv}(\tau_j) = \{X_j\} \text{ only} \end{aligned}$$

To show the third condition, suppose that θ' is an arbitrary successor matcher of D. We must show that there exists a substitution θ'' such that $\theta' = \theta'' \circ \theta$, thus proving $\theta' \leq \theta$, so θ is the MGSM of D. We know $\theta X_i = \tau_i$ for non-occurs pairs in D by the definition of θ constructed in the statement of this lemma. Therefore we have $\theta'(X_i) = \theta'(\tau_i) = \theta'(\theta X_i)$ so $\theta' = \theta' \circ \theta$ and $\theta' \leq \theta$. For occurs pairs $\langle X_j, \tau_j \rangle \in D$ we have by Lemma 27 that $\theta' = \theta'' \circ \theta$ where $\theta'' = \theta'\{Y_j := ((\theta' X_j)[n := \mathbf{0}])\downarrow\}$ and so $\theta' \leq \theta$ for the occurs case as well as the non-occurs case. ∎

The occurs-matching algorithm is essentially unification with a special way to handle occurs failures. Like the unification algorithm, the occurs-matching algorithm is specified in terms of a simple set of non-deterministic rules that transform disagreement sets into solved form. These rules are given in Figure 3.2. For a disagreement set D and a substitution θ, we say $\theta = \text{Smatch}(n, D)$, where n is the active dependency index, if and only if there exists some finite sequence of transformations $D \Rightarrow \ldots \Rightarrow D'$ where D' is in occurs-solved form, and θ is the most general successor matcher (MGSM) of D' constructed according to Lemma 28 above. If no such transformation of D into solved form exists, then the algorithm fails. Notice that the first five cases of the algorithm are exactly the same as the transformation rules for unification given in Figure 3.1 of Section 3.1 and the sixth case covers the occurs pairs.

Identity

(i) $\quad \{\langle \tau, \tau \rangle\} \cup D \;\Rightarrow\; D$

Decomposition

(ii) $\quad \{\langle \sigma_1 \to \tau_1, \sigma_2 \to \tau_2 \rangle\} \cup D$
$\qquad\qquad \Rightarrow \quad \{\langle \sigma_1, \sigma_2 \rangle, \langle \tau_1, \tau_2 \rangle\} \cup D$

(iii) $\quad \{\langle \Pi n.\sigma, \Pi m.\tau \rangle\} \cup D$
$\qquad\qquad \Rightarrow \quad \{\langle \sigma, \tau[m_1 := n_1] \ldots [m_{k+1} := n_{k+1}] \rangle\} \cup D$

(iv) $\quad \{\langle \mathbf{R}\ \tau_1\ (\lambda n_1'.\lambda X_1.\sigma_1)\ n, \mathbf{R}\ \tau_2\ (\lambda n_2'.\lambda X_2.\sigma_2)\ n \rangle\} \cup D$
$\qquad\qquad \Rightarrow \quad \{\langle \tau_1, \tau_2 \rangle, \langle \sigma_1, \sigma_2[n_2' := n_1'][X_2 := X_1] \rangle\} \cup D$

Variable

(v) $\quad \{\langle X, \tau \rangle\} \cup D$ or $\{\langle \tau, X \rangle\} \cup D$ and $X \notin \text{fv}(\tau)$
$\qquad\qquad \Rightarrow \quad \{\langle X, \tau \rangle\} \cup \theta D$
$\qquad\qquad\quad$ where $\theta = (\theta_{id}\{X := \tau\})$

Variable Occurs

(vi) $\quad \{\langle X, \tau \rangle\} \cup D$ where $X \in \text{fv}(\tau)$ and $\tau \neq X$ and $n \notin \text{fv}(\tau)$
$\qquad\qquad \Rightarrow \quad \{\langle X, \tau \rangle\} \cup \theta D$
$\qquad\qquad\quad$ where $\theta = \theta_{id}\{X := \mathbf{R}\ Y\ \lambda n'.\lambda X.\tau\ n\}$
$\qquad\qquad\quad$ and n is the active dependency index
$\qquad\qquad\quad$ and where Y is entirely new and $n' \notin \text{fv}(\tau)$

Figure 3.2: Transformation Rules for Occurs Matching

We let $\mathrm{SM}(D)$ denote the set of successor matchers for a system of pairs D. The following lemma shows that the occurs-matching transformations preserve the set of successor matchers.

Lemma 29 (Preservation of successor matchers) *If $S \Rightarrow S'$ using any one of the transformations in Figure 3.2, then* $\mathrm{SM}(S) = \mathrm{SM}(S')$.

Proof First consider the identity and decomposition transformations (i) to (iv) where neither of the two types we want to match is a variable. We can argue these cases just as we did in the proof of the preservation of unifiers, Lemma 21, Section 3.1. To successor match two non-variable types, the head symbols must successor match and all subterms must successor match. But the decomposition transformations (i) to (iv) say the same thing, for they match the head symbols and then expand the disagreement set of types that must match by all corresponding pairs of subterms, with appropriate variable renaming. Thus we have for each of the first four cases that

$$\theta' \in \mathrm{SM}(S) \text{ if and only if } \theta' \in \mathrm{SM}(S')$$

For case (v) we have $S = \{\langle X, \tau\rangle\} \cup D$ or $S = \{\langle \tau, X\rangle\} \cup D$ with $X \notin \mathrm{fv}(\tau)$ and $\theta = \theta_{id}\{X := \tau\}$ by the construction of the algorithm. We know that if θ' is a successor matcher of S, and if X is not an occurs variable, then $\theta' X = \theta' \tau$ and the reasoning now closely follows case (v) of the proof of preservation of unifiers Section 3.1, Lemma 21. Specifically, because θ is the single assignment $\theta = \theta_{id}\{X := \tau\}$, then θ' and $\theta' \circ \theta$ differ only at X where $\theta(X) = \tau$, that is, we have $\theta'(X) = \theta'(\tau) = \theta'(\theta X)$ and therefore $\theta' = \theta' \circ \theta$ at all domain elements. We use this last equality in reaching the third equivalence below. Thus we have

$$\begin{aligned}
\theta' \in \mathrm{SM}(S) &\Leftrightarrow \theta' \in \mathrm{SM}(\{\langle X, \tau\rangle\} \cup D) \text{ or } \theta' \in \mathrm{SM}(\{\langle \tau, X\rangle\} \cup D) \text{ and } X \notin \mathrm{fv}(\tau) \\
&\Leftrightarrow ((\theta' X)[n := \mathbf{S}\ n])\downarrow = \theta'\tau \text{ and } \theta' \in \mathrm{SM}(D) \\
&\Leftrightarrow ((\theta' X)[n := \mathbf{S}\ n])\downarrow = \theta'\tau \text{ and } \theta' \circ \theta \in \mathrm{SM}(D) \\
&\Leftrightarrow ((\theta' X)[n := \mathbf{S}\ n])\downarrow = \theta'\tau \text{ and } \theta' \in \mathrm{SM}(\theta D) \\
&\Leftrightarrow \theta' \in \mathrm{SM}(\{\langle X, \tau\rangle\} \cup \theta D) \\
&\Leftrightarrow \theta' \in \mathrm{SM}(S')
\end{aligned}$$

Finally, for the occurs case (vi) we have reasoning similar to case (v), provided we take into account the occurs variables. According to the construction in the algorithm, $S = \{\langle X, \tau\rangle\} \cup D$ where $\theta = \theta_{id}\{X := \mathbf{R}\ Y\ \lambda n'.\lambda X.\tau\ n\}$ and where $n, n' \notin \mathrm{fv}(\tau)$ and $X \in \mathrm{fv}(\tau)$ but $\tau \neq X$. In reaching the second equivalence below we use the fact that θ' is a successor matcher of S, so $((\theta' X)[n := \mathbf{S}\ n])\downarrow = \theta'(\tau)$. To justify the third equivalence we can appeal to Lemma 27 above to assert $\theta' = \theta'\{Y := ((\theta' X)[n := \mathbf{0}])\downarrow\} \circ \theta$. The last equivalence below is justified because the extension assigning Y the zero component of θ' does not affect the successor-matching property we are concerned with here.[4] Thus we have

$$
\begin{aligned}
\theta' \in \mathrm{SM}(S) &\Leftrightarrow \theta' \in \mathrm{SM}(\{\langle X, \tau\rangle\} \cup D) \text{ and } X \in \mathrm{fv}(\tau) \\
&\Leftrightarrow ((\theta' X)[n := \mathbf{S}\ n])\downarrow = \theta'\tau \text{ and } \theta' \in \mathrm{SM}(D) \\
&\Leftrightarrow ((\theta' X)[n := \mathbf{S}\ n])\downarrow = \theta'\tau \text{ and } \theta'\{Y := ((\theta' X)[n := \mathbf{0}])\downarrow\} \circ \theta \in \mathrm{SM}(D) \\
&\Leftrightarrow ((\theta' X)[n := \mathbf{S}\ n])\downarrow = \theta'\tau \text{ and } \theta'\{Y := ((\theta' X)[n := \mathbf{0}])\downarrow\} \in \mathrm{SM}(\theta D) \\
&\Leftrightarrow \theta'\{Y := ((\theta' X)[n := \mathbf{0}])\downarrow\} \in \mathrm{SM}(\{\langle X, \tau\rangle\} \cup \theta D) \\
&\Leftrightarrow \theta'\{Y := ((\theta' X)[n := \mathbf{0}])\downarrow\} \in \mathrm{SM}(S') \\
&\Leftrightarrow \theta' \in \mathrm{SM}(S')
\end{aligned}
$$

∎

The following theorem supports the proof of the matching theorem by separately analyzing the successor matching.

Theorem 30 (Successor matching) *Let σ and τ be weak recursively based types. Then $\theta = \mathrm{Smatch}(n, \{\langle \sigma, \tau\rangle\})$ if and only if θ is the most general successor matcher of $\{\langle \sigma, \tau\rangle\}$.*

Proof The Smatch algorithm terminates by the same reasoning as in the proof of termination of the Unification Theorem 22. The complexity measure (a, b) on disagreement sets D of pairs of types has a as the number of occurrences of free variables in D and b

[4]Note that most general successor matchers always have a unique zero variable Y in the zero position of the **R** sequence type assigned to each each occurs variable X. There are never zero variables Y in the successor position of sequence types of MGSMs.

as the sum of the sizes of the types. Then just as in the proof of the Unification Theorem, the identity and decomposition transformations (i) to (iv) strictly decrease b and the variable transformations (v) and (vi) strictly decrease a. Therefore all transformation sequences correspond to a strictly decreasing sequence in the lexicographic ordering (a, b) of complexity measures on disagreement sets.

The soundness and completeness parts of this proof also closely follow the arguments in the proof of the Unification Theorem. To show soundness, suppose $\theta = \text{Smatch}(n, \{\langle \sigma, \tau \rangle\})$, that is, there is a finite sequence of transformations on disagreement sets $D \Rightarrow \ldots \Rightarrow D'$ such that D' is a solved system with MGSM θ constructed according to Lemma 28. We can use induction on k, the length of the transformation sequences, to show that $\text{SM}(D) = \text{SM}(D')$ and therefore any successor matcher of the final solved system D' is also a successor matcher of the original system D. The basis $k = 0$ of the induction is trivial and the step cases of the induction are given by the preservation of successor matchers, Lemma 29.

To show completeness we have to show $\theta' \leq \theta$ for every SM θ' of $D = \{\langle \sigma, \tau \rangle\})$, where $\theta = \text{Smatch}(n, D)$. We know Smatch terminates on D yielding substitution θ, so there must exist a finite sequence of transformations $D \Rightarrow \ldots \Rightarrow D'$ with D' in solved form, and with θ the MGSM of D' by Lemma 28. By the soundness proof above, we know θ is also a successor matcher of the original system D, Let θ' be an arbitrary SM of D. By the preservation of successor matchers, Lemma 29, θ' is also a unifier of D'. But θ is the MGSM of D', so $\theta' \leq \theta$ in the solved system D'. But then we also have θ and θ' both SMs of D with $\theta' \leq \theta$, therefore θ is the MGSM of D. ∎

The second part of the matching algorithm uses the unification algorithm of Section 3.1 to find the base case of a recursively specified type after successor matching. The final matching algorithm shown in Figure 3.3 is a combination of the algorithms to find the occurs case (the successor matcher) and the base case (the zero matcher) of a recursively specified type. The only tricky part is the replacement that is applied to σ before applying θ_1 in line 2. This trick is used to remove the successor matching assignment for each occurs variable X_j, namely $\{X_j := \mathbf{R}\ Y_j\ \lambda n'.\lambda X_j.\tau_j\ n\}$, and replace it by the assignment

$\{X_j := Y_j\}$ of just the zero components of $\theta_1 X_j$. Replacing the X_js by the Y_js allows us to use our earlier Unify algorithm to find a substitution for the Y_j zero component of the sequence type for X_j. Note that for any occurs variable assignment $\theta_1 X_j$, the Y_j zero component variable is just the zero projection, $((\theta_1 X_j)[n := \mathbf{0}])\downarrow$.

$$\begin{aligned}\text{Match}(n, \langle \rho, \sigma, \tau \rangle) = \quad & \text{Let } \theta_1 = \text{Smatch}(n, \{\langle \sigma, \tau \rangle\}) \\ & \text{and } \theta_2 = \text{Unify}(\{\langle \rho, \theta_1(\sigma[X_j := \eta])\rangle\}) \\ & \text{where } \eta = ((\theta_1(X_j))[n := \mathbf{0}])\downarrow \\ & \text{in } \theta_2 \circ \theta_1 \\ & \text{where } n \notin \text{fv}(\rho, \sigma, \tau)\end{aligned}$$

Figure 3.3: Matching Algorithm for Weak-recursion Schemes

The matching theorem shows termination, soundness, and completeness of the matching algorithm. According to this theorem, if there exists some matcher, then the algorithm is guaranteed to return the most general matcher. This aspect of the matching theorem is used in the proof of the principal types theorem for the $\mathcal{T}^\pi$ system. Matching is complete with respect to the weak $\mathcal{T}^\pi$ system specified in Definition 4, Section 2.1.

Theorem 31 (Matching Theorem) *Let ρ, σ, τ be any three weak recursively based $\mathcal{T}^\pi$ types. Then*

1. **Termination** *The matching algorithm $Match(n, \{\langle \rho, \sigma, \tau \rangle\})$ always terminates, either successfully returning a substitution θ, or else failing.*

2. **Soundness and Completeness** *If σ, τ do not unify and $\text{fv}(\rho) \cap \text{fv}(\sigma, \tau) = \{\}$, then $\theta = Match(n, \{\langle \rho, \sigma, \tau \rangle\})$ if and only if θ is a most general n-indexed weak-recursion matcher of $\{\langle \rho, \sigma, \tau \rangle\}$. Otherwise Match returns fail.*

Proof of termination and soundness The proof of termination is immediate from the termination of Smatch and Unify. To prove soundness, assume that σ and τ do not unify and together do not share free variables with ρ. Let $\theta = \text{Match}(n, \{\langle \rho, \sigma, \tau \rangle\})$. We want to show that θ is an n-indexed weak recursion matcher for $\langle \rho, \sigma, \tau \rangle$, that is,

1. $n \in \text{fv}(\theta\sigma), \text{wrb}(\theta\sigma)$

2. $((\theta\sigma)[n := \mathbf{S}\ n])\downarrow = \theta\tau$

3. $((\theta\sigma)[n := 0])\downarrow = \theta\rho$

For item 1 we know that $n \in \mathrm{fv}(\theta\sigma)$ after the call to the Smatch algorithm provided there is at least one occurs variable in $\mathrm{dom}(\theta)$. But there must be at least one occurs variable because σ, τ do not unify and yet Smatch succeeds by our assumption. To show $\theta\sigma$ is weak recursively based, we must show that $\theta_2 \circ \theta_1(\sigma)$ is weak recursively based. But both the Smatch algorithm and the Unify algorithm preserve weak recursion, therefore, $\theta_1(\sigma)$ is weak recursivly based and so is $\theta_2(\theta_1(\sigma))$.

