WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Thesis Examination Committee:
Robert Pless, Chair
Aaron Stump
Ronald Cytron
Jeremy Buhler
José Burmudez
Nik Weaver

HIGHER-ORDER ENCODINGS WITH CONSTRUCTORS

by

Edwin M. Westbrook, B.S.

A dissertation presented to the School of Engineering
of Washington University in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2008
Saint Louis, Missouri

ABSTRACT OF THE THESIS

Higher-Order Encodings with Constructors

by

Edwin M. Westbrook

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2008

Research Advisor: Aaron Stump and Robert Pless

As programming languages become more complex, there is a growing call in the research community for machine-checked proofs about programming languages. A key obstacle to this goal is in formalizing *name binding*, where a new name is created in a limited scope. Name binding is used in almost every programming language to refer to the formal arguments to a function. For example, the function $f\ (x) = x * 2$, which doubles its argument, binds the name x for its formal argument. Though this concept is intuitively straightforward, it is complex to define precisely because of the intended properties of name binding. For example, the above function is considered "syntactically equivalent" to $f\ (y) = y * 2$.

It is the goal of this dissertation to posit a new technique for encoding name binding, called Higher-Order Encodings with Constructors or HOEC. HOEC encodes name binding with a construct called the ν-abstraction, which binds new constructors in a limited scope. These constructors can then be used to encode names. ν-abstractions already have the required properties of name bindings, so name binding need only be

formalized once, in the definition of the ν-abstraction. The user thus then gets name binding "for free" and need not define it explicitly.

To demonstrate the usefulness of HOEC, this dissertation gives a language, CPTT, with ν-abstractions. CPTT is a form of Intensional Constructive Type Theory, a formalism which is both a programming language and a logical theory whose proofs can be checked by machine. The key novelty that enables CPTT is the *constructor predicates*, which limits the constructors that may appear in a term, allowing pattern-matching to be defined. The consistency proof for CPTT also has independent interest, as it demonstrates how a simpler form of name binding can be added to a wide variety of theories.

Acknowledgments

Finishing this degree and this dissertation was the most difficult challenge of my life, and I have many people to thank for helping me along the way. My first thanks goes to my advisor, Aaron Stump. Aaron always fostered my interest in learning, and always made sure I had something new and interesting to learn that was useful in my career. My knowledge of many of the subjects touched upon in this dissertation began with the seminars Aaron directed at Wash. U. Aaron then had the magnanimity to let me apply this knowledge as I saw fit, stopping me only when I started down a wrong path, and always supported me with enthusiasm.

My second thanks goes to Joe Ullian. Joe taught me about logic and set theory, two important fundamentals without which this dissertation would not be possible. More than that, however, Joe taught me how to think about these topics. He always claimed that many foundational theories such as set theory defy intuition, and consist merely of formal symbol manipulation, and yet it is from him that I learned many intuitions about these subjects. It is also from Joe that I learned to appreciate the wonder and majesty involved in questions of the foundations of mathematics.

A special thanks goes to all the members of the Computational Logic Group, including Joel Brandt, Li-Yang Tan, Ben Delaware, Morgan Deters, and Ian Wehrman, both for their friendship and comraderie and for their discussions and feedback on many of my ideas. An extra thanks goes to Ian, for reviewing every paper I have yet written, for shaping much of my grammatical style, and for initiating my interests in the fields of logical, computational, and linguistic pedantics.

I would also like to thank my committee. Each member of my committee has supported or inspired me in a meaningful way throughout my degree. The feedback I received on my dissertation from them has also been invaluable. I am especially grateful to Robert Pless, both for agreeing to serve as chair of my committee and for his numerous pieces of sound, practical advice throughout my five years in Saint Louis.

None of this would have been possible without the support of my family, who gave me sound advice on everything from how to successfully navigate a Ph.D. program to how

to handle girl problems to what I should cook with my pasta. I would additionally like to thank all of my friends in Saint Louis, including many past and present students and faculty, both for helping me learn to give good talks and for giving me something to do to unwind.

Last, and most importantly, I would like to thank Nicci Cobb for all of her work that went into this dissertation. Without her enormous amount of help, including cooking meals, reviewing my slides, and packing up our apartment, I could not have finished this document and the work it represents. Her tenderness, patience, encouragement, and good humor helped me through the monumental task that I think comes with any dissertation, and for all of these I am eternally grateful.

Edwin M. Westbrook

Washington University in Saint Louis
December 2008

To everyone who believed in me.

Contents

List of Figures

Chapter 1

Introduction

The concept of a name is ubiquitous in Computer Science: programming language variables are names that refer to the formal parameters of functions; pointers in Java or C are names that refer to data stored on the heap; and file descriptors are names that refer to files and other system resources in the UNIX operating system. The concept of naming is powerful exactly because it is simple. Using names abstracts away the specifics of what an entity is and how it is accessed. For example, variables hide the details of where exactly a formal parameter is stored, and pointers obviate concerns of how to access the virtual page that contains a particular piece of data in the heap. Names are also a convenient mechanism to express access and permissions. For instance, if a program is not given a file descriptor for a file, the program cannot even express accesses on that file.

One question concerning names is how they are created. Under the *name binding* paradigm, names can be created only in a limited scope. A common example is the declaration of formal parameters in programming languages. Consider for example the following definintion of the doubling function in a programming language of arithmetic functions:

$$f\ (x) = x * 2$$

The definition of this function creates the new name x to refer to the formal parameter of the function. This particular function uses the name x in the expression $x * 2$, specifying that the input to the function is to be doubled. It is nonsensical, however, to refer to the formal parameter x outside the definition of the doubling function. To ensure this cannot happen, x is created only in the limited scope of the function definition, and exists nowhere else. Any object that creates a name with a limited

scope in this fashion is called a name binding. x in this case is said to be *bound* by the function definition $f\ (x) = x * 2$.

Name binding has the following four properties:

1. **Freshness**: When the name x is bound, the name x itself becomes a new object that is distinct from all other objects, including other names. This is why the two functions

$$f\ (x, y) = x * 2 \qquad\qquad f\ (x, y) = y * 2$$

   are disinct. The variable x is even distinct from other names written as x. For example, if we allow nested functions, meaning a functions returning functions, then the function

$$f\ (x) = (g\ (x) = x * 2)$$

   takes any argument and returns the doubling function. In this example, the x in $x * 2$ refers to the innermost argument, while the outer-most argument is not used. The two copies of x are thus distinct.

2. α**-equivalence**: It should not matter what symbol is used in name binding. Thus name bindings are equal modulo renaming of bound names, and the function $f\ (x) = x * 2$ should be indistinguishable from the function $f\ (y) = y * 2$, even if the observer can "look inside" the definitions.

3. **Scoping**: Names cannot escape their bindings, and the funciton

$$f\ (x) = x + y$$

   is malformed, unless it is a nested function inside another function with argument y.

4. **Typing**: When a name is bound, it is often given a specific type or classifier indicating how it can be used. For example, the doubling function above requires its argument to be a number. For example, if the notation $[1, 2, 3]$ is used for the list of the numbers 1, 2, and 3, and $l_1 + + l_2$ denotes appending of lists, then the function

$$f\ (x) = x + + [x, (x * 2)]$$

is malformed, as x is used as both a list and a number.

Note that typing is not always necessary, for instance in a language with just number expressions. Thus we consider it as an optional property here. A formalization of variable binding that includes typing will be called *higher-order* below, while others are called *first-order*.

These four properties can be viewed as axioms for defining name binding, and have a number of implications. Scoping implies that names cannot be created except by name bindings, or else names could be used out of their scope. α-equivalence implies that names are interchangeable, and thus a program must behave identically no matter which actual name is used. Names must therefore have no observable internal structure, unless this structure is the same across all names.

Unfortunately, although these concepts are intuitively straightforward, it is cumbersome to define name binding formally. This means it is likewise cumbersome to formalize programming languages and prove properties about them, as almost all programming languages rely in an integral way on name binding. Programming languages, however, are making stronger guarantees about software. For example, strongly-typed languages such as ML [45] and Haskell [69] ensure that executing a program can never yield a type error such as trying to add to objects that are not numbers. As another example, ownership types can be used to a strict locking protocol [9]. In order that programmers may trust these guarantees, there is a growing call in the research community for formalized, machine-checkable proofs about programming languages. As suggested by the POPLmark Challenge [4], at least two workshops [54, 75], and much research [42, 21, 74, 71, 62, 55, 57, 63, 60, 10], formalizing name binding is a key obstacle to this goal.

It is the purpose of this dissertation to propose a new approach to formalizing variable binding. This approach, called Higher-Order Encodings with Constructors or HOEC, enriches the standard concept of algebraic datatypes with a construct, called the ν-abstraction, that is useful for encoding name binding. Algebraic datatypes encode data with a set of constructors, or function symbols, for building elements of the datatype. For example, the natural numbers are often encoded as an algebraic datatype with the nullary constructor zero and the unary constructor succ.

HOEC enriches this notion by adding ν-abstractions, which introduce a new constructor in a local scope. ν-abstractions can then be used to encode name bindings, while the locally-scoped constructors they introduce can be used to encode names. ν-abstractions themselves are binding constructs, and so already have the aforementioned four properties of name binding. Thus name binding need only be formalized once, in the meta-language, and the user then gets these properties "for free" and need not formalize them explicitly. Note that the ν-abstraction itself was introduced in other work [63], but its use here is novel.

HOEC is similar to Higher-Order Abstract Syntax, or HOAS [52]. The key difference is that HOAS uses meta-language variables to encode names, while HOEC uses constructors. The difficulty with the HOAS approach is that meta-language variables cannot be considered distinct from any other term. This is because of the Substitution principle, which states that properties that can be expressed in the meta-language are closed under substitution for variables. Thus if $x \neq M$ were true for any M then the Substitution principle implies that M itself can be substituted for x, yielding the contradiction $M \neq M$! This means that, in a sense, HOAS encodings *do not satisfy freshness*. This is discussed in Section 8.1 below. In contrast, in the paradigm of algebraic datatypes, constructors are always distinct from all other terms.

HOEC is developed in two ways in this dissertation. First, the formalism of Higher-Order Name-Binding Rewriting is developed. This is an extension of Higher-Order Rewriting, which is itself an extension of standard Term Rewriting [67, 15]. Term Rewriting allows the definition of computation on algebraic datatypes. For example, the definition of computation in a programming language of functions over the natural numbers might include the rewrite rules

$$\begin{array}{lcl} \mathsf{plus}\ x\ \mathsf{zero} & \rightsquigarrow & x \\ \mathsf{plus}\ x\ (\mathsf{succ}\ y) & \rightsquigarrow & \mathsf{succ}\ (\mathsf{plus}\ x\ y) \end{array}$$

to specify how to compute the value of addition expressions. Term Rewriting uses algebraic datatypes as the base language over which computation is defined. Higher-Order Rewriting enriches this base language to the simply-typed λ-calculus, adding simple functions. Higher-Order Name-Binding Rewriting enriches the base language further to the $\lambda\nu$-calculus, allowing locally-bound constructors as well as simple functions. The need for a rewriting formalism in the context of names and name binding

is already motivated in other work [22], and will also be indispensible for defining CPTT and CNIC, introduced below.

The second direction in which HOEC is developed in this dissertation is that it is integrated with Intensional Constructive Type Theory. Intensional Constructive Type Theory, or ICTT, is a logical theory that is also a programming language. This allows the user to write programs and prove properties of them in the same language. The correctness of the proofs can also be checked by machine, increasing user confidence in them. This is especially useful in proving properties of programming languages, where the often repetitive and voluminous nature of the proofs make human analysis error-prone.

HOEC is integrated with ICTT here in two steps. The first step is the Calculus of Nominal Inductive Constructions, or CNIC. CNIC extends the Calculus of Inductive Constructions, a well-known form of ICTT, to give the user the first three properties of name binding, namely, freshness, α-equivalence, and scoping. Typing is omitted to simplify the theory. Instead, all names reside in a single type of names. This is accomplished with a restricted form of the ν-abstraction where constructors can only be bound with the type Name. Recursion is allowed inside ν-abstractions and names can be compared for equality. CNIC is itself of independent interest, as it is closely related to Nominal Logic [59, 23] and similar research [10, 71].

The second step of this development is a language called Constructor Predicate Type Theory, or CPTT, which allows ν-abstractions to bind constructors of any type. The key difficulty here is that recursive functions over algebraic datatypes are defined by pattern-matching, but since arbitrarily many constructors can be introduced at arbitrary types, a pattern-matching function would need infinitely many cases to specify its behavior for each possible constructor. To solve this difficulty, CPTT introduces the novel concept of *constructor predicates*, which specify what constructors may occur in a term. A pattern-matching function must only be specified for the finitely many cases specified by its constructor predicate, and so again becomes finite.

The remainder of this document is structured as follows. The remainder of this Introduction, Section 1.1, introduces the notions of HOEC informally through examples. Chapter 2 gives the necessary for the remainder of the document. Chapter 4

defines and proves some properties of Higher-Order Name-Binding Rewriting. Chapter 3 gives a brief introduction to Intensional Constructive Type Theory. Chapter 5 defines CNIC and proves Type Safety. Chapter 6 proves CNIC consistent with a reduction to a slightly modified version of Intensional Constructive Type Theory already known to be consistent. Chapter 7 defined CPTT, proves Type Safety for it, and informally describes how a subset of it is consistent. The full consistency proof is left as future work. Finally, Chapter 8 concludes with a discussion of related work and potential future work.

1.1 Higher-Order Encodings with Constructors

As suggested above, the goal of HOEC is to enrich algebraic datatypes with a means for encoding name binding. To illustrate this concept, this section considers again the concept of a programming language of arithmetic functions over the natural numbers. Section 1.1.1 introduces the concept of encodings in algebraic datatypes by giving an example encoding of just arithmetic expressions using natural numbers, addition, and multiplication. Section 1.1.2 extends this example to include simple arithmetic functions. The reader conversant with Higher-Order Abstract Syntax will find these sections familiar, as a HOEC encoding is almost identical to a HOAS encoding except that ν-abstractions are used in place of λ-abstractions. The difference between the two approaches is to be found in Section 1.1.3, which gives some examples of operations defined over the given encoding. Here and in the below the phrase *object language* is used to denote the language of arithmetic functions being encoded, while the phrase *meta-language* is used for language in which that encoding is written. Note that the meta-language is not made precise here, as various forms of meta-language are the subject of the remainder of this dissertation.

1.1.1 Algebraic Datatypes

In this section we consider how to encode arithmetic expressions built from the natural numbers and the binary operators $+$ and $\times$. To encode an expression E as an algebraic

datatype, the encoding function $\ulcorner E \urcorner$ is used, defined as

$$\begin{array}{lcl} \ulcorner n \urcorner & = & \mathsf{lit}\ \ulcorner n \urcorner^{\mathrm{nat}} \\ \ulcorner E_1 + E_2 \urcorner & = & \mathsf{plus}\ \ulcorner E_1 \urcorner\ \ulcorner E_2 \urcorner \\ \ulcorner E_1 \times E_2 \urcorner & = & \mathsf{mult}\ \ulcorner E_1 \urcorner\ \ulcorner E_2 \urcorner \\ \ulcorner 0 \urcorner^{\mathrm{nat}} & = & \mathsf{zero} \\ \ulcorner n+1 \urcorner^{\mathrm{nat}} & = & \mathsf{succ}\ \ulcorner n \urcorner^{\mathrm{nat}} \end{array}$$

where $\ulcorner n \urcorner^{\mathrm{nat}}$ is used to encode the natural number n, called a *literal* of the language, by cases on whether n is 0 or not. As an exmple, the expression $1 + 1$ is encoded as plus (succ zero) (succ zero). $+$ and $\times$ are assumed to associate to the left, so that $1 + 1 + 1$ is really $(1 + 1) + 1$, and thus would be encoded as

$$\mathsf{plus}\ (\mathsf{plus}\ (\mathsf{succ}\ \mathsf{zero})\ (\mathsf{succ}\ \mathsf{zero}))\ (\mathsf{succ}\ \mathsf{zero})$$

The encoding function $\ulcorner E \urcorner$ assumes two algebraic datatypes, the type expr for expressions and the type nat for natural numbers. These are defined by the signature

$$\begin{array}{lcl} \mathsf{zero} & : & \mathsf{nat} \\ \mathsf{succ} & : & \mathsf{nat} \Rightarrow \mathsf{nat} \\ \mathsf{lit} & : & \mathsf{nat} \Rightarrow \mathsf{expr} \\ \mathsf{plus} & : & \mathsf{expr} \Rightarrow \mathsf{expr} \Rightarrow \mathsf{expr} \\ \mathsf{mult} & : & \mathsf{expr} \Rightarrow \mathsf{expr} \Rightarrow \mathsf{expr} \end{array}$$

where the symbols on the left of the colons are called *constructors* and the syntax to the right of the colons gives a *type* for each constructor, or specification of how it can be used. The zero constructor is given type nat, meaning it is a nullary constructor for the algebraic datatype nat. succ is given the type nat $\Rightarrow$ nat, specifying that succ M, called the *application* of succ to M, has type nat for any M of type nat. Thus nat is a unary constructor that builds an expression from another. lit is a unary constructor that builds elements of expr from elements of nat. $\Rightarrow$ associates to the right, so the type expr $\Rightarrow$ expr $\Rightarrow$ expr given to plus and mult is really expr $\Rightarrow$ (expr $\Rightarrow$ expr). This specifies that plus M has type expr $\Rightarrow$ expr for M of type expr, and so (plus M) N has type expr for N of type expr. Application associates to the left, so (plus M) N is also written plus M N. Thus plus and mult are thus binary constructors for the type

expr. This approach of using nested applications to encode application to multiple arguments is called *currying* in the literature.

It is straightforward to see that the encoding functions $\ulcorner E \urcorner$ and $\ulcorner n \urcorner^{\mathsf{nat}}$ given above are isomorphisms from the arithmetic expressions to terms of type expr and from the natural numbers and the terms of type nat, respectively. This means that all terms of type expr or nat are encodings of actual expressions or natural numbers, and two different expressions or natural numbers are encoded as two distinct terms of type expr or nat, respectively. Encodings with this property are said to be *adequate*.

1.1.2 Encoding Name Binding

Encoding becomes more difficult when functions are added to the expressions above. This requires the definition of $\ulcorner f\ (x) = E \urcorner$ for functions of the language as well as the definition of $\ulcorner x \urcorner$ for variables. The difficulties come in trying to satisfy the four properties of name binding discussed above. α-equivalence requires that $\ulcorner f\ (x) = E(x) \urcorner$ should be equal to $\ulcorner f\ (y) = E(y) \urcorner$. This means that $\ulcorner x \urcorner$ in the context of encoding $f\ (x) = E(x)$ should be equal to $\ulcorner y \urcorner$ in the context of encoding $f\ (y) = E(y)$. Freshness requires that $\ulcorner x \urcorner$ and $\ulcorner y \urcorner$ be different otherwise. Scoping requires not only that $\ulcorner x \urcorner$ be undefined when not in the context of encoding $f\ (x) = E(x)$, but also that the value of $\ulcorner x \urcorner$ in such a context *cannot be created* outside of such a context. Typing requires that $\ulcorner x \urcorner$ cannot, for example, be used as a natural number in a literal expression.

To accommodate these requirements, HOEC adds the ν-abstraction to the concept of algebraic datatypes. A ν-abstraction has the form $\nu\, c\!:\!A\,.\,M$, where c is a constructor, A is a type, and M is a term. The type A is often omitted when clear from context, and the above ν-abstraction is written $\nu\, c\,.\,M$. Intuitively, $\nu\, c\!:\!A\,.\,M$ creates a new constructor c of type A that can be used only in M. This affects the *world*, a word which here means the set of constructors available for building terms. For example, in the constructor declarations given above for encoding arithmetic expressions, the world includes the four constructors zero, succ, plus, and mult. A ν-abstraction thus extends the world, though this extended world is only available inside the ν-abstraction. The definition of ν-abstractions themselves satisfies the four properties

of name binding given above: the new constructor c is guaranteed to be distinct from all other constructors in the world, so freshness is satisfied; $\nu\, c\!:\!A\,.\,M$ is by definition equal to $\nu\, d : A\,.\,N$ when N is got from M by replacing occurrences of c by d, so α-equivalence is satisfied; a constructor c is only a valid term in a world containing it, so scoping is satisfied; and ν-abstractions allow constructors of different types A to be added to the world, so typing is satisfied.

The HOEC approach then uses ν-abstractions to encode name binding. Such an encoding has the four properties automatically, and need not be formalized explicitly. For example, consider the addition of functions to the above language of arithmetic expressions, where a function of n arguments is written $f\ (x_1, \ldots, x_n) = E$ for some expression E. The function that computes the polynomial $(x*y)+x+y$, for instance, is written in this extension as $f\ (x, y) = (x * y) + x + y$. To encode this extended language, the encoding function $\ulcorner E \urcorner$ is extended with a function f mapping object-language variables to locally-bound constructors that encode them. This is written $\ulcorner E \urcorner^f$. $\ulcorner E \urcorner^f$ is then defined as

$$
\begin{array}{lcl}
\ulcorner n \urcorner^f & = & \mathsf{lit}\ \ulcorner n \urcorner^{\mathrm{nat}} \\
\ulcorner E_1 + E_2 \urcorner^f & = & \mathsf{plus}\ \ulcorner E_1 \urcorner^f\ \ulcorner E_2 \urcorner^f \\
\ulcorner E_1 \times E_2 \urcorner^f & = & \mathsf{mult}\ \ulcorner E_1 \urcorner^f\ \ulcorner E_2 \urcorner^f \\
\ulcorner x \urcorner^f & = & f(x) \\
\ulcorner f\ (x) = E \urcorner^f & = & \mathsf{fun\text{-}one}\ (\nu\, c\,.\,\ulcorner E \urcorner^{f, x \mapsto c}) \\
\ulcorner f\ (x_1, x_2, \ldots) = E \urcorner^f & = & \mathsf{fun\text{-}many}\ (\nu\, c\,.\,\ulcorner f\ (x_2, \ldots) = E \urcorner^{f, x \mapsto c})
\end{array}
$$

where $f, x \mapsto c$ is the function that is identical to f except the result for argument x is changed to be c. To encode a function of a single argument, the constructor $\mathsf{fun\text{-}one}$ is used. For example, the doubling function $f\ (x) = x * 2$ given above would be encoded as

$$\mathsf{fun\text{-}one}\ (\nu\, c\!:\!\mathsf{expr}\,.\,\mathsf{mult}\ c\ (\mathsf{succ}\ (\mathsf{succ}\ \mathsf{zero}))).$$

Note that the variable x gets encoded as the locally bound constructor c. To encode a function of two or more arguments, the constructor $\mathsf{fun\text{-}many}$ is used, so the function $f\ (x, y) = (x * y) + x + y$ would be encoded as

$$\mathsf{fun\text{-}many}\ (\nu\, c_1\!:\!\mathsf{expr}\,.\,\mathsf{fun\text{-}one}\ (\nu\, c_2\!:\!\mathsf{expr}\,.\,\mathsf{plus}\ (\mathsf{mult}\ c_1\ c_2)\ (\mathsf{plus}\ c_1\ c_2)))$$

recalling that + associates to the left. The associated datatypes are then the types nat and expr given above along with the type fun-expr of function expressions, defined as

$$\begin{array}{lcl} \mathsf{fun\text{-}one} & : & (\nabla \mathsf{expr}\,.\,\mathsf{expr}) \Rightarrow \mathsf{fun\text{-}expr} \\ \mathsf{fun\text{-}many} & : & (\nabla \mathsf{expr}\,.\,\mathsf{fun\text{-}expr}) \Rightarrow \mathsf{fun\text{-}expr} \end{array}$$

where $\nabla \mathsf{expr}\,.\,\mathsf{expr}$ is the type of ν-abstractions $\nu\, c : \mathsf{expr}\,.\,M$ where M has type expr and $\nabla \mathsf{expr}\,.\,\mathsf{fun\text{-}expr}$ is similarly the type of similar ν-abstractions $\nu\, c : \mathsf{expr}\,.\,M$ where M has type fun-expr.

It is the case that $\ulcorner F \urcorner^{\cdot}$ is an adequate encoding of the function F, where $\cdot$ is the function that is everywhere undefined. To see this, the adequacy of $\ulcorner E \urcorner^{f}$ must first be established. The domain of $\ulcorner E \urcorner^{f}$ is the set of expressions E over variables in the domain of f. The range of $\ulcorner E \urcorner^{f}$ is the terms of type expr in the world containing the above constructors plus the unary constructors in the range of f. Given this domain and range, it is straightforward to see that $\ulcorner E \urcorner^{f}$ is an isomorphism from the one set to the other. Adequacy of $\ulcorner f\ (x_1, \ldots, x_n) = E \urcorner^{\cdot}$ then follows by induction on the number of variables n.

1.1.3 Operations on HOEC Data

In this section a number of operations are considered on the HOEC encoding given above. These operations are defined by pattern-matching and recursion on the structure of the input. Again, the meta-language is not made precise here. Many of the operations given here can in fact be defined in both the Higher-Order Name-Binding Rewriting formalism of Chapter 4 and the Constructor Predicate Type Theory of Chapter 7, though the syntax differs between the two formalisms. The focus here is instead on the benefits of the HOEC approach for writing such operations.

As a first example, the function countvars is given that computes the number of occurrences of variables in an arithmetic expression. countvars is defined by the

following cases:

$$\begin{array}{lcl}
\mathsf{countvars}\ (\mathsf{lit}\ x) \setminus x & = & \mathsf{zero} \\
\mathsf{countvars}\ c \setminus c & = & \mathsf{succ}\ \mathsf{zero} \\
\mathsf{countvars}\ (\mathsf{plus}\ x\ y) \setminus x, y & = & \mathsf{add}\ (\mathsf{countvars}\ x)\ (\mathsf{countvars}\ y) \\
\mathsf{countvars}\ (\mathsf{mult}\ x\ y) \setminus x, y & = & \mathsf{add}\ (\mathsf{countvars}\ x)\ (\mathsf{countvars}\ y)
\end{array}$$

The syntax $M \setminus \Gamma$ is a *pattern*, where M is a term and Γ is a list of variables and constructors. Such a pattern is said to *match* any term that can be got from M by replacing variables and constructors listed in Γ by arbitrary terms and constructors, respectively. The cases above stipulate the value of countvars on inputs that match the various patterns. The first case thus states that countvars of a literal returns zero, while the third and fourth cases state that countvars of an addition or multiplication expression recursively computes countvars of the two arguments and adds the results. Adding is done with the function add, which itself can be defined as

$$\begin{array}{lcl}
\mathsf{add}\ x\ \mathsf{zero} \setminus x & = & x \\
\mathsf{add}\ x\ (\mathsf{succ}\ y) \setminus x, y & = & \mathsf{succ}\ (\mathsf{add}\ x\ y)
\end{array}$$

The second case of countvars states that countvars of an arbitrary nullary constructor c returns succ zero, as such constructors are used to encode object language variables. This case cannot be written directly for a HOAS encoding, because a pattern cannot distinguish whether an input is a meta-language variable. As stated above, if a pattern could distinguish meta-language variables then the Substitution principle would be violated, as the property of being a meta-language variable is not closed under substitution for meta-language variables.

To extend countvars to the full language of arithmetic functions requires recursing inside ν-abstractions. This is done with the function countvars-fun, defined as follows:

$$\begin{array}{lcl}
\mathsf{countvars\text{-}fun}\ (\mathsf{fun\text{-}one}\ x) \setminus x & = & \mathsf{lift\text{-}nat}\ (\nu\, c\,.\, \mathsf{countvars}\ x\ \langle c \rangle) \\
\mathsf{countvars\text{-}fun}\ (\mathsf{fun\text{-}many}\ x) \setminus x & = & \mathsf{lift\text{-}nat}\ (\nu\, c\,.\, \mathsf{countvars\text{-}fun}\ x\ \langle c \rangle)
\end{array}$$

The intent of this definition is that if the input to countvars-fun is a one-argument function of the form fun-one $(\nu\, c\,.\, E)$, then countvars is called on the expression E, and otherwise the input is a many-argument function of the form fun-many $(\nu\, c\,.\, F)$ and

countvars-fun recurses on F. The difficulty, though, is that pattern-matching cannot match a term of the form fun-one $(\nu\, c\,.\, E)$ and extract E itself. Such an operation would be unsafe, as E would be removed from the scope of the ν-abstraction, possibly allowing c to escape the scope of the ν-abstraction. Scoping states that this cannot happen.

To extract E from $\nu\, c\,.\, E$ is safe, however, if another constructor d is supplied to use for c in E. Intuitively this is because the only freshness stipulation made by $\nu\, c\,.\, E$ is that the new constructor c be fresh for all other constructors that E "knows about." If d is not such a constructor then it is perfectly valid to use it in place of c in E. In a sense, this states that $\nu\, c\,.\, E$ can be viewed as a partial function whose domain excludes all constructors for which c is required to be fresh. The construct $M\ \langle c\rangle$, called a *constructor replacement*, calls this function with argument c. This is equivalent to the concretion operator of [10]. Such a term is only well-formed if c is fresh for M.