For item 2, the successor-matching case, suppose $\theta = \theta_2 \circ \theta_1$ where θ_1 and θ_2 are computed according to the Match algorithm, Figure 3.3. That is, θ_1 comes from the Smatch call in the first step and θ_2 comes from the Unify call in the second step. Then we have

$$
\begin{aligned}
((\theta\sigma)[n := \mathbf{S}\ n])\downarrow &= (((\theta_2 \circ \theta_1)\sigma)[n := \mathbf{S}\ n])\downarrow \\
&= ((\theta_2(\theta_1\sigma))[n := \mathbf{S}\ n])\downarrow \\
&= \theta_2((\theta_1\sigma)[n := \mathbf{S}\ n])\downarrow && \text{because } n \text{ is not introduced by } \theta_2 \\
&= \theta_2(\theta_1\tau) && \text{by the soundness of Smatch,} \\
& && \text{Theorem 28, used in} \\
& && \text{line one of the Match algorithm} \\
&= (\theta_2 \circ \theta_1)\tau \\
&= \theta\tau
\end{aligned}
$$

Finally, for item 3, the zero matching case, assume again that we have computed θ_1 according to the Smatch call in line 1 of the algorithm and computed θ_2 according to the

unify in line 2 of the algorithm. Then we have the following equations

$$
\begin{aligned}
(\theta\sigma[n := \mathbf{0}])\downarrow &= (((\theta_2 \circ \theta_1)\sigma)[n := \mathbf{0}])\downarrow \\
&\text{by the definition of } \theta \text{ in the algorithm} \\
&= ((\theta_2(\theta_1\sigma))[n := \mathbf{0}])\downarrow \\
&= \theta_2((\theta_1\sigma)[n := \mathbf{0}])\downarrow \\
&\text{because } n \text{ is not introduced by } \theta_2 \\
&= \theta_2((\theta_1|_{-\{X_j\}}(\sigma[X_j := \theta_1 X_j]))[n := \mathbf{0}])\downarrow \\
&\text{separating out the occurs assignments} \\
&= \theta_2(\theta_1|_{-\{X_j\}}(\sigma[X_j := ((\theta_1 X_j)[n := \mathbf{0}])\downarrow] \ ) \ ) \\
&\text{by distributing the zero projection inward} \\
&\text{note } n \text{ is not introduced by } \theta_1 \text{ except at } X_j \\
&= \theta_2(\theta_1(\sigma[X_j := ((\theta_1 X_j)[n := \mathbf{0}])\downarrow] \ ) \ ) \\
&\text{because } X_j \notin \sigma[X_j := ((\theta_1 X_j)[n := \mathbf{0}])\downarrow] \\
&= \theta_2\rho \\
&\text{by the soundness of the Unify algorithm} \\
&\text{called in line 2 of the Match algorithm} \\
&\text{to unify } \rho \text{ and } \sigma[X_j := ((\theta_1 X_j)[n := \mathbf{0}])\downarrow] \\
&= \theta_2(\theta_1\rho) \\
&\text{because } \mathrm{dom}(\theta_1) \cap \mathrm{fv}(\rho) = \{\} \\
&= (\theta_2 \circ \theta_1)\rho \\
&= \theta\rho
\end{aligned}
$$

We have covered the three items we wished to prove and so our soundness proof is now complete. ∎

Now we can conclude the proof of the Matching theorem with the proof of completeness.

Proof of Completeness

1. Assume θ is a weak-recursive matcher of the triple of types $\langle \rho, \sigma, \tau \rangle$.

2. Then θ is a successor matcher (SM) of $\langle \sigma, \tau \rangle$.

3. By the preservation of SM's, Theorem 29, θ is also an SM of the occurs-solved system of the initial pair $\{\langle\sigma,\tau\rangle\}$.

4. Let $\theta_1 = \text{Smatch}(\langle\sigma,\tau\rangle)$.

5. By the successor-matching theorem, then θ_1 is the most general SM for the pair $\langle\sigma,\tau\rangle$.

6. This implies $\theta \leq \theta_1$, that is, $\theta = \theta' \circ \theta_1$ for some θ'.

7. Because θ a weak-recursive matcher of the triple of types $\langle\rho,\sigma,\tau\rangle$, we have that θ is also a zero matcher (ZM) of the pair $\langle\rho,\sigma\rangle$, thus $((\theta\sigma)[n := \mathbf{0}])\downarrow = \theta\rho$.

8. From the previous two steps showing that θ is a ZM and $\theta = \theta' \circ \theta_1$, we have $(((\theta' \circ \theta_1)\sigma)[n := \mathbf{0}])\downarrow = (\theta' \circ \theta_1)\rho$ that is, $((\theta'\theta_1\sigma)[n := \mathbf{0}])\downarrow = \theta'\theta_1\rho$.

9. We know $\theta_1\rho = \rho$ because $\text{dom}(\theta_1) \cap \text{fv}(\rho) = \{\}$ as a result of the assumption that $\text{fv}(\rho) \cap \text{fv}(\sigma,\tau) = \{\}$. The intuition here is that successor matchers do not affect zero case types.

10. Thus we have that θ' is a ZM for $\{\langle\rho,\theta_1\sigma\rangle\}$ because the previous two steps imply $((\theta'\theta_1\sigma)[n := \mathbf{0}])\downarrow = \theta'\theta_1\rho = \theta'\rho$, which is the definition of a zero matcher for $\{\langle\rho,\theta_1\sigma\rangle\}$.

11. If θ' is a ZM for $\langle\rho,\theta_1\sigma\rangle$ then θ' must be a unifier for $\langle\rho,\theta_1(\sigma[X_j := \eta_j])\rangle$ for all X_j occurs variables in $\text{dom}(\theta_1)$, where $\eta_j = ((\theta_1 X_j)[n := \mathbf{0}])\downarrow = Y_j$, the zero variable of the assignment to the occurs variable X_j. To see that θ' is such a unifier, first note that the definition of what it means for θ' to be a zero matcher for types $\langle\sigma,\tau\rangle$ collapses to the same definition as unify when there are no dependency indices n in $\theta'\sigma$. Now all dependency indices arise from occurs variables $X_j \in \text{dom}(\theta')$ and we are replacing them with non-occurs variables Y_i obtained from the zero component of $\theta' X_j$. Therefore, θ' collapses from a zero matcher to a unifier of ρ and $\theta_1(\sigma[X_j := \eta_j])$ for all X_j occurs variables in $\text{dom}(\theta_1)$. Thus we have $\theta'(\theta_1(\sigma[X_j := \eta_j])) = \theta'\rho$ where $\eta_j = ((\theta_1 X_j)[n := \mathbf{0}])\downarrow = Y_j$, the zero variable of the assignment to the occurs variable X_j.

12. Now, by Lemma 21 (Preservation of Unifiers), θ' is a unifier of the solved system D' of $D = \{\langle \rho, \theta_1(\sigma[X_j := \eta_j])\rangle\}$. Again, the variables X_j range over all occurs variables in $\mathrm{dom}(\theta_1)$ and the η_js are the zero variables of the assignment to the occurs variable X_j, that is, $\eta_j = ((\theta_1 X_j)[n := \mathbf{0}])\!\downarrow\!= Y_j$.

13. Let $\theta_2 = \mathrm{Unify}(\{\langle \rho, \theta_1(\sigma[X_j := \eta_j])\rangle\})$. By the unification theorem, θ_2 is the MGU of the solved system D' of D and therefore $\theta' \leq \theta_2$, that is, $\theta = \theta'' \circ \theta_2$ for some substitution θ''.

14. From step 6 we have $\theta = \theta' \circ \theta_1$ and from the previous step we have $\theta' = \theta'' \circ \theta_2$, thus $\theta = \theta'' \circ \theta_2 \circ \theta_1$ and so $\theta \leq \theta_2 \circ \theta_1$.

15. Finally, observe that $\theta_2 \circ \theta_1$ is exactly what the Match algorithm computes (see Figure 3.3), thus from the soundness part of this theorem we know that $\theta_2 \circ \theta_1$ is a weak-recursive matcher of $\langle \rho, \sigma, \tau \rangle$ and, from the previous step, it is the most general one.

∎

3.3 Principal Types

We will prove a principal types theorem for the weak $\mathcal{T}^\pi$ type system using the weak $\mathcal{T}^\pi$ unification and matching theorems of the previous sections. This theorem is analogous to the principal types theorem for the simply typed lambda calculus that relies on unification. The proof is constructive and, in fact, the reconstruction algorithm given in Section 3.4 is extracted from this proof. The reader may wish to consult the algorithm in Figure 3.4 of that section to get a feel for the structure of this proof.

Let us extend type subsumption to contexts and to sequences of types and contexts. We say $\Gamma' \leq \Gamma$ over a set of type variables V if and only if $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma)$ and $\forall x \in \mathrm{dom}(\Gamma), \Gamma'(x) \leq \Gamma(x)$[5]. Extend the subsumption order to sequences by letting

[5]Recall contexts are finite mappings.

$(\tau_1, \ldots, \tau_n) \leq (\sigma_1, \ldots, \sigma_n)$ mean the conjunction of the subsumptions $\tau_i \leq \sigma_i$ *by the same substitution.*

Definition 32 (Principal Types) *A type τ is a* Principal Type (PT) *for a term δ if and only if for some Γ*

1. $\Gamma \vdash \delta : \tau$,

2. *for every other Γ' and τ' such that $\Gamma' \leq \Gamma$ and $\Gamma' \vdash \delta : \tau'$ then $\tau' \leq \tau$.*

Definition 33 (Principal Derivations) *A derivation D in the weak $\mathcal{T}^\pi$ type system is a* Principal Derivation (PD) *for a term δ if and only if for every other $\Gamma' \vdash \delta : \tau'$ for term δ, then $(\Gamma', \tau') \leq (\Gamma, \tau)$.*

If $\Gamma \vdash \delta : \tau$ is a principal derivation for δ, then τ is a principal type for δ by examination of the definitions above. The existence proof for principal types is therefore a corollary of the existence proof for principal derivations. Principal types and derivations are unique modulo variable name changes, should they exist.

The proof of the principal derivation theorem is constructive and and uses the constructive proofs for the unification and matching theorems. We will extract from these proofs an algorithm for constructing principal weak $\mathcal{T}^\pi$ types for untyped $\mathcal{T}^\pi$ terms in Section 3.4.

Theorem 34 (Existence of Principal Derivations) *If there is a derivation $\Gamma' \vdash \delta : \tau'$ for a term δ in the weak $\mathcal{T}^\pi$ type system, then there is a principal derivation $\Gamma \vdash \delta : \tau$ for δ.*

Proof The proof is by induction on the structure of the given derivation $\Gamma' \vdash \delta : \tau'$ and depends on the weak $\mathcal{T}^\pi$ unification and matching theorems. The most interesting cases are the $\rightarrow$*elim* and Π*intro-seq* rules, which are covered here along with the cases Π*intro* and Π*elim*. The remaining cases follow standard proofs of principal derivations for the typed lambda calculus.

$\rightarrow$*elim* case. Suppose the last step in the given derivation uses the $\rightarrow$*elim* rule, so the form of the final sequent is

$$\Gamma' \vdash \delta\ \delta' : \beta \tag{3.4}$$

and the form of the immediate premises are

$$\Gamma' \vdash \delta : (\alpha \to \beta) \tag{3.5}$$

$$\Gamma' \vdash \delta' : \alpha \tag{3.6}$$

By the induction hypothesis, δ and δ' have principal derivations that we can suppose have the following forms, where we assume the variables in each of these derivations do not overlap

$$\Gamma_1 \vdash \delta : \pi \tag{3.7}$$

$$\Gamma_2 \vdash \delta' : \xi \tag{3.8}$$

By the definition of principal derivations, we can infer the following subsumptions for the principal derivations 3.7 and 3.8.

$$(\Gamma_1, \pi) \quad \geq_{\theta_1} \quad (\Gamma', \alpha \to \beta) \tag{3.9}$$

$$(\Gamma_2, \xi) \quad \geq_{\theta_2} \quad (\Gamma', \alpha) \tag{3.10}$$

We know that the type π must have an arrow structure or else itself be a type variable, so consider unifying the following pairs

$$(\Gamma_1, \pi) \tag{3.11}$$

$$(\Gamma_2, \xi \to B) \tag{3.12}$$

where B is an entirely new type variable, specifically, B is not in π or ξ. We know these sequences are unifiable according to the unification theorem for $\mathcal{T}^\pi$ types because they both share a common instance $(\Gamma', \alpha \to \beta)$. To see how 3.9 and 3.10 share a common instance, examine the following subsumption constructed from the subsumptions 3.9 and 3.10 above with an extension to the substitution θ_2 by the assignment of the new variable B.

$$(\Gamma_2, \xi \to B) \quad \geq_{\theta_2\{B:=\beta\}} \quad (\Gamma', \alpha \to \beta) \tag{3.13}$$

Rewriting the subsumptions 3.9 and 3.13 we get

$$\begin{array}{rll} (\Gamma_1, \pi) & \geq_{\theta_1} & (\Gamma', \alpha \to \beta) \\ (\Gamma_2, \xi \to B) & \geq_{\theta_2\{B:=\beta\}} & (\Gamma', \alpha \to \beta) \end{array} \tag{3.14}$$

Note that the free type variables of π and ξ do not overlap by our assumed conventions in 3.7 and 3.8 above, and neither π nor ξ include B. Therefore, $\mathrm{dom}(\theta_1) \cap \mathrm{dom}(\theta_2) \cap \{B\} = \{\}$, that is, the substitutions are independent and can be combined to form a single substitution $\hat{\theta} \stackrel{\mathrm{def}}{=} \theta_1 + \theta_2\{B := \beta\}$ where the $+$ is disjoint set union. Now we can apply $\hat{\theta}$ to each of 3.11 and 3.12 to get the common instance $(\Gamma', \alpha \to \beta)$.

Because 3.11 and 3.12 have a common instance by a single substitution $\hat{\theta}$, we can now appeal to the unification theorem for weak $\mathcal{T}^{\pi}$ types and assert the existence of a most general common instance $(\Gamma, \sigma \to \tau)$ by a most general unifier θ. Therefore, by the definition of most general unifiers, $\hat{\theta} = \theta' \circ \theta$ for some θ' and we have the following desired subsumptions by θ and θ'.

$$\begin{array}{lll} (\Gamma_1, \pi) & \geq_{\theta} (\Gamma, \sigma \to \tau) & \geq_{\theta'} (\Gamma', \alpha \to \beta) \\ (\Gamma_2, \xi) & \geq_{\theta} (\Gamma, \sigma) & \geq_{\theta'} (\Gamma', \alpha) \end{array} \tag{3.15}$$

By applying the substitution θ to the principal derivations 3.7 and 3.8 of the induction hypothesis we get the following derivations.

$$\theta\Gamma_1 \vdash \delta : \theta\pi \;=\; \Gamma \vdash \delta : \sigma \to \tau \tag{3.16}$$

$$\theta\Gamma_2 \vdash \delta' : \theta\xi \;=\; \Gamma \vdash \delta' : \sigma \tag{3.17}$$

Using the $\to$*elim* rule, we can construct a new derivation from 3.16 and 3.17

$$\Gamma \vdash \delta\ \delta' : \tau \tag{3.18}$$

We now claim that the derivation 3.18 is a principal derivation. To show derivation 3.18 is principal, we must show that for all other derivations $\Gamma_3 \vdash \delta\ \delta' : \tau_3$ with $\Gamma_3 \leq \Gamma$ then $(\Gamma_3, \tau_3) \leq (\Gamma, \tau)$. In as much as the original derivation 3.4 was completely arbitrary, it is enough to show that $(\Gamma', \beta) \leq (\Gamma, \tau)$. But $(\Gamma', \beta) \leq (\Gamma, \tau)$ follows directly from 3.15 where

$$(\Gamma', \alpha \to \beta) \;\leq (\Gamma, \sigma \to \tau) \tag{3.19}$$

The proof of the $\to$*elim* case is complete, that is, $\Gamma \vdash \delta\ \delta' : \tau$ is a principal derivation for $\delta\ \delta'$.

Π*intro-seq* case. Suppose the last step in the derivation uses the Π*intro-seq* rule so that the form of the conclusion of the final step is the sequent

$$\Gamma' \vdash \mathbf{R}\ \delta'\ \delta\ : \Pi n.\gamma \tag{3.20}$$

The form of the immediate premises and conditions of this last step are

$$\Gamma' \vdash \delta' : \alpha \tag{3.21}$$

$$\Gamma'\{n : Nat\} \vdash \delta : Nat \to \gamma \to \beta \tag{3.22}$$

$$\begin{array}{ll} \text{where} & n \notin \mathrm{fv}(\delta), n \in \mathrm{fv}(\gamma), \mathrm{wrb}(\gamma) \\ \text{and where} & (\gamma[n := 0])\downarrow = \alpha \\ & (\gamma[n := \mathbf{S}\ n])\downarrow = \beta \end{array} \tag{3.23}$$

By the induction hypothesis, δ and δ' have principal derivations that we can suppose have the forms

$$\Gamma_1 \vdash \delta' : \xi \tag{3.24}$$

$$\Gamma_2 \vdash \delta : \pi \tag{3.25}$$

where we assume that the free variables of these two derivations do not overlap. We can infer from the properties of principal derivations that the following subsumptions hold

$$(\Gamma_1, \xi) \ \geq_{\theta_1} \ (\Gamma'\{n : Nat\}, \alpha) \tag{3.26}$$

$$(\Gamma_2, \pi) \ \geq_{\theta_2} \ (\Gamma'\{n : Nat\}, Nat \to \gamma \to \beta) \tag{3.27}$$

We expect to construct the next step of the principal derivation using the Π*intro-seq* rule and therefore the type π must have the form $Nat \to \sigma \to \tau$, so consider unifying the following sequences, where A, B, and C are entirely new.

$$(\Gamma_1, \xi, \pi) \tag{3.28}$$

$$(\Gamma_2, C, Nat \to A \to B) \tag{3.29}$$

We know these sequences are unifiable according to the unification theorem because they both share the common instance

$$(\Gamma'\{n : Nat\}, \alpha, Nat \to \gamma \to \beta)$$

To see this, examine the following subsumptions constructed from equations 3.26 and 3.27.

$$\begin{array}{rll} \Gamma_1 & \geq_{\theta_1} & \Gamma'\{n : Nat\} \\ \Gamma_2 & \geq_{\theta_2} & \Gamma'\{n : Nat\} \\ \\ \xi & \geq_{\theta_1} & \alpha \\ C & \geq_{\theta_{id}\{C:=\alpha\}} & \alpha \\ \\ \pi & \geq_{\theta_2} & Nat \rightarrow \gamma \rightarrow \beta \\ Nat \rightarrow A \rightarrow B & \geq_{\theta_{id}\{A:=\gamma\}\{B:=\beta\}} & Nat \rightarrow \gamma \rightarrow \beta \end{array} \tag{3.30}$$

By our assumptions on the derivations 3.24 and 3.25, we know the type variables of π and ξ do not overlap and do not contain the new type variables A, B, or C so that $\text{dom}(\theta_1) \cap \text{dom}(\theta_2) \cap \{A, B, C\} = \{\}$. Therefore we can define $\hat{\theta} \stackrel{\text{def}}{=} \theta_1 + \theta_2 + \theta_{id}\{C := \alpha\}\{A := \gamma\}\{B := \beta\}$, where $+$ is disjoint set union, and get the same subsumptions as above, but with the single unifier $\hat{\theta}$. Therefore, the two type sequences (Γ_1, ξ, π) and $(\Gamma_2, C, Nat \rightarrow A \rightarrow B)$ of 3.28 and 3.29 above have a common sequence $(\Gamma'\{n : Nat\}, \alpha, Nat \rightarrow \gamma \rightarrow \beta)$ by the unifier $\hat{\theta}$.