The first case of countvars-fun above takes an input of the form fun-one M, binds a fresh constructor c, and calls countvars on $M\ \langle c\rangle$. This yields $\nu\, c : \mathsf{expr}\,.\, n$ for some (encoding of a) natural number n. Note that c could not possibly be used in n, as neither zero or succ take any arguments of type expr. Thus it should be possible to remove n from this ν-abstraction without using a constructor replacment. This is done with the function lift-nat, which *lifts* n out of the ν-abstraction. Similarly, the second case of countvars-fun takes an input of the form fun-one M, binds a fresh constructor c, recurses on $M\ \langle c\rangle$, and calls lift-nat on the resulting ν-abstraction of the form $\nu\, c : \mathsf{fun\text{-}expr}\,.\, n$. Note that, technically speaking, the two copies of lift-nat are different, as one lifts past a constructor of type expr and the other lifts past a constructor of type fun-expr. The two functions are otherwise identical, so we use the same name here.

lift-nat is a pattern-matching function over a ν-abstraction, and can be defined as follows:

$$\begin{aligned} \mathsf{lift\text{-}nat}\ (\nu\, c\,.\, \mathsf{zero})\ \backslash & = \mathsf{zero} \\ \mathsf{lift\text{-}nat}\ (\nu\, c\,.\, \mathsf{succ}\ x\ \langle c\rangle)\ \backslash\ x & = \mathsf{succ}\ (\mathsf{lift\text{-}nat}\ (\nu\, c\,.\, x\ \langle c\rangle)) \end{aligned}$$

The first case takes the input $(\nu\, c\,.\, \mathsf{zero})$ to the output zero. The intent of the second case is to take an input of the form $\nu\, c\,.\, \mathsf{succ}\ x$, recurse on $\nu\, c\,.\, x$, and the return succ

of the result. For the same safety reasons given above, however, a pattern cannot directly match $\nu\, c\,.\, \mathsf{succ}\ x$, as matching x could remove c from its scope. Instead the pattern $\nu\, c\,.\, \mathsf{succ}\ x\ \langle c \rangle$ is used. If the input to $\mathsf{lift\text{-}nat}$ is $\nu\, c\,.\, \mathsf{succ}\ N$, this pattern matches $x\ \langle c \rangle$ against N, meaning that x becomes a ν-abstraction whose constructor replacement is N. This is equivalent to setting x to $\nu\, c\,.\, N$. $\mathsf{lift\text{-}nat}$ then recurses on $\nu\, c\,.\, x\ \langle c \rangle$ and returns its successor.

Chapter 2

Background and Technical Preliminaries

In this chapter, various background material is given that is relevant to this dissertation. Section 2.1 discusses general concepts, such as sets and sequences, and defines some notation for these. Section 2.2 define graphs and a number of related concepts. Section 2.3 discusses term rewriting.

2.1 General Concepts

In the below, standard notation for set theory is assumed. For example, $\{1, 2, 3\}$ denotes the set containing exactly the elements 1, 2, and 3. $\cup$ denotes union and $\cap$ denotes intersection. Ordered pairs are written (x, y). Relations are sets of ordered pairs. If R is a relation, then the notation xRy denotes that the ordered pair (x, y) is in the set R. For any relation R, $R^=$ denotes the reflexive closure of R, meaning $xR^=y$ if and only if xRy or $x = y$. RS denotes the composition of R and S, meaning $xRSy$ if and only if $xRzSy$ for some z. R^+ denotes the transitive closure of R, meaning xR^+y if and only if $xRx_1R \ldots Rx_nRy$ for some sequence of zero or more elements $x_1, \ldots, x_n$. R^* is the reflexive-transitive closure of R, meaning xR^*y if and only if xR^+y or $x = y$. R^{-1} is the inverse of R, meaning $xR^{-1}y$ if and only if yRx. Arrows $\rightsquigarrow$ are sometimes used in the below for relations. In this case, the reflexivie-symmetric-transitive closure of the relation $\rightsquigarrow$ is written $\leftrightsquigarrow^*$. This is equivalent to $(\rightsquigarrow \cup \rightsquigarrow^{-1})^*$.

Sequences are written $x_1, x_2, \ldots, x_n$. For any symbol x, the notation $\vec{x}$ denotes the sequence $x_1, x_2, \ldots, x_n$ of the symbol x with different natural number subscripts. The notation $|\vec{x}|$ is sometimes used for the length of the sequence, meaning the biggest i such that x_i is an object in the sequence. Often these notions are left somehwat implicit.

A category is triple $(O, A, \circ)$ of a set O, a family of sets $A_{(o_1,o_2)}$ indexed by ordered pairs (o_1, o_2) of elements of O, and a binary function $\circ$ mapping elements of $A_{o_3,o_2} \times A_{o_2,o_1}$ to A_{o_3,o_1}. The elements of O are called the *objects*, the elements of A_{o_2,o_1} are called the *morphisms from* o_1 *to* o_2, and $\circ$ is called the *composition* operator. $\circ$ is further required to be associative, meaning $a_3 \circ (a_2 \circ a_1) = (a_3 \circ a_2) \circ a_1$, and every object o is required to have an identity mapping id in $A_{o,o}$ such that $a \circ \mathsf{id} = a$ and $\mathsf{id} \circ a = a$ for every mapping a.

2.2 Graphs

In this section some concepts related to graphs are briefly defined. These are standard, and can be found in many textbooks [13].

Definition 2.2.1 (Graph). *A* directed graph *is a pair of a set V, called the* vertices *of the graph, and a binary relation E on V, called the* edges *of the graph.*

Viewing binary relations as sets of ordered pairs, an edge is thus an ordered pair (v_1, v_2) of vertices $v_1, v_2 \in V$.

Definition 2.2.2 (Graph Concepts). *Given a graph $G = (V, E)$, the following are useful definitions:*

- *A* finite *graph is one where V and E are both finite. Since E is not a multiset here, V being finite implies that E is also finite.*
- *The* in-degree *of $v \in V$ is the number of edges $(v', v) \in E$ for some $v' \in V$.*
- *The* out-degree *of $v \in V$ is the number of edges $(v, v') \in E$ for some $v' \in V$.*

- *A* path *from v_1 to v_n in G is a sequence $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$ of edges in E such that $v_i \neq v_j$ for $i \neq j$, except for the special case, called a* loop*, where $v_1 = v_n$ is allowed. v_1 is the* beginning *of the path and v_2 is the* end*. Excluding the loops, paths by this definition are also called* simple *paths elsewhere.*

- *An* empty path *is a path that is an empty sequence. Note that empty sequences of edges are distinguished by their beginning vertices.*

- *Vertex v_2 is* reachable *from v_1 in G if and only if there exists a path from v_1 to v_2 in G.*

- *Vertices $v_1, v_2 \in V$ are* connected *in G if and only if there exists a path from one to the other in G; i.e. , if one is reachable from the other.*

- *A vertex $v \in V$ is* on a path *p if and only if p contains either (v, v') or (v', v) for some $v' \in V$ or if p is the empty path from v to itself.*

- *Two paths* share a vertex *if and only if there is some $v \in V$ such that v is on both paths.*

- *Two loops are* identical up to their starting points *if and only if one of the loops is the sequence $(v_1, v_2), (v_2, v_3), \ldots, (v_n, v_1)$ and the other is the sequence $(v_i, v_{i+1}), \ldots, (v_n, v_1), (v_1, v_2), \ldots, (v_{i-1}, v_i)$ for some i.*

- *A* maximal path *in G is a path that is not a proper subsequence of any valid paths in G.*

- *The* maximal paths modulo loops *of G include all maximal paths of G that are not loops as well as the equivalence classes of the loops under the equivalence relation of being identical up to starting points.*

- *The* union *of G with some other $G' = (V', E')$, written $G \cup G'$, is the graph $(V \cup V', E \cup E')$.*

- *A* graph homomorphism *from G to some other $G' = (V', E')$ is a function from V to V' such that if $(v_1, v_2) \in E$ then $(f(v_1), f(v_2)) \in E'$.*

- *A* graph isomorphism *between G and some other $G' = (V', E')$ is a bijective graph homomorphism whose inverse is also a graph homomorphism. Stated*

differently, a graph isomorphism is a bijective function f from V to V' such that $(v_1, v_2) \in E$ if and only if $(f(v_1), f(v_2)) \in E'$.

2.3 Term Rewriting

This section is meant as a brief overview of aspects of Term Rewriting that are relevant to this dissertation. A more in-depth discussion can be found in many standard sources [15, 67, 6]. Term rewriting was originally developed as an approach to automated theorem proving in theories of equality. Consider for example the equality axioms

$$\begin{aligned} \mathsf{plus}\ x\ \mathsf{zero} &= x \\ \mathsf{plus}\ x\ (\mathsf{succ}\ y) &= \mathsf{succ}\ (\mathsf{plus}\ x\ y) \end{aligned}$$

for addition over the natural numbers. It is apparent that equality modulo these axioms can be decided by repeatedly replacing expressions of the form $x + 0$ with x and replacing expressions of the form $x + (\mathsf{succ}\ y)$ with $\mathsf{succ}\ (x + y)$. This yields the system

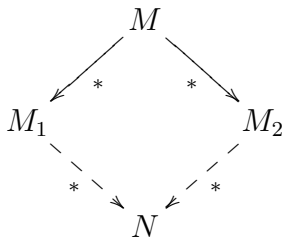
$$\begin{aligned} \mathsf{plus}\ x\ \mathsf{zero} &\rightsquigarrow x \\ \mathsf{plus}\ x\ (\mathsf{succ}\ y) &\rightsquigarrow \mathsf{succ}\ (\mathsf{plus}\ x\ y) \end{aligned}$$

of simplification rules for addition expressions over the natural numbers. We call this system $\mathcal{R}_{\mathsf{plus}}$ below. For example, the above rules would simplify $\mathsf{plus}\ (\mathsf{succ}\ \mathsf{zero})\ \mathsf{zero}$ to $\mathsf{succ}\ \mathsf{zero}$. Such a system is called a *term rewrite system* or TRS.

Term rewrite systems are also useful for defining computation. For example, $\mathcal{R}_{\mathsf{plus}}$ can also be seen as a definition of the function plus. In this view, a term is a computation that is in the process of executing. Rewriting specifies the possible "next steps" or *reductions* of a computation. The *value* of a computation is then found by repeatedly finding a next step, or *reducing*, a term until there are no more rules to apply. $\mathsf{plus}\ (\mathsf{succ}\ \mathsf{zero})\ \mathsf{zero}$ is thus a computation whose value by the above rules is $\mathsf{succ}\ \mathsf{zero}$.

Term rewriting is in general a non-deterministic notion of computation. For example, the term $\mathsf{plus}\ (\mathsf{plus}\ \mathsf{zero}\ (\mathsf{succ}\ \mathsf{zero}))\ \mathsf{zero}$ reduces to the two different terms $\mathsf{plus}\ \mathsf{zero}\ (\mathsf{succ}\ \mathsf{zero})$ and $\mathsf{plus}\ (\mathsf{succ}\ (\mathsf{plus}\ \mathsf{zero}\ \mathsf{zero}))\ \mathsf{zero}$. In this case, however, two

more reductions on each term lead to the common value succ zero. The property that different sequences of reductions starting at the same term can always be brought back together is called *confluence*. Graphically, confluence can be illistrated as



where the existence of the solid lines implies the existence of the dashed lines and the symbols $*$ indicate zero or more steps of reduction. Confluence of a TRS implies that every computation has at most one value. Note that not every TRS is confluent. A simple counterexample is the system

$$c \rightsquigarrow d$$
$$c \rightsquigarrow e$$

In this case, c has two values, d and e.

In order that every computation has at least one value, every term must have some sequence of reductions that terminates. A TRS is said to be *weakly normalizing* if this property holds. In a weakly normalizing system, however, there might still be infinite sequences of reductions, and reductions performed in the wrong order might lead to such an infinite sequence. For example, the system

$$\begin{array}{lcl} c & \rightsquigarrow & f\ c \\ f\ x & \rightsquigarrow & d \end{array}$$

is weakly normalizing, as every term has value d, but the term c has the infinite reduction $c \rightsquigarrow f\ c \rightsquigarrow f\ (f\ c) \rightsquigarrow \ldots$. The stronger property of *strong normalization* or *termination* states that this is not possible, that is, that there are no infinite sequences of reductions. This ensures not only that every computation has a value but also that reductions can be applied in any order and some value will eventually be reached.

When a rewrite system is both terminating and confluent it is called *convergent*. Viewed as a system of computation, this ensures that every term has a value in the given rewrite system. Convergence is also useful for the automated theorem proving aspect of rewriting, as it ensures that equality in the associated equational theory is decidable. This is because any two terms may be reduced in finitely many steps to their unique values, and these values will be identical if and only if the original terms are equal in the theory. For example, $\mathcal{R}_{\mathsf{plus}}$ is convergent, and thus equality in the associated equational theory is decidable.

Unfortunately, the plus operator in this theory is neither associative nor commutative as one would expect. A simple counterexample to commutativity is the term plus zero x. This term is a value, because it matches neither of the left-hand sides in $\mathcal{R}_{\mathsf{plus}}$. Commuting the arguments, however, leads to plus x zero which reduces to the value x. Commutativity is in general problematic, as the rule

$$f\ x\ y \rightsquigarrow f\ y\ x$$

will make any system non-terminating. In some theories, associativity can be handled by orienting it in one way or the other, but this is not always the case. Examples are in many standard references.

The standard solution to this problem is to incorporate associativity and commutativity into the definition of rewriting. The resulting formalism is called AC-rewriting. An associtive-commutative rewrite system or ACRS is a set of rules along with a set of binary operations to be considered associative and commutative. Terms are then considered equal up to associativity and commutativity of the given operators, and a term reduces if and only if it is equal up to associativity and commutativity to a left-hand side in the given ACRS. For example, if $\mathcal{R}_{\mathsf{plus}}$ is considered as an ACRS, where plus is stipulated as associative and commutative, then plus x zero reduces to x.

The remainder of this section is organized as follows. First, Section 2.3.1 defines standard rewriting. Next, Section 2.3.2 defines Associative-Commutative Rewriting. Finally, Section 2.3.3 concludes with some standard results on proving confluence. Standard results on termination are not addressed here as they will not be used in this document.

2.3.1 Term Rewrite Systems

In this section, term rewrite systems are defined. This is done by first defining the term language on which rewriting operates, and then term rewriting systems themselves are defined. To define the terms, we assume two disjoint sets $\mathcal{C}$ and $\mathcal{V}$ of symbols are given, along with a function mapping each element of $\mathcal{C}$ to a natural number. The elements of $\mathcal{C}$ are called the *constructors* and those of $\mathcal{V}$ are called the *variables* below. while number associated with an element of $\mathcal{C}$ is called its *arity*. A constructor with arity 0 is called nullary, one with arity 1 is called unary, and one with arity 2 is called binary. c, d, e, and f are used for constructors and x, y, and z are used for variables below, all possibly with subscripts. The terms are then defined inductively as follows:

- $x \in \mathcal{V}$ is a term; and
- If $M_1, \ldots, M_n$ are all terms then so is $c\, M_1 \ldots M_n$, where n is the arity of c.

M, N, l, and r, possibly with subscripts, are used for terms below, where l and r are reserved for the left- and right-hand sides of rewrite rules. The notation $c\, \vec{M}$ is sometimes used below for $c\, M_1 \ldots M_n$. A term N is a *subterm* of M if and only if M contains N. N is in addition said to be a *strict subterm* if $M \neq N$. The set of *free variables* of M, written $\mathrm{FV}(M)$, is then the set of all variables that occur as subterms of M.

Another useful concept is the substitutions. A *substitution* is a mapping from variables to terms that is the identity for all but finitely many variables. σ is used for substitutions below. The *domain* of a substitution σ, written $\mathrm{Dom}(\sigma)$, is the set of all x such that $\sigma(x) \neq x$. A substitution can also be written out as $[M_1/x_1, \ldots, M_n/x_n]$ where the $\{x_1, \ldots, x_n\}$ is the domain of the substitution and M_i is the value of the substitution on x_i. Substitutions can also be extended to terms by setting $\sigma(c\, M_1 \ldots M_n) = c\, \sigma(M_1) \ldots \sigma(M_n)$.

A final notion that will be necessary is the term contexts. A *term context* is a term with exactly one occurrence of the special symbol $\_$. Term contexts are written C below. Intuitively, a term context represents a term with a hole. This hole can be

filled by replacing it with a term M. This is written $C[M]$. It is straightforward to see that $M = C[N]$ for some C if and only if N is a subterm of M.

Given these definitions, a term rewrite system is a set of pairs of terms $(l \rightsquigarrow r)$ such that l is not a variable and $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$. The first condition is so that a rule does not apply to every term, while the second is because it is not clear for example what term to use for y in the rule $c\ x \rightsquigarrow d\ x\ y$. Rewrite systems are written $\mathcal{R}$ and $\mathcal{S}$ below. A rewrite system $\mathcal{R}$ induces a relation, written $\rightsquigarrow_{\mathcal{R}}$, called the *one-step $\mathcal{R}$-reduction*. $\rightsquigarrow_{\mathcal{R}}$ is defined to be the set of all pairs $C[\sigma l] \rightsquigarrow C[\sigma r]$ for C a term context, σ a substitution, and $(l \rightsquigarrow r) \in \mathcal{R}$. The symbol $\mathcal{R}$ itself is sometimes also used for this relation. A term of the form σl is called an *$\mathcal{R}$-redex*, or just a redex if $\mathcal{R}$ is clear from context. A term M is said to *$\mathcal{R}$-reduce* to N, or just reduce, if $M \rightsquigarrow_{\mathcal{R}}^{*} N$, recalling that $\rightsquigarrow_{\mathcal{R}}^{*}$ is the reflexive-transitive closure of $\rightsquigarrow_{\mathcal{R}}$. In this case, N is said to be an *$\mathcal{R}$-reduct* of M. Two terms are then said to be *$\mathcal{R}$-joinable* if and only if they share a common $\mathcal{R}$-reduct. Finally, two terms are said to be *$\mathcal{R}$-equal*, written $\leftrightsquigarrow_{\mathcal{R}}^{*}$ or $=_{\mathcal{R}}$, if and only if the two terms are related by the reflexive-symmetric-transitive closure of $\rightsquigarrow_{\mathcal{R}}$.

2.3.2 Associative-Commutative Rewrite Systems

An associative-commutative rewrite system, or ACRS, is a TRS along with a set of binary constructors to be considered associative and commutative. These are called the *AC constructors* below. Equality up to associativity and commutativity of these operators, written $=_{\mathrm{AC}}$, is defined as the reflexive-symmetric-transitive closure of the one-step rewrite relation of the system containing the two rules

$$\begin{array}{lcl} f\ x\ y & \rightsquigarrow & f\ y\ x \\ f\ (f\ x\ y)\ z & \rightsquigarrow & f\ x\ (f\ y\ z) \end{array}$$

for every AC constructor f.

AC-Rewriting is defined by the one-step AC-reduction relation $\rightsquigarrow_{\mathcal{R}/\mathrm{AC}}$. Given terms M and N, $M \rightsquigarrow_{\mathcal{R}/\mathrm{AC}} N$ holds if and only if

$$M =_{\mathrm{AC}} C[\sigma l] \wedge C[\sigma r] =_{\mathrm{AC}} N$$

for some rule $(l \rightsquigarrow r) \in \mathcal{R}$, some term context C, and some substitution σ. Note that this is essentially rewriting on $=_{\mathrm{AC}}$-equivalence classes of terms. The $\mathcal{R}$-redexes here are the terms $M =_{\mathrm{AC}} \sigma l$ for some substitution σ and left-hand side l. Note however that a term can reduce without having an $\mathcal{R}$-redex, since it might just be $=_{\mathrm{AC}}$-equivalent to a term with an $\mathcal{R}$-redex. The notion that M $\mathcal{R}$-reduces to N when $\mathcal{R}$ is an ACRS is modified to include $=_{\mathrm{AC}}$, so that this phrase means that either $M \rightsquigarrow^*_{\mathcal{R}} N$ or $M =_{\mathrm{AC}} N$. N is still an $\mathcal{R}$-reduct of M whenever M $\mathcal{R}$-reduces to N, but this means here that N could also be equal under $=_{\mathrm{AC}}$ to M. Similarly, $\mathcal{R}$-joinability is considered modulo $=_{\mathrm{AC}}$, and $\mathcal{R}$-equality, written $=_{\mathcal{R}}$, the reflexive-symmetric-transitive closure of the union $\rightsquigarrow_{\mathcal{R}/\mathrm{AC}} \cup =_{\mathrm{AC}}$, which is easily seen to be equivalent to $\leftsquigarrow\rightsquigarrow^*_{\mathcal{R}/\mathrm{AC}} \cup =_{\mathrm{AC}}$.

2.3.3 Confluence Results

Confluence is an important result for programming languages. Confluence ensures that a computation does not have different values depending on how that computation is executed. In many forms of type theory, like those introduced in Chapters 5 and 7, this is crucial to proving many of the meta-theoretic properties of the language. Termination of such systems is generally proved by special techniques, such as logical relations [25, 12], as standard techniques from rewriting do not apply. Confluence of such sytems, in contrast, is generally proved by standard rewriting techniques. The remainder of this section proceeds gives some standard confluence results. Results of abstract reduction systems, which include any binary relations, are given first, followed by the Critical Pairs Lemma for both standard and associative-commutative rewriting. Proofs are given where straightforward, and otherwise the reader is referred to standard references [15, 67, 6].

Confluence of Abstract Reduction Systems

A number of useful properties of rewrite systems can be shown without any reference to the definition of rewriting. Stated differently, these results hold for any binary relation or relations on a given set, a class which happens to include rewrite systems. Binary relations here are thus called abstract reduction systems. The word "term"

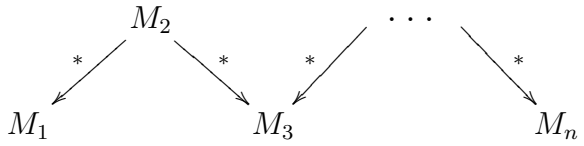
here thus refers to any element of the domain or range of the given binary relation or relations.

The first result relates to the well-known Church-Rosser property of a relation $\mathcal{R}$. This property states that equality under the reflexive-symmetric-transitive closure $\leftrightsquigarrow^*_{\mathcal{R}}$ is equivalent to joinability. This is why confluence is important: in a confluent system, equality of two terms can be checked by just running the system forward to see if the terms are joinable.

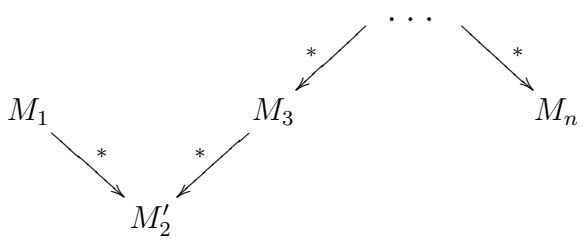
Definition 2.3.1 (Church-Rosser). *A relation $\mathcal{R}$ is* Church-Rosser *if and only if, for any M_1 and M_2, $M_1 \leftrightsquigarrow^*_{\mathcal{R}} M_2$ implies there exists an N such that $M_i \rightsquigarrow^*_{\mathcal{R}} N$ for $i \in \{1, 2\}$.*

Theorem 2.3.1. *A relation $\rightsquigarrow_{\mathcal{R}}$ is Church-Rosser if and only if it is confluent.*

Proof. The "only if" direction is immediate. The "if" direction can be shown graphically. If $M_1 \leftrightsquigarrow^*_{\mathcal{R}} M_n$ then there must exist M_2 through M_{n-1} related to M_1 and M_n as in the following figure:



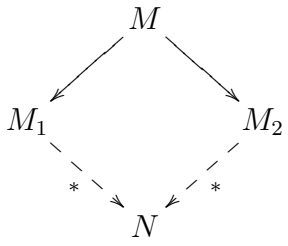
By confluence, this implies that the first "peak" may be turned into a "valley" as:



By induction on the number of peaks, this process may be iterated until there are no peaks, implying that M_1 and M_n are joinable. □

The simplest way to prove confluence is through strong confluence. Strong confluence states that if a term reduces in one step to two different terms then those terms are

joinable in a single step. Graphically, this can be depicted as



where the existence of the solid lines implies that of the dashed lines.

Definition 2.3.2. *The relation $\mathcal{R}$ is called* strongly confluent *if and only if $\mathcal{R}^{-1}\mathcal{R} \subseteq \mathcal{R}^{=}(\mathcal{R}^{=})^{-1}$.*

Lemma 2.3.1. *If $\mathcal{R}$ is strongly confluent then it is confluent.*

Proof. We first show that if $M \rightsquigarrow^{=}_{\mathcal{R}} M_1$ and $M \rightsquigarrow^{*}_{\mathcal{R}} M_2$ then $M_1 \rightsquigarrow^{*}_{\mathcal{R}} N$ and $M_2 \rightsquigarrow^{=}_{\mathcal{R}} N$ by induction on the number of steps in $M \rightsquigarrow^{*}_{\mathcal{R}} N$. The result is immediate if this number is 0, as then $M_2 = M$. The result is also immediate if $M = M_1$. Otherwise we have the picture $\begin{array}{ccccc} M & \longrightarrow & M_2' & \xrightarrow{*} & M_2 \\ \downarrow & & \vdots & & \\ M_1 & & N' & \xrightarrow{*} & N \end{array}$ where the first square is by the strong confluence of $\mathcal{R}$ and the second is by the induction hypothesis. A similar proof then extends this fact to full confluence of $\mathcal{R}$. □

It is sometimes useful to prove confluence of a relation $\mathcal{R}$ by proving confluence of some $\mathcal{R}'$ such that $\mathcal{R} \subseteq \mathcal{R}' \subseteq \mathcal{R}^{*}$. This is shown valid by the following lemma:

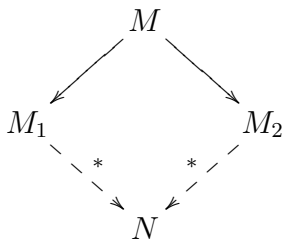
Lemma 2.3.2. *For any relations $\mathcal{R}$ and $\mathcal{R}'$ such that $\mathcal{R} \subseteq \mathcal{R}' \subseteq \mathcal{R}^{*}$, if $\mathcal{R}'$ is confluent then so is $\mathcal{R}$.*

Proof. If $M \rightsquigarrow^{*}_{\mathcal{R}} M_1$ and $M \rightsquigarrow^{*}_{\mathcal{R}} M_2$, then $M \rightsquigarrow^{*}_{\mathcal{R}'} M_i$ by the assumed subset relation. By confluence of $\mathcal{R}'$ it follows that $M_i \rightsquigarrow^{*}_{\mathcal{R}'} N$ for some N, and so $M_i \rightsquigarrow^{*}_{\mathcal{R}} N$, also by the assumed subset relation. □

It is also sometimes useful to prove confluence by proving the following *local* confluence property:

Definition 2.3.3 (Local Confluence). *A relation $\mathcal{R}$ is* locally confluent *if $M \rightsquigarrow_{\mathcal{R}} M_1$ and $M \rightsquigarrow_{\mathcal{R}} M_2$ implies there exists some N such that $M_i \rightsquigarrow^*_{\mathcal{R}} N$ for $i \in \{1, 2\}$.*

This can be displayed graphically as

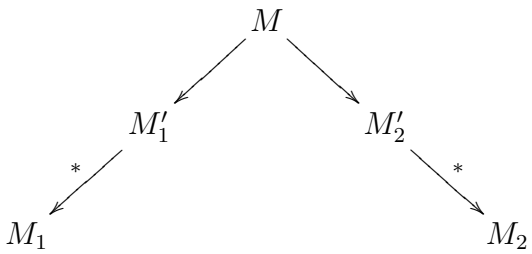


where the existence of the solid lines implies the existence of the dashed lines. Local confluence implies confluence if the relation is terminating, as shown by the following lemma. To see that termination is necessary, the relation given graphically as

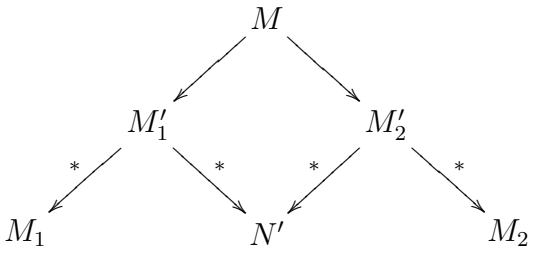
is a standard counterexample of a non-terminating relation that is locally confluent but not confluent.

Lemma 2.3.3 (Newman's Lemma). *If $\mathcal{R}$ is locally confluent and terminating then it is confluent.*

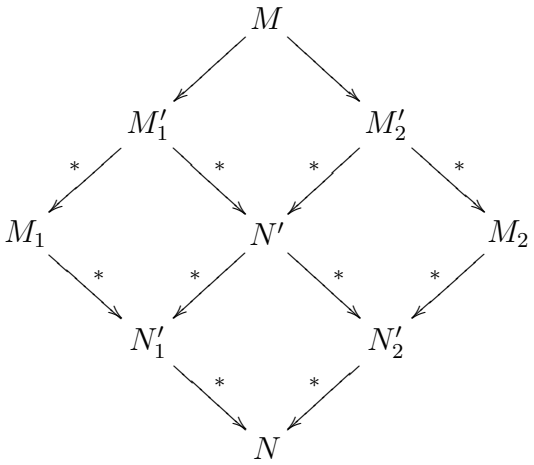
Proof. Let $M \rightsquigarrow^*_{\mathcal{R}} M_1$ and $M \rightsquigarrow^*_{\mathcal{R}} M_2$. We now need to show that there is some N such that $M_i \rightsquigarrow^*_{\mathcal{R}} N$ for $i \in \{1, 2\}$. We prove this by induction on the maximum number of $\mathcal{R}$-steps that can be taken from M, or, more precisely, the maximum number of terms M', M'', M''', etc. such that $M \rightsquigarrow_{\mathcal{R}} M' \rightsquigarrow_{\mathcal{R}} M'' \rightsquigarrow_{\mathcal{R}} M''' \rightsquigarrow_{\mathcal{R}} \ldots$. This is well-defined and finite as $\mathcal{R}$ is terminating. If M is equal to either M_1 or M_2, then the result is immediate. Otherwise the situation can be depicted graphically as



for some M_1' and M_2'. Local confluence then yields



for some N'. The number of $\mathcal{R}$-steps that can be taken from M_i' must be less than the number of $\mathcal{R}$-steps that can be taken from M, as the $\mathcal{R}$-reduction step from M to M_i' can be prepended to any sequence of $\mathcal{R}$-reductions starting at M_i'. Thus by the induction hypothesis we have



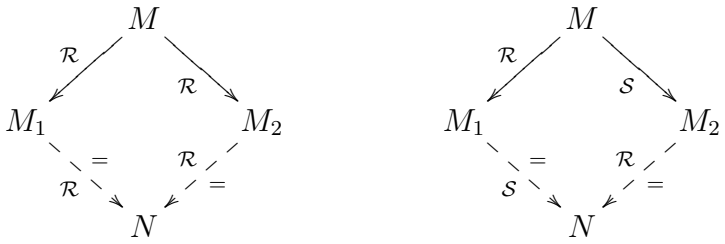
for some N_1', N_2', and N, where the existence of N follows by a third application of the induction hypothesis to N'. □

We now turn briefly to *modularity* of confluence, the property that the confluence of two relations $\mathcal{R}$ and $\mathcal{S}$ implies the confluence of their union. A standard modularity of confluence result is the Hindley-Rosen Lemma, which states that the union of two strongly confluent is confluent if the relations commute. Recall that $\mathcal{R}^=$ is the reflexive closure of $\mathcal{R}$.