According to the Theorem 22 Section 3.1 (Unification) we can assert the existence of a most general common sequence of types, $(\Gamma\{n : Nat\}, \rho, Nat \rightarrow \sigma \rightarrow \tau)$, by a most general unifier θ^*, where $\hat{\theta} = \theta' \circ \theta^*$ for some θ'. Thus we have the following subsumptions.

$$\left.\begin{array}{r} \Gamma_1 \\ \Gamma_2 \end{array}\right\} \geq_{\theta^*} \Gamma\{n : Nat\} \geq_{\theta'} \Gamma'\{n : Nat\}$$

$$\left.\begin{array}{r} \xi \\ C \end{array}\right\} \geq_{\theta^*} \rho \geq_{\theta'} \alpha \tag{3.31}$$

$$\left.\begin{array}{r} \pi \\ Nat \rightarrow A \rightarrow B \end{array}\right\} \geq_{\theta^*} Nat \rightarrow \sigma \rightarrow \tau \geq_{\theta'} Nat \rightarrow \gamma \rightarrow \beta$$

The substitution θ^* is the most general substitution that gives us the most general arrow structure of the premise types. Next we want to find the most general weak-recursion matcher θ that makes (ρ, σ, τ) into a weak-recursion scheme. According Definition 24 Section 3.2 (Weak-recursion Matchers), finding a weak-recursion matcher θ for (ρ, σ, τ)

means finding θ such that

$$\begin{aligned} & n \in \mathrm{fv}(\theta\sigma), \mathrm{wrb}(\theta\sigma) \\ & ((\theta\sigma)[n := 0])\downarrow = \theta\rho \\ & ((\theta\sigma)[n := \mathbf{S}\ n])\downarrow = \theta\tau \end{aligned} \tag{3.32}$$

We know that such a matcher exists, namely θ', because (α, γ, β) is a weak-recursion scheme by 3.23 and according to equation 3.31

$$(\theta'\rho, \theta'\sigma, \theta'\tau) = (\alpha, \gamma, \beta)$$

and therefore substituting into our original assumptions 3.23 we get

$$\begin{aligned} & n \in \mathrm{fv}(\theta'\sigma), \mathrm{wrb}(\theta'\sigma) \\ & ((\theta'\sigma)[n := 0])\downarrow = \theta'\rho \\ & ((\theta'\sigma)[n := \mathbf{S}\ n])\downarrow = \theta'\tau \end{aligned} \tag{3.33}$$

These equations tell us that θ' makes the triple of types (ρ, σ, τ) into a weak-recursion scheme and thus θ' is by definition a weak-recursion matcher.

Knowing that the weak-recursion matcher θ' *exists* for (ρ, σ, τ) means we can appeal to Theorem 31 Section 3.2 (Matching) and assert that there is a *most general* weak-recursion matcher θ and that $(\theta\rho, \theta\sigma, \theta\tau)$ is the most general weak-recursion scheme obeying the equations 3.32.

Now compose the substitution θ^* and then the weak-recursion matcher θ and apply them to the principal derivations 3.24 and 3.25 to get the derivations[6]

$$\theta\theta^*\Gamma_1 \vdash \delta' : \theta\theta^*\xi \quad = \quad \theta\Gamma\{n : Nat\} \vdash \delta' : \theta\rho \Rightarrow \theta\Gamma \vdash \delta' : \theta\rho \tag{3.34}$$

$$\theta\theta^*\Gamma_2 \vdash \delta : \theta\theta^*\pi \quad = \quad \theta\Gamma\{n : Nat\} \vdash \delta : Nat \to \theta\sigma \to \theta\tau \tag{3.35}$$

Next use the premises 3.34 and 3.35 and the weak-recursion matching conditions of 3.32 (that we've shown to hold for θ) and invoke the Π*intro-seq* rule to construct the conclusion

$$\theta\Gamma \vdash \mathbf{R}\ \delta'\ \delta : \Pi n.\theta\sigma \tag{3.36}$$

We now claim that sequent 3.36 is a principal derivation. In as much as the original postulated derivation $\Gamma' \vdash \mathbf{R}\ \delta'\ \delta : \Pi n.\gamma$ was completely arbitrary, we must show that

[6]Note in 3.34 the context $\theta\gamma\{n : Nat\}$ can be restricted to $\theta\gamma$ because $n \notin \mathrm{fv}(\theta\rho, \delta')$.

$(\Gamma', \Pi n.\gamma) \leq (\theta\Gamma, \Pi n.\theta\sigma)$. But this last subsumption $(\Gamma', \Pi n.\gamma) \leq (\theta\Gamma, \Pi n.\theta\sigma)$ follows directly from subsumptions 3.31, where $(Nat \to \sigma \to \tau) \geq (Nat \to \gamma \to \beta)$. The proof of the Π*intro-seq* case is complete.

Π*intro* case. Suppose the last step in the derivation uses the Π*intro* rule, so that the form of the final sequent is

$$\Gamma' \vdash \lambda n.\delta : \Pi n.\tau$$

and the form of the immediate premise is

$$\Gamma'\{n : Nat\} \vdash \delta : \tau$$

By the induction hypothesis, δ has a principal derivation that we can suppose has the form

$$\Gamma_1\{n : Nat\} \vdash \delta : \sigma \tag{3.37}$$

We can apply the Π*intro* rule to sequent 3.37 and derive

$$\Gamma_1 \vdash \lambda n.\delta : \Pi n.\sigma \tag{3.38}$$

By the definition of principal derivations we have the subsumption $(\Gamma_1\{n : Nat\}, \sigma) \geq (\Gamma'\{n : Nat\}, \tau)$ from the premise derivation 3.37. From this last subsumption we can immediately infer the required subsumption for the principality of sequent 3.38

$$(\Gamma_1, \Pi n.\sigma) \geq_{\theta_1} (\Gamma', \Pi n.\tau) \tag{3.39}$$

In as much as the original derivation was completely arbitrary, we can conclude that the derivation 3.38 is a principal derivation.

Π*elim* case. The proof is a simplification of the proof of the $\to$*elim* case above because the types of the operand and the abstraction variable must both be *Nat* and therefore unification is not needed. This case does, however, rely on the following property.

$$\Pi n.\sigma \geq_{\theta_1} \Pi n.\tau \text{ and } \delta'{\downarrow} = \mathbf{S}^k\mathbf{0} \text{ for some } k \in \omega$$

implies

$$(\sigma[n := \delta']){\downarrow} \geq_{\theta_1} (\tau[n := \delta']){\downarrow}\,.$$

We have completed proofs for the four relevant cases of the proof of existence of principal derivations in $\mathcal{T}^\pi$. ∎

3.4 Type Reconstruction for Dependent Types

The type reconstruction algorithm for the weak $\mathcal{T}^\pi$ system is shown in Figure 3.4. The algorithm computes a principal type for a term by reconstructing a principal derivation in the weak $\mathcal{T}^\pi$ type inference system. Both the unification and matching algorithms from earlier sections are used. The algorithm is a realization of the constructive proof of the principal derivations theorem, Theorem 34, Section 3.3.

The algorithm is defined as a non-deterministic set of rules based on cases of the syntactic form. Every rule (sometimes we say *case*) in the algorithm corresponds to a $\mathcal{T}^\pi$ typing rule (compare with the typing rules of Figure 2.4, Section 2.1). The reconstruction of a type for a term works at each step by attempting to match the outermost syntactic form with a rule. If a match succeeds, then the reconstruction continues by attempting to solve a set of subcomputations involving unification, matching, or further reconstruction. By matching the outermost syntactic form at each step, the algorithm attempts to build a derivation for a term from the final conclusion backwards to the assumptions in the natural deduction tree.

Sometimes multiple rules in the algorithm apply to a syntactic form. Each case will then generate a separate *thread* of computation with different subcomputations. Some rules in the algorithm depend on the successful completion of unification or matching, or special conditions listed in the *where* clause, for example, there is an implicit fail in the *Arrow* case when $x \notin \mathrm{fv}(\tau)$. For each rule of the algorithm, if a sub-computation fails, or if any condition fails to hold, then that case fails.

The algorithm is non-deterministic because we can think of a computation in terms of several concurrent sub-computation threads that are seeking success. If none of the cases succeeds at any point in the computation, that thread of computation fails. If all threads in the computation fail, then the algorithm fails to reconstruct a type for the given term. If the algorithm does succeed, then exactly one thread will successfully complete because principal derivations are unique. The algorithm can be implemented deterministically by using backtracking.

To see how the algorithm works, consider reconstruction of the type of the *taut* function

discussed in Section 2.2. The complete trace of this example is given in Appendix B. Here we will highlight the steps that extend the usual ML type reconstruction algorithm. The *taut* program is defined as

$$taut \equiv \mathbf{R}\ (\lambda f.f)\ (\lambda n'.\lambda p.\lambda f.((p\ f\ true)\ \&\ (p\ (f\ false))))$$

The outermost syntactic form is an **R** term, so the Π*intro-seq* rule applies to this form as does the $\rightarrow$*intro-seq* rule. The $\rightarrow$*intro-seq* typing rule results in a non-dependent type for *taut* and we know from the discussion in the introduction (Section 1.2) that a non-dependent type for *taut* is not possible. Therefore the $\rightarrow$*intro-seq* thread will fail and, in fact, it fails on an occurs check during unification, as the example of Section 1.2 shows.

If we follow the alternative Π*intro-seq* thread of reconstruction we have four subcomputations to perform. (See the Π*intro-seq* rule of the algorithm in Figure 3.4.) The four subcomputations help reconstruct the final derivation step for *taut* using the Π*intro-seq* typing rule of Figure 2.4, Section 2.1.

1. Reconstruct the type for the zero term $\lambda f.f$ in the first premise of the Π*intro-seq* typing rule.

2. Reconstruct the type for the successor term $\lambda n'.\lambda p.\lambda f.((p\ f\ true)\&(p\ (f\ false)))$ in the second premise of the Π*intro-seq* typing rule.

3. Use Unify to get the arrow structure of the second premise of the Π*intro-seq* typing rule.

4. Use Matching to satisfy the *conditions* on the Π*intro-seq* typing rule.

The first two subcomputations succeed with the following types using the rules that correspond to the usual ML type reconstruction rules. To follow the details of these subcomputations, see the complete reconstruction in Appendix B.

$\{\} \vdash \lambda f.f : N \rightarrow N$

$\{\} \vdash \lambda n'.\lambda p.\lambda f.((p\ (f\ true))\&(p\ (f\ false))) : C \rightarrow (K \rightarrow Bool) \rightarrow ((Bool \rightarrow K) \rightarrow Bool)$

Subcomputation 3 unifies the type of the second subcomputation to get the proper arrow form and obtain the k^{th} and $(k+1)^{st}$ types of the recursive scheme.

$$\text{Unify}(\{\langle C \rightarrow (K \rightarrow Bool) \rightarrow ((Bool \rightarrow K) \rightarrow Bool), Nat \rightarrow A \rightarrow B\rangle\})$$

At this stage we have the base type $\sigma_0 = N \to N$, the k^{th} type $\sigma_k = K \to Bool$ and the $(k+1)^{st}$ type $\sigma_{k+1} = (Bool \to K) \to Bool$. Finally, subcomputation 4 uses the matching algorithm to find an n-dependent recursion scheme based on the base, the k^{th} and the $(k+1)^{st}$ types obtained in the first three subcomputations. A completely new index variable n is introduced.

$$\text{Match}(n, \{\langle \sigma_0, \sigma_k, \sigma_{k+1} \rangle\}) = \text{Match}(n, \{\langle N \to N, K \to Bool, (Bool \to K) \to Bool \rangle\})$$

The result of the matching is a substitution

$$\theta = \{N := Bool\}\{Y := Bool\}\{K := \mathbf{R}\ Bool\ (\lambda n'.\lambda K.Bool \to K)\ n)\}$$

which gives a recursion scheme

$$\theta\sigma_k = \theta(K \to Bool) = (\mathbf{R}\ Bool\ (\lambda n'.\lambda K.Bool \to K)\ n) \to Bool$$

upon which the following final product type for taut is based.

$$\{\} \vdash taut : \Pi n.(\mathbf{R}\ Bool\ (\lambda n'.\lambda K.Bool \to K)\ n) \to Bool$$

Conjecture 35 (Reconstruction Theorem) *Let $\Gamma \vdash \delta : \tau$ be a derivation for a term δ in the weak $\mathcal{T}^\pi$ type system.*

1. **Termination** *The algorithm W terminates.*
2. **Soundness and Completeness** *$W(\Gamma, \delta) = (\theta, \sigma)$ if and only if $(\theta\ \Gamma) \vdash \delta : \sigma$ is a principal derivation for δ in the weak $\mathcal{T}^\pi$ type system.*

Proof We do not include a proof of this theorem, but it is a straightforward translation from the constructive proof of the existence of principle derivations, Theorem 34, Section 3.3. Note that we already have the termination, soundness, and completeness proofs for the Unify and Match functions from the Unification and Matching Theorems 22 and 31. ∎

In this chapter we introduced a unification algorithm for $\mathcal{T}^\pi$ types and a matching algorithm for constructing recursive families of $\mathcal{T}^\pi$ types. We used these algorithms to

prove the existence of principal derivations and principal types for $\mathcal{T}^{\pi}$. Finally, we extracted a type reconstruction algorithm for $\mathcal{T}^{\pi}$ from the proof of principal derivations and conjectured its soundness and completeness.

$W(\Gamma, \delta) = (\theta, \tau)$ where

Zero	$W(\Gamma, \mathbf{0})$	=	(θ_{id}, Nat)
Succ	$W(\Gamma, \mathbf{S})$	=	$(\theta_{id}, Nat \to Nat)$
Var	$W(\Gamma, x)$	=	(θ_{id}, τ) if $\{x : \tau\} \in \Gamma$
$\to$ *intro*	$W(\Gamma, \lambda x.\delta)$	=	Let $(\theta, \tau) = W(\Gamma\{x : A\}, \delta)$ and $x \notin \mathrm{fv}(\tau)$ in $(\theta, \theta A \to \tau)$ where A is a new type variable
$\to$ *intro-seq*	$W(\Gamma, \mathbf{R}\ \delta'\ \delta)$	=	Let $(\theta_1, \sigma) = W(\Gamma, \delta)$ and $(\theta_2, \tau) = W(\theta_1\Gamma, \delta')$ and $\theta_3 = \mathrm{Unify}(\{\langle \theta_2\sigma, Nat \to A \to B\rangle\})$ and $\theta_4 = \mathrm{Unify}(\{\langle \theta_3 A, \theta_3 B\rangle\})$ and $\theta_5 = \mathrm{Unify}(\{\langle \theta_4\theta_3\tau, \theta_4\theta_3 A\rangle\})$ in $(\theta_5\theta_4\theta_3\theta_2\theta_1, Nat \to \theta_5\theta_4\theta_3 A)$ where A, B are new type variables
$\to$ *elim*	$W(\Gamma, \delta\ \delta')$	=	Let $(\theta_1, \tau) = W(\Gamma, \delta)$ and $(\theta_2, \sigma) = W(\theta_1\Gamma, \delta')$ and $\theta_3 = \mathrm{Unify}(\{\langle \theta_2\tau, \sigma \to B\rangle\})$ in $(\theta_3\theta_2\theta_1, \theta_3 B)$ where B is a new type variable
Π*intro*	$W(\Gamma, \lambda n.\delta)$	=	Let $(\theta, \tau) = W(\Gamma\{n : Nat\}, \delta)$ and $n \in \mathrm{fv}(\tau)$ in $(\theta, \Pi n.\tau)$
Π*intro-seq*	$W(\Gamma, \mathbf{R}\ \delta'\ \delta)$	=	Let $(\theta_1, \sigma) = W(\Gamma, \delta)$ and $(\theta_2, \tau) = W(\theta_1\Gamma, \delta')$ and $\theta_3 = \mathrm{Unify}(\{\langle \theta_2\sigma, Nat \to A \to B\rangle\})$ and $\theta_4 = \mathrm{Match}(n, \langle \theta_3\tau, \theta_3 A, \theta_3 B\rangle)$ in $(\theta_4\theta_3\theta_2\theta_1, \Pi n.\theta_4\theta_3 A)$ where n, A, B are new
Π*elim*	$W(\Gamma, \delta\ \delta')$	=	Let $(\theta_1, \tau) = W(\Gamma, \delta)$ and $(\theta_2, Nat) = W(\theta_1\Gamma, \delta')$ and $\theta_3 = \mathrm{Unify}(\{\langle \theta_2\tau, \Pi n.B\rangle\})$ and $n \in \mathrm{fv}(\theta_3 B)$ in $(\theta_3\theta_2\theta_1, (\theta_3 B[n := \delta'])\downarrow)$ where B is a new type variable

Figure 3.4: Type Reconstruction for Weak $\mathcal{T}^{\pi}$ Types

Chapter 4

Comparisons and Conclusions

4.1 Summary of Results

We have presented an extension of an ML-style type inference system called $\mathcal{T}^\pi$ supporting first order Martin-Löf style dependent product types for Gödel's theory $\mathcal{T}$ of primitive recursive functionals. The intent of the work is to demonstrate the theoretical feasibility of reconstructing dependent types for functions defined by primitive recursion over well founded types.