Definition 2.3.4 (Strongly Confluent). *A relation $\mathcal{R}$ is* strongly confluent *if and only if $\mathcal{R}^{-1}\mathcal{R} \subseteq \mathcal{R}^=(\mathcal{R}^=)^{-1}$.*

Definition 2.3.5 (Commutativity of Relations). *The relations $\mathcal{R}$ and $\mathcal{S}$ are said to* commute *if and only if $\mathcal{R}^{-1}\mathcal{S} \subseteq \mathcal{S}^{=}(\mathcal{R}^{=})^{-1}$.*

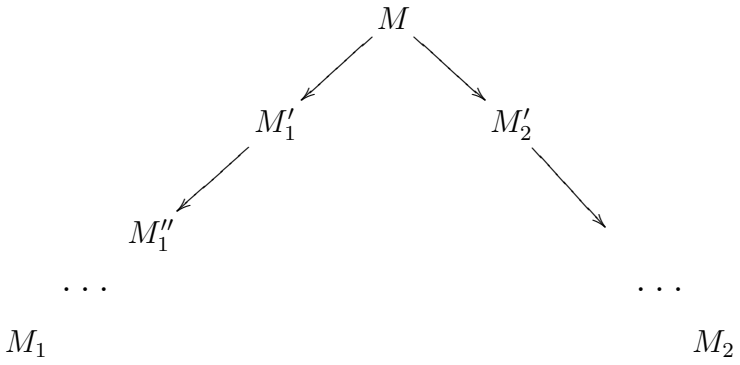
These two notions can be conveyed graphically as



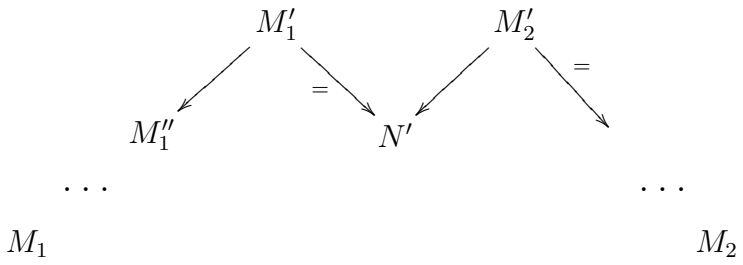
where the existence of the solid lines implies the existence of the dashed lines. The first diagram describes strong confluence of $\mathcal{R}$ while the second describes commutativity of $\mathcal{R}$ and $\mathcal{S}$.

Lemma 2.3.4 (Hindly-Rosen Lemma). *If $\mathcal{R}$ and $\mathcal{S}$ commute and are both strongly confluent then their union is confluent.*

Proof. Assume $M \rightsquigarrow_{\mathcal{R}\cup\mathcal{S}} M_1$ and $M \rightsquigarrow_{\mathcal{R}\cup\mathcal{S}} M_2$. These reductions are equivalent to a sequence of individual steps of $\mathcal{R}$ and $\mathcal{S}$. Thus we have a peak



where each line is a reduction of either $\mathcal{R}$ or $\mathcal{S}$. Any peak may be "pushed down" to form a valley as in the proof of Newman's Lemma:



Because the "sides" of the new valley have length at most one, induction is not needed on the number of steps to a normal form. Thus neither of the relations $\mathcal{R}$ or $\mathcal{S}$ need be terminating. Instead, induction may proceed on the number of steps in $M \rightsquigarrow_{\mathcal{R}\cup\mathcal{S}} M_1$ and $M \rightsquigarrow_{\mathcal{R}\cup\mathcal{S}} M_2$, and the process of pushing down peaks may be repeated until an N is found such that $M_i \rightsquigarrow_{\mathcal{R}\cup\mathcal{S}} N$. □

Note that if $\mathcal{R}$ is confluent then $\mathcal{R}^*$ is strongly confluent for any relation $\mathcal{R}$. Thus a direct corollary of the Hindley-Rosen Lemma is that if $\mathcal{R}^*$ and $\mathcal{S}^*$ commute then $\mathcal{R}$ and $\mathcal{S}$ satisfy modularity of confluence.

The Critical Pairs Lemma

We now consider a confluence result, the Critical Pairs Lemma, that relies on more specific properties of rewrite systems. The Critical Pairs Lemma shows how local confluence may be proved by checking joinability of only finitely many terms. Coupled with Newman's Lemma this can then be used to prove confluence of terminating systems. The key idea of the Critical Pairs Lemma is that the only difficulty in proving local confluence is when two rules overlap in a non-trivial way. In this case, reducing one redex of one rule could cause a redex for the other rule to disappear. In all other cases, local confluence is straightforward. After the Critical Pairs Lemma is introduced, it is generalized to include ACRSs.

We turn first to the standard Critical Pairs Lemma without associativity or commutativity. The first notion needed is that of overlap:

Definition 2.3.6 (Overlap). *A term M is said to* overlap *a term N if and only if there exists some substitution σ and some subterm N' of N such that $\sigma M = \sigma N$. M is said to* non-trivially overlap *N if and only if M is not equal to N and the given N' is not a variable.*

When $\sigma M = \sigma N$ σ is said to be a *unifier* of M and N. If such a σ exists, M and N are said to *unify*. In general, if two terms unify then they have infinitely many unifiers. Consider the terms $c\ x$ and $c\ (d\ y)$ for some constructors c, d, and e. Any substitution that makes $\sigma x = d\ (\sigma y)$ is a unifier of these terms. There are infinitely many such substitutions. What is needed is the idea of most general unifier.

Definition 2.3.7 (Most General Unifier). *A substitution σ is said to be* more general *than a substitution σ' if and only if there exists some substitution σ_1 such that $\sigma' x = \sigma_1(\sigma x)$ for all x. Two substitutions are called* incomparably general *if and only if neither is more general than the other. A* most general unifier *of M and N is then a unifier of M and N that is more general than any other unifier of M and N.*

It is a standard result that any two terms that unify in fact have a most general unifier [7]. The concepts of non-trivial overlap and most general unifier can then be used to define the critical pairs. A non-trivial overlap of the left-hand sides of two rules yields some term that can be rewritten two different ways by the two rules. A critical pair is the (most general) pair of terms resulting from this rewriting:

Definition 2.3.8 (Critical Pair). *Let $(l_1 \rightsquigarrow r_1)$ and $(l_2 \rightsquigarrow r_2)$ be two (not necessarily distinct) rules in $\mathcal{R}$ whose variables are renamed to be distinct. If $l_1 \equiv C[M]$ for some term context C and non-variable term M, and σ is a most general unifier of M and l_2, then the pair $(\sigma r_1, C[\sigma r_2])$ is called a* critical pair *of the rules $(l_1 \rightsquigarrow r_1)$ and $(l_2 \rightsquigarrow r_2)$. If the two rules are the same then C is required to not be the trivial term context $\_$.*

Lemma 2.3.5 (Critical Pairs Lemma). *If $\mathcal{R}$ is terminating and all of its critical pairs are joinable then it is locally confluent and thus confluent.*

Proof. The proof goes as described at the beginning of this section. The details are quite involved, so the reader is referred to any standard reference. See Baader and Nipkow [6] for an especially readable development. □

For AC-rewriting, the situation is more complicated. First, two unifiable terms need not have a most general unifier. For example, if f is an AC constructor and c and d are two other constructors then $f\ x\ y$ and $f\ c\ d$ has the two incomparable unifiers $[c/x, d/y]$ and $[d/x, c/y]$. It is however a standard result that any terms that unify with respect to $=_{\mathrm{AC}}$ will always have a finite set of unifiers that cover all the necessary cases [7], where this notion can be defined precisely as follows:

Definition 2.3.9 (Minimal Complete Set of AC-Unifiers). *A* complete set of AC-unifiers *for M and N is a set of unifiers of M and N with respect to $=_{\mathrm{AC}}$ such that for any unifier σ' of M and N with respect to $=_{\mathrm{AC}}$ there is some more general σ in the given set. A* minimal complete set of AC-unifiers *is a complete set of AC-unifiers such that every pair of elements in the set is incomparably general.*

In this terminology, any two terms that unify with respect to $=_{\mathrm{AC}}$ will have a finite minimal complete set of AC-unifiers. Note that there are infinitely many minimal complete sets of AC-unifiers for a particular pair of terms, but since each set is complete it suffices to consider just one minimal complete set of AC-unifiers for every unifiable pair of terms. This leads to the following adapted definition of critical pairs:

Definition 2.3.10 (AC Critical Pairs). *Let $(l_1 \rightsquigarrow r_1)$ and $(l_2 \rightsquigarrow r_2)$ be two (not necessarily distinct) rules in $\mathcal{R}$ whose variables are renamed to be distinct. If $l_1 =_{\mathrm{AC}} C[M]$ for some term context C and non-variable term M, and σ is in a minimal complete set of AC-unifiers for M and l_2, then the pair $(\sigma r_1, C[\sigma r_2])$ is an* AC critical pair *of the rules $(l_1 \rightsquigarrow r_1)$ and $(l_2 \rightsquigarrow r_2)$.*

Because of the swapping performed by $=_{\mathrm{AC}}$, two AC rewrite rules can still interfere with each other even if they have no critical pairs. For example, if f is an AC constructor then the system

$$\begin{aligned} f\ c_1\ c_2 &\rightsquigarrow d \\ f\ c_2\ c_3 &\rightsquigarrow e \end{aligned}$$

is not confluent, even though it is obviously terminating and has no critical pairs. This is because $f\ c_1\ (f\ c_2\ c_3)$ can reduce both to $f\ c_1\ e$ and to $f\ d\ c_3$. Peterson and Stickel [51] showed, however, that it is enough to consider all AC critical pairs in an extended system got by adding the rule

$$f\ l\ x \rightsquigarrow f\ r\ x$$

for every rule $(l \rightsquigarrow r)$ in the original system where l is of the form $f\ M\ N$ and f is an AC constructor. Note that the original rules are retained. The above example does have a critical pair in this extended system, since the left-hand side $f\ c_1\ c_2$ has a non-trivial overlap (modulo $=_{\mathrm{AC}}$) with the new rule $f\ (f\ c_2\ c_3)\ x$. The resulting unifier is $[c_1/x]$, leading to the critical pair $(f\ e\ c_1, f\ d\ c_3)$, which is exactly the pair that could not be joined above. Peterson and Stickel proved the following result for AC critical pairs:

Theorem 2.3.2. *Let $\mathcal{R}$ be an AC-rewrite system such that, for every rule of the form $(f\ M\ N \rightsquigarrow r)$ in $\mathcal{R}$, $\mathcal{R}$ also contains a rule of the form $(f\ (f\ M\ N)\ x \rightsquigarrow f\ r\ x)$ for some fresh variable x not occurring in M, N, and r. If all AC critical pairs of $\mathcal{R}$ are joinable, then $\mathcal{R}$ is locally confluent.*

It follows that if $\mathcal{R}$ is extended to contain the above new rules, and all the AC critical pairs in the resulting system are joinable, then $\mathcal{R}$ is locally confluent. This is because any reduction performed by one of the new extended rules could also be performed by the original rule, so reduction in the extended system is identical to reduction in $\mathcal{R}$.

Chapter 3

A Brief Introduction to Intensional Constructive Type Theory

Intensional Constructive Type Theory, or ICTT, is a logical theory that is also a programming language. This allows the user to write programs and prove properties of them in the same language. Originally introduced by Per Martin-Löf [40], ICTT is a foundational theory of mathematics based on the ideas of Type Theory that came from Russell. It can thus be compared to Zermelo-Frankel Set Theory, a more common foundational theory of mathematics.

ICTT is a syntactic theory organized, as its name suggests, around a combination of the philosophy of Constructivism and the notion of intensionality. Constructivism is a broad topic, but the basic tenet is that the only mathematical objects whose existence can be accepted are those that can be constructed in some concrete fashion by the mathematician. Thus an object cannot be proved to exist by *reductio ad absurdum*, as such a proof only shows that an object cannot fail to exist and does not show how the object may be constructed. ICTT is constructive, meaning it adheres to Constructivism, in the sense that the only objects that exist in the theory are those that can be constructed syntactically. This is in contrast to ZF Set Theory, in which the set of real numbers has uncountably many elements that cannot be written down, as the language of ZF Set Theory is countable.

ICTT is also *intensional*, meaning equality of objects is defined as a purely syntactic notion. In contrast, an *extensional* theory makes object equality a semantic notion, based on properties defined in the theory itself. Thus, for example, Zermelo-Frankel set theory is an extensional theory, because two sets are equal in ZF set theory if and

only if they have the same elements, which is a semantic notion. Equality in ICTT is instead defined by the syntactic transformations of a rewrite system (see Section 2.3). This defines a notion of computation: an element of ICTT computes, or evaluates, to another term through the reductions of the given rewrite system. Thus, for example, the term for $2 + 2$ reduces to the term for 4. The rewrite system for ICTT is in fact convergent, meaning all terms have a canonical form, which can be computed. This is an important property of ICTT, as it is how the theory is proved consistent. It also means that all computations are guaranteed to terminate. Equality in ICTT can thus be decided, by checking if two terms have the same canonical form.

The remainder of this chapter is organized as follows. Section 3.1 discusses Constructivism and introduces the Curry-Howard Isomorphism, a central concept in ICTT. Section 3.2 then gives an informal introduction to the Calculus of Inductive Constructions, a particular version of ICTT, by example.

3.1 Constructivism and the Curry-Howard Isomorphism

This section gives a brief overview of Constructivism and the Curry-Howard Isomorphism. For more details, see [70, 30]. Constructivism is the philosophical stance that mathematical objects can only be considered to exist if the mathematician demonstrates how they can be *constructed*, or built up from some starting objects via some allowed operations. For example, the set of natural numbers might be an allowed starting object, assumed to exist *a priori*, and the allowed operations might include taking the union of two sets or taking the image of a set under some partial function. Constructivism rejects existence proofs that use *reductio ad absurdum*. The reasoning is that such proofs only posit that an object cannot *not* exist, and do not give any means to construct the object.

Rejecting *reductio ad absurdum* existence proofs also requires the rejection of the Law of the Excluded Middle. The Law of the Excluded Middle states that, for any formula ϕ, the formula $\phi \vee \neg\phi$ is true. Stated differently, the Law of the Excluded Middle means that every formula is either true or false. It is well-known that the Law of the Excluded

Middle is equivalent (modulo the standard rules for disjunction and implication) to the Law of Double Negation, written $\neg\neg\phi \rightarrow \phi$. Substituting $\exists x.\psi(x)$ for ϕ, the Law of Double Negation implies the validity of *reductio ad absurdum* existence proofs, and hence cannot be accepted by Constructivism. Thus, under Constructivism, a proposition is not necessarily either true or false.

One approach to understanding Constructivism is in terms of the Brouwer-Heyting-Kolmogorov interpretation, or BHK-interpretation, of first-order logic. This interpretation gives a Constructivist meaning to the first-order logical connectives by defining what constitutes a proof of each of these connectives. The BHK-interpretation can be summarized as follows:

- A proof of $\phi \wedge \psi$ is a pair of a proof of ϕ and a proof of ψ.
- A proof of $\phi \vee \psi$ contains either a proof of ϕ or a proof of ψ.
- A proof of $\phi \rightarrow \psi$ is a function that creates proofs of ψ from proofs of ϕ.
- A proof of $\exists x.\phi(x)$ is a pair of an individual i and a proof of $\phi(i)$.
- A proof of $\forall x.\phi(x)$ is a function that creates a proof of $\phi(x)$ from any x.

Negations $\neg\phi$ are read as implications $\phi \rightarrow \bot$, where $\bot$ is the absurd proposition that has no proof. The BHK-interpretation yields a form of what is called *intuitionistic* first-order logic. (The philosophies of Intuitionism and Constructivism are closely tied; see [70].) Intuitionistic first-order logic is known to be strictly weaker than standard, or *classical* first-order logic. Specifically, as discussed in the previous paragraph, $\exists x.\phi(x)$ can only be proved by giving (constructing) a particular i and proving $\phi(i)$. In addition, $\phi \vee \neg\phi$ cannot be proved without giving either a proof of ϕ or one of $\neg\phi$, so if neither of these is provable then the disjunction is also not provable.

The reader will note that the BHK-interpretation is not actually fully defined. The one piece that was left implicit was the stipulation of what constitutes a function. This is intended: the BHK-interpretation is parameterized by the class of allowed functions. Different classes of functions will give different logics. The class of functions, however, should include only *total* functions. Otherwise every implication $\phi \rightarrow \psi$ would be provable using the (partial) function that is undefined on all inputs. In

most uses (including CTT), the functions are also required to be recursive. This makes ideological sense in Constructivism, because a mathematician cannot truly be said to construct the result of a non-recursive function. Thus, as a side note, in order for the class of functions itself to be decidable it must not contain all recursive functions, since this set is known to not even be recursively enumerable.

The BHK-interpretation has a similar flavor to algebraic datatype definitions. This similarity is made formal with the Curry-Howard Isomorphism, also known as the Propositions-as-Types principle. Under the Curry-Howard Isomorphism, propositions are identified with the sets of their proofs. Provability of a proposition then becomes non-emptiness of its set. Moving to programming languages, these sets can be viewed as datatypes, where provability is non-emptiness of the datatype. This follows from the following re-statement of the BHK-interpretation:

- $\phi \wedge \psi$ is the type of pairs whose first element has type ϕ and whose second element has type ψ;
- $\phi \vee \psi$ is the type of variants, or disjoint sets, which contain either an element of type ϕ or one of type ψ;
- $\phi \rightarrow \psi$ is the type of functions from type ϕ to type ψ;
- $\exists x.\phi(x)$ is the type of pairs whose first element i has some type A, reflecting the domain of quantification, and whose second element has type $\phi(i)$; and
- $\forall x.\phi(x)$ is the type of functions from argument i in the domain A of quantification to type $\phi(i)$.

Note that the last two clauses require *dependent types*, or types that can contain data. For example, the proposition that all natural numbers are either odd or even might be interpreted by the formula $\forall x.(\mathsf{isodd}\ x) \vee (\mathsf{iseven}\ x)$. Viewed as a datatype this type include the functions whose input is a natural number x and whose output is a variant type containing either an element of the datatype $\mathsf{isodd}\ x$ or an element of the datatype $\mathsf{iseven}\ x$. Note again that this re-interpretation still requires functions to be total, since, for example, a proof that all natural numbers are odd or even must return a proof for all inputs x.

3.2 Informal Calculus of Inductive Constructions

In this Section a particular version of ICTT, called the Calculus of Inductive Constructions or CIC, is introduced by example. For a more in-depth treatment of ICTT in various forms, see [49, 3]. For an implementation of CIC, see the Coq system [68].

The data of CIC are defined by the user with *constructors*. A constructor is an atomic, named piece of syntax in CIC. These allow the user to define standard algebraic datatypes, such as natural numbers and lists, as well as properties of data, such as the concept that a number is less than another number. Types defined by constructors are in general called *inductive types*. CIC also allows the user to write functions over these data. Functions on algebraic datatypes define operations on data, whereas functions on properties of data can be viewed as proofs of implications, as discussed above. Syntactic restrictions ensure that pattern-matching functions are total.

The remainder of this section proceeds as follows. Section 3.2.1 gives some example inductive types, and discusses how these can encode both data and proofs. Section 3.2.2 then describes some example functions over those inductive types, including examples of both operations and proofs. Finally, Section 3.2.3 briefly introduces the type universes alluded to in the previous sections.

3.2.1 Example Datatypes

Figure 3.1 gives some example constructor declarations in CIC. In an implementation of ICTT, such as Coq, these declarations could be entered by the user to define the given constructors. Each line of the figure declares a constructor by putting it to the left of a colon. For example, the first line declares the constructor nat. This declaration adds the symbol nat as a term, or syntactic element, of the theory. Every term must have a type, so when nat is declared it must be also given a type. This is done by putting a type, or term that is itself a term of type Type_i for some i, to the right of the colon. In this case, nat is given the type Type_0, classifying nat itself as a type. Such a constructor is called a *type constructor*. The terms Type_i for each i are called the *type universes*, and are discussed below in Section 3.2.3. For the present

$$
\begin{array}{lcl}
\mathsf{nat} & : & \mathsf{Type}_0 \\
\mathsf{zero} & : & \mathsf{nat} \\
\mathsf{succ} & : & \mathsf{nat} \Rightarrow \mathsf{nat} \\
 & & \\
\mathsf{list} & : & \mathsf{Type}_0 \Rightarrow \mathsf{Type}_0 \\
\mathsf{nil} & : & \Pi A : \mathsf{Type}_0 \,.\, \mathsf{list}\ A \\
\mathsf{cons} & : & \Pi A : \mathsf{Type}_0 \,.\, A \Rightarrow \mathsf{list}\ A \Rightarrow \mathsf{list}\ A \\
 & & \\
\mathsf{True} & : & \mathsf{Type}_0 \\
\mathsf{true\text{-}i} & : & \mathsf{True} \\
\mathsf{False} & : & \mathsf{Type}_0 \\
 & & \\
\mathsf{eq} & : & \Pi A : \mathsf{Type}_0 \,.\, A \Rightarrow A \Rightarrow \mathsf{Type}_0 \\
\mathsf{eq\text{-}refl} & : & \Pi A : \mathsf{Type}_0 \,.\, \Pi x : A \,.\, \mathsf{eq}\ x\ x \\
 & & \\
\mathsf{le} & : & \mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{Type}_0 \\
\mathsf{le\text{-}refl} & : & \Pi x : \mathsf{nat} \,.\, \mathsf{le}\ x\ x \\
\mathsf{le\text{-}succ} & : & \Pi x : \mathsf{nat} \,.\, \Pi y : \mathsf{nat} \,.\, \mathsf{le}\ x\ y \Rightarrow \mathsf{le}\ x\ (\mathsf{succ}\ y) \\
 & & \\
\mathsf{is\text{-}sorted} & : & \Pi A : \mathsf{Type}_0 \,.\, \Pi R : (A \Rightarrow A \Rightarrow \mathsf{Type}_0) \,.\, \mathsf{list}\ A \Rightarrow \mathsf{Type}_0 \\
\mathsf{is\text{-}sorted\text{-}nil} & : & \Pi A : \mathsf{Type}_0 \,.\, \Pi R : (A \Rightarrow A \Rightarrow \mathsf{Type}_0) \,.\, \mathsf{is\text{-}sorted}\ A\ R\ (\mathsf{nil}\ A) \\
\mathsf{is\text{-}sorted\text{-}one} & : & \Pi A : \mathsf{Type}_0 \,.\, \Pi R : (A \Rightarrow A \Rightarrow \mathsf{Type}_0) \,. \\
 & & \quad \Pi x : A \,.\, \mathsf{is\text{-}sorted}\ A\ R\ (\mathsf{cons}\ A\ x\ (\mathsf{nil}\ A)) \\
\mathsf{is\text{-}sorted\text{-}many} & : & \Pi A : \mathsf{Type}_0 \,.\, \Pi R : (A \Rightarrow A \Rightarrow \mathsf{Type}_0) \,.\, \Pi x_1 : A \,.\, \Pi x_2 : A \,.\, \Pi l : \mathsf{list}\ A \,. \\
 & & \quad (R\ x_1\ x_2) \Rightarrow \mathsf{is\text{-}sorted}\ A\ R\ (\mathsf{cons}\ A\ x_2\ l) \Rightarrow \\
 & & \quad\quad \mathsf{is\text{-}sorted}\ A\ R\ (\mathsf{cons}\ A\ x_1\ (\mathsf{cons}\ A\ x_2\ l))
\end{array}
$$

Figure 3.1: Example Constructor Declarations in CIC

discussion, it suffices to say that all terms used as types here, including Type_0 itself, will have type Type_i for some i.

The next two lines of Figure 3.1 declare the constructors zero and succ. zero is given type nat, while succ is given type nat $\Rightarrow$ nat. This latter type, called a *function* type, means that the term succ M is of type nat for any M which itself is of type nat. (nat $\Rightarrow$ nat is actually an abbreviation for the type $\Pi x : \mathsf{nat} \,.\, \mathsf{nat}$, introduced below.) Since zero and succ can both be used to construct terms of type nat, they are called constructors of nat. It is clear from the definitions that there is an isomorphism from the natural numbers to the terms of type nat built from zero and succ. It will further be true in CIC that terms built from different constructors are not equal, and that all terms of some type A built from a type constructor a will be equal to some term built from the constructors of a. Thus there is an isomorphism from the natural

numbers to the terms of type nat modulo equality in CIC. nat is then said to be an *adequate encoding* of the natural numbers. In the below, terminology is loosened a bit and a type that is an adequate encoding of a set of mathematic objects is said to be the type of that set of objects. Thus nat is said to be the type of natural numbers, and elements of nat are said to be natural numbers. This is despite the fact that, technically, the natural numbers themselves are external to CIC.

The next lines in Figure 3.1 define the type of (encodings of) finite polymorphic lists, or lists of objects of any given type. The first line declares list A as a type for any type A. The next line declares nil as a constructor for this list. nil is given the type $\Pi A : \mathsf{Type}_0 \,.\, \mathsf{list}\ A$, which is a dependent function type. This is a function type where the input can appear in the type of the output. nil A thus has type list A for any A of type Type_0. For example, nil nat has type list nat. The non-dependent function type $A \Rightarrow B$ is actually just an abbreviation for the dependent function type $\Pi x : A \,.\, B$ when x does not occur in B. The following line then declares cons as a second constructor for list. cons is given type $\Pi A : \mathsf{Type}_0 \,.\, A \Rightarrow \mathsf{list}\ A \Rightarrow \mathsf{list}\ A$, which is equivalent to $\Pi A : \mathsf{Type}_0 \,.\, (A \Rightarrow (\mathsf{list}\ A \Rightarrow \mathsf{list}\ A))$, as function types associate to the right. This means that $((\mathsf{cons}\ A)\ M)\ L$ for any type A, any term M of type A, and any term L of type list A. Note that we are here using the notion of *currying*, where arguments of multiple functions are represented as functions that return functions. $((\mathsf{cons}\ A)\ M)\ L$ can also be written cons A M L, as application associates to the left.

list A is intended as an encoding of finite lists of elements of the type A. The empty list is encoded as nil A, while the list of one or more elements is encoded as cons A M L, where M is the encoding of the first element and L is an encoding of the remainder of the list. For example, the list $0, 1, 0$ of natural numbers would be encoded as

$$\mathsf{cons\ nat\ zero\ (cons\ nat\ (succ\ zero)\ (cons}\ A\ \mathsf{zero\ nil))}$$

which itself has type list nat. It clear that this is an adequate encoding of finite lists.

The next two types defined in Figure 3.1, True, and False, reflect the Curry-Howard Isomorphism. Recall from Section 3.1 that propositions are encoded into CIC as types. True is an encoding of the vacuously true proposition. It correspondingly has a constructor true-i for proving it. False is an encoding of the absurd proposition that

has no proof. Accordingly False has no constructors. It is straightforward to see that these are adequate encodings of the notions of truth and falsity.

The next type, eq, is an encoding of the notion of equality. For any type A and elements a_1 and a_2 of A, the type eq A a_1 a_2 encodes the proposition that a_1 and a_2 are equal. eq has one constructor, eq-refl, which, for any type A and any element a of A, returns a proof that a equals itself. Thus eq-refl encodes the reflexivity of equality. For example, the term eq-refl nat zero zero is a proof of, or term of type, eq nat zero zero. It is straightforward to see that this is an adequate encoding of the notion of equality, as an object should be equal to itself and no other objects.

As suggested in Section 3.1, False can be used to encode negation. Specifically, for any type A, the type $A \Rightarrow$ False is the type of functions that create an element of type False from an element of type A. If this type is inhabited, meaning there is an object f of type $A \Rightarrow$ False, then there can be no element a of A, as $f\ a$ would then be an element of the type False. Note, however, that it is not the case that there is an object f of type $A \Rightarrow$ False whenever there is not an object of type A, as this would imply the Law of the Excluded Middle, which is rejected by Constructivism. Combining False with eq, the proposition that two objects a_1 and a_2 of type A are not equal can now be stated as the type eq A a_1 $a_2 \Rightarrow$ False. For example, it will be possible prove, or create a term of type, (eq nat zero (succ zero)) $\Rightarrow$ False indicating that 0 is not equal to 1.

The next three lines of Figure 3.1 define the type le m n of proofs that $m \leq n$ for natural numbers m and n. (m and n are actually terms M and N that are encodings of natural numbers m and n.) The first constructor for le is le-refl, which produces a proof that $n \leq n$ for any n. The second constructor for le is le-succ. le-succ m n P is a proof that $m \leq n+1$, provided that P is a proof that $m \leq n$. To construct a proof that $m \leq m+k$ for arbitrary m, then, le-refl can be used to prove that $m \leq m$, and k applications of le-succ can be applied to build the desired proof. For example, the following is a proof that $1 \leq 3$:

```
le-succ (succ; zero) (succ (succ (succ; zero)))
    (le-succ (succ zero) (succ zero) (le-refl (succ zero)))
```

If M and N are encodings of natural numbers m and n, then it is straightforward to see that $\mathsf{le}\ M\ N$ is inhabited, meaning there is a term of this type, if and only if $m \leq n$. Thus it is not possible to construct a proof of $\mathsf{le}\ m\ n$, and this type is an *empty* type, when $m \leq n$ does not hold. Note that the type le could instead have been defined using one constructor for proving $0 \leq m$ for any m and one constructor for proving $m+1 \leq n+1$ given a proof that $m \leq n$. The form used here will be more convenient below.

The final type in Figure 3.1 defines the proposition that a list is sorted. Specifically, $\mathsf{is\text{-}sorted}\ A\ R\ L$ is inhabited if and only if the list L of elements of type A is sorted with respect to binary relation R. Abstractly, a binary relation on a set is a proposition that can hold (or fail to hold) on any two elements of the set. Thus a binary relation on a type A is encoded into CIC as a term R of type $A \Rightarrow A \Rightarrow \mathsf{Type}_0$. Terms M and N of type A are considered to be in the relation defined by R if and only if the type $R\ M\ N$ is inhabited. Thus, for example, le is a relation on nat, as would be expected. The type $\mathsf{is\text{-}sorted}\ A\ R\ L$ is then defined as follows. $\mathsf{is\text{-}sorted\text{-}nil}$ constructs a proof that the empty list is sorted, while $\mathsf{is\text{-}sorted\text{-}one}$ constructs a proof that the singleton list $\mathsf{cons}\ A\ M\ (\mathsf{nil}\ A)$ is also sorted. These are the trivial cases. For a list of two or more elements, $\mathsf{is\text{-}sorted\text{-}many}$ constructs a proof that the list is sorted if the first element is related by R to the second and if the remainder of the list beginning at the second element is sorted. For example, the term

```
is-sorted-many (list nat) le zero zero
    (cons nat zero (cons nat zero (cons (succ zero) (nil nat)))) (le-zero zero)
    (is-sorted-many (list nat) le zero zero (cons nat zero (cons (succ zero) (nil nat)))
        (le-zero (succ zero))(is-sorted-one (list nat) le zero))
```

is a proof that the list $0, 0, 1$ is sorted with respect to $\leq$.

3.2.2 Example Functions

CIC defines two sorts of function, λ-abstractions and pattern-matching functions. A λ-abstraction is a term of the form $\lambda\, x : A\,.\, M$, where x is a variable, A is a type, and M is a term. This represents the function that takes any argument N of

type A and returns the result of substituting N for x in M. The substitution of N for x in M is denoted $[N/x]M$ as above. As a simple example, the λ-abstraction $\lambda\, x : \mathsf{nat}\,.\,\mathsf{succ}\ (\mathsf{succ}\ x)$ represents the function that takes any argument x and returns $x+2$. The type of a λ-abstraction is a function type, so this λ-abstraction for example has type $\mathsf{nat} \Rightarrow \mathsf{nat}$. This means that, as expected, a λ-abstraction can be applied to an argument of its input type. Stated differently, if N is a term of type A and M is a term of type B then $(\lambda\, x : A\,.\,M)\ N$ is a term of type B. The reduction relation of CIC is then defined so that $(\lambda\, x : A\,.\,M)\ N$ reduces, or evaluates, to $[N/x]M$. Thus, for example, $(\lambda\, x : \mathsf{nat}\,.\,\mathsf{succ}\ (\mathsf{succ}\ x))\ \mathsf{zero}$ reduces to, and is thus equal to, $\mathsf{succ}\ (\mathsf{succ}\ \mathsf{zero})$.

The remainder of this sub-section focuses on pattern-matching functions. First, pattern-matching functions are introduced by giving examples with types of the form $A \Rightarrow B$, meaning that the return type does not depend on the scrutinee, or input that pattern-matching examines. The full case of pattern-matching functions with dependencies is then broached second, as this makes the notion of typing more complex.

Pattern-Matching without Dependencies

A pattern-matching function in CIC examines the form of its input and returns a different value, depending on this form. The input that is examined is called the *scrutinee*. The examining is done with a list of pairs of patterns and return values. If a scrutinee matches a pattern, then the associated return value is returned. Patterns are given in the form $P \setminus x_1 : A_1, \ldots, x_n : A_n$. A term N *matches* this pattern if there is some substitution of terms N_i of type A_i for the x_i that makes P identical to N. The x_i are called the *pattern variables*. The types are often omitted from the pattern variables, and the symbol $\cdot$ is used if there are no pattern variables. As an example of matching, $\mathsf{succ}\ \mathsf{zero}$ matches the pattern $\mathsf{succ}\ x \setminus x$ by substituting zero for x. Pattern cases are of the form $P \setminus x_1, \ldots, x_n \to M$, which stipulate that if a scrutinee matches P with some substitution σ then σM, the application of σ to M, is returned.

The following is an example pattern-matching function which implements the standard predecessor function on natural numbers:

$$\textbf{fun}\ (\mathsf{zero} \setminus \cdot \to \mathsf{zero} \mid \mathsf{succ}\ x \setminus x \to x)$$

If the scrutinee is zero, then it matches the first pattern, and the first return value, zero, is returned. If the scrutinee is instead of the form $\mathsf{succ}\ N$ for some N, then it matches the second pattern with the substitution $[N/x]$, and N is returned. The type of the above example has type $\mathsf{nat} \Rightarrow \mathsf{nat}$, because the scrutinee has type nat and each return value also has type nat. Also as with λ-abstractions, the behavior of patern-matching functions is defined by the reduction relation of CIC, so, for example, the term

$$(\textbf{fun}\ (\mathsf{zero} \setminus \cdot \to \mathsf{zero} \mid \mathsf{succ}\ x \setminus x \to x))\ (\mathsf{succ}\ N)$$

reduces to N.

Two similar examples are the functions head and tail, which return the first element and the remainder of the list after the first element, respectively. head is defined as follows:

$$\textbf{fun}\ (A : \mathsf{Type}_0, x : A)\ (\mathsf{nil}\ A \setminus \cdot \to x \mid \mathsf{cons}\ A\ x\ l \setminus x, l \to x)$$

Note that $(A : \mathsf{Type}_0, x : A)$ are additional formal parameters of head with the types indicated. This function thus takes three arguments, two for the parameters A and x and one for the scrutinee. The types of parameters are often omitted when they are apparent. head thus returns the first element of a list, or x if the list is empty. tail is defined as follows:

$$\textbf{fun}\ (A : \mathsf{Type}_0)\ (\mathsf{nil}\ A \setminus \cdot \to \mathsf{nil}\ A \mid \mathsf{cons}\ A\ x\ l \setminus x, l \to l)$$

If the scrutinee to tail is the empty list, then tail returns the empty list. Otherwise, if the scrutinee is $\mathsf{cons}\ A\ x\ l$, tail returns l.

In their full form, pattern-matching functions are also recursive. As an example, the following is the definition of the addition function for natural numbers:

$$\begin{aligned}&\textbf{fun}\ \mathsf{add}\ (x : \mathsf{nat})\ (\mathsf{zero} \setminus \cdot \to x \mid \\ &\qquad\qquad \mathsf{succ}\ y \setminus y \to \mathsf{succ}\ (\mathsf{add}\ x\ y) \\ &\qquad\qquad)\end{aligned}$$

The symbol add after the symbol **fun** is used to refer to the whole function when making recursive calls. Recursion behaves as follows. When a scrutinee matches a pattern, the appropriate return value is returned as before, but with the extra stipulation that the whole function is also substituted for the given symbol. Thus we have that, if F is the above function, then $F\ M\ (\mathsf{succ}\ N)$ reduces to $\mathsf{succ}\ (F\ M\ N)$. The zero case works as before, with $F\ M\ \mathsf{zero}$ reducing to M. This function has type $\mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}$, as the parameter has type nat, the scrutinee has type nat, and each return value also has type nat.

Multiplication of natural numbers can be defined similarly. This is done with the following function:

$$\begin{aligned}&\textbf{fun}\ \mathsf{multiply}\ (x : \mathsf{nat})\ (\mathsf{zero} \setminus \cdot \to \mathsf{zero} \mid \\ &\qquad\qquad \mathsf{succ}\ y \setminus y \to \mathsf{plus}\ (\mathsf{multiply}\ x\ y)\ x \\ &\qquad\qquad)\end{aligned}$$

If the second argument is zero then the result is zero, as anything times 0 is 0. If the second argument is $\mathsf{succ}\ M$ for some M, then the result is $\mathsf{plus}\ (\mathsf{multiply}\ x\ y)\ x$. This makes sense because $x * (y + 1)$ is equal to $(x * y) + x$.

As discussed above, all computations in CIC must terminate. This is required for CIC to be consistent. To ensure this holds, pattern-matching functions are not considered as valid terms unless recursive calls are on *structurally smaller* terms. In all the cases considered here, a term is structurally smaller than another if the first is a strict subterm of the second. For example, the recursive call in the add function above is of the form $\mathsf{add}\ x\ y$, while the scrutinee pattern for that case is $\mathsf{succ}\ y$, so the second argument, which is the pattern-matching argument, is structurally smaller than the scrutinee. Requiring recursive calls to be on structurally smaller terms ensures that successive recursive calls are on smaller and smaller terms, and so the recursion must

eventually terminate. Note that, as a second restriction, pattern-matching functions are also required to have a case for every constructor of the given type.

Another example of a pattern-matching function without dependencies is the list append function, which concatenates two lists. This is defined as follows:

$$\begin{array}{l}\textbf{fun}\ \mathsf{append}\ (A)\ (\mathsf{nil}\ A \setminus A \to \lambda\, l : \mathsf{list}\ A\,.\,l \mid \\ \qquad\qquad \mathsf{cons}\ A\ x\ l_1 \setminus A \to \lambda\, l_2 : \mathsf{list}\ A\,.\,\mathsf{cons}\ A\ x\ (\mathsf{append}\ A\ l_1\ l_2) \\ \qquad\qquad)\end{array}$$

append has type $\Pi A : \mathsf{Type}_0\,.\,\mathsf{list}\ A \Rightarrow \mathsf{list}\ A \Rightarrow \mathsf{list}\ A$. If the first list argument is the empty list, append returns the second. If the first list argument is $\mathsf{cons}\ A\ x\ l_1$, then the result of appending l_2 is calculated by first appending l_2 to l_1 with a recursive call, and then prepending x onto the result with cons. Note that the recursive call uses the structurally smaller argument l_1.

As a final case here, the proof that falsity implies anything is considered. In logic this proposition is called *ex falso quod libet*. This is proved with the pattern-matching function

$$\textbf{fun}\ ()$$

with no cases, as False has no constructors. This pattern-matching function can in fact have any type of the form $\mathsf{False} \Rightarrow A$, as there are no return values and so no requirements on A.

Pattern-Matching with Dependencies

We turn now to some pattern-matching functions for creating and manipulating proofs. Such functions generally generally have a dependent function type $\Pi x : A\,.\,B$, as the return type B must mention the argument. The rules for finding the type of such a function is more complex.

The first example here is a proof that le is reflexive, meaning that any number is less than or equal to itself. This can be written as follows:

$$\textbf{fun}\ f\ (\mathsf{zero} \setminus \cdot \to \mathsf{le\text{-}zero}\ \mathsf{zero} \mid \mathsf{succ}\ x \setminus x \to \mathsf{le\text{-}succ}\ x\ x\ (f\ x))$$

This function is intended to have type $\Pi x : \mathsf{nat} \,.\, \mathsf{le}\ x\ x$. For the case where the input is zero, the return value is $\mathsf{le\text{-}zero}\ \mathsf{zero}$, which has type $\mathsf{le}\ \mathsf{zero}\ \mathsf{zero}$. For the case where the input matches $\mathsf{succ}\ x$, the return value is $\mathsf{le\text{-}succ}\ x\ x\ (f\ x)$, which has type $\mathsf{le}\ (\mathsf{succ}\ x)\ (\mathsf{succ}\ x)$. Note that this requires that the variable f used for recursive calls has the desired type of the function. The types of the two return values are thus both $\mathsf{le}\ P\ P$, where P is the corresponding pattern. It thus makes sense to say that the whole function has type $\Pi x : \mathsf{nat} \,.\, \mathsf{le}\ x\ x$ because, for any M that matches either of the patterns, applying the function to M evaluates to a term with type $\mathsf{le}\ M\ M$. Generalizing, pattern-matching functions have type $\Pi x : A \,.\, B$ if each return value has type $[P/x]B$, where P is the corresponding pattern.

If a pattern-matching function has extra parameters that are in the type of the input, then pattern-matching on the input can constrain these parameters. For example, consider the following casting function, which takes a proof about x and creates a proof about y when $x = y$:

$$\mathbf{fun}\ (A, x, y)\ (\mathsf{eq\text{-}refl}\ t\ z \setminus t, z \to \lambda P : (B \Rightarrow \mathsf{Type}_0) \,.\, \lambda p : P\ z \,.\, p)$$

This function has type

$$\Pi A : \mathsf{Type}_0 \,.\, \Pi x : A \,.\, \Pi y : A \,.\, \mathsf{eq}\ A\ x\ y \Rightarrow \Pi P : (A \Rightarrow \mathsf{Type}_0) \,.\, P\ x \Rightarrow P\ y$$

To see why this holds, consider that the sole pattern, $\mathsf{eq\text{-}refl}\ B\ z$, has type $\mathsf{eq}\ B\ z\ z$. In this case, then, A must be equal to B, and x and y must both be equal to z. Thus it makes sense that the return value for this pattern could have type $\Pi P : (t \Rightarrow \mathsf{Type}_0) \,.\, P\ z \Rightarrow P\ z$, as this is the result of applying the substitution $[B/A, z/x, z/y]$ to the intended return type of the function. The return value of the function is $\lambda P : (B \Rightarrow \mathsf{Type}_0) \,.\, \lambda p : P\ z \,.\, p$, which has exactly this type. Generalizing, if a pattern-matching function has scrutinee x of type A and return type B, and if a pattern P has type σA for some σ, then the type of the associated return value should be $[P/x]\sigma A$.

Note that the above casting function can be used to prove that zero is not equal to $\mathsf{succ}\ x$ for any x. Recalling that negation is encoded as the implication of falsity, the type associated with this proposition is thus $\Pi x : \mathsf{nat} \,.\, \mathsf{eq}\ \mathsf{nat}\ \mathsf{zero}\ (\mathsf{succ}\ x) \Rightarrow \mathsf{False}$.

Let F be the above casting function and let G be the function

$$\textbf{fun}\ (\mathsf{zero} \setminus \cdot \to \mathsf{True} \mid \mathsf{succ}\ x \setminus x \to \mathsf{False})$$

of type $\mathsf{nat} \Rightarrow \mathsf{Type}_0$. The term

$$\lambda\, x : \mathsf{nat}\,.\,\lambda\, e : \mathsf{eq}\ \mathsf{nat}\ \mathsf{zero}\ (\mathsf{succ}\ x)\,.\,F\ \mathsf{nat}\ \mathsf{zero}\ (\mathsf{succ}\ x)\ e\ G\ \mathsf{true\text{-}i}$$

thus has type $G\ (\mathsf{succ}\ x)$, which by the definition of G reduces to False. Note that this requires that $\mathsf{true\text{-}i}$ have type $G\ \mathsf{zero}$, which holds, as $G\ \mathsf{zero}$ reduces to True.

As another example of a pattern-matching function with dependencies, consider the following proof that le is transitive. The most convenient approach is to write a pattern-matching function of type

$$\Pi y : \mathsf{nat}\,.\,\Pi z : \mathsf{nat}\,.\,\mathsf{le}\ y\ z \Rightarrow \Pi x : \mathsf{nat}\,.\,\mathsf{le}\ x\ y \Rightarrow \mathsf{le}\ x\ z$$

This is given as follows:

$$\begin{array}{l}\textbf{fun}\ \mathsf{le\text{-}trans}\ (y, z)\ (\mathsf{le\text{-}refl}\ y' \setminus y' \to \lambda\, x : \mathsf{nat}\,.\,\lambda\, p : \mathsf{le}\ x\ y'\,.\,p \mid \\ \qquad\qquad \mathsf{le\text{-}succ}\ y'\ z'\ p_1 \setminus y', z', p_1 : \mathsf{le}\ y'\ z' \to \\ \qquad\qquad\quad \lambda\, x : \mathsf{nat}\,.\,\lambda\, p_2 : \mathsf{le}\ x\ y'\,.\,\mathsf{le\text{-}succ}\ x\ z'\ (\mathsf{le\text{-}trans}\ y'\ z'\ p_1\ x\ p_2) \\ \qquad\qquad)\end{array}$$

The first pattern matches the proof $\mathsf{le\text{-}refl}\ y'$ of type $\mathsf{le}\ y'\ y'$. This induces the substitution $[y'/y, y'/z]$ for the type of the return value, which must therefore be $\Pi x : \mathsf{nat}\,.\,\mathsf{le}\ x\ y' \Rightarrow \mathsf{le}\ x\ y'$. The return value constructs a term of this type with λ-abstractions that take in the argument x and the proof of $\mathsf{le}\ x\ y'$ and just return this proof. The second pattern matches the proof $\mathsf{le\text{-}succ}\ y'\ z'\ p_1$ of type $\mathsf{le}\ y'\ (\mathsf{succ}\ z')$, so the return value must have type $\Pi x : \mathsf{nat}\,.\,\mathsf{le}\ x\ y' \Rightarrow \mathsf{le}\ x\ (\mathsf{succ}\ z')$. This is accomplished with λ-abstractions that take in x and the proof of $\mathsf{le}\ x\ y'$ and then recurse on the proof p_1 (which is structurally smaller than the input) to form a proof of type $\mathsf{le}\ x\ z'$. $\mathsf{le\text{-}succ}$ is then applied to produce the required proof of $\mathsf{le}\ x\ (\mathsf{succ}\ z')$. The standard form of transitivity, written

$$\Pi x : \mathsf{nat}\,.\,\Pi y : \mathsf{nat}\,.\,\Pi z : \mathsf{nat}\,.\,\mathsf{le}\ x\ y \Rightarrow \mathsf{le}\ y\ z \Rightarrow \mathsf{le}\ x\ z$$

can then be proved with the term

$$\lambda\, x : \mathsf{nat}\,.\,\lambda\, y : \mathsf{nat}\,.\,\lambda\, z : \mathsf{nat}\,.\,\lambda\, p_1 : \mathsf{le}\ x\ y\,.\,\lambda\, p_2 : \mathsf{le}\ y\ z\,.\,F\ y\ z\ p_2\ x\ p_1$$

where F is the above function.

As a final example, we consider a proof that x is less than or equal to $\mathsf{add}\ x\ y$ for any y. This is a proof about the addition function add. The most convenient approach is to write a pattern-matching function of type $\Pi y : \mathsf{nat}\,.\,\Pi x : \mathsf{nat}\,.\,\mathsf{le}\ x\ (\mathsf{add}\ x\ y)$. This function is given as follows:

$$\begin{array}{l}\textbf{fun}\ \mathsf{le\text{-}add}\ (\mathsf{zero} \setminus \cdot \rightarrow \mathsf{le\text{-}refl} \mid \\ \qquad\qquad \mathsf{succ}\ y' \setminus y' \rightarrow \lambda\, x : \mathsf{nat}\,.\,\mathsf{le\text{-}succ}\ x\ (\mathsf{add}\ x\ y')\ (\mathsf{le\text{-}add}\ y'\ x) \\ \qquad\qquad)\end{array}$$

In the first case, y is zero, so the return type should be $\Pi x : \mathsf{nat}\,.\,\mathsf{le}\ x\ (\mathsf{add}\ x\ \mathsf{zero})$, which reduces to $\Pi x : \mathsf{nat}\,.\,\mathsf{le}\ x\ x$. This can be proved simply by using the constructor $\mathsf{le\text{-}refl}$. In the second case, y is $\mathsf{succ}\ y'$, so the return value is required to have type $\Pi x : \mathsf{nat}\,.\,\mathsf{le}\ x\ (\mathsf{add}\ x\ (\mathsf{succ}\ y'))$. This reduces to $\Pi x : \mathsf{nat}\,.\,\mathsf{le}\ x\ (\mathsf{succ}\ (\mathsf{add}\ x\ y'))$, which can be proved by taking in x as an argument and applying $\mathsf{le\text{-}succ}$ to the result of the recursive call.

3.2.3 Type Universes and Impredicativity

One of the stipulations of CIC is that every term has a type. This includes the type Type_0. What should be the type of Type_0, which is itself the type of types? One idea would be to have Type_0 have Type_0 itself. Unfortunately, this leads to a contradiction where every type is inhabited, meaning every proposition is provable, and the theory is thus inconsistent [11].[1] Instead, either Type_0 must be a special term which does not have a type, or there must be a new term, Type_1, to be the type of Type_0, and another term, Type_2, to be the type of Type_1, etc. This is the approach taken in CIC.

[1] The contradiction, known as Girard's Paradox, stems from the fact that there is a type of all types, and is similar to Russell's Paradox, which stems from the existence of a set of all sets.

The Type_i for i a natural number are called the *predicative type universes*. A term is in the universe of Type_i if it is either a term whose type is of type Type_j for $j \leq i$ or it is itself a type of type Type_j for $j \leq i$. The type universes are so called because they are closed: any combination of terms in a universe results in a term still in the universe. If the types A or B are not in the universe, however, then neither is $A \Rightarrow B$. Thus it is impossible to write a proof about all elements of the universe of Type_i inside this universe, as such a proof would have type $\Pi A : \mathsf{Type}_i \,.\, B$ for some B, and this type is not in the universe of Type_i.

The predicative universes are inconvenient, however, for expressing general propositions. This is because types in the type universe Type_i cannot be "about" anything outside of Type_i. Thus we need to invent a new concept of proposition for each universe Type_i. Instead, CIC includes an *impredicative* universe of propositions. We call this universe $\mathsf{Type}_{\mathcal{P}}$ for consistency with the other universe notation. If B is a $\mathsf{Type}_{\mathcal{P}}$, then the type $A \Rightarrow B$ is a $\mathsf{Type}_{\mathcal{P}}$ for any A, no matter what universe. Impredicativity allows a degree of circularity: if B is a $\mathsf{Type}_{\mathcal{P}}$, then $\mathsf{Type}_{\mathcal{P}} \Rightarrow B$ is the type of functions that take any elements of $\mathsf{Type}_{\mathcal{P}}$, including the type $\mathsf{Type}_{\mathcal{P}} \Rightarrow B$ itself. For more discussion of impredicativity, see for example the Coq reference manual [68].

Chapter 4

Higher-Order Name-Binding Rewriting

Though standard term rewriting is useful for defining specific operations, it is difficult to define programming languages in that formalism. This is because many programming languages require the notion of capture-avoiding substitution. Although capture-avoiding substitution can be formalized in standard term rewriting by adding explicit substitution terms to the language [1, 14], this approach is complex. Instead, a number of extensions of standard term rewriting have evolved to add capture-avoiding substitution as a primitive notion, including Higher-Order Rewriting [48, 41], Combinatory Reduction Systems [38], and Expression Reduction Systems [37]. These have all been shown to be equivalent in a certain sense [73], so we focus on Higher-Order Rewriting here.

Higher-Order Rewriting extends the term language of standard Term Rewriting to the simply-typed λ-calculus. A benefit of this approach is that λ-abstractions can used encode variable binding. For example, the doubling function $f\ (x) = x * 2$ might be encoded as the λ-calculus term

$$\mathsf{fun\text{-}one}\ (\lambda\, x\,.\, \mathsf{mult}\ x\ (\mathsf{succ}\ (\mathsf{succ}\ \mathsf{zero}))).$$

A second benefit of this approach is that function application can be used to express capture-avoiding substitution for a name binding, which is useful in defining computation in a programming language. For example, the operation apply-fun that applies

a function to an argument might be defined with the rewrite rule

$$\mathsf{apply\text{-}fun}\ (\mathsf{fun\text{-}one}\ f)\ x \rightsquigarrow f\ x$$

This specifies that the value of applying the function encoded as $\mathsf{fun\text{-}one}\ f$ to an argument x is equal to the λ-calculus application of the λ-calculus function f to x. Returning to the above example of the doubling function, the term

$$\mathsf{apply\text{-}fun}\ (\mathsf{fun\text{-}one}\ (\lambda\, x\,.\,\mathsf{mult}\ x\ (\mathsf{succ}\ (\mathsf{succ}\ \mathsf{zero}))))\ \mathsf{zero}$$

reduces to $\mathsf{mult}\ \mathsf{zero}\ (\mathsf{succ}\ (\mathsf{succ}\ \mathsf{zero}))$, meaning that applying the doubling function to 0 is equal to $0 * 2$. This expression can then be further reduced by standard rewrite rules.

Unfortunately, λ-abstractions are not good at encoding name-bindings that are not meant as functions. This is because, by the Substitution principle, it is not possible to distinguish variables from other terms. For example, consider the operation $\mathsf{countvars}$ meant to count the number of variables occurring in an arithmetic expression. The only way to have $\mathsf{countvars}\ x$ reduce to $\mathsf{succ}\ \mathsf{zero}$ when x is a variable would be to add the rule

$$\mathsf{is\text{-}var}\ x \rightsquigarrow (\mathsf{succ}\ \mathsf{zero})$$

but this rule would match *any* x, not just those that are variables. Similarly, it is not possible to define a construct for testing equality unless that construct either does not produce an answer for variables or has the non-left-linear rule

$$\mathsf{is\text{-}eq}\ x\ x \rightsquigarrow (\mathsf{succ}\ \mathsf{zero})$$

which causes problems for confluence.

To solve these problems, Higher-Order Name-Binding Rewriting is introduced. This formalism allows rewrite rules to be defined over the simply-typed $\lambda\nu$-calculus, an extension of the simply-typed λ-calculus withx ν-abstractions. Rather than binding variables, ν-abstractions bind locally bound constructors. Under the HOEC approach, ν-abstractions are then used to encode name bindings, while the constructors they bind are used to encode names. The advantage of this approach is that locally-bound constructors do not have to adhere to the Substitution principle, and a rule such as

the proposed rule for countvars above can exactly match the constructors used to encode variables. Also, the proposed rule for is-eq above can become left-linear, as it now must match two copies of the same constructor instead of two copies of the same variable. In addition, since λ-abstractions are retained, λ-abstractions can still be used to define name-binding constructs that are meant as functions.

The remainder of this chapter is organized as follows. Section 4.1 defines the simply-typed $\lambda\nu$-calculus, the term language of Higher-Order Name-Binding Rewriting. Section 4.2 defines Higher-Order Name-Binding Rewrite systems, or HNRSs. Section 4.3 proves confluence of the orthogonal HNRSs. Section 4.4 proves convergence of the union of an orthogonal HNRS and a terminating ACRS, a result which will be used in Chapter 7.

4.1 The $\lambda\nu$-Calculus

To define HNRSs, the term language must first be fixed. This defines the syntactic objects on which HNRSs operate. The underlying language of HNRSs is the simply-typed $\lambda\nu$-calculus. This is the simply-typed λ-calculus enriched with ν-abstractions, a construct that locally binds constructors. In the remainder of this section, a precise definition is given for the $\lambda\nu$-calculus. Section 4.1.1 introduces the types and terms of the $\lambda\nu$-calculus. Section 4.1.2 defines the typing relation of the $\lambda\nu$-calculus. Finally, Section 4.1.3 defnes equality in the $\lambda\nu$-calculus.

4.1.1 Types and Terms

We assume three disjoint sets of symbols $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{V}$ have been given, with $\mathcal{C}$ and $\mathcal{V}$ both countably infinite. Elements of these sets are referred to as *base types*, *constructors*, and *variables*, respectively. In the below, b is used for base types, x, y, and z are used for variables, and c, d, e, and f are used for constructors.

Definition 4.1.1 (Type). *The* types *are given by the following inductive definition:*

- *Any $b \in \mathcal{B}$ is a type;*

- *If A and B are types built from $\mathcal{B}$ then $A \Rightarrow B$ is a type, called the* function type from A to B*; and*
- *If A and B are types then $\nabla A \,.\, B$ is a type, called a* ∇-type.

Types are denoted below as A or B, possibly with subscripts. The intended meaning of the types are as classifiers of terms, where the phrase *M is of type A* is used to denote that the term M belongs to the set defined by A. This notion is defined precisely below. For now the intended meaning of the types is given. Each $b \in \mathcal{B}$ is intended as an algebraic datatype, with terms of type b being elements of the algebraic datatype. Terms of type $A \Rightarrow B$ are intended as functions that build terms of type B from terms of type A. The type $\nabla A \,.\, B$ is used to classify ν-abstractions that bind a local constructor of type A and contain a term of type B. For the reader unfamiliar with the symbol, ∇ is pronounced "nabla". If Γ is a list of pairs of the form $x : A$ and $c : A$, (Γ is thus a typing context, a notion defined below) then $\Pi\Gamma \,.\, M$ is defined recursively as

$$\begin{array}{lcl} \Pi \cdot \,.\, A & = & A \\ \Pi x : A, \Gamma \,.\, B & = & A \Rightarrow (\Pi\Gamma \,.\, B) \\ \Pi c : A, \Gamma \,.\, B & = & \nabla A \,.\, (\Pi\Gamma \,.\, B) \end{array}$$

It is thus apparent that every type is of the form $\Pi\Gamma \,.\, b$ for some Γ and b. Note that $\Rightarrow$ associates to the right, so $A_1 \Rightarrow (A_2 \Rightarrow A_3)$ is also written $A_1 \Rightarrow A_2 \Rightarrow A_3$.