By starting with Gödel's theory $\mathcal{T}$ we are able to express primitive recursive families of terms indexed by natural numbers using an **R** combinator. Using a like **R** combinator at the level of types, we define a language specifying how to code primitive recursive families of *types* indexed by natural numbers. Upon these families we specify how to construct the $\mathcal{T}^\pi$ first-order product types dependent on natural numbers. What we call the $\mathcal{T}^\pi$ type inference system is then defined by typing rules for deriving so called *well typed* terms of $\mathcal{T}^\pi$ type including, of course, terms with dependent types. Our study of *type reconstruction* is the problem of finding an algorithm to determine types for untyped terms in a sound and complete way with respect to the inference system.

The $\mathcal{T}^\pi$ typing rules are analogous to the type inference rules of Damas and Milner [DM82] for an ML-style system. The Damas and Milner system is based on earlier work by Milner [Mil78] and originally Hindley [Hin69] showing the existence of principal type schemes for the simply typed lambda calculus. Gödel's theory $\mathcal{T}$ is exactly the simply typed lambda calculus with primitive recursion and natural numbers added, and by extension of Hindley's results it is known that the simple type system for theory $\mathcal{T}$ has

principal types.

The rules for the $\mathcal{T}^\pi$ system include all the rules of the simple type system for $\mathcal{T}$. We claim that the $\mathcal{T}^\pi$ type system extends the simple type system for theory $\mathcal{T}$ with dependent types and give an example of a function that can be typed in $\mathcal{T}^\pi$ but not in the simple type system (nor for that matter, in the ML type system). We conjecture, though do not prove, that $\mathcal{T}^\pi$ could easily be enhanced to include the Damas and Milner style *Let* polymorphism in the same way that Damas and Milner enhanced Hindley's work, and therefore the $\mathcal{T}^\pi$ system would extend ML style polymorphism with dependent types.

The system $\mathcal{T}^\pi$ is closely related to an earlier system $\mathcal{T}^\infty$ studied by Tait and by Martin-Löf in the context of proof theory [Tai65, Mar72a]. Their system has infinite sequences of terms and types but is studied in a mathematical setting where specific codings for the sequences are left unspecified. The papers by Tait and Martin-Löf demonstrate primarily strong normalization of $\mathcal{T}^\infty$ derivations. They indicate that the results hold for recursive formulations of sequences, and thus we conjecture that these results carry over to the $\mathcal{T}^\pi$ system where we have chosen a specific formulation, however, we do not give a formal proof of this. Upon this conjecture of strong normalization of $\mathcal{T}^\pi$ we research the type inference system and the reconstruction algorithm.

To show that the $\mathcal{T}^\pi$ type system is sensible according to standard mathematical intuitions, we give a simple set theoretical model closely following a model by Hindley [Hin69] and to some extent Friedman [Fri75] for the simply typed lambda calculus. In this model we let types denote sets of equivalence classes of terms by convertibility and let typing statements represent set memberships. We prove a soundness and completeness theorem for the model that verifies typing statements as exactly set memberships in the model.

Our model, like Hindley's model, expects the addition of an *Eq* rule for terms. The *Eq* rule makes intra-convertible terms members of the same type. We point out an example by Ohori [Oho89] suggesting there are better models, because the requirement of the *Eq* rule allows types to be assigned to terms that should not be typable. In this sense, the model does not match the intended behavior of the typing system. Harper and Mitchell recently introduced the term *coherence* to discuss this property of the relationship between models

and reconstruction [HM93]. We do not further investigate the problem of coherence.

We claim the $\mathcal{T}^\pi$ type system with its addition of dependent types continues to support the principal types of the simply typed lambda calculus – provided we restrict the recursive types to what we call *weak recursion*. Our definition of weak recursion means that neither the recursive unfolding nor the base case of a recursively constructed type can depend in any way on the recursion index.[1] We prove a principal type theorem for weak $\mathcal{T}^\pi$.

The key part of the proof of the principal types theorem relies on extending the unification algorithm to handle dependent types in $\mathcal{T}^\pi$. To handle dependent types, we add a specialized matching algorithm that matches triples of type schemes to a primitive recursive family of types representable in $\mathcal{T}^\pi$. The triples of type schemes represent the base, n^{th}, and $n+1^{\text{st}}$ cases of a potential sequence of types and the matching algorithm actually finds a $\mathcal{T}^\pi$ primitive recursive representation of the sequence. We use a combination of unification and matching to reconstruct a product type for a recursive sequence term similar to the way unification alone is used in the usual way to reconstruct an arrow type given an applicative term.

Our last result is a type reconstruction algorithm extracted from the proof of the principal derivations theorem. We claim in our reconstruction conjecture that this algorithm is sound and complete with respect to the weak recursively based $\mathcal{T}^\pi$ type system. Our claim is left unproved. However, our unification and matching theorems together with the principal derivations theorem provide the essential components of the reconstruction theorem.

4.2 Relationships with Other Work

The $\mathcal{T}^\pi$ system has a historical lineage beginning in the 1930s with work on theories of functionality by Curry and Church. In 1958 Gödel introduced theory $\mathcal{T}$ to prove the consistency of arithmetic. (See Hindley and Seldin [HS86].) Roger Hindley contributed two results we follow closely, namely, his discovery of principal type schemes [Hin69] and

[1] We have no reason to believe the restriction to weak recursion is necessary, rather, the restriction was chosen to simplify the proof of the principal types theorem.

his construction of a term model and completeness results for the simply typed lambda calculus [Hin83]. Our work applies the methods of Hindley to the new system $\mathcal{T}^{\pi}$ that turns out to be closely related to excursions by Tait and Martin-Löf into the theory $\mathcal{T}^{\infty}$ of lambda calculus with infinite sequences of terms and types [Tai65, Mar72b]. The way we have formulated sequences of types as primitive recursive formulas follows Martin-Löf's early intensional type theory [Mar75].

Our work belongs with the family of research on languages without general recursion. Under this paradigm, all programs terminate and programs are constructed with a variety of powerful systems of well founded recursion. Such languages have the property of *strong normalization.* There is a lively debate about the value of this strong assumption on expressing computations. General recursion often offers simplicity of expression, but proofs of correctness are more difficult and require separate proofs of termination. On the other hand, strong normalization supports much better algebraic properties and supports the paradigm known as *propositions as types*, where programs are strongly correlated their proofs of correctness.

Our model for the $\mathcal{T}^{\pi}$ system is a very simple set-theoretic term model that we construct primarily to assure that our system is sensible. The model is an adaptation of Hindley's model for the simply typed lambda calculus [Hin83]. There is much research in model theoretic constructions for the typed lambda calculi that we have not explored, but could be applicable to our system, for example, see the overview on ML models by Harper and Mitchell [HM93].

In the next few subsections we discuss various ways our research on type systems and type reconstruction relates to other work.

4.2.1 Type Systems and Philosophies

We have encountered three categorizations that help distinguish type systems. Are we typing according to a Church or a Curry philosophy? Do we have explicit or implicit types? What kind of polymorphism do we have in the system and do we have impredicativity or predicativity? There are good discussions about all of these questions in Hindley and Seldin, Harper and Mitchell, and Pierce, et al [HS86, HM93, PDM89]. We use these

characterizations as a way to situate our system among others in the literature.

We assume a Curry philosophy where terms have an existence regardless of whether they are typable in our system or any other. We also have entirely implicit types; programs have absolutely no type information and we reconstruct all types. With a Curry philosophy, the type system becomes a way to categorize terms; either as non-typable, or in some type category. Furthermore, types are meta-level constructions; they never appear in programs. In the alternative Church philosophy, no term would exist unless it were typable and all terms would be assumed to have types associated with them, even if those types were not explicitly given. Thus, for example, Harper and Mitchell [HM93] view the implicit typing of ML as a shorthand for an explicitly typed variation of ML. Harper and Mitchell argue that an explicitly typed ML "scales up" to full ML with type definitions and modules and that it is much simpler to model the systems under the Church philosophy because one does not have to worry about providing meaning to untypable terms. Reconstruction for partially typed terms becomes a relevant issue in a Church philosophy. Hindley and Seldin [HS86], on the other hand, argue the Curry approach for us – that the behavior of terms is fundamental and that typing systems help us categorize behaviors in various ways at the meta-level.

Polymorphism and predicativity or impredicativity can occur many ways in languages and type systems. Our system is based on type schemes, or what Leivant refers to as *quantifier-free universal parametric polymorphism* [Lei91]. This polymorphism is, in short, implicit quantifiers at only the outermost scope of all type expressions. Because type schemes have implicit quantification only at the outermost scope of a type expression, no quantified type appears as a constituent in another type, leaving us with effectively only one level typing, that is, a first order system. Our $\mathcal{T}^{\pi}$ system is related to a very similar family of systems that have level-2 types constructed by quantification over level one types [Lei91, KT92]. Type systems with stratification of types into levels are all essentially predicative. At the other extreme are essentially impredicative type systems, such as the second order lambda calculus [Gir72, Rey74], that allow types with quantification over all types. The $\mathcal{T}^{\pi}$ type system is essentially predicative in another sense. The $\mathcal{T}^{\pi}$ dependent types are based on well founded recursion schemes in contrast to type systems

with recursive types that have no finite basis [MPS86, CC91].

Within these dimensions of strong normalization versus general recursion, Curry versus Church philosophies, explicit versus implicit types, and predicative versus impredicative polymorphic constructions we can distinguish our system from others. Thus, for example, we differ from the Calculus of Constructions (COC) [CH88], Martin-Löf type theory [Mar75], and the *Logical Framework* (LF) [HHP93, CH88] because our types are Curry-style meta-constructions only implicitly associated with programs via type reconstruction. We differ from systems with recursively defined types [MPS86, CC91] because our dependent types are predicative; either finite or well founded recursions. On the other hand, we are similar to other implicitly typed and first order or predicative systems in the ML family [KTU93a, Hen93, Lei91, KT92, McC84].

4.2.2 Polymorphism and Dependent Types

There are many kinds of polymorphism. To simplify our study, we use monomorphic type schemes as in the original work by Hindley [Hin69]. By monomorphic we mean that no types are formed using universal quantifiers over type variables. By type schemes we mean, as Hindley does, that types have variables that can range over any type. Type expressions are implicitly assumed to be universally quantified at the outermost scope, but quantifiers never appear within type expressions. Damas and Milner ML-style polymorphism introduces quantified polymorphism in the *Let* construct to extend Hindley's work with type schemes for the lambda calculus. We believe that our work could be easily extended along these lines to handle such *Let* polymorphism.

Leivant gives an insightful perspective on polymorphism and predicativity in [Lei91]. He portrays polymorphic typing systems as ranging between two paradigms: the quantifier-free parametric polymorphism of ML typing systems with decidable type inference on one hand, and impredicative and explicitly quantified disciplines of the Girard-Reynolds second order lambda calculus on the other hand [Gir72, Rey74]. The former systems have practical advantages with decidable type inference, yet lack the expressive power of full type quantification.[2] The latter systems have great expressive power but exceed the bounds of

[2]The lack of expressive power refers to the limitation of allowing quantifiers only at the outermost level.

practical type reconstruction. Researchers are looking for practical type systems between these extremes. Leivant himself proposes a finite stratification of quantificational type systems to avoid impredicative abstraction and thus introduce a certain theoretical and practical tractability for type systems. In this system of stratification, types are assigned levels of quantification, and quantifiers always range over lesser levels. Our research clearly rests at the practical end of this spectrum of polymorphic systems, for we have the simplist kind of type schemes and concentrate on exploring the addition of dependent types in a first order system.

There are also many kinds of dependent types. Our system introduces product types dependent on terms, specifically terms that normalize to numerals representing natural numbers. Our formulation of dependent types follows Martin-Löf's early intentional type theory, [Mar75]. Like Martin-Löf, our dependent types are well founded recursive expressions built up from other already-defined type expressions. Like all our types, dependent types can have type variables. One type variable is reserved for the recursive unfolding and it is bound with a lambda binder to distinguish it from the other schematic type variables. As far as we know, the combination of dependent types and type schemas and the study of their interaction is new. Usually dependent types are found in systems like LF, COC, or Martin-Löf type theory [CH88, HHP93, Mar75], where polymorphism or dependencies on either terms or types is represented by explicit abstractions.

4.2.3 Type Reconstruction

Much work has been done with enhancing the ML type reconstruction system. The main threads of this work cover type reconstruction with subtypes [Wan91, Sta88, Oho89, FM90, Rée89], type reconstruction with recursive types [CC91], type reconstruction with polymorphic recursion [KTU93a, Hen93], and type reconstruction in stratified polymorphic systems [Lei91, KT92, McC84]. There has been some work in dependent type inference [Ell89] that we discuss shortly. There has also been recent work in type reconstruction for partially typed programs, where some explicit type information is present and the rest is implicit, [Tiu90, HM93]. There are many and various other special case results associated with proposals for new types, where ML type reconstruction is extended to handle those

types.

Conal Elliott [Ell89] shows how higher order unification (HOU) introduced by Huet [Hue75] can be modifed to work with the first-order dependent-product types of the Edinburgh *Logical Framework* (LF) [HHP93]. He goes on to show how HOU can be used to do type reconstruction for object logics encoded in LF. Elliott's technique involves what he calls *term inference* in LF. Because an object logic is encoded in LF, its type system will be representable using LF terms, thus an appropriate LF term-inference algorithm could reconstruct LF terms representing object logic types.

Elliott's term inference correlates with our methods of matching recursion schemes during the reconstruction of types for recursive sequences. Elliott adapted Huet's second order unification algorithm whereas we developed our matching algorithm by adapting a special case of Huet and Lang's second-order matching algorithm [HL78]. Some techniques of Elliott's algorithm might be applicable to our matching problem. We differ from Elliot's work primarily in our formulation of dependent-product types using well founded recursion based on our specific object language. Elliott's work in LF, on the other hand, leaves the object language dependencies unspecified. Nevertheless, Elliott's paper suggests it would be possible to *encode* our $\mathcal{T}^{\pi}$ system using LF and then derive our type inference system from the term-inference method of his paper. Such an exercise might set our work in a more general framework and help with generalizing our results to more complex recursion schemes. On the other hand, a complete LF formalization is exceedingly tedious.

Cardone and Coppo [CC91] present a principal type theorem and a model for a simple type system extended with recursive types. Their paper focuses mainly on model theory for the system. However, they make it clear that principal types exist and it is possible to reconstruct them. In their type system one can formulate recursive types $\mu A.\sigma$ where $A \in \text{fvty}(\sigma)$. These types are intended to be interpreted as infinite unfoldings, for example, the type $\mu A.Bool \to A$ would be the type $Bool \to Bool \to Bool \to \cdots$. Cardone and Coppo present rules for type assignment and a principal types theorem, as well as a model with completeness results, very similar to the style of our presentation.

Both our system and Cardone and Coppo's system have types defined by recursive

forms. The fundamental distinction lies in the interpretation of these forms. In Cardone and Coppo's system, the recursive formulas are impredicative and denote a single infinitely unfolding type. In our case, recursive formulas are well founded (predicative) and define infinite families of *finite types* over which we build a products. Thus, for example, our system would interpret the above recursive formula $\mu A.\sigma$ as the family of types $\{Bool, Bool \rightarrow Bool, Bool \rightarrow Bool \rightarrow Bool, \ldots\}$.

The problem of type reconstruction with polymorphic recursion comes from an anomaly in ML-style *Let* polymorphism and its interaction with mutual recursion. Kfoury, Tiuryn, and Urzyczyn [KTU93a] and Henglein [Hen93] both discuss this practical problem and prove the undecidability of reconstruction for various solutions to the problem. The interesting aspect of both of these papers is the demonstration of equivalence between the the reconstruction problem and the more general problem of semi-unification. Their undecidability result for reconstruction is based on the undecidability of semi-unification, shown recently in another paper by Kfoury, Tiuryn, and Urzyczyn [KTU93b]. Note that these reconstruction problems assume we have general recursion, unlike our system.