Definition 4.1.2 (Term). *The* terms *are defined inductively as follows:*

- *If $c \in \mathcal{C}$ then c is a term;*
- *If $x \in \mathcal{V}$ then x is a term;*
- *If M and N are terms then so is $M\ N$, called the* application of M to N*;*
- *If M is a term and $c \in \mathcal{C}$ then $M\ \langle c \rangle$ is a term, called the* constructor replacement of c in M*;*
- *If M is a term, A is a type, and $x \in \mathcal{V}$ then $\lambda\, x : A \,.\, M$ is a term, called a* λ-abstraction*; and*
- *If M is a term, A is a type, and $c \in \mathcal{C}$ then $\nu\, c : A \,.\, M$ is a term, called a* ν-abstraction*.*

M and N are used for terms below. The type annotations are often dropped on λ- and ν-abstractions when clear from context, writing $\lambda\, x\,.\, M$ in place of $\lambda\, x\!:\!A\,.\, M$ and writing $\nu\, c\,.\, M$ in place of $\nu\, c\!:\!A\,.\, M$. The following notations relating to sequences are used below to abbreviate repetitive uses of notation. If Γ is again a list of pairs of the form $x : A$ and $c : A$ then $\lambda\Gamma\,.\, M$ is defined recursively as

$$\begin{array}{lcl} \lambda\cdot\,.\, M & = & M \\ \lambda x : A, \Gamma\,.\, M & = & \lambda\, x\!:\!A\,.\,\lambda\Gamma\,.\, M \\ \lambda c : A, \Gamma\,.\, M & = & \nu\, c\!:\!A\,.\,\lambda\Gamma\,.\, M \end{array}$$

This generalizes the similar notions of λ- and ν-abstracion. An *argument* is either a term or the form $\langle c\rangle$ for some constructor c. Arguments are written R below. The notation $M\ \vec{R}$ can then be defined recursively as

$$\begin{array}{lcl} M\ \cdot & = & M \\ M\ (N, \vec{R}) & = & (M\ N)\ \vec{R} \\ M\ (\langle c\rangle, \vec{R}) & = & (M\ \langle c\rangle)\ \vec{R} \end{array}$$

This generalizes the similar notions of application and constructor replacment. The notation $M\ \Gamma$ is then defined by considering the variables in Γ as term arguments and the constructors as constructor replacements, so that for example $M\ (x_1 : A_1, c_2 : A_2, c_3 : A_3)$ abbreviates the term $((M\ x_1)\ \langle c_2\rangle)\ \langle c_3\rangle$. Note that both applications and constructor replacements associate to the left, so this term can also be written $M\ x_1\ \langle c_2\rangle\ \langle c_3\rangle$.

The concept of a *term context* is now defined. Term contexts are not to be confused with the type contexts, introduced below. To form the term contexts, the definition of term is enriched to include the special symbol $\_$. A term context is then a term by this enriched definition that contains exactly one occurrence of $\_$. Term contexts are written as C below. Intuitively, a term context represents a term with a hole. A term context may be *filled* with another term M by replacing $\_$ with M in C. This is written $C[M]$. Note that the filling operation is different from substitution, defined below, as filling is not capture-avoiding. The filling operation is also called *grafting* in the literature. A term context C_1 can also be filled with another term context C_2, written $C_1[C_2]$, yielding a bigger term context.

The term contexts enable a number of useful definitions. A term M is a *subterm* of $C[M]$ for any term context C. M is additionally said to be a *strict subterm* if C is not the trivial context $\_$. Conversely, $C[M]$ is said to be a *superterm* of M, and is also called a *strict superterm* of M if C is again not the trivial context $\_$. An *occurrence* of M in N is a term context C such that $C[M] = N$. An occurrence of a variable x in M is said to be *bound* if and only if the occurrence is of the form $C_1[\lambda\, x\, .\, \_[C_2]]$. Stated differently, an occurrence of x is bound in M if it is inside a λ-abstraction using x as argument. An occurrence of a variable x that is not bound in M is said to be *free* in M. The set of all free variables in M is denoted $\mathrm{FV}(M)$. Similarly, an occurrence of constructor c in M is bound if and only if the occurrence is of the form $C_1[\nu\, c\, .\, \_[C_2]]$. Otherwise an occurrence of c in M is free. The set of all free constructors in M is denoted $\mathrm{FC}(M)$.

4.1.2 Typing

Typing is used to classify terms and also acts as a well-formedness condition on terms. Typing is defined in terms of typing contexts, which associate types with variables and constructors. These are defined inductively as follows:

- $\cdot$ is a typing context, called the *empty context*;
- If Γ is a typing context and A is a type then $\Gamma, x : A$ is a typing context; and
- If Γ is a typing context and A is a type then $\Gamma, c : A$ is a typing context.

Γ is used here and below to denote contexts. Typically, the terminating $\cdot$ is dropped, and contexts are written, for example, $x : A_1, c : A_2$. $\mathrm{Dom}(\Gamma)$ denotes the *domain* of Γ, meaning the set of constructors and variables to the left of a colon in Γ. $x : A \in \Gamma$ denotes that $x \in \mathrm{Dom}(\Gamma)$ and that the last (right-most) occurrence of x in Γ is paired with A. $c : A \in \Gamma$ denotes the similar notion for c.

A constructor can be removed from a context by removing it and all variables after it. This is denoted $\mathbf{remove}_c(\Gamma)$, and is defined as follows:

$$\begin{array}{lcl} \mathbf{remove}_c(\Gamma, c : A) & = & \Gamma \\ \mathbf{remove}_c(\Gamma, x : A) & = & \mathbf{remove}_c(\Gamma) \\ \mathbf{remove}_c(\Gamma, c' : A) & = & \mathbf{remove}_c(\Gamma) \quad \text{if } c \not\equiv c' \end{array}$$

The intuitive notion behind this definition is that a term typable in $\mathbf{remove}_c(\Gamma)$ cannot possibly contain c, even after substitutions for variables.

Terms are associated with types by the judgment $\Gamma \vdash M : A$. When this judgment holds, M is said to have type A in context Γ. This is defined according to the following rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{c : A \in \Gamma}{\Gamma \vdash c : A}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda\, x\,.\, M : A \Rightarrow B} \qquad \frac{\Gamma \vdash M : A \Rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$$\frac{\Gamma, c : A \vdash M : B}{\Gamma \vdash \nu\, c\,.\, M : \nabla A\,.\, B} \qquad \frac{\mathbf{remove}_c(\Gamma) \vdash M : \nabla A\,.\, B \quad \Gamma \vdash c : A}{\Gamma \vdash M\ \langle c \rangle : B}$$

Typing serves as a well-formedness condition on terms. In the below all terms are assumed be well-typed with respect to some context.

4.1.3 Equality in the $\lambda\nu$-Calculus

The $\lambda\nu$-calculus defines two notions of equality, α-equivalence and $\beta\nu\eta$-equivalence. α-equivalence enriches the standard notion of being equal up to renaming of bound variables by also considering bound constructors. $\beta\nu\eta$-equivalence includes the standard β and η rules of the simply-typed λ-calculus along with a rule for reducing constructor replacements and an extensionality rule for ν-abstractions.

Terms are considered syntactically equal up to renaming of bound variables and constructors. More precisely, α-equivalence of terms, denoted $\equiv$, is defined by the

following set of rules:

$$\frac{}{x \equiv x} \qquad \frac{}{c \equiv c}$$

$$\frac{x \notin \mathrm{FV}(M_1) \cup \mathrm{FV}(M_2) \quad M_1\{x_1 \mapsto x\} \equiv M_2\{x_2 \mapsto x\}}{\lambda\, x_1 \,.\, M_1 \equiv \lambda\, x_2 \,.\, M_2} \qquad \frac{M_1 \equiv M_2 \quad N_1 \equiv N_2}{M_1\ N_1 \equiv M_2\ N_2}$$

$$\frac{c \notin \mathrm{FC}(M_1) \cup \mathrm{FC}(M_2) \quad M_1\{c_1 \mapsto c\} \equiv M_2\{c_2 \mapsto c\}}{\nu\, c_1 \,.\, M \equiv \nu\, c_2 \,.\, M_2} \qquad \frac{M_1 \equiv M_2}{M_1\ \langle c \rangle \equiv M_2\ \langle c \rangle}$$

where $M\{x_1 \mapsto x\}$ and $M\{c_1 \mapsto c\}$ are the terms resulting from replacing all free occurrences in M of x_1 by x and those of c_1 by c, respectively. It is straightforward to see that this notion of equality is decidable, reflexive, and symmetric, and a straightforward inductive argument shows that it is transitive.

Capture-avoiding substitution is written $[M/x]N$, which is read as the substitution of M for x in N. This can be defined as follows:

$$\begin{array}{lcll}
[M/x]x & = & M & \\
[M/x]y & = & y & \text{if } x \not\equiv y \\
[M/x]c & = & c & \\
[M/x](\lambda\, y \,.\, N) & = & \lambda\, y \,.\, [M/x]N & \text{if } y \notin \mathrm{FV}(M) \cup \{x\} \\
[M/x](N_1\ N_2) & = & [M/x]N_1\ [M/x]N_2 & \\
[M/x](\nu\, c \,.\, N) & = & \nu\, c \,.\, [M/x]N & \text{if } c \notin \mathrm{FC}(M) \\
[M/x](N\ \langle c \rangle) & = & [M/x]N\ \langle c \rangle &
\end{array}$$

Note that this operation is defined up to α-equivalence of N. Thus, despite the side conditions on substituting into λ- and ν-abstractions, renaming the bound variable or constructor can always ensure that substitution is defined. A straightforward inductive argument shows that substitution is well-defined, meaning that if $N_1 \equiv N_2$ then $[M/x]N_1 \equiv [M/x]N_2$ when these are both defined.

It is now possible to define $\beta\nu\eta$-equality. This is defined in terms of the one-step $\beta\nu\eta$-reduction relation $\rightsquigarrow_{\beta\nu\eta}$, defined as follows:

$$\begin{array}{llll}
(\beta) & C[(\lambda\, x\,.\, N)\ M] & \rightsquigarrow_{\beta\nu\eta} & C[[M/x]N] \\
(\nu_\beta) & C[(\nu\, c\,.\, M)\ \langle d\rangle] & \rightsquigarrow_{\beta\nu\eta} & C[M\{c \mapsto d\}] \\
(\eta) & C[\lambda\, x\,.\, M\ x] & \rightsquigarrow_{\beta\nu\eta} & C[M] \qquad \text{if } x \notin \mathrm{FV}(M) \\
(\nu_\eta) & C[\nu\, c\,.\, M\ \langle c\rangle] & \rightsquigarrow_{\beta\nu\eta} & C[M]
\end{array}$$

The one-step β-reduction relation $\rightsquigarrow_\beta$ is similar but only uses the rule β. Similarly, $\rightsquigarrow_\nu$ uses only the rule ν and $\rightsquigarrow_{\beta\nu}$ uses only the rules β and ν. $\rightsquigarrow_\eta$ refers to both η rules. Note that η here is split into the standard λ-calculus η rule, here called η_β, and an extensionality rule η_ν for ν-abstractions.

The following auxiliary definitions will be used below. A term of the form $(\lambda\, x\,.\, M)\ N$ is called a *β-redex*, a term of the form $(\nu\, c\,.\, M)\ \langle d\rangle$ is called a *ν-redex*, and a term of the form $\nu\, c\,.\, M\ \langle c\rangle$ or $\lambda\, x\,.\, M\ x$ with $x \notin \mathrm{FV}(M)$ is called an η-redex. M is said to *$\beta\nu\eta$-reduce* to N if and only if $M \rightsquigarrow^*_{\beta\nu\eta} N$ holds. The $\beta\nu\eta$-equality relation $=_{\beta\nu\eta}$ is defined as the reflexive-symmetric-transitive closure of $\rightsquigarrow_{\beta\nu\eta}$. It is straightforward to show that $\rightsquigarrow_{\beta\eta}$ preserves typing, meaning that if $\Gamma \vdash M : A$ and $M \rightsquigarrow_{\beta\eta} N$ then $\Gamma \vdash N : A$. This follows from preservation of typing for substitution.

To ensure that every (well-typed) term of the simply-typed $\lambda\nu$-calculus has a unique normal form, $\rightsquigarrow_{\beta\nu\eta}$ must be guaranteed to be confluent and strongly normalizing on the well-typed terms. This is straightforward by a translation $[\![\cdot]\!]$ from the terms and types of the simply-typed $\lambda\nu$-calculus to those of the simply-typed λ-calculus, where the simply-typed λ-calculus is exactly the simply-typed $\lambda\nu$-calculus with ∇-types, ν-abstractions, and name replacements removed. The translation $[\![\cdot]\!]$ is the identity on most types and terms, except on the constructs of the $\lambda\nu$-calculus that are not in the λ-calculus, in which case it behaves as follows:

$$\begin{array}{lcl}
[\![\nabla A\,.\, B]\!] & = & [\![A]\!] \Rightarrow [\![B]\!] \\
[\![\nu\, c\,.\, M]\!] & = & \lambda\, x_c\,.\, [\![M]\!]\{c \mapsto x_c\} \\
[\![M\ \langle c\rangle]\!] & = & [\![M]\!]\ c
\end{array}$$

Again, $M\{c \mapsto x_c\}$ replaces free occurrences of c in M with x_c. Straightforward inductive arguments yield $\Gamma \vdash M : A$ iff $[\![\Gamma]\!] \vdash [\![M]\!] : [\![A]\!]$ and $M \rightsquigarrow N$ iff $[\![M]\!] \rightsquigarrow [\![N]\!]$,

where $[\![\Gamma]\!]$ translates all types in Γ and replaces constructors on the left with variables. Thus the properties of confluence and strong normalization, which are known to hold for the simply-typed λ-calculus, carry directly over to the simply-typed $\lambda\nu$-calculus. Note that a similar translation is also carried out to the same effect in [10].

This proof also allows the definition of the *long $\beta\nu\eta$-normal form* of a term. A term M is in long $\beta\nu\eta$-normal form if and only if it contains no β- or ν-redexes and every occurrence of a variable or constructor is *fully applied*, meaning the subterm at that occurrence is of the form $x\ \vec{R}$ or $c\ \vec{R}$ and has base type. Long $\beta\nu\eta$-normal form is useful because the only long $\beta\nu\eta$-normal forms of functional or ∇-type are λ-abstractions or ν-abstractions, respectively.

To find a long $\beta\eta$-normal form for a particular M requires first taking the *η-expansion* of M. This is defined as the result of replacing every variable x or constructor c of type $\Pi\Gamma\,.\,b$ with $\lambda\Gamma\,.\,x\ \Gamma$ or $\lambda\Gamma\,.\,c\ \Gamma$, respectively. The long $\beta\nu\eta$-normal form of a term is then the $\beta\nu$-normal form of the η-expansion of that term. The resulting term is $\beta\nu\eta$-equal to the original term, as the η-expansion of a term is η-equal to the original term, as η-expansion simply does η-reduction in reverse, and $\beta\nu$-reduction obviously preserves $\beta\nu\eta$-equality. The resulting term is also in long $\beta\nu\eta$-normal form, as η-expansion ensures that every constructor and free variable is fully applied and $\beta\nu$-reduction removes all $\beta\nu$-redexes but cannot decrease the number of arguments to which a free variable or constructor is applied. To see that the long $\beta\nu\eta$-normal form is unique, note that η-reducing it eventually yields the unique (by confluence) $\beta\nu\eta$-normal form, and a straightforward inductive argument shows that taking the η-expansion and $\beta\nu$-reducing the result yields the same term again. Thus it is a well-defined notion to talk of *the* long $\beta\eta$-normal form of a term M. This is denoted $\mathrm{NF}(M)$.

4.2 HNRSs Defined

A higher-order name-binding rewrite system, or HNRS, is a typing context Γ^{c} of constructors along with a set of rules. A *rule* is a triple $\Gamma.l \rightsquigarrow r$, for terms l and r and context Γ, that adherers to the following restrictions:

1. l and r must be in long $\beta\eta$-normal form;
2. l and r must be of the same base type $b \in \mathcal{B}$ with respect to the typing context Γ^c, Γ;
3. l must not be a single variable;
4. $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$; and
5. l is required to be a *pattern*, meaning that every occurrence of a free variable x in l is of the form $x\ \vec{R}$ where the term arguments in $\vec{R}$ are distinct variables bound in l.

The first three conditions are useful for technical reasons. The fourth is because it is not clear for example what term to use for y in the rule $x \rightsquigarrow c\ x\ y$. The final condition is to ensure that the rewrite relation itself is decidable. It is known that finding a substitution σ such that $\sigma M =_{\beta\eta} N$ is decidable in the simply-typed λ-calculus [43, 20], and the translation given above from the simply-typed $\lambda\nu$-calculus to the λ-calculus shows that this decidability result extends in a straightforward manner to the simply-typed $\lambda\nu$-calculus. As a side note, many definitions of HRSs do not enforce the patterns condition. Authors often refer to HRSs that do satisfy this condition as pattern rewrite systems or PRSs. Since even the question of whether a term rewrites in a single step to another term is undecidable in general with the patterns condition, and because most instances of HRSs in the literature are PRSs, this condition is made part of the definition of HNRSs here. This also simplifies the proof that orthogonality implies confluence below.

As with AC-rewriting, an HNRS $\mathcal{R}$ induces a one-step $\mathcal{R}$-reduction relation. For any given terms M and N, $M \rightsquigarrow_{\mathcal{R}} N$ if and only if

$$M \equiv C[M'] \wedge N \equiv C[N'] \wedge (\lambda\Gamma\,.\,l)\ \vec{R} =_{\beta\nu\eta} M' \wedge (\lambda\Gamma\,.\,r)\ \vec{R} =_{\beta\nu\eta} N'$$

for some rule $(\Gamma.l \rightsquigarrow r) \in \mathcal{R}$, some term context C, some terms M' and N', and some sequence of arguments $\vec{R}$ with the same length as Γ. This last condition assures that only terms of base type will be replaced by $\rightsquigarrow_{\mathcal{R}}$. An $\mathcal{R}$-redex for HNRS $\mathcal{R}$ is then defined as any $M' =_{\beta\nu\eta} (\lambda\Gamma\,.\,l)\ \vec{R}$ for some rule $(\Gamma.l \rightsquigarrow r) \in \mathcal{R}$ and some sequence $\vec{R}$ of arguments. As before, M $\mathcal{R}$-reduces to N if and only if $M \rightsquigarrow^*_{\mathcal{R}} N$ holds, in which

case N is called an $\mathcal{R}$-reduct of M. Two terms are again said to be $\mathcal{R}$-joinable if and only if they share a common $\mathcal{R}$-reduct. Two terms are said to be $\mathcal{R}$-*equal*, written $=_{\mathcal{R}}$, if and only if they are related by the reflexive-symmetric-transitive closure of the union $\rightsquigarrow_{\mathcal{R}} \cup \rightsquigarrow_{\beta\nu\eta}$.

One difference between the HNRS definition of $\mathcal{R}$-reduction and the corresponding ACRS version is that the $\mathcal{R}$-reduction relation here does not use the equalities $C[M'] =_{\beta\nu\eta} M$ and $C[N'] =_{\beta\nu\eta} M$. Instead, the equality relation used here is $\equiv$. It is a standard result for HRSs that this does not matter, as $M =_{\mathcal{R}} N$ if and only if $\mathrm{NF}(M) \leftrightsquigarrow^*_{\mathcal{R}} \mathrm{NF}(N)$ for HRSs [41]. This result is rather complex to prove, however. Instead, in this document it is assumed that all terms are in long $\beta\nu\eta$-normal form. $\mathcal{R}$-Equality can then be tested by $\mathcal{R}$-reduction steps on long $\beta\nu\eta$-normal forms. The following theorem shows that this is a valid approach:

Theorem 4.2.1. *If $M =_{\mathcal{R}} N$ for M in long $\beta\nu\eta$-normal form then $M \leftrightsquigarrow^*_{\mathcal{R}} \mathrm{NF}(N)$.*

Proof. If $M \rightsquigarrow_{\mathcal{R}} N$ then $M \equiv C[M']$ and $N \equiv C[N']$ for some term context C, some rule $(\Gamma.l \rightsquigarrow r) \in \mathcal{R}$, some $M' =_{\beta\eta} (\lambda\Gamma\,.\,l)\ \vec{R}$, some $N' =_{\beta\eta} (\lambda\Gamma\,.\,r)\ \vec{R}$, and some $\vec{R}$. Since $\mathrm{NF}(N') =_{\beta\nu\eta} N' =_{\beta\nu\eta} (\lambda\Gamma\,.\,r)\ \vec{R}$, we have that $M \rightsquigarrow_{\mathcal{R}} C[\mathrm{NF}(N')]$. Since $(\lambda\Gamma\,.\,l)\ \vec{R}$ and $(\lambda\Gamma\,.\,r)\ \vec{R}$ must be of base type, so must M' and N'. This implies that $C[\mathrm{NF}(N')]$ must be in long $\beta\nu\eta$-normal form, as $C[M']$ is, and replacing a term of base type with another in long $\beta\nu\eta$-normal form cannot either create any β- or ν-redexes or cause any constructors or variables to no longer be fully applied. Further, since $\mathrm{NF}(N') =_{\beta\nu\eta} N'$ it follows that $C[\mathrm{NF}(N')] =_{\beta\nu\eta} N$ and so $C[\mathrm{NF}(N')]$ is the long $\beta\nu\eta$-normal form of N. Similarly, if N is in long $\beta\eta$-normal form then $\mathrm{NF}(M) \rightsquigarrow_{\mathcal{R}} N$. The desired result then follows by induction over the number of steps in a deduction of $M =_{\mathcal{R}} N$. □

4.3 Orthogonality

The Critical Pairs Lemma is useful for terminating systems, but does not necessarily hold for non-terminating systems. For HRSs, the infamous Klop Counterexample demonstrates that a (non-terminating) rewrite system can have no critical pairs and still fail confluence (see, for example, [72]). One reason confluence fails for the Klop

Counterexample is because it is not left linear. A rule is called *left-linear* if its left-hand side contains at most one occurrence of any free variable. A rule that is not left-linear therefore requires that two or more subterms be equal. For example, the rule

$$f\ x\ x \rightsquigarrow r$$

only applies to terms of the form $f\ M\ N$ where $M =_{\beta\nu\eta} N$. Thus any rule (including the non-left-linear rule itself) can rewrite M to M', leaving N as it is, turning the redex $f\ M\ N$ into the non-redex $f\ M'\ N$. This is true even when the two rules have no non-trivial overlap. If, however, the rules of a rewrite system are all left-linear, and additionally there is no non-trivial overlap, then the rules can no longer interfere with each other in this manner. Such a system is called *orthogonal*, and this is enough to prove confluence. Note that, technically speaking, confluence is only proved here for terms in long $\beta\nu\eta$-normal form. This will be all that is needed in this document.

For the remainder of this section, an orthogonal HNRS $\mathcal{R}$ is fixed. To prove confluence for $\mathcal{R}$, we follow here the second proof given for HRSs by Mayr and Nipkow [41], using complete superdevelopments. The approach is to find a confluent relation $\geqslant$ between $\mathcal{R}$ and $\mathcal{R}^*$, or more precisely such that $\mathcal{R} \subseteq \geqslant \subseteq \mathcal{R}^*$. By Lemma 2.3.2 this then implies that $\mathcal{R}$ is confluent. The relation used here, called *simultaneous reduction*, is defined as follows:

$$\frac{\forall i(R_i \geqslant R'_i)}{c\ \vec{R} \geqslant c\ \vec{R'}}\ (\mathsf{C}) \quad \frac{\forall i(R_i \geqslant R'_i)}{x\ \vec{R} \geqslant x\ \vec{R'}}\ (\mathsf{X}) \quad \frac{M \geqslant N}{\lambda\, x\,.\, M \geqslant \lambda\, x\,.\, N}\ (\mathsf{L}) \quad \frac{M \geqslant N}{\nu\, c\,.\, M \geqslant \nu\, c\,.\, N}\ (\mathsf{N})$$

$$\frac{\forall i(R_{1,i} \geqslant R'_{1,i}) \quad (\Gamma.l \rightsquigarrow r) \quad c\ \vec{R'_1} =_{\beta\nu\eta} ((\lambda\Gamma\,.\,l)\ \vec{R_2})}{c\ \vec{R_1} \geqslant \mathrm{NF}((\lambda\Gamma\,.\,r)\ \vec{R_2})}\ (\mathsf{R})$$

The notation $\vec{R} \geqslant \vec{R'}$ means that R_i and R'_i have the same length, the same indicies are terms, $R_i \geqslant R'_i$ for every term R_i, and $R_j = R'_j$ for every constructor argument R_j. Intuitively, simultaneous reduction can reduce any number of redexes in a term simultaneously, though it is actually slightly stronger since it can reduce an inner reduction to create a redex in a strict superterm which can then be reduced in the same step of $\geqslant$.

We first give some important properties of $\geqslant$.

Lemma 4.3.1. *$M \geqslant M$ for all M.*

Proof. Immediate by induction on M. □

Lemma 4.3.2. *$\lambda\, x\,.\, M \geqslant N$ implies $N \equiv \lambda\, x\,.\, N'$ for some N' and $\nu\, c\,.\, M \geqslant N$ implies $N \equiv \nu\, c\,.\, N'$ for some N'.*

Proof. Immediate by the definition of $\geqslant$, as only rule (L) applies to λ-abstractions and only rule (N) applies to ν-abstractions. □

Lemma 4.3.3. $\rightsquigarrow_{\mathcal{R}} \subseteq \geqslant \subseteq \rightsquigarrow_{\mathcal{R}} *$

Proof. $\rightsquigarrow_{\mathcal{R}} \subseteq \geqslant$ is immediate by induction on terms and $\geqslant \subseteq \rightsquigarrow_{\mathcal{R}} *$ is immediate by induction on derivations of $\geqslant$. □

Lemma 4.3.4. *$\geqslant$ preserves long $\beta\nu\eta$-normal forms.*

Proof. By induction on derivations of $\geqslant$. For rule (R), a long $\beta\nu\eta$-normal form is used on the right-hand side in the definition. The property follows for all other rules by the induction hypothesis. □

The following lemma states that, if a term matches a left-hand side of $\mathcal{R}$, then no steps of $\geqslant$ in its strict subtersm can make this matching disappear.

Lemma 4.3.5. *Let $c\ \vec{R} \equiv \mathrm{NF}((\lambda\Gamma\,.\,l)\ \vec{R_l})$ for some constructor c, some rule $(\Gamma.l \rightsquigarrow r) \in \mathcal{R}$, and some sequences of arguments $\vec{R}$ and $\vec{R_l}$. If $c\ \vec{R} \geqslant c\ \vec{R'}$ then $c\ \vec{R'} \equiv \mathrm{NF}((\lambda\Gamma\,.\,l)\ \vec{R'_l})$ for some sequence of arguments $\vec{R'_l}$ with $\vec{R_l} \geqslant \vec{R'_l}$.*

Proof. If $c\ \vec{R} \equiv \mathrm{NF}((\lambda\Gamma\,.\,l)\ \vec{R_l})$ then there is some "top part" of $c\ \vec{R}$ that matches l. No steps of $\mathcal{R}$-reduction on $c\ \vec{R}$ can cause this mathing to disappear, as this would imply a non-trivial overlap of left-hand sides in $\mathcal{R}$. Lemma 4.3.3 shows that steps of $\geqslant$ therefore cannot make this matching disappear either. Thus $c\ \vec{R} \geqslant c\ \vec{R'}$ can only hold because terms matching $\vec{R_l}$ have been changed by $\geqslant$, and so $\vec{R_l} \geqslant \vec{R'_l}$ for some $\vec{R'_l}$ such that $c\ \mathsf{var}R' \equiv \mathrm{NF}((\lambda\Gamma\,.\,l)\ \vec{R_l})$ as required. □

The other technical lemma needed for the proof of confluence states that simultaneous reduction on the arguments substituted into a left-hand side implies simultaneous reduction on the substitution of those arguments into that left-hand side.

Lemma 4.3.6. *Let $\vec{R}$ and $\vec{R'}$ be sequences of arguments whose terms are in long $\beta\nu\eta$-normal form. If $\vec{R} \geqslant \vec{R'}$ then $\mathrm{NF}(\lambda\Gamma \,.\, l\ \vec{R}) \geqslant \mathrm{NF}(\lambda\Gamma \,.\, l\ \vec{R'})$ for any left-hand side l in $\mathcal{R}$.*

Proof. By induction on l. The only interesting case is for free variables in l, which always must occur as $x\ \vec{R_x}$ where the term arguments in $\vec{R_x}$ are distinct variables bound in l. In $\mathrm{NF}(\lambda\Gamma \,.\, l\ \vec{R})$ this occurrence of x becomes $\mathrm{NF}(R_i\ \vec{R_x})$ for some term argument R_i in $\vec{R}$, and similarly for $\vec{R'}$. Since R_i is in long $\beta\nu\eta$-normal form it must be $\lambda\Gamma' \,.\, M$ for some M, and thus $\mathrm{NF}(R_i\ \vec{R_x})$ is just a renaming of the variables and constructors in R_i, since R_i cannot have any variables or constructors in $\vec{R_x}$ free. Similarly, R'_i must be $\lambda\Gamma' \,.\, N$ for some N, and $\mathrm{NF}(\lambda\Gamma \,.\, l\ \vec{R'})$ turns the given occurrence of x to a renaming of N. $R_i \geqslant R'_i$ then yields $M \geqslant N$, and a straightforward induction then yields that $\geqslant$ holds between the given renamings of the variables and constructors in M and N. □

To show confluence, we will now consider the biggest possible simultaneous reduction on a term. This is called the *complete superdevelopment* of a term, following the literature. The proof then shows that if M reduces by a simultaneous reduction to N, then N can be reduced by another simultaneous reduction to the complete superdevelopment of M. This then shows that the relation $\geqslant$ is strongly confluent, as if $M \geqslant M_1$ and $M \geqslant M_2$ then each $M_i \geqslant N$ where N is the complete superdevelopment of M. Thus $\geqslant$ is confluent, and by Lemma 2.3.2, so is $\mathcal{R}$.

The complete superdevelopment of M in long $\beta\nu\eta$-normal form is defined by induction on M as follows:

$$
\begin{array}{lcll}
(c\ \vec{R})^* & = & c\ \vec{R}^* & \text{if } c\ \vec{R} \text{ is not a redex} \\
(c\ \vec{R})^* & = & \mathrm{NF}((\lambda\Gamma \,.\, r)\ \vec{R_l}) & \text{if } c\ \vec{R}^* =_{\beta\nu\eta} (\lambda\Gamma \,.\, l)\ \vec{R_l} \text{ for } (\Gamma.l \rightsquigarrow r) \in \mathcal{R} \\
(x\ \vec{R})^* & = & x\ \vec{R}^* & \\
(\lambda\, x \,.\, M)^* & = & \lambda\, x \,.\, M^* & \\
(\nu\, x \,.\, M)^* & = & \nu\, x \,.\, M^* &
\end{array}
$$

where $\vec{R}^*$ is defined as the constructors and complete superdevelopments of the terms of $\vec{R}$.

We now turn to the main theorem:

Theorem 4.3.1. *If M is in long $\beta\nu\eta$-normal form and $M \geqslant N$ then $N \geqslant M^*$.*

Proof. This is proved by induction on the derivation of $M \geqslant N$.

Case: $c\ \vec{R_M} \geqslant c\ \vec{R_N}$ by rule (C).

$\vec{R_N} \geqslant \vec{R_M}^*$ by the induction hypothesis. If $c\ \vec{R_M}^* =_{\beta\nu\eta} (\lambda\Gamma\,.\,l\ \vec{R_l})$ for some rule $(\Gamma.l \rightsquigarrow r) \in \mathcal{R}$, then $(c\ \vec{R_M})^* = \mathrm{NF}((\lambda\Gamma\,.\,r)\ \vec{R_l})$ and $c\ \vec{R_N} \geqslant \mathrm{NF}((\lambda\Gamma\,.\,r)\ \vec{R_l})$ by rule (R). Otherwise $(c\ \vec{R_M})^* = c\ \vec{R_M}^*$ and $c\ \vec{R_N} \geqslant c\ \vec{R_M}^*$ by rule (C).

Case: $x\ \vec{R_M} \geqslant x\ \vec{R_N}$ by rule (X).

$\vec{R_N} \geqslant \vec{R_M}^*$ by the induction hypothesis, and so $c\ \vec{R_N} \geqslant c\ \vec{R_M}^*$ by rule (X).

Case: $\lambda\,x\,.\,M \geqslant \lambda\,x\,.\,N$ by rule (L).

$N \geqslant M^*$ by the induction hypothesis, so $\lambda\,x\,.\,N \geqslant \lambda\,x\,.\,M^*$ by rule (L).

Case: $\nu\,x\,.\,M \geqslant \nu\,x\,.\,N$ by rule (N).

$N \geqslant M^*$ by the induction hypothesis, so $\nu\,x\,.\,N \geqslant \nu\,x\,.\,M^*$ by rule (N).

Case: $c\ \vec{R_M} \geqslant \mathrm{NF}((\lambda\Gamma\,.\,r)\ \vec{R_l})$ by rule (R) for some $\vec{R_l}$ and some rule $(\Gamma.l \rightsquigarrow r) \in \mathcal{R}$ such that $\vec{R_M} \geqslant \vec{R_N}$ and $c\ \vec{R_N} =_{\beta\nu\eta} (\lambda\Gamma\,.\,l)\ \vec{R_l}$.

$\vec{R_N} \geqslant \vec{R_M}^*$ by the induction hypothesis. By Lemma 4.3.5 there is some sequence $\vec{R_l'}$ of arguments such that $c\ \vec{R_M}^* \equiv \mathrm{NF}((\lambda\Gamma\,.\,l)\ \vec{R_l'})$ and $\vec{R_l} \geqslant \vec{R_l'}$. By Lemma 4.3.6 we thus have that $\mathrm{NF}((\lambda\Gamma\,.\,r)\ \vec{R_l}) \geqslant \mathrm{NF}((\lambda\Gamma\,.\,r)\ \vec{R_l'})$.

□

4.4 Modularity of Convergence on a Restricted Set

In Chapter 7 it shall be necessary to combine an orthogonal HNRS defining a programming language with another rewrite system for simplifying the constructor predicates in a term. The constructor predicates form a boolean algebra using the usual boolean connectives $\wedge$, $\vee$, and $\neg$. Unfortunately, the widely-accepted rewrite system for simplifying boolean connectives is an ACRS that is not left-linear, so the combined system is not orthogonal [31]. Though there is a wide literature on the preservation, or *modularity*, of confluence and termination in the union of two systems [17, 18, 16, 26], little of this research has considered either HRSs or ACRSs. Given the increased complexity of the critical pair criteria for ACRSs, as well as the fact that many properties shown for standard TRSs are known to fail for HRSs [72], modularity of confluence and termination might or might not hold in this case, as HNRSs are generalizations of HRSs.

Instead, we focus here on modularity of *restricted* confluence and termination, that is, confluence and termination of a union system when restricted to a set of terms with some useful properties. The difficulty with respect to modularity properties of non-left-linear rules is that such rules require two distinct subterms to be syntactically identical. Thus even if two systems $\mathcal{R}$ and $\mathcal{S}$ are non-overlapping, if $\mathcal{S}$ is non-left-linear then steps of $\mathcal{R}$ can create or destroy $\mathcal{S}$-redexes by rewriting distinct subterms to identical ones or rewriting identical subterms to distinct ones. If $\mathcal{R}$ and $\mathcal{S}$ are non-overlapping, this can only happen when $\mathcal{R}$-reduction happens on a subterm of an $\mathcal{S}$-redex. If, in contrast, $\mathcal{R}$ and $\mathcal{S}$ are restricted to a set T of terms with no $\mathcal{R}$-redexes in subterms of $\mathcal{S}$-redexes, then this problem disappears. This is demonstrated with the following theorem:

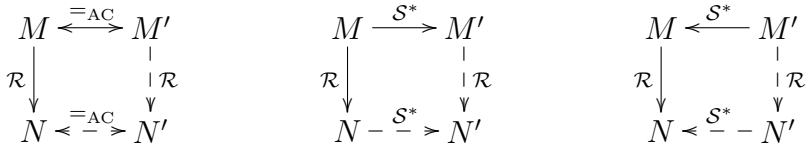
Theorem 4.4.1. *Let $\mathcal{R}$ be an HNRS, $\mathcal{S}$ be an ACRS, and T be any set of terms closed under $\mathcal{R}$-reduction, $\mathcal{S}$-reduction, and $=_{\mathrm{AC}}$ given the associative-commutative operators in $\mathcal{S}$. Further, assume that the following hold:*

- *$\mathcal{R}$ is left-linear;*
- *the rules for $\mathcal{R}$ contain no associative-commutative operators from $\mathcal{S}$;*

- *the left-hand sides in $\mathcal{R}$ have no non-trivial overlap with any left- or right-hand sides in §; and*
- *and no $\mathcal{R}$-redex appears as a subterm of an $\mathcal{S}$-redex in any term in T.*

It is then the case that both confluence and termination on T are modular for $\mathcal{R}$ and $\mathcal{S}$.

Proof. It is first proved that $\mathcal{R}$ commutes with $=_{\mathrm{AC}}$, $\mathcal{R}^{-1}\mathcal{S} \subseteq (\mathcal{S}^{-1})^*\mathcal{R}$, and $\mathcal{S}\mathcal{R} \subseteq \mathcal{R}^{-1}(\mathcal{S}^{-1})^*$, all on terms in T. Graphically, these properties can be displayed as



where the existence of the solid lines implies that of the dashed lines. For all properties we assume $M \rightsquigarrow_{\mathcal{R}} N$. Further, let Q_M be the redex that is reduced in going from M to N, and let Q_N be the term that replaces it in N, so that M and N are equivalent under $\equiv$ except that a copy of Q_M in M is replaced in N with Q_N. Thus $Q_M =_{\beta\nu\eta} (\lambda\Gamma \,.\, l)\ \vec{R}$ and $Q_N =_{\beta\nu\eta} (\lambda\Gamma \,.\, r)\ \vec{R}$ for some $\vec{R}$. For the first property, note that $=_{\mathrm{AC}}$ cannot change Q_M itself as $\mathcal{R}$ is assumed to have no AC operators. $=_{\mathrm{AC}}$ can only move the occurrence of Q_M in M to a different position and change the associative commutative arrangement of AC operators in $\vec{R}$. Thus M' has some subterm $(\lambda\Gamma \,.\, l)\ \vec{R}'$ where $R_i =_{\mathrm{AC}} R'_i$ for every term argument R_i and R_j and R'_j are identical for every constructor argument R_j. Let N' be the result of replacing the appropriate copy of $(\lambda\Gamma \,.\, l)\ \vec{R}'$ in M' with $(\lambda\Gamma \,.\, r)\ \vec{R}'$. It is then straightforward to see that $M' \rightsquigarrow_{\mathcal{R}} N'$ and $N =_{\mathrm{AC}} N'$. In addition, N' must be in T as T is closed under $\rightsquigarrow_{\mathcal{R}}$ and $=_{\mathrm{AC}}$.

For the second property, we may ignore any steps of $=_{\mathrm{AC}}$ in $\rightsquigarrow_{\mathcal{S}}$ by the previous property, so let $\sigma_{\mathcal{S}} l_{\mathcal{S}}$ be the subterm of M replaced by $\mathcal{S}$-rewriting with $\sigma_{\mathcal{S}} r_{\mathcal{S}}$ in M'. If neither of Q_M and $\sigma_{\mathcal{S}} l_{\mathcal{S}}$ is a subterm of the other, the result is immediate. By the requirements on T, it is not possible that Q_M be a subterm of $\sigma_{\mathcal{S}} l_{\mathcal{S}}$. The only other possibility is that $\sigma_{\mathcal{S}} l_{\mathcal{S}}$ is a strict subterm of Q_M. Since l and $l_{\mathcal{S}}$ have no non-trivial overlap by assumption, $\sigma_{\mathcal{S}} l_{\mathcal{S}}$ is a subterm of some R_i. Let $\vec{R}'$ be a new sequence of

arguments that is identical to $\vec{R}$ except $\sigma_{\mathcal{S}} l_{\mathcal{S}}$ is replaced in R_i by $\sigma_{\mathcal{S}} r_{\mathcal{S}}$. Further, let N' be the result of replacing the appropriate copy of Q_N in N by $\mathrm{NF}((\lambda\Gamma\,.\,r)\;\vec{R'})$. It is then apparent that $M' \rightsquigarrow_{\mathcal{R}} N'$. To see that $N \rightsquigarrow_{\mathcal{S}}^{*} N'$, note that $R_i \rightsquigarrow_{\mathcal{S}} R_i'$ by construction. Since ith element of Γ may occur zero, one, or multiple times in r, and since N and N' only differ in N having R_i where N' has R_i', it follows that $N \rightsquigarrow_{\mathcal{S}}^{*} N'$. In addition, N' must be in T as T is closed under $\rightsquigarrow_{\mathcal{R}}$ and $\rightsquigarrow_{\mathcal{S}}$. The proof of the third property is similar.

To show modularity of confluence, note that $\mathcal{R}$ and $\mathcal{S}^*$ commute by an inductive argument on the number of $\mathcal{S}$-steps in $\mathcal{S}^*$ using the second property above. A second inductive argument on the number of $\mathcal{R}$-steps in $\mathcal{R}^*$ then shows that $\mathcal{R}^*$ and $\mathcal{S}^*$ commute, so by the Hindley-Rosen Lemma it follows that confluence is modular for $\mathcal{R}$ and $\mathcal{S}$ for terms in T, meaning that if $M \in T$ then any two $(\mathcal{R} \cup \mathcal{S})$-reducts of M can be joined. Further, all $(\mathcal{R} \cup \mathcal{S})$-reducts of any $M \in T$ must also be in T, as T is closed under $(\mathcal{R} \cup \mathcal{S})$-reduction, so confluence restricted to T is modular for $\mathcal{R}$ and $\mathcal{S}$.

To show modularity of termination, the combination of two similar inductive arguments and the third property above show that $\mathcal{S}^+\mathcal{R}^+ \subseteq \mathcal{R}^+\mathcal{S}^*$. It is also trivially the case that $\mathcal{R}^+ \subseteq \mathcal{R}^+\mathcal{S}^*$, since $\mathcal{S}^*$ is reflexive. Thus, since $\mathcal{S}*$ equals the union of the identity relation and $\mathcal{S}^+$, it follows that $\mathcal{S}^*\mathcal{R}^+ \subseteq \mathcal{R}^+\mathcal{S}^*$. Now assume that $\mathcal{R}$ and $\mathcal{S}$ are both terminating but that their union is not. It follows that there must be an infinite sequence of alternating $\mathcal{R}^+$ and $\mathcal{S}^+$ steps, as any other infinite reduction in the union would imply an sequence of reductions in $\mathcal{R}$ or $\mathcal{S}$. By the above, however, any series of n steps of $\mathcal{R}^+\mathcal{S}^+$ implies a series of n steps of $\mathcal{R}^+$ starting from the same point, since $\mathcal{S}^+ \subseteq \mathcal{S}^*$ and any $\mathcal{S}^*$ steps can be permuted past any $\mathcal{R}^+$ steps. Thus an infinite reduction of alternating $\mathcal{R}^+$ and $\mathcal{S}^+$ steps implies the existence of an infinite sequence of $\mathcal{R}$ steps, which is a contradiction. □

Chapter 5

The Calculus of Nominal Inductive Constructions

In this chapter we introduce CNIC, the Calculus of Nominal Inductive Constructions. CNIC adds untyped names and name binders, along with the ability to recurse over these binders, to CIC. By untyped names it is meant that all names are in a single type Name of names. Thus encoding name binding in CNIC does not satisfy the fourth property, typing, above. There do exist other formalisms in the literature with untyped names in this sense, however, and these have proven useful [59, 23, 10].

The remainder of this chapter is organized as follows. Section 5.1 informally introduces CNIC and gives example CNIC programs. Finally, Section 5.3 develops the metatheory for CNIC. Strong normalization and consistency of CNIC are deferred to Chapter 6.

5.1 Examples

As an example of how the untyped ν-abstraction is used to encode name-binding constructs, consider the following constructor declarations:

$$
\begin{array}{lcl}
\mathsf{nat} & : & \mathsf{Type}_0 \\
\mathsf{zero} & : & \mathsf{nat} \\
\mathsf{succ} & : & \mathsf{nat} \Rightarrow \mathsf{nat} \\
 & & \\
\mathsf{expr} & : & \mathsf{Type}_0 \\
\mathsf{var\text{-}inj} & : & \mathsf{Name} \Rightarrow \mathsf{expr} \\
\mathsf{lit} & : & \mathsf{nat} \Rightarrow \mathsf{expr} \\
\mathsf{plus} & : & \mathsf{expr} \Rightarrow \mathsf{expr} \Rightarrow \mathsf{expr} \\
\mathsf{mult} & : & \mathsf{expr} \Rightarrow \mathsf{expr} \Rightarrow \mathsf{expr} \\
 & & \\
\mathsf{fun\text{-}expr} & : & \mathsf{Type}_0 \\
\mathsf{fun\text{-}one} & : & (\nabla\alpha\,.\,\mathsf{expr}) \Rightarrow \mathsf{fun\text{-}expr} \\
\mathsf{fun\text{-}many} & : & (\nabla\alpha\,.\,\mathsf{fun\text{-}expr}) \Rightarrow \mathsf{fun\text{-}expr}
\end{array}
$$

The type nat is a straightforward encoding of the natural numbers. The type expr is an encoding of arithmetic expressions, including literals, addition expressions, multiplication expressions, and variables. Since names have type Name, names used to encode variables in the object language must be wrapped in var-inj to create an element of type expr. The type fun-expr is the type of functions. fun-one takes as argument the type $\nabla\alpha\,.\,\mathsf{expr}$ of ν-abstractions that bind a name α and build an element of expr. Note that this is different from the ∇ type in the $\lambda\nu$-calculus above. Here, a type need not be given for the name, but, because of dependent types, the name itself might appear in the result type so it must be given. fun-many takes a similar argument of type $\nabla\alpha\,.\,\mathsf{fun\text{-}expr}$ which encodes a name binding over the type fun-expr.

We now consider some operations over this encoding. The first of these is countvars, which counts the number of variable occurrences in an expression. This can be defined

as follows:

$$\begin{aligned}&\textbf{fun}\ \mathsf{countvars}\ (\mathsf{lit}\ n \setminus n \to \mathsf{zero}\\ &\qquad\qquad \mathsf{var\text{-}inj}\ x \setminus x \to \mathsf{succ}\ \mathsf{zero}\\ &\qquad\qquad \mathsf{plus}\ x\ y \setminus x, y \to \mathsf{add}\ (\mathsf{countvars}\ x)\ (\mathsf{countvars}\ y)\\ &\qquad\qquad \mathsf{mult}\ x\ y \setminus x, y \to \mathsf{add}\ (\mathsf{countvars}\ x)\ (\mathsf{countvars}\ y)\\ &\qquad\qquad)\end{aligned}$$

This function behaves in a straightforward manner, returning 0 for literals, 1 for variables, and the sum of the number of variables in the subexpressions for addition and multiplication expressions. To count the number of variables in a function, countvars-fun is used, defined as follows:

$$\begin{aligned}&\textbf{fun}\ \mathsf{countvars\text{-}fun}\ (\mathsf{fun\text{-}one}\ E \setminus E \to \mathsf{lift\text{-}nat}\ (\nu\,\alpha\,.\,\mathsf{countvars}\ E\ \langle\alpha\rangle)\\ &\qquad\qquad \mathsf{fun\text{-}many}\ F \setminus F \to \mathsf{lift\text{-}nat}\ (\nu\,\alpha\,.\,\mathsf{countvars\text{-}fun}\ F\ \langle\alpha\rangle)\\ &\qquad\qquad)\end{aligned}$$

For the fun-one case, countvars-fun introduces a new name α so that the body of E may be accessed with the name replacement $E\ \langle\alpha\rangle$. countvars is then called on this name replacement, and lift-nat is used to lift the result out of the ν-abstraction. The fun-many case is similar, except recursion is required instead of a call to countvars. The lift-nat function used in both cases is defined as

$$\begin{aligned}&\textbf{fun}\ \mathsf{lift\text{-}nat}\ (\nu\,\alpha\,.\,\mathsf{zero} \setminus \cdot \to \mathsf{zero}\ |\\ &\qquad\qquad \nu\,\alpha\,.\,\mathsf{succ}\ x\ \langle\alpha\rangle \setminus x \to \mathsf{succ}\ (\mathsf{lift\text{-}nat}\ x)\\ &\qquad\qquad)\end{aligned}$$

If the input to lift-nat is $\nu\,\alpha\,.\,\mathsf{zero}$ then zero is returned. If the input is $\nu\,\alpha\,.\,\mathsf{succ}\ M$ for some M then succ (lift-nat $\nu\,\alpha\,.\,M$) is returned.

We next consider operations for finding the value of an arithmetic function for a given input. For expressions, this is defined by eval-expr, given as:

$$
\begin{aligned}
&\textbf{fun}\ \mathsf{eval\text{-}expr}\ (\mathsf{lit}\ n \setminus n \to n \\
&\qquad\qquad \mathsf{var\text{-}inj}\ x \setminus x \to \mathsf{zero} \\
&\qquad\qquad \mathsf{plus}\ x\ y \setminus x \to \mathsf{add}\ (\mathsf{eval\text{-}expr}\ x)\ (\mathsf{eval\text{-}expr}\ y) \\
&\qquad\qquad \mathsf{mult}\ x\ y \setminus x \to \mathsf{multiply}\ (\mathsf{eval\text{-}expr}\ x)\ (\mathsf{eval\text{-}expr}\ y) \\
&\qquad\qquad)
\end{aligned}
$$

eval-expr of a literal is the literal, of an addition expression is the sum, and of a multiplication expression is the product. The value for a free variable is arbitrarily set as 0.

To extend eval-expr to the functions, the substitution function is first needed. This is defined as follows:

$$
\begin{aligned}
&\textbf{fun}\ \mathsf{subst}\ (z)\ (\nu\,\alpha\,.\,\mathsf{lit}\ n\ \langle\alpha\rangle \setminus n \to \mathsf{lit}\ (\mathsf{lift\text{-}nat}\ n) \\
&\qquad\qquad \nu\,\alpha\,.\,\mathsf{var\text{-}inj}\ x\ \langle\alpha\rangle \setminus x \to (\textbf{nfun}\ (\nu\,\alpha\,.\,\alpha \setminus \cdot \to z \mid \nu\,\alpha\,.\,\alpha' \setminus \alpha' \to \alpha'))\ x \\
&\qquad\qquad \nu\,\alpha\,.\,\mathsf{plus}\ x\ \langle\alpha\rangle\ y\ \langle\alpha\rangle \setminus x \to \mathsf{plus}\ (\mathsf{subst}\ z\ x)\ (\mathsf{subst}\ z\ y) \\
&\qquad\qquad \nu\,\alpha\,.\,\mathsf{mult}\ x\ \langle\alpha\rangle\ y\ \langle\alpha\rangle \setminus x \to \mathsf{mult}\ (\mathsf{subst}\ z\ x)\ (\mathsf{subst}\ z\ y) \\
&\qquad\qquad)
\end{aligned}
$$

subst substitutes the expression z into the binding $\nu\,\alpha\,.\,M$ by replacing var-inj α in M with z and removing the ν-abstraction. In a way this is like lift-nat, except names can occur in expressions, so these are replaced by z. subst of a literal just calls lift-nat. subst of a variable must determine whether the variable is α or not. This is done with a name-matching function, which behaves similarly to a standard pattern-matching function except names are matched instead of other terms. The plus and mult cases simply recurse on the subterms, pulling the plus and mult cases outside of the recursion.

We now write apply-fun, which substitutes a list of arguments into a function:

$$
\begin{array}{l}
\textbf{fun}\ \mathsf{apply\text{-}fun}\ (\mathsf{fun\text{-}one}\ E \setminus E \to \lambda\, l : \mathsf{list\ nat}\,.\,\mathsf{subst}\ (\mathsf{head}\ A\ x\ l)\ E \mid \\
\qquad \mathsf{fun\text{-}many}\ F \setminus F \to \\
\qquad\quad \lambda\, l : \mathsf{list\ nat}\,.\quad \mathsf{subst}\ (\mathsf{head}\ A\ x\ l)\ (\nu\, \alpha\,.\,\mathsf{apply\text{-}fun}\ F\ \langle \alpha \rangle\ (\mathsf{tail}\ A\ l)) \\
\qquad)
\end{array}
$$

apply-fun has type fun-expr $\Rightarrow$ (list nat) $\Rightarrow$ expr. If the scrutinee is fun-one E, apply-fun calls subst to substitute the head of the list argument into E. If the scrutinee is fun-many F, apply-fun recurses on F inside a ν-abstraction to produce an expr. In the recursion, tail l is passed as the list argument to ensure that the next argument in the list gets passed for the next variable in the function. apply-fun then calls subst to pass the head of the list to the ν-abstraction with the result of the recursive call. Putting all the pieces together, eval-expr (apply-fun F l) applies a function F to a list l of natural numbers and evaluates the resulting expression, yielding a natural number.

5.2 CNIC Formalized

In this section CNIC is formalized by giving the syntax of the term language as well as the operational and static semantics. One of the design goals of this formalization was to ensure that pattern-matching functions can always be lifted, meaning they can be brought to the top level. This implies that a pattern-matching function have no free names or variables. Any input must come directly via its parameters or its scrutinee. One exception is that recursive calls can still be made from inside a pattern-matching function to another pattern-matching function containing it. Lifting is known to be useful for implementing functional languages [33], but this design goal was more to enable the translation in Chapter 6. As vague as this may sound here, something about the translation seemed to require this to hold.

Constructors and variables representing recursive calls must thus be separated from the other variables and the names. This is done by defining two sorts of variable, one for recursive calls and one for normal variables, as well as two sorts of typing context, one for constructors and recursive calls and the other for names and normal variables.

$$
\begin{array}{lrl}
\text{Terms} & M ::= & \mathsf{Type}_i \parallel \Pi x{:}A\,.\,B \parallel \nabla\alpha\,.\,A \parallel a \parallel \mathsf{Name} \parallel u \parallel x \parallel c \parallel \alpha \\
& & \parallel \nu\,\alpha\,.\,M \parallel M\,\langle\alpha\rangle \parallel \lambda\,x{:}A\,.\,M \parallel M_1\,M_2 \\
& & \parallel \mathbf{fun}\;u\;(\vec{\alpha},\Gamma^{\mathrm{x}})\;(P_1^c \to M_1 \mid \ldots \mid P_n^c \to M_n) \\
& & \parallel \mathbf{nfun}\;(\vec{\alpha})\;(P_1^\alpha \to M_1 \mid \ldots \mid P_n^\alpha \to M_n) \\
c\text{-Patterns} & P^c ::= & \nu\,\vec{\alpha}\,.\,c\;x_1\;\langle\vec{\alpha}\rangle \ldots x_n\;\langle\vec{\alpha}\rangle \setminus x_1 : \nabla\vec{\alpha}\,.\,A_1, \ldots, x_n : \nabla\vec{\alpha}\,.\,A_n \\
\alpha\text{-Patterns} & P^\alpha ::= & \nu\,\vec{\alpha}\,.\,\alpha \setminus \cdot \parallel \nu\,\vec{\alpha}\,.\,\alpha \setminus \alpha \\
\text{Contexts} & \Gamma ::= & \Gamma, x : A \parallel \Gamma, \alpha \parallel \cdot \\
\text{Modal Contexts} & \Sigma ::= & \Sigma, c : A \parallel \Sigma, u : A \parallel \cdot \\
\text{Substitutions} & \sigma ::= & [M/x, \sigma] \parallel \cdot
\end{array}
$$

Figure 5.1: Syntax of CNIC

The variables for recursive calls are called *modal* variables here, and the contexts that assign types to constructors and modal variables are called modal contexts.

With this stipulation, the syntax of CNIC can be given in Figure 5.1. In it and the below, x, y, and z are used for variables, u for modal variables, α, β, and γ for names, c for constructors, a for type constructors, M and N for terms, A and B for terms intended to be types, P for patterns, Γ for contexts, Γ^{x} for *variable contexts*, or contexts only containing bindings for variables, and σ for substitutions. All of these may appear with subscripts or primes (as in M') to distinguish different elements of these classes.

Figure 5.1 defines the syntax of CNIC by giving a grammar for a number of syntactic categories. The terms have all been introduced and discussed above. Note that there is a small bureaucratic stipulation that the parameters in a pattern-matching function put the name parameters first and the term ones second. A further stipulation requires that the only parameters to name-matching functions be names. The c-patterns are patterns that match constructors. As discussed above, these can contain ν-abstractions, but all variables in the pattern must have a name replacement for all names bound in the pattern. The α-patterns are patterns for matching names. These can either match one of the name parameters of the function, one of the names bound in the pattern, or some other name external to all of these names. Normal contexts assign types to variables and list the names considered in scope. Modal contexts assign types to constructors and modal variables. Substitutions list terms to substitute for variables.

$$(\mathbf{fun}\ u\ (\vec{\alpha}, \Gamma^{\mathrm{x}})\ (\ldots \mid \nu\,\vec{\beta}\,.\,c_i\ \vec{x}\ \langle\vec{\beta}\rangle \setminus \vec{x} \to M_i \mid \ldots))\ \langle\vec{\alpha}\rangle\ \vec{N'}\ (\nu\,\vec{\beta}\,.\,c_i\ \vec{N}) \rightsquigarrow [\mathbf{fun}\ u\ (\vec{\alpha}, \Gamma^{\mathrm{x}})\ (\ldots \mid \nu\,\vec{\beta}\,.\,c_i\ \vec{x}\ \langle\vec{\beta}\rangle \setminus \vec{x} \to M_i \mid \ldots)/u, (\nu\,\vec{\beta}\,.\,\vec{N})/\vec{x}]M_i$$

$$\begin{array}{rcl}
(\mathbf{nfun}\ (\vec{\alpha})\ (\ldots \mid \nu\,\vec{\beta}\,.\,\alpha_i \setminus \cdot \to \vec{M_{\alpha_i}} \mid \ldots))\ \langle\vec{\alpha}\rangle\ (\nu\,\vec{\beta}\,.\,\alpha_i) & \rightsquigarrow & M_{\alpha_i} \\
(\mathbf{nfun}\ (\vec{\alpha})\ (\ldots \mid \nu\,\vec{\beta}\,.\,\beta_i \setminus \cdot \to \vec{M_{\beta_i}} \mid \ldots))\ \langle\vec{\alpha}\rangle\ (\nu\,\vec{\beta}\,.\,\alpha_i) & \rightsquigarrow & M_{\beta_i} \\
(\mathbf{nfun}\ (\vec{\alpha})\ (\ldots \mid \nu\,\vec{\beta}\,.\,\gamma \setminus \gamma \to M_\gamma))\ \langle\vec{\alpha}\rangle\ (\nu\,\vec{\beta}\,.\,\gamma) & \rightsquigarrow & M_\gamma \\
(\lambda\,x{:}A\,.\,M)\ N & \rightsquigarrow & [N/x]M \\
(\nu\,\alpha\,.\,M)\ \langle\alpha\rangle & \rightsquigarrow & M
\end{array}$$

Figure 5.2: Operational Semantics of CNIC

The operational semantics for CNIC is given by the rewrite system of Figure 5.2. This is an HNRS, as defined in Chapter 4. Further, note that it is orthogonal, since the there is trivially no non-trivial overlap, and the only constructs that are repeated on the left-hand side are variables. Thus this is a confluent HNRS. Note that this rewrite system could not be written without an HNRS, as the rules for reducing name-matching functions must determine if the name in the argument is equal to one of the α_i or not. The equality judgment $\vdash M = N$ is then defined as the reflexive-symmetric-transitive closure of this rewrite relation.