Based on stratified polymorphism discussed above [Lei91], Leivant in earlier work [Lei83] and then McCracken [McC84] present papers on type reconstruction for polymorphic systems of two levels. More recently Kfoury and Tiuryn [KT92] complete this work by proving that typability in the second order lambda calculus restricted to two levels of polymorphism is decidable. Furthermore, Kfoury and Tiuryn prove in the same paper that the reconstruction problem for the second order lambda calculus at levels of polymorphism greater than two is undecidable. Leivant, McCracken, and Kfoury and Tiuryn are working with explicitly quantified polymorphic types, rather than implicit ML-style parametric polymorphism at the outer level, as we do with type schemes.

The work on type reconstruction and subtypes covers a range of research that investigates the extension of ML with properties of object-oriented and database programming [Wan91, Sta88, Oho89, FM90, Ré89]. All of these works propose new extensions to ML representing general language facilities and then investigate type reconstruction. There are also many other miscellaneous extensions to ML for which type reconstruction has been extended.

4.2.4 Unification and Matching

The heart of the thesis lies in the unification and matching theorems for the recursive sequences of types comprising the dependent products. In developing the matching algorithm we worked with ad-hoc methods, variations of second-order unification [Hue75], and variations of second-order matching [HL78] combined with first-order unification. We believe there may be a more general framework for our unification and matching that could greatly simplify the presentation and clarify the results – in much the same way that semi-unification turned out to be a general framework for viewing ML-style unification and the enhancements for polymorphic recursion [KTU93a]. In our search for an algorithm we are walking the boundary of decidability. Huet's second-order unification is only semi-decidable [Hue75, Gol81]. Semi-unification is also undecidable in general [KTU93b]. Yet unification and second-order matching [HL78] are decidable. We hope to clarify the general class of matching and unification problems upon which our reconstruction depends.

4.3 Reflections, Criticisms, and Future Work

This thesis establishes a new type system about which little is known, so there is much research that remains. Most immediately relevant is to determine how useful the language and type system are. We have postulated the value of dependent types and proposed a way to express them, but we have not experimented with their use. Are the restrictions we place on the language and types too strong? Can we write intersting programs with dependent types that we cannot write in ML? Can we exclude ill-typed programs that the ML type checker does not catch? We must try to program more with our language and type system to illustrate the capabilities.

We also must closely examine the restrictions on the language and type system and discover to what extent we have excluded interesting programs. The restrictions with the most impact appear to be the requirement for strong normalization of terms and types, plus the restriction on type dependencies to closed type indices that reduce to numerals.

There is an on-going debate about the restriction to terminating programs and we are not concerned about being amidst research exploring programming without general

recursion. We must, however, prove our strong normalization conjecture before continuing further. Many of the results depend on this fact.

It would be prudent to take a close look at our requirements on indices to dependent types. What is the effect of restricting the application of terms with product type to closed terms that normalize to numerals? Does this restriction too severely constrain the kind of functions or function applications that we can type? We do not fully understand the implications of this restriction.

We also want to look closely at the implications of limiting ourselves to weak recursion. Our definition of weak recursion means that no dependency index can nest within a recursive type dependent on another index. What is the practical result of this limitation on the nesting of dependencies? Is there some simple way and intuitive way to characterize this limitation? Is the limitation serious? Some thought should also be put into these limitations before pursuing further research.

If we successfully address these immediate questions, then there is a long list of elaborations we wish to pursue. We want to find a simpler framework for presenting the system. Perhaps a better syntax; a way to cope with the technical clutter of working with well founded recursion. Predicative systems that incorporate realizability within the language always seem to have more special cases to handle.

One approach to the clarifying the presentation would be to look for uniformity by examining the $\mathcal{T}^{\pi}$ system within a categorical framework. Some work has been done in this area for dependent types [Car86]. Spencer [Spe91] discusses the advantages of the categorical approach for strongly normalizing languages.

Another approach to simplifying the presentation would be to look at basing the system on *iteration* rather than *weak recursion.* Girard [GLT89] shows that the combinator we use can be simulated using an iterator combinator plus pairing. Can we gain some uniformity that would simplify the system?

We have presented in this thesis a type system dependent only on numerals. A natural extension to this work is the extension to recursion over richer types, such as lists and trees. This leads us to consider type inference over the full range of well founded recursion schemes, such as any of those within the first-level universe of Martin-Löf's type theory

[Mar75, NPS90]. What power of expression do we gain from such a system? Further, what about extending the results to multiple levels of universes? A related question concerns the possibilities and implications of embedding this system in one with full recursion like ML. Is there a sensible way to do this?

The simple term model we use suffices for our purposes. However, like other term models, it is not very informative – it does little more than categorize the syntactic behavior of terms. We would like to enhance the semantic treatment with more informative models. Harper and Mitchell [HM93] suggest several possible methods of modeling, including set-theoretic as well as domain-theoretic models. They introduce the concept of *coherence* also; we should investigate more closely the *coherence* of our model with respect to our reconstruction method. It appears that by assuming a *decent* semantics like Hindley, we gain completeness at the expense of coherence.

Our models do not need to model non-terminating terms, so we might be able to simplify the model by adopting the Church philosophy and viewing our implicit typing as a shorthand for a particular explicitly typed system, such as promoted by Harper and Mitchell, [HM93]. Our experience with the problems of the *Eq* rule in semantically equating any two convertible terms might be resolved by this approach. On the other hand, it is difficult to give up the Curry philosophy that allows us to believe that there are meaningful terms out there and our type system just does not quite organize them right.

Our matching algorithm turns out to be a somewhat ad-hoc adaption of Huet's second order matching algorithm [HL78]. We believe there must be more general characterizations of our matching problem that would provide a more uniform framework in which to express our algorithm. We also want to explore matching more powerful recursion schemes such as those proposed above. What are the decidability results? Is there a general characterization of the problem?

It is important to complete the work here by proving the reconstruction conjecture of soundness and completeness of the algorithm with respect to our weak recursively based type system. Then it would be fun to write an experimental program implementing the reconstruction algorithm.

Finally, from a practical viewpoint we want to look at the potential applications. For

example, would our dependent typing make a useful abstraction for building libraries of size-dependent array operations? Should research results appear promising, we would build a test system and experiment with the language.

Bibliography

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[Car86] John Cartmell. Generalized algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

[CC91] Felice Cardone and Mario Coppo. Type inference with recursive types: Syntax and semantics second order lambda calculus. *Information and Computation*, 92:48–80, 1991.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM, January 1982.

[Ell89] Conal M. Elliott. Higher-order unification with dependent function types. In N. Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 121–136, Berlin, Apr 1989. Springer-Verlag.

[FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73:155–175, 1990.

[Fri75] Harvey Friedman. Equality between functionals. *Lecture Notes in Mathematics*, 453:22–37, 1975.

[Gir72] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* Thèse de doctorat d'etat, University of Paris VII, 1972.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge University Press, New York, 1989.

[Gol81] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15:253–289, 1993.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40:143–184, 1993.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[Hin83] R. Hindley. The completeness theorem for typing lambda terms. *Theoretical Computer Science*, 22:1–17, 1983.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HM93] Robert Harper and John C. Mitchell. On the type structure of standard ML. *ACM Transactions on Programming Languages and Systems*, 15:211–252, 1993.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, Cambridge, 1986.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hue80] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27:797–821, 1980.

[KT92] A. J. Kfoury and J. Tiuryn. Type reconstruction in finite rank fragments of the second order lambda calculus. *Information and Computation*, 98:228–257, 1992.

[KTU93a] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type recursion in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15:290–311, 1993.

[KTU93b] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102:83–101, 1993.

[Lei83] Daniel Leivant. Polymorphic type inference. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 88–98. ACM, January 1983.

[Lei91] Daniel Leivant. Finitely stratified polymorphism. *Information and Computation*, 93:93–113, 1991.

[Mar72a] Per Martin-Löf. Infinite terms and a system of natural deduction. In *Compositio Mathematica*, volume 24, pages 93–103. Wolters-Noordhoof, 1972.

[Mar72b] Per Martin-Löf. An intuitionistic theory of types. An unpublished mimeographed manuscript [Mar75] containing a presentation of Girard's paradox that does not appear in the final version, 1972.

[Mar75] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118. North Holland, 1975.

[McC84] N. McCracken. The typechecking of programs with implicit type structure. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France*, volume 173 of *Lecture Notes in Computer Science*, pages 301–315, Berlin, June 1984. Springer-Verlag.

[Mey82] Albert R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52:87–122, 1982.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MPS86] David B. MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[NPS90] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory.* Oxford University Press, New York, NY, 1990.

[Oho89] Atsushi Ohori. A simple semantics for ML polymorphism. In *Functional Programming and Computer Architecture*, pages 281–292. ACM, 1989.

[Pau91] Laurence Paulson. *ML for the working programmer.* Cambridge University Press, New York, 1991.

[PDM89] Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, School of Computer Science, Carnegie Mellon University, March 1989.

[Ré89] D. Rémy. Type checking records and variants in a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 77–88. ACM, January 1989.

[Rey74] John C. Reynolds. Towards a theory of types structure. In *Proceedings Colloque sur la Programmation*, pages 408–423. Springer-Verlag, New York, 1974.

[SG89] Wayne Snyder and Jean Gallier. Higher order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8:101–140, 1989.

[Spe91] Dwight Spencer. A survey of categorical computation: fixed points, partiality, combinators, ... control? *Bulletin of the European Association of Theoretical Computer Science*, 43:285–312, 1991.

[Sta88] R. Stansifer. Type inference with subtypes. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 88–97. ACM, January 1988.

[Tai65] W. W. Tait. Infinitely long terms of transfinite type. In Crossley and Dummett, editors, *Formal Systems and Recursive Functions*, pages 177–185. North Holland, 1965.

[Tiu90] J. Tiuryn. Type inference problems: A survey. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *Lecture Notes in Computer Science*, pages 105–120, Berlin/New York, 1990. Springer-Verlag.

[Tur76] David A. Turner. SASL language manual. Technical report, University of St. Andrews, 1976.

[Wan91] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93:1–15, 1991.

Appendix A

Dependent Typing Examples

We show how to type tupler and projection operations in the $\mathcal{T}^\pi$ type system consistently dependent on the size of the tuples. For these examples we will be using the letters A, B, C, D, E to stand for simple types. The examples use dependent types of a special left linear form like the type $\Pi n.(A \rightarrow)^n\ B$ introduced for the *taut* example (where in *taut* $A = B =$ *Bool*). We use the symbol $\equiv$ to mean *syntactically equal*, that is, an abbreviation. Abbreviations make the derivation significantly more readable. The usual equality symbol = identifies semantic values. In some cases we stretch the notation and say $((\Pi n.\tau)\ \delta)\downarrow$ which, strictly speaking, should be $(\tau[n := \delta])\downarrow$. We do this to accommodate abbreviations for Π types, for example, in the type $(Swapmn\ n\ n)\downarrow$ that appears below we do not want to expand the abbreviation of *Swapmn*. Also we regularly simplify the form $((\Pi n.\tau)\ n)\downarrow$, where the reduction is simply a variable change, by leaving off the $\downarrow$ symbol for reduction. For example, we often write $(Swapmn\ n)$ or $(Projector\ n)$ without the necessary reduction explicitly indicated.

A.1 Tuples

The representation of tuples is similar to the usual lambda calculus formulations. Tuples are constructed using a combinator *tupler* having a tuple length as first argument, tuple elements as middle arguments and the last argument a *Projector*. A *concrete tuple*, is *tupler* applied to a length and all of its elements up to that length. We have no less than one tuples for compatibility with projections in Appendix A.2. Informally, the family of tupler terms is given by the following table.

The *tupler* combinator is composed of a *pretupler* operation, which expects a projector as the second argument rather than last argument, and a *swap* operation to reorganize the arguments so the projector is correctly the last argument to the tupler. This two stage formulation of the tupler makes dependent typing possible according to the $\mathcal{T}^\pi$ rules. We do not know if the intermediate swapping construction that complicates this example is

$$
\begin{array}{lcll}
\mathit{tupler}\ 0 & = & \lambda a_0.\lambda z.z\ a_0 & 1\text{-}\mathit{tupler} \\
\mathit{tupler}\ 1 & = & \lambda a_0.\lambda a_1.\lambda z.z\ a_0\ a_1 & 2\text{-}\mathit{tupler} \\
 & \vdots & & \\
\mathit{tupler}\ k & = & \lambda a_0 \ldots \lambda a_k.\lambda z.z\ a_0 \ldots a_k & (k+1)\text{-}\mathit{tupler}
\end{array}
$$

Table A.1: A Family of Tupler Terms

necessary. Informally, the pretupler terms are:

$$
\begin{array}{lcl}
\mathit{pretupler}\ 0 & = & \lambda z.\lambda a_0.z\ a_0 \\
\mathit{pretupler}\ 1 & = & \lambda z.\lambda a_0.\lambda a_1.z\ a_0\ a_1 \\
 & \vdots & \\
\mathit{pretupler}\ k & = & \lambda z.\lambda a_0 \ldots \lambda a_k.z\ a_0 \ldots a_k
\end{array}
$$

The swapper terms look like

$$
\begin{array}{lcl}
\mathit{swap}\ 0 & = & \lambda f.\lambda a_0.\lambda z.f\ z\ a_0 \\
\mathit{swap}\ 1 & = & \lambda f.\lambda a_0.\lambda a_1.\lambda z.f\ z\ a_0\ a_1 \\
 & \vdots & \\
\mathit{swap}\ k & = & \lambda f \lambda a_0 \ldots \lambda a_k.\lambda z.f\ z\ a_0 \ldots a_k
\end{array}
$$

To type the swap operation, it seems necessary to formulate a more general *swapmn* to keep track of both recursion indices—the arity of f and the number of swaps. Here is the complete formulation of tupler terms.

$$
\begin{array}{lcl}
\mathit{pretupler} & \equiv & \mathbf{R}\ (\lambda z.\lambda a.z\ a)\ (\lambda k.\lambda p.\lambda z.\lambda a.p\ (z\ a)) \\
\mathit{swapmn} & \equiv & \lambda m.\mathbf{R}\ (\lambda f.\lambda a.\lambda z.f\ z\ a)\ (\lambda k.\lambda p.\lambda f.\lambda a.p\ (\lambda z.f\ z\ a)\ ) \\
\mathit{swap} & \equiv & \lambda n.\mathit{swapmn}\ n\ n \\
\mathit{tupler} & \equiv & \lambda n.\mathit{swap}\ n\ (\mathit{pretupler}\ n)
\end{array}
$$

By expansion of the above definitions and the equality theory of $\mathcal{T}^\pi$ types, the following

equations hold.

$$\begin{array}{lcll}
\mathit{pretupler}\ \mathbf{0} & = & \lambda z.\lambda a.z\ a & (= \mathit{Zero}) \\
\mathit{pretupler}\ (\mathbf{S}\ k) & = & \lambda z.\lambda a.\mathit{pretupler}\ k\ (z\ a) & (= \mathit{Succ}) \\
 & & & \\
\mathit{swapmn}\ m\ \mathbf{0} & = & \lambda f.\lambda a.\lambda z.f\ z\ a & (= \beta, = \mathit{Zero}) \\
\mathit{swapmn}\ m\ (\mathbf{S}\ k) & = & \lambda f.\lambda a.\mathit{swapmn}\ m\ k\ (\lambda z.f\ z\ a) & (= \beta, = \mathit{Succ}) \\
 & & & \\
\mathit{swap}\ \mathbf{0} & = & \lambda f.\lambda a.\lambda z.f\ z\ a & (= \beta, = \mathit{Zero}) \\
\mathit{swap}\ (\mathbf{S}\ k) & = & \lambda f.\lambda a.\mathit{swap}\ k\ (\lambda z.f\ z\ a) & (= \beta, = \mathit{Succ})
\end{array}$$

The following type definitions are useful for typing the tupler and other terms above. Note the abbreviations we introduce for the types *Projector* and *Swapmn*.

$$\begin{array}{lcl}
\mathit{Projector} & \equiv & \mathbf{R}\ (A \to A)\ (\lambda k.\lambda X.A \to X) \\
 & \equiv & \Pi n.(A \to)^n (A \to A) \\
 & & \\
\mathit{Pretupler} & \equiv & \Pi n.(\mathit{Projector}\ n \to \mathit{Projector}\ n) \\
 & & \\
\mathit{Tuple} & \equiv & \Pi n.(\mathit{Projector}\ n \to A) \\
 & & \\
\mathit{Swapmn} & \equiv & \Pi m.\Pi n.(\ (\mathit{Projector}\ m) \to (\mathit{Projector}\ n)\ ) \\
 & & \to (\mathbf{R}\ (A \to \mathit{Tuple}\ m)\ (\lambda k.\lambda t.A \to t)\ n) \\
 & \equiv & \Pi m.\Pi n.(\ (\mathit{Projector}\ m) \to (\mathit{Projector}\ n)\ ) \\
 & & \to (\ (A \to)^n (A \to \mathit{Tuple}\ m))
\end{array}$$

From the definition of *Projector* the equality theory of the type system of $\mathcal{T}^\pi$ the following equations hold.

$$\begin{array}{lcll}
\mathit{Projector}\ \mathbf{0} & = & A \to A & (= \Pi \mathit{Zero}) \\
\mathit{Projector}\ (\mathbf{S}\ k) & = & A \to (\mathit{Projector}\ k) & (= \Pi \mathit{Succ})
\end{array}$$

We will derive the following typings using the non-weak $\mathcal{T}^\pi$ typing rules. Note the abbreviation we introduce for the type of *tupler*.

$$\begin{array}{lcl}
\mathit{pretupler} & : & \mathit{Pretupler} \\
 & & \\
\mathit{swapmn} & : & \mathit{Swapmn} \\
 & & \\
\mathit{swap} & : & \Pi n.(\mathit{Swapmn}\ n\ n)\downarrow \\
 & & \\
\mathit{tupler} & : & \Pi n.\mathbf{R}\ (A \to \mathit{Tuple}\ n)\ (\lambda k.\lambda X.A \to X)\ n \\
 & : & \Pi n.(A \to)^n (A \to \mathit{Tuple}\ n)
\end{array}$$

First assume typings for *swap* and *pretupler* and derive the type of *tupler*.