Typing in CNIC is given by the the typing judgment $\Sigma; \Gamma \vdash M : A$ which gives M type A relative to two contexts, a modal and a normal one. This judgment is defined in Figure 5.4. Many of these are straightforward. The first rule gives a term type B if it already has type A for some subtype A of B. Subtyping is as in CIC, and is repeated in Figure 5.3 for convenience. Type_i has type Type_{i+1}. $\Pi x : A\,.\,B$ has type Type_i if both A and B do, and it has type $\mathsf{Type}_{\mathcal{P}}$ if A is a type and B has type $\mathsf{Type}_{\mathcal{P}}$. $\nabla\alpha\,.\,A$ has type Type_i is A does as welll. Name is a Type_0. Constructors, type constructors, modal variabbles, and normal variables have the type given them by their appropriate context. Names α have type Name as long as they are in the normal context. $\nu\,\alpha\,.\,M$ has type $\nabla\alpha\,.\,A$ if M has type A in the context extended with α. $\lambda\,x{:}A\,.\,M$ has type $\Pi x{:}A\,.\,B$ if M has type B in the context extended with $x : A$. Applications $M\ N$ have type $[N/x]B$ if M has type $\Pi x{:}A\,.\,B$ and N has type A.

$$\frac{\vdash A = B}{\vdash A \lesssim B}\ \textsf{st-eq} \qquad \frac{i \leq j}{\vdash \mathsf{Type}_i \lesssim \mathsf{Type}_j}\ \textsf{st-type} \qquad \frac{\vdash B_1 \lesssim A_1 \quad \vdash A_2 \lesssim B_2}{\vdash \Pi x{:}A_1\,.\,A_2 \lesssim \Pi x{:}B_1\,.\,B_2}\ \textsf{st-pi}$$

Figure 5.3: Subtyping for CNIC

$$\frac{\Sigma;\Gamma \vdash M : A \quad \Sigma;\Gamma \vdash B : \mathsf{Type}_i \quad \vdash A \lesssim B}{\Sigma;\Gamma \vdash M : B}\ \textsf{t-subt} \qquad \frac{}{\Sigma;\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}}\ \textsf{t-type}$$

$$\frac{\Sigma;\Gamma \vdash A : \mathsf{Type}_i \quad \Sigma;\Gamma, x : A \vdash B : \mathsf{Type}_i}{\Sigma;\Gamma \vdash \Pi x{:}A\,.\,B : \mathsf{Type}_i}\ \textsf{t-pi-p} \qquad \frac{\Sigma;\Gamma \vdash A : \mathsf{Type}_i \quad \Sigma;\Gamma, x : A \vdash B : \mathsf{Type}_{\mathcal{P}}}{\Sigma;\Gamma \vdash \Pi x{:}A\,.\,B : \mathsf{Type}_{\mathcal{P}}}\ \textsf{t-pi-i}$$

$$\frac{\Sigma;\Gamma, c : A \vdash B : \mathsf{Type}_i}{\Sigma;\Gamma \vdash \nabla\alpha\,.\,A : \mathsf{Type}_i}\ \textsf{t-nabla} \qquad \frac{a : A \in \Sigma}{\Sigma;\Gamma \vdash a : A}\ \textsf{t-tctor} \qquad \frac{}{\Sigma;\Gamma \vdash \mathsf{Name} : \mathsf{Type}_0}\ \textsf{t-name}$$

$$\frac{x : A \in \Gamma}{\Sigma;\Gamma \vdash x : A}\ \textsf{t-var} \qquad \frac{u : A \in \Sigma}{\Sigma;\Gamma \vdash u : A}\ \textsf{t-mvar} \qquad \frac{c : A \in \Sigma}{\Sigma;\Gamma \vdash c : A}\ \textsf{t-ctor} \qquad \frac{\alpha \in \Gamma}{\Sigma;\Gamma \vdash \alpha : \mathsf{Name}}\ \textsf{t-alpha}$$

$$\frac{\Sigma;\Gamma, \alpha \vdash M : A}{\Sigma;\Gamma \vdash \nu\,\alpha\,.\,M : \nabla\alpha\,.\,A}\ \textsf{t-nu} \qquad \frac{\Sigma;\mathbf{remove}_\alpha(\Gamma) \vdash M : \nabla\alpha\,.\,A}{\Sigma;\Gamma \vdash M\ \langle\alpha\rangle : A}\ \textsf{t-namerepl}$$

$$\frac{\Sigma;\Gamma, x : A \vdash M : B}{\Sigma;\Gamma \vdash \lambda\,x{:}A\,.\,M : \Pi x{:}A\,.\,B}\ \textsf{t-lambda} \qquad \frac{\Sigma;\Gamma \vdash M : \Pi x{:}A\,.\,B \quad \Sigma;\Gamma \vdash N : A}{\Sigma;\Gamma \vdash M\ N : [N/x]B}\ \textsf{t-app}$$

$$\frac{\begin{array}{c}\Sigma;\cdot \vdash \nabla\vec{\alpha}\,.\,\Pi\Gamma^{\mathrm{x}}{\uparrow^{\vec{\beta}}}\ .\,\Pi x{:}(a\ \Gamma^{\mathrm{x}}){\uparrow^{\vec{\beta}}_{\nabla}}\ .\,B : \mathsf{Type}_i \qquad \forall i(\Sigma;\cdot \vdash c_i : \Pi\Gamma^{\mathrm{x}}_{c_i}\,.\,a\ \vec{M}_i) \\ \forall i(\Sigma, u : (\nabla\vec{\alpha}\,.\,\Pi\Gamma^{\mathrm{x}}{\uparrow^{\vec{\beta}}}\ .\,\Pi x{:}(a\ \Gamma^{\mathrm{x}}){\uparrow^{\vec{\beta}}_{\nabla}}\ .\,B)\ ;\ \vec{\alpha}, \Gamma^{\mathrm{x}}_i{\uparrow^{\vec{\beta}}} \vdash N_i : [(M_i){\uparrow^{\vec{\beta}}}\ /\Gamma^{\mathrm{x}}\ ,\ (c_i\ \Gamma^{\mathrm{x}}_i){\uparrow^{\vec{\beta}}}\ /x]B) \\ \Gamma^{\mathrm{x}} \text{ fully applied w.r.t. } \vec{\beta} \text{ in } B \qquad \forall i(\vdash \mathbf{app\text{-}check}_u(\Gamma^{\mathrm{x}}_i; N_i)) \qquad \Gamma \vdash \vec{c}\ \mathbf{covers}\ a\end{array}}{\Sigma;\Gamma \vdash \mathbf{fun}\ u\ (\vec{\alpha}, \Gamma^{\mathrm{x}}{\uparrow^{\vec{\beta}}})\ ((c\ \Gamma^{\mathrm{x}}_c){\uparrow^{\vec{\beta}}}\ \setminus \vec{\Gamma^{\mathrm{x}}_c}{\uparrow^{\vec{\beta}}} \to \vec{N}) : \nabla\vec{\alpha}\,.\,\Pi\Gamma^{\mathrm{x}}{\uparrow^{\vec{\beta}}}\,.\,\Pi x{:}((a\ \Gamma^{\mathrm{x}}){\uparrow^{\vec{\beta}}_{\nabla}})\,.\,B}\ \textsf{t-pmfun}$$

$$\frac{\begin{array}{c}\Sigma;\cdot \vdash \nabla\vec{\alpha}\,.\,\Pi x{:}(\nabla\vec{\beta}\,.\,\mathsf{Name})\,.\,B : \mathsf{Type}_i \\ \forall i(\Sigma;\vec{\alpha}, \Gamma^{\alpha}_i \vdash M_i : \nabla\vec{\beta}\,.\,\mathsf{Name}) \quad \forall i(\Sigma;\vec{\alpha}, \Gamma^{\alpha}_i \vdash N_i : [M_i/x]B)\end{array}}{\Sigma;\Gamma \vdash \mathbf{nfun}\ (\vec{\alpha})\ (\nu\,\vec{\beta}\,.\,\vec{M}_i \setminus \vec{\Gamma^{\alpha}} \to \vec{N}) : \nabla\vec{\alpha}\,.\,\Pi x{:}(\nabla\vec{\beta}\,.\,\mathsf{Name})\,.\,B}\ \textsf{t-nfun}$$

Figure 5.4: Typing for CNIC

To type the name replacement $M\ \langle c\rangle$, M must have type $\nabla\alpha\,.\,A$ in the context $\mathbf{remove}_\alpha(\Gamma)$. This context operation is defined as follows:

$$\begin{array}{lcl}\mathbf{remove}_\alpha(\Gamma, \alpha) & = & \Gamma \\ \mathbf{remove}_\alpha(\Gamma, \alpha') & = & \mathbf{remove}_\alpha(\Gamma, \alpha') \quad \text{if } \alpha \not\equiv \alpha' \\ \mathbf{remove}_\alpha(\Gamma, x : A) & = & \mathbf{remove}_\alpha(\Gamma)\end{array}$$

Intuitively, $\mathbf{remove}_\alpha(\Gamma)$ removes α from Γ. Any variables bound after α, however, could possibly be instantiated with a term containing α. Thus these variables must also be removed from Γ.

Typing the name-matching function $\mathbf{nfun}\ (\vec{\alpha})\ (\nu\,\vec{\beta}\,.\,\vec{M}_i \setminus \vec{\Gamma^{\alpha}} \to \vec{N})$ makes three requirements. First, the intended type, $\nabla\vec{\alpha}\,.\,\Pi x : (\nabla\vec{\beta}\,.\,\mathsf{Name})\,.\,B$, must be well-typed in the empty non-modal context. This is because, as discussed above, pattern- and

name-matching functions must be able to be lifted, meaning they must have no reliance on the non-modal context. The second requirement is that the pattern be well-typed in the non-modal context including only the name parameters and any name bound by the pattern. Finally, the return value M_i must have type $[M_i/x]B$, also in the non-modal context including only the name parameters and any name bound by the pattern.

To discuss the typing rule for pattern-matching functions, the notion of raising must be defined. The notation $M\uparrow^{\vec{\beta}}$ is called the *raising* of M by the names $\vec{\beta}$. This denotes the term $\nu\,\vec{\beta}\,.\,M'$ where M' is the result of replacing all free variables x in M with $x\,\langle\vec{\beta}\rangle$. Similarly, the notation $A\uparrow^{\vec{\beta}}_{\nabla}$ raises A as a type, and denotes the term $\nabla\vec{\beta}\,.\,A'$ where A' is again the result of replacing all free variables x in M with $x\,\langle\vec{\beta}\rangle$. Contexts can also be raised with the notation $\Gamma\uparrow^{\vec{\beta}}$. This operation replaces every pair $x : A$ in Γ by $x : A\uparrow^{\vec{\beta}}_{\nabla}$. It is then straightforward to see that $\Sigma;\Gamma \vdash M : A$ implies $\Sigma;\Gamma\uparrow^{\vec{\beta}} \vdash M\uparrow^{\vec{\beta}} : A\uparrow^{\vec{\beta}}_{\nabla}$.

Returning to the typing rule for pattern-matching functions, six requirements are made. First, the intended type must be well-typed in the empty non-modal context. This is again per the requirement that pattern- and name-matching functions be able to be lifted. In general, a pattern-matching function can match inside ν-abstractions binding some names β. Thus the scrutinee type is $(a\ \Gamma^{\mathrm{x}})\uparrow^{\vec{\beta}}_{\nabla}$. By the above discussion about raising, it then makes sense to require the context of parameters to be $\Gamma^{\mathrm{x}}\uparrow^{\vec{\beta}}$. Thus the expected type for the whole pattern-matching function is $\nabla\vec{\alpha}\,.\,\Pi\Gamma^{\mathrm{x}}\uparrow^{\vec{\beta}}\,.\,\Pi x{:}\,(a\ \Gamma^{\mathrm{x}})\uparrow^{\vec{\beta}}_{\nabla}\,.\,B$ for some B. The second requirement ensures that the constructor c_i in each pattern has type $\Pi\Gamma^{\mathrm{x}}_{c_i}\,.\,a\ \vec{M}_i$ for some $\vec{M}_i$. The third requirement checks that each return value N_i has the appropriate instance of the type B in the modal context extended with the modal variable u for recursive calls and the non-modal context containing just the names $\vec{\alpha}$ and the lifted pattern variables for the c_i pattern. The fourth requirement, that the variables in Γ^{x} be fully applied with respect to $\vec{\beta}$ in B. This is a technical condition which requires that these variables always occur as $x\,\langle\vec{\gamma}\rangle$ for some names γ. The fifth requirement, $\vdash \mathbf{app\text{-}check}_u(\Gamma^{\mathrm{x}}_i; N_i)$, is just notation for stating that u is only applied to terms that are structurally smaller than $c_i\ \Gamma^{\mathrm{x}}_i$ in N_i, while the sixth requirement, $\Gamma \vdash \vec{c}\ \mathbf{covers}\ a$, states that the constructors $\vec{c}$ must be all of the constructors for a.

The typing judgment $\Sigma; \Gamma \vdash M : A$ also implicitly assumes that Σ and Γ are well-formed. The judgment $\vdash \Sigma$ states that Σ is a well-formed modal context. The rules for this judgment are as follows:

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \Sigma \quad \Sigma; \cdot \vdash A : \mathsf{Type}_i}{\vdash \Sigma, u : A}$$

$$\frac{\vdash \Sigma \quad \Sigma; \cdot \vdash A : \mathsf{Type}_i \quad A \equiv \Pi x_1 : A_1 \,.\, \ldots \Pi x_n : A_n \,.\, \mathsf{Type}_j}{\vdash \Sigma, a : A}$$

$$\frac{\vdash \Sigma \quad \Sigma; \cdot \vdash A : \mathsf{Type}_i \quad A \equiv \Pi x_1 : B_1 \,.\, \ldots \Pi x_n : B_n \,.\, a\ \vec{M}}{\vdash \Sigma, c : A}$$

These rules ensure that the types associated with each modal variable u, each type constructor a, and each object constructor c are all well-typed in the preceeding signature. Type constructors are also required to have a type that makes them a function of zero or more arguments to a type, and object constructors must be functions of zero or more arguments to an element of a previously defined inductive type. The judgment $\vdash \Sigma$ also implicitly includes standard side conditions on the formation of inductive types, ensuring that the A_i in the type of a are not of a higher level than that of a, and that the B_i in the type of c only contain a strictly positively. For more precise definitions of these conditions, the rules for the Coq system may be consulted [68, 50].

The well-formedness judgment $\Sigma \vdash \Gamma$ for normal contexts is very similar to that for modal contexts. This judgment is given by the rules

$$\frac{}{\Sigma \vdash \cdot} \qquad \frac{\Sigma \vdash \Gamma \quad \Sigma; \Gamma \vdash A : \mathsf{Type}_i}{\Sigma \vdash \Gamma, x : A}$$

$$\frac{\Sigma \vdash \Gamma}{\Sigma \vdash \Gamma, \alpha}$$

These rules simply ensure that all types in a context are well-typed as types.

5.3 Metatheory of CNIC

In this section the metatheory of CNIC is developed, including Subect Reduction and Canonical Forms. Subject Reduction states that computation does not alter the type of a term. Canonical Forms states that objects of type $a\ \vec{M}$ for some $\vec{M}$ must be built from a constructor. This allows adequacy to be proved.

To prove Canonical Forms, the following lemma is needed:

Lemma 5.3.1. *If the forms of the types A and B are different elements of the list $a\ \vec{M}$, $\Pi x{:}A_1\,.\,A_2$, $\nabla\alpha\,.\,A_1$, Type_i, and Name, then neither of A or B can be a subtype of the other.*

Proof. The only way a type of one of these forms could be a subtype of a type of another of these forms is through the equality st-eq. By confluence, however, no two of these forms can be equal, as it is straightforward that none of them share a common reduct. □

Canonical Forms is then straightforward:

Lemma 5.3.2 (Canonical Forms). *If $\Sigma;\cdot \vdash M : a\ \vec{N}$ for some Σ with no modal variables and some M in normal form with respect to reduction in* CNIC*, then M is of the form $c\ \vec{M'}$ for some constructor c of a and some terms M'.*

Proof. We consider the typing rules that can possibly yield the type $a\ \vec{N}$ for a term. t-subt cannot change the form of the type. t-var is impossible as the non-modal signature is empty. t-mvar is impossible as there are no modal variables in Σ. Thus M must either be a constructor or a term of the form $M'\ \vec{R}$ for some arguments and/or name replacements R with M' not an application or a constructor replacment. If M' is a constructor then again we are finished. M' cannot be a λ- or ν-abstraction, as then M would not be in normal form. Further, M' cannot be a variable or modal variable, as there are no such variables in scope. Thus M must be of the required form. □

Proving Subject Reduction requires a number of lemmas. The first of these are Weakening and Substitution:

Lemma 5.3.3 (Weakening). *If* $\Sigma; \Gamma \vdash M : A$ *then* $\Sigma; \Gamma' \vdash M : A$ *for any context* Γ' *containing all the variables and names of* Γ *and satisfying the property that if* x *comes before* α *in* Γ *then it also does in* Γ'*.*

Proof. By induction on the typing judgment $\Sigma; \Gamma \vdash M : A$. The result is mostly straightforward, and so is omitted here. Note, however, that the side condition about ordering is needed for the t-namerepl case to show that if $x : A$ is in $\mathbf{remove}_\alpha(\Gamma)$ then it is also in $\mathbf{remove}_\alpha(\Gamma')$. This is because if $x : A$ is in $\mathbf{remove}_\alpha(\Gamma)$ then it must come before α in Γ, so the side condition ensures that it must also come before α in Γ' and is therefore in $\mathbf{remove}_\alpha(\Gamma')$. □

Lemma 5.3.4 (Substitution). *If* $\Sigma; \Gamma_1 \vdash M : A$ *and* $\Sigma; \Gamma_1, x : A, \Gamma_2 \vdash N : B$ *then* $\Sigma; \Gamma_1, \Gamma_2 \vdash [M/x]N : [M/x]B$*.*

Proof. By induction on the typing judgment $\Sigma; \Gamma \vdash M : A$. The result is straightforward, and so is omitted here. □

Lemma 5.3.5 (Modal Substitution). *If* $\Sigma; \Gamma_1 \vdash M : A$ *and* $\Sigma; \Gamma_1, u : A, \Gamma_2 \vdash N : B$ *then* $\Sigma; \Gamma_1, \Gamma_2 \vdash [M/u]N : [M/u]B$*.*

Proof. By induction on the typing judgment $\Sigma; \Gamma \vdash M : A$. Again, the result is straightforward and similar to normal Substitution, and so is omitted here. □

Lemma 5.3.6. *If* $\vec{x}$ *include all the free variables of* M*, then* $[\nu\,\vec{\beta}\,.\,\vec{N}/\vec{x}](M{\uparrow}^{\vec{\beta}})$ *is equal to* $\nu\,\vec{\beta}\,.\,M$*.*

Proof. By straightforward induction on M. □

Lemma 5.3.7 (Preservation). *If* $\Gamma \vdash M : A$ *and* $M \rightsquigarrow N$*, then* $\Gamma \vdash N : A$*.*

Proof. Proof is by induction on $\mathcal{D}$, the proof of $\Gamma \vdash M : A$. If the rewrite $M \rightsquigarrow N$ happens in a strict subterm of M, then in most cases the result follows directly from an invocation of the induction hypothesis, possibly with an extra application of the t-subt rule in the case that M is an application and the rewrite happens in the argument of M. Thus we consider only cases where M itself is the redex being reduced.

Case:

$$\mathcal{D} = \dfrac{\dfrac{\Sigma; \mathbf{remove}_\alpha(\Gamma), \alpha \vdash M : A}{\Sigma; \mathbf{remove}_\alpha(\Gamma) \vdash \nu\, \alpha \,.\, M : \nabla \alpha \,.\, A}\ \textsf{t-nu}}{\Sigma; \Gamma \vdash \nu\, \alpha \,.\, M\ \langle \alpha \rangle : A :}\ \textsf{t-namerepl}$$

In this case $(\nu\, \alpha \,.\, M)\ \langle \alpha \rangle \rightsquigarrow M$, and so it must be shown that $\Sigma; \Gamma \vdash M : A$. Note that Γ has all the variables and names as $\mathbf{remove}_\alpha(\Gamma), \alpha$, as the latter is got from the former by removing variables and moving α to the end. Further, any variable x before name β in $\mathbf{remove}_\alpha(\Gamma), \alpha$ is still before β in Γ, and any variable x before α in $\mathbf{remove}_\alpha(\Gamma), \alpha$ must be before α in Γ or it would not be in $\mathbf{remove}_\alpha(\Gamma)$. Thus Weakening can be used on the proof of $\Sigma; \mathbf{remove}_\alpha(\Gamma), \alpha \vdash M : A$ to achieve the desired result.

Case:

$$\mathcal{D} = \dfrac{\Sigma; \Gamma \vdash N : A \quad \dfrac{\Sigma; \Gamma, x : A \vdash M : B}{\Sigma; \Gamma \vdash \lambda\, x{:}A \,.\, M : \Pi x{:}A \,.\, B}\ \textsf{t-lambda}}{\Sigma; \Gamma \vdash (\lambda\, x{:}A \,.\, M)\ N : [N/x]B}\ \textsf{t-app}$$

In this case $(\lambda\, x{:}A \,.\, M)\ N \rightsquigarrow [N/x]M$. The result is immediate by Substitution.

Case:

$$\mathcal{D} = \dfrac{\dfrac{\begin{array}{c}\Sigma; \cdot \vdash \nabla \vec{\alpha} \,.\, \Pi x{:}(\nabla \vec{\beta} \,.\, \mathsf{Name}) \,.\, B : \mathsf{Type}_i \\ \forall i (\Sigma; \vec{\alpha}, \Gamma_i^\alpha \vdash M_i : \nabla \vec{\beta} \,.\, \mathsf{Name}) \quad \forall i (\Sigma; \vec{\alpha}, \Gamma_i^\alpha \vdash N_i : [M_i/x]B)\end{array}}{\Sigma; \mathbf{remove}_{\vec{\alpha}}(\Gamma) \vdash \mathbf{nfun}\ (\vec{\alpha})\ (\nu\, \vec{\beta} \,.\, \vec{M_i} \setminus \vec{\Gamma^\alpha} \to \vec{N}) : \nabla \vec{\alpha} \,.\, \Pi x{:}(\nabla \vec{\beta} \,.\, \mathsf{Name}) \,.\, B}\ \textsf{t-nfun}}{\vdots}}{\Sigma; \Gamma \vdash (\mathbf{nfun}\ (\vec{\alpha})\ (\nu\, \vec{\beta} \,.\, \vec{M_i} \setminus \vec{\Gamma^\alpha} \to \vec{N}))\ \langle \vec{\alpha} \rangle\ (\nu\, \vec{\beta} \,.\, \gamma) : [\nu\, \vec{\beta} \,.\, \gamma/x]B}\ \textsf{t-app}$$

In this case $(\mathbf{nfun}\ (\vec{\alpha})\ (\nu\, \vec{\beta} \,.\, \vec{M_i} \setminus \vec{\Gamma^\alpha} \to \vec{N}))\ \langle \vec{\alpha} \rangle\ (\nu\, \vec{\beta} \,.\, \gamma) \rightsquigarrow N_\gamma$. It follows immediately that $\Sigma; \vec{\alpha}, \gamma \vdash N_\gamma : [\nu\, \vec{\beta} \,.\, \gamma/x]B$ by the typing requirement for return values in name-matching functions, and Weakening then yields the desired result. Note that the other cases of evaluating a name-matching function are similar, so they are omitted.

Case:

$$\frac{\begin{array}{c}\Sigma;\cdot\vdash\nabla\vec{\alpha}\,.\,\Pi\Gamma^{\mathrm{x}}{\uparrow}^{\vec{\beta}}\;.\,\Pi x{:}(a\;\Gamma^{\mathrm{x}}){\uparrow}^{\vec{\beta}}_{\nabla}\;.\,B:\mathsf{Type}_i\qquad\forall i(\Sigma;\cdot\vdash c_i:\Pi\Gamma^{\mathrm{x}}_{c_i}\,.\,a\;\vec{Q}_i)\\ \forall i(\Sigma,u:(\nabla\vec{\alpha}\,.\,\Pi\Gamma^{\mathrm{x}}{\uparrow}^{\vec{\beta}}\;.\,\Pi x{:}(a\;\Gamma^{\mathrm{x}}){\uparrow}^{\vec{\beta}}_{\nabla}\;.\,B)\;;\;\vec{\alpha},\Gamma^{\mathrm{x}}_i{\uparrow}^{\vec{\beta}}\vdash M_i:[(Q_i){\uparrow}^{\vec{\beta}}\,/\Gamma^{\mathrm{x}}\;,\;(c_i\;\Gamma^{\mathrm{x}}_i){\uparrow}^{\vec{\beta}}\,/x]B)\\ \Gamma^{\mathrm{x}}\text{ fully applied w.r.t. }\vec{\beta}\text{ in }B\qquad\forall i(\vdash\mathbf{app\text{-}check}_u(\Gamma^{\mathrm{x}}_i;M_i))\qquad\Gamma\vdash\vec{c}\;\mathbf{covers}\;a\end{array}}{\Sigma;\Gamma\vdash\mathbf{fun}\;u\;(\vec{\alpha},\Gamma^{\mathrm{x}}{\uparrow}^{\vec{\beta}})\;((c\;\Gamma^{\mathrm{x}}_c){\uparrow}^{\vec{\beta}}\;\setminus\vec{\Gamma^{\mathrm{x}}_c}{\uparrow}^{\vec{\beta}}{\to}\;\vec{M}):\nabla\vec{\alpha}\,.\,\Pi\Gamma^{\mathrm{x}}{\uparrow}^{\vec{\beta}}.\,\Pi x{:}((a\;\Gamma^{\mathrm{x}}){\uparrow}^{\vec{\beta}}_{\nabla})\,.\,B}\;\textsf{t-pmfun}$$

$$\vdots$$

$$\Sigma;\Gamma\vdash(\mathbf{fun}\;u\;(\vec{\alpha},\Gamma^{\mathrm{x}}{\uparrow}^{\vec{\beta}})\;((c\;\Gamma^{\mathrm{x}}_c){\uparrow}^{\vec{\beta}}\;\setminus\vec{\Gamma^{\mathrm{x}}_c}{\uparrow}^{\vec{\beta}}{\to}\;\vec{M})\;\langle\vec{\alpha}\rangle\;\vec{N}'\;(\nu\,\vec{\beta}\,.\,c_i\;\vec{N}))\,:\,[(\nu\,\vec{\beta}\,.\,\vec{N}')/\Gamma^{\mathrm{x}},(\nu\,\vec{\beta}\,.\,c_i\;\vec{N})/x]B$$

In this case the following holds:

$$\begin{array}{c}(\mathbf{fun}\;u\;(\vec{\alpha},\Gamma^{\mathrm{x}})\;(\ldots\mid\nu\,\vec{\beta}\,.\,c_i\;\Gamma^{\mathrm{x}}_{c_i}\;\langle\vec{\beta}\rangle\setminus\Gamma^{\mathrm{x}}_{c_i}\to M_i\mid\ldots))\;\langle\vec{\alpha}\rangle\;\vec{N}'\;(\nu\,\vec{\beta}\,.\,c_i\;\vec{N})\\ \rightsquigarrow\\ [\mathbf{fun}\;u\;(\vec{\alpha},\Gamma^{\mathrm{x}})\;(\ldots\mid\nu\,\vec{\beta}\,.\,c_i\;\Gamma^{\mathrm{x}}_{c_i}\;\langle\vec{\beta}\rangle\setminus\Gamma^{\mathrm{x}}_{c_i}\to M_i\mid\ldots)/u,(\nu\,\vec{\beta}\,.\,\vec{N})/\Gamma^{\mathrm{x}}_{c_i}]M_i\end{array}$$

Since c_i has type $\Pi\Gamma^{\mathrm{x}}_{c_i}\,.\,a\;\vec{Q}_i$ and $(\nu\,\vec{\beta}\,.\,c_i\;\vec{N})$ has type $\nabla\vec{\beta}\,.\,a\;(\vec{N}'\;\langle\vec{\beta}\rangle)$ it is apparent that $[\vec{N}/\Gamma^{\mathrm{x}}_{c_i}]\vec{Q}_i$ equals $\vec{N}'\;\langle\vec{\beta}\rangle$. Further, M_i has type $[(Q_i){\uparrow}^{\vec{\beta}}\,/\Gamma^{\mathrm{x}}\;,\;(c_i\;\Gamma^{\mathrm{x}}_i){\uparrow}^{\vec{\beta}}\,/x]B$. Letting F be the whole pattern-matching function above, we thus have

$$\begin{array}{rcl}[F/u,(\nu\,\vec{\beta}\,.\,\vec{N})/\Gamma^{\mathrm{x}}_{c_i}]M_i&:&[F/u,(\nu\,\vec{\beta}\,.\,\vec{N})/\Gamma^{\mathrm{x}}_{c_i}]([(Q_i){\uparrow}^{\vec{\beta}}\,/\Gamma^{\mathrm{x}}\;,\;(c_i\;\Gamma^{\mathrm{x}}_{c_i}){\uparrow}^{\vec{\beta}}\,/x]B)\\ &=&[F/u,(\nu\,\vec{\beta}\,.\,\vec{N}'\;\langle\vec{\beta}\rangle)/\Gamma^{\mathrm{x}},\nu\,\vec{\beta}\,.\,c_i\;\vec{N}/x]B\\ &=&[\vec{N}'/\Gamma^{\mathrm{x}},\nu\,\vec{\beta}\,.\,c_i\;\vec{N}/x]B\end{array}$$

where the first line follows by Substitution, the second line follows from Lemma 5.3.6, and the third follows because u cannot be free in B and because every instance of a variable in Γ^{x} is required to be fully applied with respect to $\vec{\beta}$ in B.

□

Chapter 6

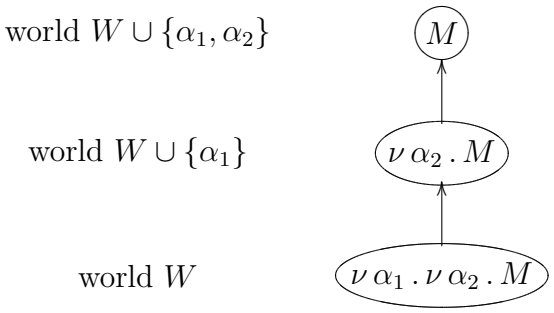
Consistency of CNIC

In this chapter, CNIC is proved consistent and strongly normalizing. This is done with a translation from well-typed terms of CNIC to well-typed terms of some other theory, called the *target* theory. Here, the target theory is an extension of CIC, called CIC + $\mathbb{T}$, which is introduced below. CIC + $\mathbb{T}$ is the combination of the Calculus of Inductive Constructions with the category $\mathbb{T}$, defined below. The result of the translation is that, if the contradictory type $\Pi A : \mathsf{Type}_i \,.\, A$ is inhabited by some term M in CNIC (and thus the contradictory proposition is provable), then the translation of M will prove a contradiction in the target theory. If the target theory is known to be consistent, then this will not be possible, and CNIC must be consistent as well. Further, the translation will also be shown to preserve reduction, so that an infinite reduction sequence starting from M in CNIC will result in an infinite reduction sequence starting from the translation of M in the target theory. If the target theory is strongly normalizing, then this is not possible, and so CNIC must be strongly normalizing as well.