Theorem

$$tupler \;:\; \Pi n.(A \to)^n(A \to Tuple\ n)$$

Assumptions

$$A_1.\ \vdash swap \;:\; \Pi n.(Swapmn\ n\ n)\downarrow$$
$$A_2.\ \vdash pretupler \;:\; \Pi n.(Projector\ n) \to (Projector\ n)$$

Proof

1.	$\{n : Nat\} \vdash n$	$: Nat$	(Var)
2.	$\{n : Nat\} \vdash swap$	$: \Pi n.(Swapmn\ n\ n)\downarrow$	$(Cext\ A_1)$
3.	$\{n : Nat\} \vdash swap\ n$	$: (Swapmn\ n\ n)\downarrow$	$(\Pi elim\ 2\ 1)$
		$: (\ (Projector\ n) \to (Projector\ n)\ )$	
		$\to (\ (A \to)^n(A \to Tuple\ n)\ )$	$(\equiv Swapmn)$
4.	$\{n : Nat\} \vdash pretupler$	$: \Pi n.(Projector\ n) \to (Projector\ n)$	$(Cext\ A_2)$
5.	$\{n : Nat\} \vdash pretupler\ n$	$: (Projector\ n) \to (Projector\ n)$	$(\Pi elim\ 4\ 1)$
6.	$\{n : Nat\}$		
	$\vdash swap\ n\ (pretupler\ n)$	$: (A \to)^n(A \to Tuple\ n)$	$(\to elim\ 3\ 5)$
7.	$\vdash \lambda n.swap\ n\ (pretupler\ n)$	$: \Pi n.(A \to)^n(A \to Tuple\ n)$	$(\Pi intro)$
	$\vdash tupler$	$: \Pi n.(A \to)^n(A \to Tuple\ n)$	$(\equiv tupler)$

∎

Derive a typing for *swap* assuming a typing for *swapmn*.

Theorem

$$swap \;:\; \Pi n.(Swapmn\ n\ n)\downarrow$$

Assumptions

$$A_1.\ \vdash swapmn \;:\; Swapmn$$

Proof

1.	$\{n : Nat\} \vdash n$	$: Nat$	(Var)
2.	$\{n : Nat\} \vdash swapmn$	$: Swapmn$	$(Cext\ A_1)$
3.	$\{n : Nat\} \vdash swapmn\ n$	$: (Swapmn\ n)\downarrow$	$(\Pi elim\ 2\ 1)$
4.	$\{n : Nat\} \vdash swapmn\ n\ n$	$: (Swapmn\ n\ n)\downarrow$	$(\Pi elim\ 3\ 1)$
5.	$\vdash \lambda n.swapmn\ n\ n$	$: \Pi n.(Swapmn\ n\ n)\downarrow$	$(\Pi intro)$
	$\vdash swap$	$: \Pi n.(Swapmn\ n\ n)\downarrow$	$(\equiv swap)$

Now type *swapmn*.

Theorem

$$swapmn \;:\; Swapmn$$

Steps 7a and 7b below, together with steps 16b and 16c, represent the proof of the reduction conditions on the Π*intro-seq* rule used in step 18.

Proof

Zero case.

1. $\{m : Nat\}$
   $\{f : (Projector\ m) \to (A \to A)\}$
   $\{z : Projector\ m\}\{a : A\} \vdash f \quad : \quad (Projector\ m) \to (A \to A)$ (*Var*)
2. $\{m : Nat\}$
   $\{f : (Projector\ m) \to (A \to A)\}$
   $\{z : Projector\ m\}\{a : A\} \vdash z \quad : \quad (Projector\ m)$ (*Var*)
3. $\{m : Nat\}$
   $\{f : (Projector\ m) \to (A \to A)\}$
   $\{z : Projector\ m\}\{a : A\} \vdash f\ z \quad : \quad A \to A$ ($\to$ *elim* 1 2)
4. $\{m : Nat\}$
   $\{f : (Projector\ m) \to (A \to A)\}$
   $\{z : Projector\ m\}\{a : A\} \vdash a \quad : \quad A$ (*Var*)
5. $\{m : Nat\}$
   $\{f : (Projector\ m) \to (A \to A)\}$
   $\{z : Projector\ m\}\{a : A\} \vdash f\ z\ a \quad : \quad A$ ($\to$ *elim* 3 4)
6. $\{m : Nat\}$
   $\{f : (Projector\ m) \to (A \to A)\}$
   $\vdash \lambda a.\lambda z.f\ z\ a \quad : \quad A \to (Projector\ m \to A)$ ($\to$ *intro*, $\to$ *intro*)

7a. $\{m : Nat\}$
$\vdash \lambda f.\lambda z.\lambda a.f\ z\ a \quad : \quad ((Projector\ m) \to (A \to A))$
$\to (A \to (Projector\ m \to A))$ ($\to$ *intro*)

7b. $: \quad (Projector\ m \to (Projector\ \mathbf{0}))\downarrow$
$\to (\ (A \to)^{\mathbf{0}}(A$
$\to (Projector\ m \to A))\ )\downarrow$ ($= \Pi Zero, = \Pi Zero$)
$: \quad (Projector\ m \to (Projector\ \mathbf{0}))\downarrow$
$\to (\ (A \to)^{\mathbf{0}}(A \to Tuple\ m)\ )\downarrow$ ($\equiv$ *Tuple*)
$: \quad (Swapmn\ m\ \mathbf{0})\downarrow$ ($\equiv$ *Swapmn*)

Successor case.

8. $\{m : Nat\}\{n : Nat\}\{k : Nat\}$
   $\{p : (Swapmn\ m\ n)\downarrow\}$
   $\{f : (Projector\ m) \to (A \to Projector\ n)\}$
   $\{a : A\}\{z : Projector\ m\}$
   $\vdash f \quad : \quad (Projector\ m)$
   $\to (A \to Projector\ n) \qquad (Var)$

9. $\{m : Nat\}\{n : Nat\}\{k : Nat\}$
   $\{p : (Swapmn\ m\ n)\downarrow\}$
   $\{f : (Projector\ m) \to (A \to Projector\ n)\}$
   $\{a : A\}\{z : Projector\ m\}$
   $\vdash z \quad : \quad (Projector\ m) \qquad (Var)$

10. $\{m : Nat\}\{n : Nat\}\{k : Nat\}$
    $\{p : (Swapmn\ m\ n)\downarrow\}$
    $\{f : (Projector\ m) \to (A \to Projector\ n)\}$
    $\{a : A\}\{z : Projector\ m\}$
    $\vdash f\ z \quad : \quad A \to Projector\ n \qquad (\to elim\ 8\ 9)$

11. $\{m : Nat\}\{n : Nat\}\{k : Nat\}$
    $\{p : (Swapmn\ m\ n)\downarrow\}$
    $\{f : (Projector\ m) \to (A \to Projector\ n)\}$
    $\{a : A\}\{z : Projector\ m\}$
    $\vdash a \quad : \quad A \qquad (Var)$

12. $\{m : Nat\}\{n : Nat\}\{k : Nat\}$
    $\{p : (Swapmn\ m\ n)\downarrow\}$
    $\{f : (Projector\ m) \to (A \to Projector\ n)\}$
    $\{a : A\}\{z : Projector\ m\}$
    $\vdash (f\ z)\ a \quad : \quad Projector\ n \qquad (\to elim\ 10\ 11)$

13. $\{m : Nat\}\{n : Nat\}\{k : Nat\}$
    $\{p : (Swapmn\ m\ n)\downarrow\}$
    $\{f : (Projector\ m) \to (A \to Projector\ n)\}$
    $\{a : A\}$
    $\vdash \lambda z.(f\ z)\ a \quad : \quad (Projector\ m)$
    $\to (Projector\ n) \qquad (\to intro)$

14. $\{m : Nat\}\{n : Nat\}$
$\{k : Nat\}$
$\{p : (Swapmn\ m\ n)\downarrow\}$
$\{f : (Projector\ m)$
$\rightarrow (A \rightarrow Projector\ n)\}$
$\{a : A\} \vdash p$: $(Swapmn\ m\ n)\downarrow$ (*Var*)
: $((Projector\ m) \rightarrow (Projector\ n))$
$\rightarrow ((A \rightarrow)^n (A \rightarrow Tuple\ m))$ ($\equiv Swapmn$)

15. $\{m : Nat\}\{n : Nat\}$
$\{k : Nat\}$
$\{p : (Swapmn\ m\ n)\downarrow\}$
$\{f : (Projector\ m)$
$\rightarrow (A \rightarrow Projector\ n)\}$
$\{a : A\} \vdash p\ (\lambda z.(f\ z)\ a)$: $(A \rightarrow)^n (A \rightarrow Tuple\ m)$ ($\rightarrow$ *elim* 14 13)

16*a*. $\{m : Nat\}\{n : Nat\}$
$\{k : Nat\}$
$\{p : (Swapmn\ m\ n)\downarrow\}$
$\vdash \lambda f.\lambda a.p\ (\lambda z.(f\ z)\ a)$: $((Projector\ m) \rightarrow$
$(A \rightarrow Projector\ n))$
$\rightarrow A \rightarrow (A \rightarrow)^n (A \rightarrow Tuple\ m)$ ($\rightarrow$ *intro*, $\rightarrow$ *intro*)

16*b*. : $((Projector\ m)$
$\rightarrow (Projector\ (\mathbf{S}\ n))\downarrow)$
$\rightarrow (A \rightarrow)^{\mathbf{S}\ n}\downarrow (A \rightarrow Tuple\ m)$ ($= \Pi Succ, = \Pi Succ$)

16*c*. : $(Swapmn\ m\ (\mathbf{S}\ n))\downarrow$ ($\equiv Swapmn$)

17. $\{m : Nat\}\{n : Nat\}$
$\vdash \lambda k.\lambda p.\lambda f.\lambda a.p\ (\lambda z.(f\ z)\ a)$
: $Nat \rightarrow (Swapmn\ m\ n)\downarrow)$
$\rightarrow (Swapmn\ m\ (\mathbf{S}\ n))\downarrow$ ($\rightarrow$ *intro*, $\rightarrow$ *intro*)

Recursive sequence introduction.

18. $\{m : Nat\}$
$\vdash \mathbf{R}\ (\lambda f.\lambda z.\lambda a.f\ z\ a)(\lambda k.\lambda p.\lambda f.\lambda a.p\ (\lambda z.(f\ z)\ a))$
$: \Pi n.(Swapmn\ m\ n)\downarrow$ ($\Pi intro\text{-}seq\ 7b\ 17$)
$: \Pi n.(\ (Projector\ m) \rightarrow (Projector\ n)\ )$
$\rightarrow (\ (A \rightarrow)^n(A \rightarrow Tuple\ m)\ )$ ($\equiv Swapmn$)

19. $\vdash \lambda m.\mathbf{R}\ (\lambda f.\lambda z.\lambda a.f\ z\ a)(\lambda k.\lambda p.\lambda f.\lambda a.p\ (\lambda z.(f\ z)\ a))$
$: \Pi m.\Pi n.(\ (Projector\ m)$
$\rightarrow (Projector\ n)\ )$
$\rightarrow (\ (A \rightarrow)^n(A \rightarrow Tuple\ m)\ )$ ($\Pi intro$)
$: Swapmn$ ($\equiv Swapmn$)
$\vdash swapmn : Swapmn$ ($\equiv swapmn$)

∎

Finally type *pretupler*. Steps 5b and 12b represent the proof of the reduction conditions on the $\Pi intro\text{-}seq$ rule in step 14.

Theorem

$$pretupler\ :\ \Pi n.(Projector\ n) \rightarrow (Projector\ n)$$

Proof

Zero case.

1.	$\{z : A \rightarrow A\}\{a : A\} \vdash z$	:	$A \rightarrow A$	(Var)
2.	$\{z : A \rightarrow A\}\{a : A\} \vdash a$	:	A	(Var)
3.	$\{z : A \rightarrow A\}\{a : A\} \vdash z\ a$	:	A	($\rightarrow elim\ 1\ 2$)
4.	$\{z : A \rightarrow A\} \vdash \lambda a.z\ a$	:	$A \rightarrow A$	($\rightarrow intro$)
5a.	$\vdash \lambda a.\lambda z.z\ a$	:	$(A \rightarrow A) \rightarrow (A \rightarrow A)$	($\rightarrow intro$)
5b.		:	$(Projector\ \mathbf{0})\downarrow$	
			$\rightarrow (Projector\ \mathbf{0})\downarrow$	($= \Pi Zero, = \Pi Zero$)
		:	$(Pretupler\ \mathbf{0})\downarrow$	($\equiv Pretupler$)

Successor case.

6. $\{n : Nat\}\{k : Nat\}$
   $\{p : Pretupler\ n\}$
   $\{z : A \to (Projector\ n)\}\{a : A\}$
   $\vdash p$: $Pretupler\ n$ (*Var*)
   : $(Projector\ n)$
   $\to (Projector\ n)$ ($\equiv$ *Pretupler*)

7. $\{n : Nat\}\{k : Nat\}$
   $\{p : Pretupler\ n\}$
   $\{z : A \to (Projector\ n)\}\{a : A\}$
   $\vdash z$: $A \to (Projector\ n)$ (*Var*)

8. $\{n : Nat\}\{k : Nat\}$
   $\{p : Pretupler\ n\}$
   $\{z : A \to (Projector\ n)\}\{a : A\}$
   $\vdash a$: A (*Var*)

9. $\{n : Nat\}\{k : Nat\}$
   $\{p : Pretupler\ n\}$
   $\{z : A \to (Projector\ n)\}\{a : A\}$
   $\vdash z\ a$: $Projector\ n$ ($\to$ *elim* 7 8)

10. $\{n : Nat\}\{k : Nat\}$
    $\{p : Pretupler\ n\}$
    $\{z : A \to (Projector\ n)\}\{a : A\}$
    $\vdash p\ (z\ a)$: $Projector\ n$ ($\to$ *elim* 6 9)

11. $\{n : Nat\}\{k : Nat\}$
    $\{p : Pretupler\ n\}$
    $\{z : A \to (Projector\ n)\}$
    $\vdash \lambda a.p\ (z\ a)$: $A \to (Projector\ n)$ ($\to$ *intro*)

12a. $\{n : Nat\}\{k : Nat\}$
$\{p : Pretupler\ n\}$
$\vdash \lambda z.\lambda a.p\ (z\ a)$: $(A \to (Projector\ n))$
$\to (A \to (Projector\ n))$ $(\to intro)$

12b. : $(Projector\ (\mathbf{S}\ n))\downarrow$
$\to (Projector\ (\mathbf{S}\ n))\downarrow$ $(= \Pi Succ, = \Pi Succ)$
: $Pretupler\ (\mathbf{S}\ n)\downarrow$ $(\equiv Pretupler)$

13. $\{n : Nat\}$
$\vdash \lambda k.\lambda p.\lambda z.\lambda a.p\ (z\ a)$: $Nat \to$
$(Pretupler\ n)\downarrow$
$\to Pretupler\ (\mathbf{S}\ n)\downarrow$ $(\to intro, \to intro)$

Recursive sequence introduction.

14. $\vdash \mathbf{R}\ (\lambda a.\lambda z.z\ a)$
$(\lambda k.\lambda p.\lambda z.\lambda a.p\ (z\ a))$: $\Pi n.(Pretupler\ n)\downarrow$ $(\Pi intro\text{-}seq\ 5b\ 13)$
: $\Pi n.(Projector\ n)$
$\to (Projector\ n)$ $(\equiv Pretupler)$
$\vdash pretupler$: $\Pi n.(Projector\ n)$
$\to (Projector\ n)$ $(\equiv pretupler)$

∎

A.2 Projections

Projections work on concrete tuples and are parameterized by the tuple length in the first argument and the projection number in the second argument. Recall from Appendix A.1 that tuples of length one are indexed by zero, tuples of length two are indexed by one, and so forth. Thus in a projection, if the tuple length parameter is zero, then we are projecting from one tuples. Likewise, the first projection is indexed by zero, the second projection is indexed by one, and so forth. Any attempt to project beyond the length of the tuple is equivalent to a projection of the last element of the tuple. The following table shows the entire family of projection terms.

Preproj terms look like the following.

(1-*tuple*)	*preproj* 0	=	$\lambda z.z$
(2-*tuple*)	*preproj* 1	=	$\lambda z.\lambda a_1.z$
⋮	⋮		
$((k+1)$-*tuple*)	*preproj* k	=	$\lambda z \lambda a_1 \ldots \lambda a_k.z$

Tuple size	Projection number				
	0	1	2	3	...
(1-*tuple*) 0	$\lambda z.z$	$\lambda z.z$	$\lambda z.z$	$\lambda z.z$	...
(2-*tuple*) 1	$\lambda z.\lambda y.z$	$\lambda y.\lambda z.z$	$\lambda y.\lambda z.z$	$\lambda y.\lambda z.z$	...
(3-*tuple*) 2	$\lambda z.\lambda x.\lambda y.z$	$\lambda x.\lambda z.\lambda y.z$	$\lambda x.\lambda y.\lambda z.z$	$\lambda x.\lambda y.\lambda z.z$	...
(4-*tuple*) 3	$\lambda z.\lambda w.\lambda x.\lambda y.z$	$\lambda w.\lambda z.\lambda x.\lambda y.z$	$\lambda w.\lambda x.\lambda z.\lambda y.z$	$\lambda w.\lambda x.\lambda y.\lambda z.z$	...
⋮	⋮	⋮	⋮	⋮	⋮

Table A.2: A family of Projection Terms

Pswap terms look like the following. The role of *pswap* is to adjust the *preproj* terms, which are essentially all first projections, into j-projections for some projection number j. The pswap is similar to the swap term of tupling, except that the number of swapping steps is dependent on the projection-number second argument rather than the tuple size. Interestingly, the type of *pswap* is dependent on the tuple length first argument that is not actually used in the *pswap* terms.

$$\begin{aligned}
pswap\ n\ 0 &= \lambda f.\lambda z.f\ z \\
pswap\ n\ 1 &= \lambda f.\lambda a_1.\lambda z.f\ z\ a_1 \\
pswap\ n\ 2 &= \lambda f.\lambda a_1.\lambda a_2.\lambda z.f\ z\ a_2\ a_1 \\
pswap\ n\ 3 &= \lambda f.\lambda a_1.\lambda a_2.\lambda a_3.\lambda z.f\ z\ a_3\ a_2\ a_1 \\
&\vdots \\
pswap\ n\ k &= \lambda f.\lambda a_1 \ldots \lambda a_k.\lambda z.f\ z\ a_k \ldots a_1
\end{aligned}$$

The projection, preproj, and pswap terms are constructed according to the following definitions.

$$preproj \equiv \mathbf{R}\ (\lambda z.z)\ (\lambda k.\lambda p.\lambda z.\lambda a.p\ z)$$

$$pswap \equiv \lambda n.\lambda j.\mathbf{R}\ (\lambda f.\lambda z.f\ z)\ (\lambda k.\lambda p.\lambda f.\lambda a.(p\ (\lambda z.f\ z))\ a)\ j$$

$$projection \equiv \lambda n.\lambda j.pswap\ n\ j\ (preproj\ n)$$

The following type definitions are convenient for typing projections. We borrow the definitions for *Projector* from Appendix A.1.

$$\begin{aligned}
Projector &\equiv \mathbf{R}\ (A \to A)\ (\lambda k.\lambda t.A \to t) \\
&\equiv \Pi n.(A \to)^n (A \to A)
\end{aligned}$$

$$Pswap \equiv \Pi n.(Projector\ n \to Projector\ n)$$

We will derive the following typings in the non-weak $\mathcal{T}^{\pi}$ type system.

$$\begin{array}{lcl} \mathit{preproj} & : & \mathit{Projector} \\ \mathit{pswap} & : & \Pi n.(\mathit{Nat} \to \mathit{Pswap}\ n) \\ \mathit{projection} & : & \Pi n.(\mathit{Nat} \to \mathit{Projector}\ n) \end{array}$$

First derive the typing for *projection* assuming typings for *preproj* and *pswap* terms.

Theorem

$$\mathit{projection} : \Pi n.(\mathit{Nat} \to \mathit{Projector}\ n)$$

Assumptions

$$\begin{array}{lcl} A_1 \vdash \mathit{preproj} & : & \mathit{Projector} \\ A_2 \vdash \mathit{pswap} & : & \Pi n.(\mathit{Nat} \to \mathit{Pswap}\ n) \end{array}$$

Proof

1.	$\{n : Nat\}\{j : Nat\} \vdash preproj$	:	$Projector$	$(Cext\ A_1)$
2.	$\{n : Nat\}\{j : Nat\} \vdash n$	:	Nat	(Var)
3.	$\{n : Nat\}\{j : Nat\} \vdash preproj\ n$	:	$Projector\ n$	$(\Pi elim\ 1\ 2)$
4.	$\{n : Nat\}\{j : Nat\} \vdash pswap$	:	$\Pi n.(Nat \to Pswap\ n)$	$(Cext\ A_2)$
5.	$\{n : Nat\}\{j : Nat\} \vdash pswap\ n$	:	$Nat \to Pswap\ n$	$(\Pi elim\ 4\ 2)$
6.	$\{n : Nat\}\{j : Nat\} \vdash j$	:	Nat	(Var)
7.	$\{n : Nat\}\{j : Nat\} \vdash pswap\ n\ j$	:	$Pswap\ n$	$(\to elim\ 5\ 6)$
		:	$Projector\ n \to Projector\ n$	$(\equiv Pswap)$
8.	$\{n : Nat\}\{j : Nat\}$			
	$\vdash pswap\ n\ j\ (preproj\ n)$	:	$Projector\ n$	$(\to elim\ 7\ 3)$
9.	$\{n : Nat\}$			
	$\vdash \lambda j.pswap\ n\ j\ (preproj\ n)$	:	$Nat \to Projector\ n$	$(\to intro)$
10.	$\vdash \lambda n.\lambda j.pswap\ n\ j\ (preproj\ n)$	:	$\Pi n.(Nat \to Projector\ n)$	$(\Pi intro)$
	$\vdash projection$	:	$\Pi n.(Nat \to Projector\ n)$	$(\equiv projection)$

Now derive a typing for the *preproj* term.

Theorem

$$\mathit{preproj} : \mathit{Projector}$$

Proof

Zero case.

1.	$\{z : A\} \vdash z$	:	A	(*Var*)
2*a*.	$\vdash \lambda z.z$	:	$A \to A$	($\to$ *intro*)
2*b*.		:	$(Projector\ \mathbf{0})\downarrow$	($= \Pi Zero$)

Successor case.

3.	$\{n : Nat\}\{k : Nat\}\{p : Projector\ n\}$			
	$\{z : A\}\{a : A\} \vdash p$	:	$Projector\ n$	(*Var*)
		:	$(A \to)^n(A \to A)$	($\equiv$ *Projector*)
4.	$\{n : Nat\}\{k : Nat\}\{p : Projector\ n\}$			
	$\{z : A\}\{a : A\} \vdash z$	:	A	(*Var*)
5.	$\{n : Nat\}\{k : Nat\}\{p : Projector\ n\}$			
	$\{z : A\}\{a : A\} \vdash p\ z$	:	$(A \to)^{n-1}(A \to A)$	($\to$ *elim* 3 4)
6.	$\{n : Nat\}\{k : Nat\}\{p : Projector\ n\}$			
	$\{z : A\} \vdash \lambda a.p\ z$	:	$(A \to)^n(A \to A)$	($\to$ *intro*)
		:	$Projector\ n$	($\equiv$ *Projector*)
7.	$\{n : Nat\}\{k : Nat\}\{p : Projector\ n\}$			
	$\vdash \lambda z.\lambda a.p\ z$	:	$A \to Projector\ n$	($\to$ *intro*)
7*b*.		:	$Projector\ (\mathbf{S}\ n)\downarrow$	($= \Pi Succ$)

8.	$\{n : Nat\} \vdash \lambda k.\lambda p.\lambda z.\lambda a.p\ z$	:	$Nat \to (Projector\ n)\downarrow$	
			$\to (Projector\ (\mathbf{S}\ n))\downarrow$	($\to$ *intro*, $\to$ *intro*)

Primitive recursion introduction.

9.	$\vdash \mathbf{R}\ (\lambda z.z)\ (\lambda k.\lambda p.\lambda y.\lambda x.p\ y)$	:	$\Pi n.(Projector\ n)\downarrow$	(Π*intro-seq* 2*b* 8)
		:	$Projector$	($= \beta, \equiv$ *Projector*)
	$\vdash preproj$	:	$Projector$	($\equiv$ *preproj*)

∎

Finally, derive a typing for the pswap term.

Theorem

$$pswap \ : \ \Pi n.(Nat \to Pswap\ n)$$

Proof

Zero case.

1.	$\{n : Nat\}\{j : Nat\}\{f : A \to A\}$			
	$\{z : A\} \vdash f$	:	$A \to A$	(Var)
2.	$\{n : Nat\}\{j : Nat\}\{f : A \to A\}$			
	$\{z : A\} \vdash z$	:	A	(Var)
3.	$\{n : Nat\}\{j : Nat\}\{f : A \to A\}$			
	$\{z : A\} \vdash f\ z$	:	A	$(\to elim\ 1\ 2)$
4.	$\{n : Nat\}\{j : Nat\}\{f : A \to A\}$			
	$\vdash \lambda z.f\ z$	:	$A \to A$	$(\to intro\ )$
5.	$\{n : Nat\}\{j : Nat\} \vdash \lambda f.\lambda z.f\ z$	:	$(A \to A) \to (A \to A)$	$(\to intro)$
5a.		:	$(Projector\ \mathbf{0})\downarrow$	
			$\to (Projector\ \mathbf{0})\downarrow$	$(= \Pi Zero, = \Pi Zero)$
		:	$Pswap\ \mathbf{0}$	$(\equiv Pswap)$

Successor case.

6.	$\{n : Nat\}\{j : Nat\}\{k : Nat\}$			
	$\{p : Pswap\ n\}\{f : Projector\ n\}$			
	$\{a : A\}\{z : A\} \vdash f$	:	$Projector\ n$	(Var)
		:	$(A \to)^n(A \to A)$	$(\equiv Projector)$
7.	$\{n : Nat\}\{j : Nat\}\{k : Nat\}$			
	$\{p : Pswap\ n\}\{f : Projector\ n\}$			
	$\{a : A\}\{z : A\} \vdash z$	:	A	(Var)
8.	$\{n : Nat\}\{j : Nat\}\{k : Nat\}$			
	$\{p : Pswap\ n\}\{f : Projector\ n\}$			
	$\{a : A\}\{z : A\} \vdash f\ z$	:	$(A \to)^{n-1}(A \to A)$	$(\to elim\ 6\ 7)$
9.	$\{n : Nat\}\{j : Nat\}\{k : Nat\}$			
	$\{p : Pswap\ n\}\{f : Projector\ n\}$			
	$\{a : A\} \vdash \lambda z.f\ z$	:	$(A \to)^n(A \to A)$	$(\to intro\ )$
		:	$Projector\ n$	$(\equiv Projector)$
10.	$\{n : Nat\}\{j : Nat\}\{k : Nat\}$			
	$\{p : Pswap\ n\}\{f : Projector\ n\}$			
	$\{a : A\} \vdash p$	:	$Pswap\ n$	(Var)
		:	$Projector\ n \to Projector\ n$	$(\equiv Pswap)$

11. $\{n : Nat\}\{j : Nat\}\{k : Nat\}$
$\{p : Pswap\ n\}\{f : Projector\ n\}$
$\{a : A\} \vdash p\ (\lambda z.f\ z)$: $Projector\ n$ ($\rightarrow$ *elim* 10 9)
: $(A \rightarrow)^n(A \rightarrow A)$ ($\equiv$ *Projector*)

12. $\{n : Nat\}\{j : Nat\}\{k : Nat\}$
$\{p : Pswap\ n\}\{f : Projector\ n\}$
$\{a : A\} \vdash a$: A (*Var*)

13. $\{n : Nat\}\{j : Nat\}\{k : Nat\}$
$\{p : Pswap\ n\}\{f : Projector\ n\}$
$\{a : A\} \vdash p\ (\lambda z.f\ z)\ a$: $(A \rightarrow)^{n-1}(A \rightarrow A)$ ($\rightarrow$ *elim* 11 12)

14. $\{n : Nat\}\{j : Nat\}\{k : Nat\}$
$\{p : Pswap\ n\}\{f : Projector\ n\}$
$\vdash \lambda a.p\ (\lambda z.f\ z)\ a$: $(A \rightarrow)^n(A \rightarrow A)$ ($\rightarrow$ *intro*)
: $Projector\ n$ ($\equiv$ *Projector*)

15. $\{n : Nat\}\{j : Nat\}\{k : Nat\}$
$\{p : Pswap\ n\}$
$\vdash \lambda f.\lambda a.p\ (\lambda z.f\ z)\ a$: $Projector\ n$
$\rightarrow Projector\ n$ ($\rightarrow$ *intro*)
: $Pswap\ n$ ($\equiv$ *Pswap*)

16. $\{n : Nat\}\{j : Nat\}$
$\vdash \lambda k.\lambda p.\lambda f.\lambda a.p\ (\lambda z.f\ z)\ a$: $Nat \rightarrow Pswap\ n$
$\rightarrow Pswap\ n$ ($\rightarrow$ *intro*, $\rightarrow$ *intro*)

Primitive recursion introduction.

17. $\{n : Nat\}\{j : Nat\}$
$\vdash \mathbf{R}\ (\lambda f.\lambda z.f\ z)\ (\lambda k.\lambda p.\lambda f.\lambda a.p\ (\lambda z.f\ z)\ a)$
: $Nat \rightarrow Pswap\ n$ ($\rightarrow$ *intro-seq* 5*a* 16

18. $\{n : Nat\}\{j : Nat\} \vdash j$: Nat (*Var*)

19. $\{n : Nat\}\{j : Nat\}$
$\vdash \mathbf{R}\ (\lambda f.\lambda z.f\ z)$
$(\lambda k.\lambda p.\lambda f.\lambda a.p\ (\lambda z.f\ z)\ a)\ j$: $Pswap\ n$ ($\rightarrow$ *elim* 17 18)

20. $\{n : Nat\}$
$\vdash \lambda j.\mathbf{R}\ (\lambda f.\lambda z.f\ z)$
$(\lambda k.\lambda p.\lambda f.\lambda a.p\ (\lambda z.f\ z)\ a)\ j$: $Nat \rightarrow Pswap\ n$ ($\rightarrow$ *intro*)

21. $\{n : Nat\}$
$\vdash \lambda n.\lambda j.\mathbf{R}\ (\lambda f.\lambda z.f\ z)$
$(\lambda k.\lambda p.\lambda f.\lambda a.p\ (\lambda z.f\ z)\ a)\ j$: $\Pi n.(Nat \rightarrow Pswap\ n)$ (Π*intro*)
$\vdash pswap$: $\Pi n.(Nat \rightarrow Pswap\ n)$ ($\equiv$ *pswap*)

Here's an exercise for the reader. Try typing a concrete tuple applied to a projection.

Appendix B

Reconstruction Example

To improve readability of this example execution of the algorithm W, we let the notation $W : \Gamma \vdash \delta$ stand for $W(\Gamma, \delta)$. Numbered items state a reconstruction problem to be solved. For each problem, a list of the sub-problems is given, and then the sub-problems are attempted in turn. We are reconstructing a $\mathcal{T}^\pi$ type for the following *taut* program of Section 2.2.

$$taut \equiv \mathbf{R}\ (\lambda f.f)\ (\lambda n'.\lambda p.\lambda f.((p\ f\ true)\ \&\ (p\ (f\ false))))$$

For this example we have three assumptions that hold for all contexts Γ

A_1 $\Gamma \vdash \& : Bool \to Bool \to Bool$

A_2 $\Gamma \vdash true : Bool$

A_3 $\Gamma \vdash false : Bool$

1 $W : \{\} \vdash \mathbf{R}\ (\lambda f.f)\ \lambda n'.\lambda p.\lambda f.\&\ (p\ (f\ true))\ (p\ (f\ false))$. Apply the Π*intro-seq* rule to get the following sub-problems. Return $(\theta_4\theta_3\theta_2\theta_1, \Pi n.\theta_4\theta_3 A)$

- **1.1** $(\theta_1, \sigma) = W : \{\} \vdash \lambda n'.\lambda p.\lambda f.\&\ (p\ (f\ true))\ (p\ (f\ false))$
- **1.2** $(\theta_2, \tau) = W : \theta_1\{\} \vdash \lambda f.f$
- **1.3** $\theta_3 = \text{Unify}(\theta_2\sigma, \text{Nat} \to A \to B)$. The variables A and B are new.
- **1.4** $\theta_4 = \text{Match}(n, \langle \theta_3\tau, \theta_3 A, \theta_3 B\rangle)$. The variable n is new.

1.1 $(\theta_1, \sigma) = W : \{\} \vdash \lambda n'.\lambda p.\lambda f.\&\ (p\ (f\ true))\ (p\ (f\ false))$. Apply the $\to$*intro* rule three times to get the following sub-problem. Return $(\theta_1, \sigma) = (\theta, \theta C \to \theta D \to \theta E \to \tau)$.

- **1.1.1** $(\theta, \tau) = W : \{n' : C\}\{p : D\}\{f : E\} \vdash \&\ (p\ (f\ true))\ (p\ (f\ false))$ where variables C, D, and E are new.

1.1.1 $(\theta, \tau) = W : \{n' : C\}\{p : D\}\{f : E\} \vdash \&\ (p\ (f\ true))\ (p\ (f\ false))$ where variables C, D, and E are new. Apply the $\rightarrow elim$ rule to get the following sub-problems. Return $(\theta, \tau) = (\theta_3\theta_2\theta_1, \theta_3 F)$

1.1.1.1 $(\theta_1, \tau) = W : \{n' : C\}\{p : D\}\{f : E\} \vdash \&\ (p\ (f\ true))$

1.1.1.2 $(\theta_2, \sigma) = W : \theta_1\{n' : C\}\{p : D\}\{f : E\} \vdash (p\ (f\ false))$

1.1.1.3 $\theta_3 = \text{Unify}(\theta_2\tau, \sigma \rightarrow F)$. Variable F is new.

1.1.1.1 $(\theta_1, \tau) = W : \{n' : C\}\{p : D\}\{f : E\} \vdash \&\ (p\ (f\ true))$. Apply $\rightarrow elim$ rule to get the following sub-problems. Return $(\theta_1, \tau) = (\theta_3\theta_2\theta_1, \theta_3 G)$

1.1.1.1.1 $(\theta_1, \tau) = W : \{n' : C\}\{p : D\}\{f : E\} \vdash \&$

1.1.1.1.2 $(\theta_2, \sigma) = W : \theta_1\{n' : C\}\{p : D\}\{f : E\} \vdash p\ (f\ true)$

1.1.1.1.3 $\theta_3 = \text{Unify}(\theta_2\tau, \sigma \rightarrow G)$. Variable G is new.

1.1.1.1.1 $(\theta_1, \tau) = W : \{n' : C\}\{p : D\}\{f : E\} \vdash \&$. Return $(\theta_1, \tau) = (\theta_{id}, Bool \rightarrow Bool \rightarrow Bool)$ by Assumption A_1

1.1.1.1.2 $(\theta_2, \sigma) = W : \theta_{id}\{n' : C\}\{p : D\}\{f : E\} \vdash p\ (f\ true)$. Apply the $\rightarrow elim$ rule and generate the following sub-problems. Return $(\theta_2, \sigma) = (\theta_3\theta_2\theta_1, \theta_3 H)$

1.1.1.1.2.1 $(\theta_1, \tau) = W : \{n' : C\}\{p : D\}\{f : E\} \vdash p$

1.1.1.1.2.2 $(\theta_2, \sigma) = W : \theta_1\{n' : C\}\{p : D\}\{f : E\} \vdash (f\ true)$

1.1.1.1.2.3 $\theta_3 = \text{Unify}(\theta_2\tau, \sigma \rightarrow H)$. The variable H is new.

1.1.1.1.2.1 $(\theta_1, \tau) = W : \{n' : C\}\{p : D\}\{f : E\} \vdash p$. Apply the *Var* rule and return $(\theta_1, \tau) = (\theta_{id}, D)$

1.1.1.1.2.2 $(\theta_2, \sigma) = W : \theta_{id}\{n' : C\}\{p : D\}\{f : E\} \vdash (f\ true)$. Apply the $\rightarrow elim$ rule and generate the following sub-problems. Return $(\theta_2, \sigma) = (\theta_3\theta_2\theta_1, \theta_3 K)$

1.1.1.1.2.2.1 $(\theta_1, \tau) = W : \{n' : C\}\{p : D\}\{f : E\} \vdash f$

1.1.1.1.2.2.2 $(\theta_2, \sigma) = W : \theta_1\{n' : C\}\{p : D\}\{f : E\} \vdash true$

1.1.1.1.2.2.3 $\theta_3 = \text{Unify}(\theta_2\tau, \sigma \rightarrow K)$. The variable K is new.

1.1.1.1.2.2.1 $(\theta_1, \tau) = W : \{n' : C\}\{p : D\}\{f : E\} \vdash f$. Apply the *Var* rule and return $(\theta_1, \tau) = (\theta_{id}, E)$

1.1.1.1.2.2.2 $(\theta_2, \sigma) = W : \theta_{id}\{n' : C\}\{p : D\}\{f : E\} \vdash true$. Return $(\theta_2, \sigma) = (\theta_{id}, Bool)$ by Assumption A_2.

1.1.1.1.2.2.3 $\theta_3 = \text{Unify}(\theta_{id}E, Bool \to K)$. The variable K is new. Return $\theta_3 = \{E := Bool \to K\}$

1.1.1.1.2.2 Return $(\theta_2, \sigma) = (\theta_3\theta_2\theta_1, \theta_3 K) = (\{E := Bool \to K\}\theta_{id}\theta_{id}, \{E := Bool \to K\}K) = (\{E := Bool \to K\}, K)$. End of 1.1.1.1.2.2

1.1.1.1.2.3 $\theta_3 = \text{Unify}(\{E := Bool \to K\}D, K \to H) = \text{Unify}(D, K \to H)$. Return the result of the Unify, that is, return $\theta_3 = \{D := K \to H\}$.

1.1.1.1.2 Return $(\theta_2, \sigma) = (\theta_3\theta_2\theta_1, \theta_3 H) = (\{D := K \to H\}\{E := Bool \to K\}\theta_{id}, \{D := K \to H\}H) = (\{D := K \to H\}\{E := Bool \to K\}, H)$. End of 1.1.1.1.2.

1.1.1.1.3 $\theta_3 = \text{Unify}(\{D := K \to H\}\{E := Bool \to K\}Bool \to Bool \to Bool, H \to G) = \text{Unify}(Bool \to (Bool \to Bool), H \to G) = \{H := Bool\}\{G := Bool \to Bool\}$

1.1.1.1 Return $(\theta_1, \tau) = (\theta_3\theta_2\theta_1, \theta_3 G) = (\{H := Bool\}\{G := Bool \to Bool\}\{D := K \to H\}\{E := Bool \to K\}\theta_{id}, \{H := Bool\}\{G := Bool \to Bool\}G) = (\{G := Bool \to Bool\}\{D := K \to Bool\}\{E := Bool \to K\}, Bool \to Bool)$ End of 1.1.1.1.

1.1.1.2 $(\theta_2, \sigma) = W : \theta_1\{n' : C\}\{p : D\}\{f : E\} \vdash p\ (f\ false) = W : (\{G := Bool \to Bool\}\{D := K \to Bool\}\{E := Bool \to K\}\{n' : C\}\{p : D\}\{f : E\} \vdash p\ (f\ false) = W : \{n' : C\}\{p : K \to Bool\}\{f : Bool \to K\} \vdash p\ (f\ false)$. Apply the $\to elim$ rule and generate the following sub-problems. Return $(\theta_2, \sigma) = (\theta_3\theta_2\theta_1, \theta_3 L)$

1.1.1.2.1 $(\theta_1, \tau) = W : \{n' : C\}\{p : K \to Bool\}\{f : Bool \to K\} \vdash p$

1.1.1.2.2 $(\theta_2, \sigma) = W : \theta_1\{n' : C\}\{p : K \to Bool\}\{f : Bool \to K\} \vdash (f\ false)$

1.1.1.2.3 $\theta_3 = \text{Unify}(\theta_2\tau, \sigma \to L)$. The variable L is new.

1.1.1.2.1 $(\theta_1, \tau) = W : \{n' : C\}\{p : K \to Bool\}\{f : Bool \to K\} \vdash p$. Apply the *Var* rule and return $(\theta_1, \tau) = (\theta_{id}, K \to Bool)$

1.1.1.2.2 $(\theta_2, \sigma) = W : \theta_{id}\{n' : C\}\{p : K \to Bool\}\{f : Bool \to K\} \vdash (f\ false)$. Apply the $\to elim$ rule and generate the following sub-problems. Return $(\theta_2, \sigma) = (\theta_3\theta_2\theta_1, \theta_3 M)$

1.1.1.2.2.1 $(\theta_1, \tau) = W : \{n' : C\}\{p : K \to Bool\}\{f : Bool \to K\} \vdash f$

1.1.1.2.2.2 $(\theta_2, \sigma) = W : \theta_1\{n' : C\}\{p : K \to Bool\}\{f : Bool \to K\} \vdash false$

1.1.1.2.2.3 $\theta_3 = \text{Unify}(\theta_2\tau, \sigma \to M)$. The variable M is new.

1.1.1.2.2.1 $(\theta_1, \tau) = W : \{n' : C\}\{p : K \to Bool\}\{f : Bool \to K\} \vdash f$. Apply the *Var* rule and return $(\theta_1, \tau) = (\theta_{id}, Bool \to K)$

1.1.1.2.2.2 $(\theta_2, \sigma) = W : \theta_{id}\{n' : C\}\{p : K \to Bool\}\{f : Bool \to K\} \vdash false$. Return $(\theta_2, \sigma) = (\theta_{id}, Bool)$ by Assumption A_2.

1.1.1.2.2.3 $\theta_3 = \text{Unify}(\theta_{id}(Bool \to K), Bool \to M)$. The variable M is new. Return $\theta_3 = \{M := K\}$

1.1.1.2.2 Return $(\theta_2, \sigma) = (\theta_3\theta_2\theta_1, \theta_3 M) = (\{M := K\}\theta_{id}\theta_{id}, \{M := K\}M) = (\{M := K\}, K)$. End of 1.1.1.2.2

1.1.1.2.3 $\theta_3 = \text{Unify}(\{M := K\}K \to Bool, K \to L) = \text{Unify}(K \to Bool, K \to L) = \{L := Bool\}$. Return the result of the Unify, that is, return $\theta_3 = \{L := Bool\}$.

1.1.1.2 Return $(\theta_2, \sigma) = (\theta_3\theta_2\theta_1, \theta_3 L) = (\{L := Bool\}\{M := K\}\theta_{id}, \{L := Bool\}L) = (\{L := Bool\}\{M := K\}, Bool)$ End of 1.1.1.2.

1.1.1.3 $\theta_3 = \text{Unify}(\theta_2\tau, \sigma \to F) = \text{Unify}(\{L := Bool\}\{M := K\}(Bool \to Bool), Bool \to F) = \{F := Bool\}$.

1.1.1 Return $(\theta, \tau) = (\theta_3\theta_2\theta_1, \theta_3 F) = (\{F := Bool\}\{L := Bool\}\{M := K\}\{G := Bool \to Bool\}\{D := K \to Bool\}\{E := Bool \to K\}, \{F := Bool\}F) = (\{F := Bool\}\{L := Bool\}\{M := K\}\{G := Bool \to Bool\}\{D := K \to Bool\}\{E := Bool \to K\}, Bool)$. End of 1.1.1.

1.1 Return $(\theta_1, \sigma) = (\theta, \theta C \to \theta D \to \theta E \to \tau) = \{F := Bool\}\{L := Bool\}\{M := K\}\{G := Bool \to Bool\}\{D := K \to Bool\}\{E := Bool \to K\}, C \to (K \to Bool) \to (Bool \to K) \to Bool)$. End of 1.1.

1.2 $(\theta_2, \tau) = W : \theta_1\{\} \vdash \lambda f.f$. Apply $\to intro$ once to get the following sub-problem. Return $(\theta_2, \tau) = (\theta, \theta N \to \tau)$

1.2.1 $(\theta, \tau) = W : \{f : N\} \vdash f$ where N is new.

1.2.1 $(\theta, \tau) = W : \{f : N\} \vdash f$. Apply the *Var* rule and return $(\theta, \tau) = (\theta_{id}, N)$.

1.2 Return $(\theta_2, \tau) = (\theta_{id}, N \to N)$.

1.3 $\theta_3 = \text{Unify}(\theta_2\sigma, Nat \to A \to B)$, where A and B are new. So $\theta_3 = \text{Unify}(\theta_{id}(C \to (K \to Bool) \to ((Bool \to K) \to Bool))), Nat \to A \to B)$, thus $\theta_3 = \{C := Nat\}\{A := K \to Bool\}\{B := (Bool \to K) \to Bool\}$

1.4 $\theta_4 = \text{Match}(n, \langle\theta_3\tau, \theta_3 A, \theta_3 B\rangle) = \text{Match}(n, \langle N \to N, K \to Bool, (Bool \to K) \to Bool\rangle)$. See Appendix C for the trace of this match. The result is $\theta_2 \circ \theta_1 = \{N := Bool\}\{Y := Bool\}\{K := \mathbf{R}\ Y\ (\lambda n'.\lambda K.Bool \to K)\ n\}$ Note at this point we could

instead try to unify the triple of types $\theta_3\tau, \theta_3 A$, and $\theta_3 B$ and attempt to apply the $\rightarrow$ *intro-seq* rule. But that unify fails due to an occurs.

1. Return $(\theta_4\theta_3\theta_2\theta_1, \Pi n.\theta_4\theta_3 A) = (\{N := Bool\}\{Y := Bool\}\{K := \mathbf{R}\ Y\ (\lambda n'.\lambda K.Bool \rightarrow K)\ n\}\{C := Nat\}\{A := K \rightarrow Bool\}\{B := (Bool \rightarrow K) \rightarrow Bool\}\theta_{id}\{F := Bool\}\{L := Bool\}\{M := K\}\{G := Bool \rightarrow Bool\}\{D := K \rightarrow Bool\}\{E := Bool \rightarrow K\}, \Pi n.\{N := Bool\}\{Y := Bool\}\{K := \mathbf{R}\ Y\ (\lambda n'.\lambda K.Bool \rightarrow K)\ n\}\{C := Nat\}\{A := K \rightarrow Bool\}\{B := (Bool \rightarrow K) \rightarrow Bool\}A) = (\theta_4\theta_3\theta_2\theta_1, \Pi n.\mathbf{R}\ Bool\ (\lambda n'.\lambda K.Bool \rightarrow K)\ n) \rightarrow Bool)$

Appendix C

Matching Example

In this appendix we trace the matching algorithm of Section 3.2 Figure 3.3 on an example that appears as a required step in the example reconstruction of the dependent type for *taut*. (See Appendix B.)

1. $\theta = \text{Match}(n, \langle N \to N, K \to \mathit{Bool}, (\mathit{Bool} \to K) \to \mathit{Bool}\rangle)$. Compute the following sub-problems. Assume n is a new term variable of type *Nat*. Return $\theta_2\theta_1$.

1.1 Do successor matching. $\theta_1 = \text{Smatch}(n, \langle K \to \mathit{Bool}, (\mathit{Bool} \to K) \to \mathit{Bool}\rangle)$.

1.2 Do zero matching. $\theta_2 = \text{Unify}(N \to N, \eta \to \mathit{Bool})$, where $\eta = (\theta_1 X_j)[n := 0]\downarrow$ for all X_j occurs variables in $\{\langle N \to N, K \to \mathit{Bool}, (\mathit{Bool} \to K) \to \mathit{Bool}\rangle\}$.

1.1 Use Smatch to fine the successor matcher. Return θ_1.

$$
\begin{array}{rcll}
\theta_1 & = & \text{Smatch}(n, \langle K \to \mathit{Bool}, (\mathit{Bool} \to K) \to \mathit{Bool}\rangle) & \\
 & = & \text{Smatch}(n, \{\langle K, \mathit{Bool} \to K\rangle, \langle \mathit{Bool}, \mathit{Bool}\rangle\}) & \text{Smatch rule } ii \\
 & = & \text{Smatch}(n, \{\langle K, \mathit{Bool} \to K\rangle\}) & \text{Smatch rule } i \\
 & = & \{K := \mathbf{R}\ Y\ (\lambda n'.\lambda K.\mathit{Bool} \to K)\ n\} & \text{Smatch rule } vi
\end{array}
$$

1.2 Use Unify to find the zero matcher. Return θ_2.

$$
\begin{array}{rcll}
\theta_2 & = & \text{Unify}(\langle N \to N, \eta \to \mathit{Bool}\rangle) & \\
 & & \text{where } \eta = (\theta_1 K)[n := 0]\downarrow \text{ for } K \text{ an occurs variable} & \\
 & = & \text{Unify}(\{\langle N \to N, (\mathbf{R}\ Y\ (\lambda n'.\lambda K.\mathit{Bool} \to K)\ n)[n := \mathbf{0}]\downarrow \to \mathit{Bool}\rangle\}) & \\
 & = & \text{Unify}(\{\langle N \to N, (\mathbf{R}\ Y\ (\lambda n'.\lambda K.\mathit{Bool} \to K)\ \mathbf{0})\downarrow \to \mathit{Bool}\}) & \\
 & = & \text{Unify}(\{\langle N \to N, Y \to \mathit{Bool}\rangle\}) & \\
 & = & \text{Unify}(\{\langle N, Y\rangle, \langle N, \mathit{Bool}\rangle\}) & \text{Unify rule } ii \\
 & = & \text{Unify}(\{\langle N, Y\rangle, \{N := Y\}\langle N, \mathit{Bool}\rangle\}) & \text{Unify rule } v \\
 & = & \text{Unify}(\{\langle Y, \mathit{Bool}\rangle, \{Y := \mathit{Bool}\}\langle N, Y\rangle\}) & \text{Unify rule } v \\
 & = & \{N := \mathit{Bool}\}\{Y := \mathit{Bool}\} &
\end{array}
$$

1. Return $\theta_2\theta_1 = \{N := \mathit{Bool}\}\{Y := \mathit{Bool}\}\{K := \mathbf{R}\ Y\ (\lambda n'.\lambda K.\mathit{Bool} \to K)\ n\}$

Biographical Note

Neal Nelson was born September 3, 1952 in Albany, Oregon and raised in Kennewick, Washington. He attended Washington State University where he received a bachelor's degree in mathematics and a master's degree in computer science. Neal has worked in the semi-conductor industry and taught computer science and mathematics at Reed College, Portland, Oregon and at The Evergreen State College, Olympia, Washington. Neal has three children.