The translation given in this chapter also serves as a denotational semantics for CNIC. A *denotational semantics* of a language is a model of the language; i.e. , if term M evaluates to value V in the language, then the denotational semantics of the two are the same. A denotational semantics is also required to be *compositional*, meaning that the denotational semantics of a term should be built from the denotational semantics for each of its subterms. Thus a denotational semantics gives a *meaning* or *interpretation* of every term of the language, and this meaning is built from the meanings of the subterms. See any standard reference (such as John Mitchell's book [47]) for more on denotational semantics. Here, instead of using set theory or category

theory to describe the meanings of terms, the target language CIC + $\mathbb{T}$ is used instead. A set-theoretic or category-theoretic model for CNIC, such as the well-known proof-irrelevant model [46], can then be obtained from models of CIC + $\mathbb{T}$.

We turn now to a high-level motivation and description of the translation. The translation is parameterized by a *world*, or set of names. Translation is said to take *in* some given world. Intuitively, the world specifies the names that have already been bound by ν-abstractions and other constructs. As an example, the translation of the term $\nu\,\alpha_1\,.\,\nu\,\alpha_2\,.\,M$ in some world W will use the translation of M in world $W \cup \{\alpha_1, \alpha_2\}$. Pictorially, we can view the previous sentence as follows:



where the arrows denote set inclusion of worlds.

The reason the notion of world is included in the translation is because it is necessary to define the meanings of ν- and ∇-abstractions. Specifically, the behavior of $\nu\,\alpha\,.\,M$ is to bind a new name α in M, where α is distinct from all other names that have previously been bound. Stated differently, α must be distinct from all other names in the world. Thus ν- and ∇-abstractions act as quantifiers (like $\forall$ and $\exists$) whose domain of quantification is the complement of the given world. Any attempt to define a meaning for ν- and ∇-abstractions, therefore, will need to refer to worlds. This is in fact a central difficulty with formalizing name binding, that the meaning of name binding relies on implicit information which is dependent on the context of the term inside a superterm.

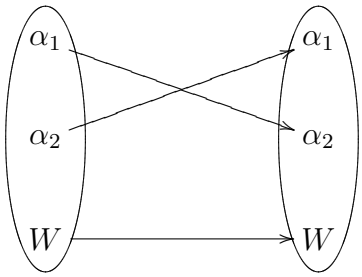
Evaluation can *move* a term to a different world. For example, consider the term $(\lambda\,x : A\,.\,\nu\,\alpha\,.\,x)\;M$. If this term is translated in world W, then so is M. One step of reduction leads to the term $\nu\,\alpha\,.\,M$, which, if translated again in W, contains a translation of M in world $W \cup \{\alpha\}$. Evaluation can also shuffle the names in a term through name replacements and pattern-matching under ν-abstractions. As an

example of the first of these, consider the term

$$\nu\,\alpha_1\,.\,\nu\,\alpha_2\,.\,(\nu\,\alpha_1\,.\,\nu\,\alpha_2\,.\,M)\;\langle\alpha_2,\alpha_1\rangle$$

Two steps of evaluating name replacements yields the term $\nu\,\alpha_1\,.\,\nu\,\alpha_2\,.\,M'$ where M' is the result of replacing free occurrences of α_1 with α_2 and vice versa. This sort of shuffling can be seen as a function mapping the names in the world of M, which here is $W \cup \{\alpha_1, \alpha_2\}$. In this case, this mapping maps back to the same world, $W \cup \{\alpha_1, \alpha_2\}$. Evaluation cannot remove names from the world of a term, however, as this might cause some terms to become invalid.

In general, a term can be moved by evaluation from world W to world W' via a *renaming function*, or *mapping* for short, from W to W'. The renaming functions are defined precisely in Section 6.1 as the morphisms of the category $\mathbb{T}$. We note here only that renaming functions must be injective, meaning distinct names in W must be mapped to distinct names in W'. As an example of a renaming function, the previous paragraph demonstrated a situation in which a term can be moved from the world $W \cup \{\alpha_1, \alpha_2\}$ to itself via the renaming function that is the identity on W and that maps α_1 and α_2 to each other. Graphically, this renaming function looks like

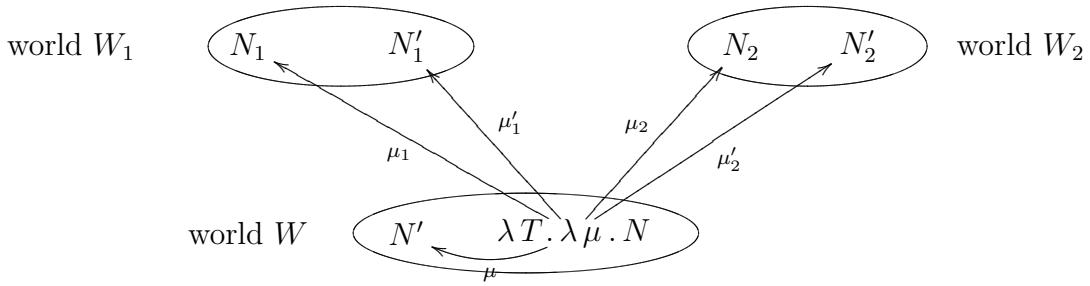


where the line from W to itself represents the identity on W.

To incorporate renaming functions into the translation, CNIC terms are translated to functions (in the target language CIC + $\mathbb{T}$) that take a renaming function, or mapping, as argument. Specifically, the translation in world W produces terms of the form $\lambda\,T\,.\,\lambda\,\mu : ((|W| \rightrightarrows T))\,.\,N$ where the first argument, T, specifies the *type* of the destination world, and the second argument, μ, specifies a mapping from the type $|W|$ of world W to some world of type T. Thus mappings are actually defined on collections of source and target worlds which all have the same world type. We leave

these notions vague here, deferring a precise definition of worlds and world types to Section 6.1 below.

The function $\lambda T.\,\lambda \mu\,.\,N$ produced by translating M in world W is intended to be viewed as a set of terms, one for each mapping μ with source W. Intuitively, each term in the set is a "version" of M after having been moved from W by the given mapping μ. For example, if μ, μ_1, μ_1', μ_2, and μ_2' are mappings from world W to worlds W, W_1, W_1, W_2, and W_2, respectively, then $\lambda T.\,\lambda \mu\,.\,N$ applied to each of these mappings results in a term in the given world that is in the term set represented by $\lambda T.\,\lambda \mu\,.\,N$. Graphically, this can be visualized as:

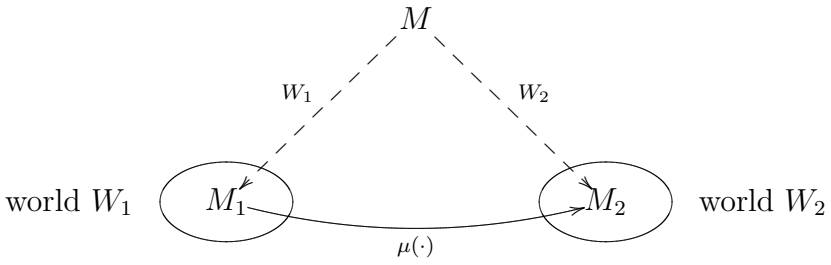


where all of the terms displayed are in the term set defined by $\lambda T.\,\lambda \mu\,.\,N$.

Since terms translate as sets, it is necessary to define the moving operation on these whole sets. The moving operation cannot just be the application of a $\lambda T.\,\lambda \mu\,.\,N$ function to a mapping, as the result could not be moved *again*. To move a $\lambda T.\,\lambda \mu\,.\,N$ term set, the operation $\mu(x)$ is used. This is called the *renaming* of x. $\mu(x)$ represents the moving of the term set by the mapping μ. $\mu(x)$ is defined as $\lambda T_2\,.\,\lambda \mu_2 : ((T \rightrightarrows T_2))\,.\,x\ T_2\ (\mu_2 \circ \mu)$, where $\circ$ is the composition of renaming functions. This operation forms another term set that takes argument T_2 and μ_2, where μ_2 is a further renaming function from worlds of type T to worlds of some type T_2. μ_2 is then composed with μ and passed to x, yielding the same result as if μ_2 and μ had been composed and passed to x without forming the renaming of x. Renaming may also be viewed as a subset operation; if F is a function that defines a term set, then $\mu(F)$ is the subset of F containing only those return values of F that can be attained by passing $\mu' \circ \mu$ to F for some μ'. This is because $\mu(F)\ T'\ \mu'$ equals $F\ T'\ (\mu' \circ \mu)$.

An important property of renaming is that, if μ maps world W_1 to W_2, then $\mu(\cdot)$ applied to the translation of any M in W_1 is equal to the translation of M in W_2.

Graphically, this can be pictured as follows:



where the dashed lines labeled with worlds represent translation in the given world. The property described here is called re-translation, and is proved as Lemma 6.2.5 in Section 6.2. Stated differently, re-translation says that moving the translation of a term does in fact yield the correct re-translation of the term in the new world. Re-translation turned out to be an important consideration that shaped the work on the translation given here. This is discused more in Section 6.2.

The discussion here is concluded here with a high-level description of how the translation is defined for some of the constructs of CNIC, as this information shaped the definition of the category $\mathbb{T}$ given in Section 6.1. The construct with the translation that is conceptually most straightforward is the name α. A name is translated in world W as a renaming function from the world $\{\alpha\}$ of exactly one name to the world W. Such a function picks out exactly one name from W. Thus the names in W can be equated with the mappings from the world of exactly one name to W, in a similar fashion to the way morphisms whose domain is a singleton set are equated with the elements of a set in the category Set of all sets. It will turn out here, however, that the world of exactly one name is not a terminal object, unlike singleton sets in Set, which are. See any standard reference, such as Pierce [58], for more on the category Set and similar constructions.

ν-abstractions are translated to terms of the form $\lambda T\,.\,\lambda\,\mu\,.\,\lambda\,T_2\,.\,\lambda\,\mu_2\,.\,M$ that take two mappings. The first mapping, μ, moves the translated term set from the world W of the translation to some world of type T. This is as described in the preceeding paragraphs. The second mapping, μ_2, is a mapping from worlds of type T to worlds of type T_2 with one name that has been removed or canceled out. Intuitively, such a world represents a world with a "hole" that is waiting for some new name to fill the hole. The body M of the translated term set then introduces a new name for the

ν-abstraction and uses it to fill the hole in the destination world that is the range of μ_2.

The reason that the translations of ν-abstractions take two mappings is to allow for name replacements, the elimination form for ν-abstractions. The translation of a name replacement $N\ \langle\alpha\rangle$ in world W translates the subterm N in the world resulting from canceling out the name α in W. This is because part of the meaning of the name replacement $N\ \langle\alpha\rangle$ is that α is fresh for N. The translation for ν-abstractions then dictates that the translation of N will create a new name α' which will eventually be used to fill the hole created by canceling out α in W. Any use of α', therefore, will be mapped by the filling action described above to a use of α in W, achieving the desired result that a name replacement eliminates a ν-abstraction.

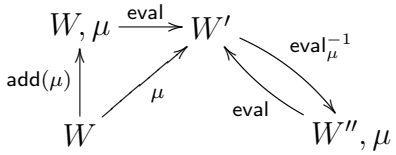
λ-abstractions were one of the more difficult constructs to translate. Much of the reason $\mathbb{T}$ was defined as it is was to enable the translation of λ-abstractions. The translation $\lambda\, x\!:\!A\,.\,M$ yields a term of the form $\lambda\, T\,.\,\lambda\,\mu\,.\,F$, where F is a function that takes translations of CNIC terms of type A as input and returns the translation of the body M. The difficulty with λ-abstractions is how to define the moving operation; i.e. , it must be defined how the mapping μ is applied to the translation of the body M. The standard approach to defining operations on functions, as given for instance in the work of Meijer and Hutton [42], is to define a new function from the old one that undoes the given opertion on the argument, passes the argument to the function, and then re-performs the operation on the result. Graphically, this can be conveyed as follows:

$$\begin{array}{ccc} \text{in} & & \text{out} \\ {\scriptstyle \text{op}^{-1}}\downarrow & & \uparrow{\scriptstyle \text{op}} \\ \text{op}^{-1}(\text{in}) & \xrightarrow[F]{} & \text{op}^{-1}(\text{out}) \end{array}$$

where op is the operation being performed and F is the given function.

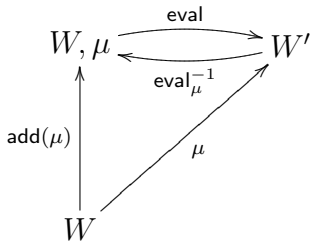
Unfortunately, it is not in general possible to undo renaming functions. For instance, the inverse of a renaming function that adds names would necessarily remove names, which is disallowed. This problem is surmounted by enriching the category $\mathbb{T}$ as follows. First, worlds are extended to also include mappings as elements. More specifically, if μ is a mapping from W to some W' then W, μ is also a world. This world is similar to a quoted or boxed program expression, waiting to be evaluated.

Three mapping constructs are then added to operate on worlds with mappings. The eval mapping performs the evaluation of quoted expressions W, μ, so, for example, eval maps W, μ to W'. The mapping $\mathsf{add}(\mu)$ adds the mapping μ to a world, and so maps W to W, μ. Finally, the mapping eval_μ^{-1} acts as a limited form of inverse for μ. eval_μ^{-1} maps from W' to some world W'', μ such that μ maps W'' to W'. Thus eval_μ^{-1} is a right inverse to eval, meaning that $\mathsf{eval} \circ \mathsf{eval}_\mu^{-1} = \mathsf{id}$, the identity mapping. These definitions yield the diagram



where the given mappings commute.

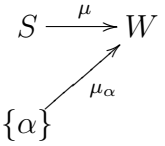
The original intent of these mappings was to have eval_μ^{-1} be a more direct "undoing" of μ; i.e. , the intent was to have W'' be the same as W, yielding a second diagram:



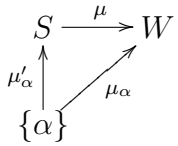
This is not possible in general, however. To see this, take any two worlds W_1 and W_2 of the same type along with mappings μ_1 and μ_2 on these worlds such that μ_i maps W_i to W' for $i \in \{1, 2\}$. It is thus the case that eval maps W_i, μ_i to W', so, taking μ in the above diagram to be eval, it must be the case that $\mathsf{eval}_{\mathsf{eval}}^{-1}$ maps W' to both W_1, μ_1 and to W_2, μ_2. If the W_i or the μ_i are distinct, a contradiction results.

The first diagram above is enough, however, to translate λ-abstractions. If μ is the mapping argument to the translation of a λ-abstraction, then eval_μ^{-1} is applied to the argument, moving it to some world W'', μ. The rest of the body of the λ-abstraction is translated in world W, μ. The moved argument is then substituted for x in the moved body, and the entire result is moved with the mapping eval. This moves all pieces to the intended world W'.

The second diagram above is needed to translate name-matching functions. A name-matching function takes a name α and a set of names S, all of which are in the current world W, and tests if α is in S. Rephrased in the language of $\mathbb{T}$, a name in W is a mapping μ_α from the singleton world of one name to W. Similarly, a set of names S in W is a mapping μ from the set S to W. Graphically, this can be displayed as follows:

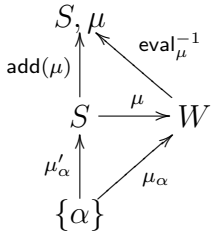


The name is then in the set if and only if μ_α picks out a name in W to which μ maps one of the names in S, which holds in turn if and only if μ_α is equal to $\mu \circ \mu'_\alpha$ for some μ'_α that maps the singleton world to S. This situation becomes the picture



where the given arrows commute.

To test whether the name given by μ_α is in the set specified by μ, eval_μ^{-1} can be composed with μ_α. The result is a mapping from the singleton world of one name to the world S, μ. By the second diagram for eval_μ^{-1} above, which does in fact hold when the domain of μ is a set of names with no mappings or holes, if μ_α is in S then the following diagram commutes:

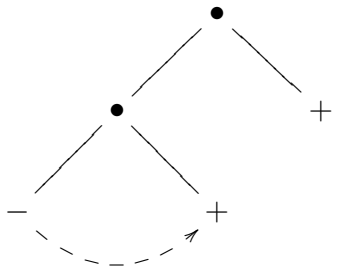


This means that $\mathsf{eval}_\mu^{-1} \circ \mu_\alpha$ maps the singleton name into the S part of the world S, μ, as $\mathsf{add}(\mu)$ maps all elements of S into this part. Otherwise, $\mathsf{eval}_\mu^{-1} \circ \mu_\alpha$ maps the singleton name into the μ part of the world S, μ. Thus eval_μ^{-1} in a sense separates the names in W that are in S from those that are not.

The remainder of this chapter is split among two sections. Section 6.1 defines the category $\mathbb{T}$, while Section 6.2 gives the translation and proves it correct.

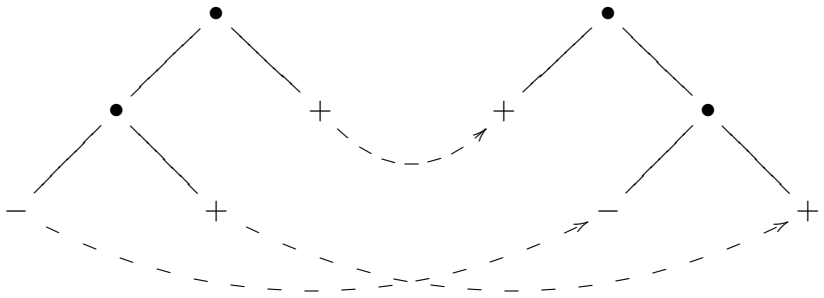
6.1 A Category of Worlds

As discussed above, a world is encoded as a tree. Trees can contain negative leaves, which cancel out positive ones. Trees can be conveyed graphically. For example, the tree



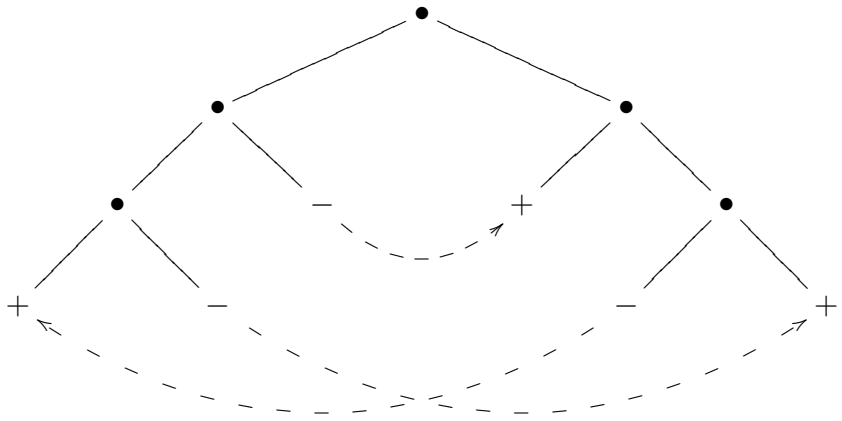
contains one negative leaf and two positive leaves. The negative leaf cancels out the left-most positive leaf. This is denoted with a dashed arrow. Thus this tree represents a world with just one name.

Renaming functions, also called *mappings* below, are injective functions mapping positive leaves to positive leaves and negative leaves to negative leaves. These can also be conveyed graphically:

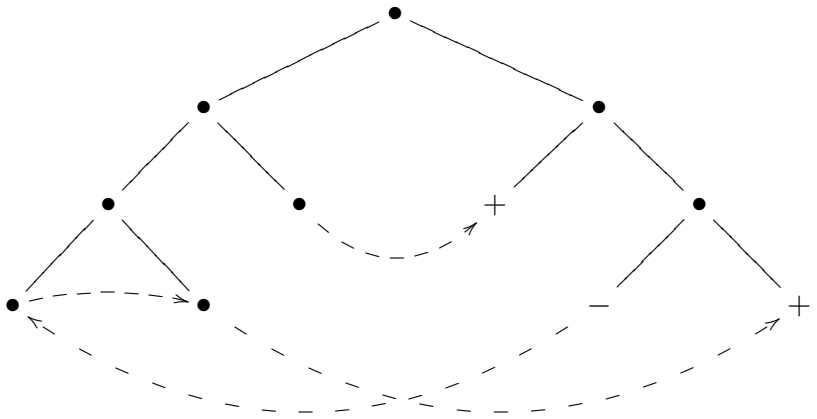


Note, however, that this figure looks very similar to a tree. In fact, if the polarities of the left tree are switched, the arrows for mapping negative leaves are reversed, and a

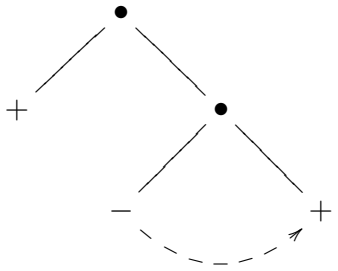
common node is added at the top, the following tree results:



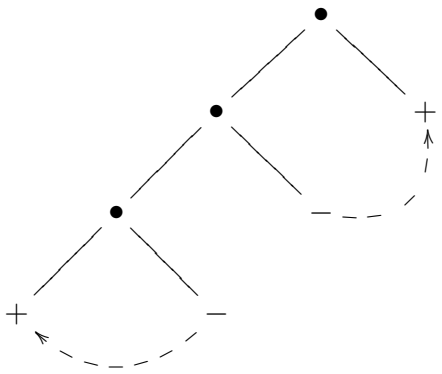
To apply this mapping to a tree, the tree is superimposed with the left subtree of the mapping. The left subtree is then removed, but any paths through it are retained in the right subtree. For example, superimposing the previous example tree on the left subtree of the above mapping yields



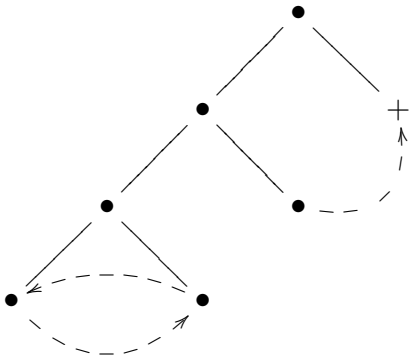
Removing the left subtree, the following tree results:



It is also possible to create a mapping from the original example to the singleton tree with one leaf, as follows:



Superimposing the original example tree yields



and removing the left subtree yields the singleton tree. Swapping the two sides of the mapping and reversing the directions of the arrows, it is also apparent that the mapping to the singleton tree can be reversed. Thus the original tree is said to be *isomorphic* to the singleton tree. This makes intuitive sense, as a world with $2 - 1$ names should be equivalent to one with 1 name.

Note that this construction is very similar to the free compact closed category construction of [36, 65]. The difference there is that all positive nodes in a free compact closed category are required to have an arrow to them. Here, this is not the case. Intuitively, the difference is that the free compact closed category is a model of a form

of linear logic, where all resources must be preserved. Here, new names can always be created, an the main requirement is simply that names cannot be destroyed.

The remainder of this section proceeds as follows. Section 6.1.1 introduces the concept of a *path disjoint graph*. These are used to model trees. Section 6.1.2 uses path disjoint graphs to formally define the category $\mathbb{T}$ of trees and tree mappings. Section 6.1.3 then briefly discusses CIC + $\mathbb{T}$.

6.1.1 Path Disjoint Graphs

Definition 6.1.1 (Path Disjoint). *A graph $G = (V, E)$ is* path disjoint *if and only if every vertex has in- and out-degree at most 1. Given such a G, the following are useful definitions:*

- *A vertex $v \in V$ is a* source *if and only if it has in-degree 0 and out-degree at most 1.*
- *A vertex $v \in V$ is a* sink *if and only if it has in-degree at most 1 and out-degree 0. A source is a* strict sink *if it has in-degree equal to 1.*
- *A vertex $v \in V$ is an* intermediate *vertex if and only if it has in-degree and out-degree 1.*
- *A vertex $v \in V$ is an* unused *vertex if and only if it has in-degree and out-degree 0.*

Lemma 6.1.1. *If $G = (V, E)$ is path disjoint and p is a path in G then p is maximal if and only if it begins at an unused vertex, is a non-empty loop, or is a path from a source vertex to a sink vertex.*

Proof. If $v \in V$ is an unused vertex then there can be no non-empty paths containing it, so the empty path beginning at v is maximal. If $p = (v_1, v_2), \ldots, (v_n, v_1)$ is a non-empty loop then any proper super-sequence would contain v_1 twice but not as the last element, so would not be a valid path. If $p = (v_1, v_2), \ldots, (v_{n-1}, v_n)$ is a path from a source vertex to a sink vertex then any proper super-sequence would either

need an edge (v, v_1), contradicting the fact that v_1 is a source vertex, or would need an edge (v_n, v), contradicting the fact that v_n is a sink vertex.

Conversely, let $p = (v_1, v_2), \ldots, (v_{n-1}, v_n)$ be any maximal path. If p is empty then v_1 must be an unused vertex, or there would be some super-sequence of p that is a valid path. So let p be non-empty. If v_1 is not a source vertex then there is some edge $(v, v_1) \in E$. Since p is maximal, $(v, v_1), p$ cannot be a valid path, and so v must be distinct from v_n and must already be on p. Let $v = v_i$. $i \neq n-1$ implies a contradiction, since this means $(v_i, v_{i+1}) \in E$ and $(v_i, v_1) \in E$, meaning either v_i has out-degree at least 2, contradicting the assumption that G is path disjoint, or $v_{i+1} = v_1$ and so p is not a valid path. Thus $i = n-1$, and either $v_1 = v_n$ and p is a loop, or v_{n-1} has out-degree at least 2, contradicting the assumption that G is path disjoint. A similar argument shows that if v_n is not a sink vertex then p must also be a loop. □

Lemma 6.1.2. *Let $G = (V, E)$ be any path disjoint graph and $v \in V$. If p_1 and p_2 are paths in G that both start at v, then one is a prefix of the other, while if p_1 and p_2 are paths in G that both end at v, then one is a suffix of the other.*

Proof. Proof is by induction on the length of the shorter of the two paths. If one of these is empty then it is trivially a prefix of the other. Otherwise $p_1 = (v, v_1), p_1'$ and $p_2 = (v, v_2), p_2'$. By the induction hypothesis, one of p_1' and p_2' is a prefix of the other. It must also be that $v_1 = v_2$, or v would have out-degree at least 2, contradicting the path disjointedness of G. Thus one of p_1 and p_2 is a prefix of the other. The case for two paths ending at v is similar. □

Lemma 6.1.3. *If $G = (V, E)$ is path disjoint then no two distinct maximal paths modulo loops share a vertex.*

Proof. Let v be any vertex in V. If there is some non-empty path p from v to itself, p is a loop and thus, by Lemma 6.1.1, is maximal. To see that p is unique modulo loops, let p_1, p_2 be another maximal path containing v, where p_1 is the prefix of this new path ending at v and p_2 is the suffix beginning at v. By Lemma 6.1.2 and the maximality of p, p_2 is a prefix of p and p_1 is a suffix of p. Thus $p = p_1', p_1, p_2, p_2'$

for some p_1' and p_2'. The maximality of p_1, p_2 ensures that p is not a proper supersequence of p_1, p_2, so p_1' and p_2' are empty, and p_1, p_2 is a loop that is identical to p up to its starting point.

Otherwise v is not on any loop, so by Lemma 6.1.1 any two maximal paths p_1, p_2 and p_1', p_2' containing v begin at a source and end at a sink node, where p_1 and p_1' are the prefixes before v and p_2 and p_2' are the suffixes after v. By Lemma 6.1.2 one of p_1 and p_1' must be a suffix of the other, but since both begin at a source node neither can be a *proper* suffix and both must be equal. Similarly, one of p_2 and p_2' must be a prefix of the other but both begin at a source node so the two must be equal. Thus the two maximal paths containing v are identical. □

Lemma 6.1.4. *If $G = (V, E)$ is path disjoint and $v \in V$ is a source or sink vertex connected in G to only finitely many vertices, then there is a maximal path p containing v.*

Proof. Proof is by induction on the number of vertices connected to v. If v is a source vertex, then either it is unused, in which case it is also a sink vertex and the empty path is a maximal path from it to itself, or there is some $(v, v') \in E$, where $v' \neq v$ follows from the fact that v is a source vertex. Let $G' = (V, E - (v, v'))$. In this graph, v' is a source vertex and is connected to one less vertex than is v in G. To see this note that if v' is connected to some v'' in G' then, because v' is a source vertex, there must be a path p in G' from v to v', and so $(v, v'), p$ is a path from v to v' in G. v' cannot, however, be connected to v in G', as v is unused in G'. Thus, by the inductive hypothesis, there is a maximal path p in G' containing v'. p must begin at v' as v' is a source vertex in G', so $(v, v'), p$ is a valid path in G. This path is maximal because no edge can be added to the beginning, as v is a source vertex in G, and no edge can be added to the end, as this edge would also be in G', contradicting the maximality of p in G'. The argument for v being a sink vertex is similar. □

Lemma 6.1.5. *If $G = (V, E)$ is path disjoint and $v \in V$ is connected to only finitely many vertices in G then v is on a maximal path.*

Proof. If v is a sink vertex then it is on a maximal path by Lemma 6.1.4. Otherwise there exists some edge $(v, v') \in E$. Let $G' = (V, E - (v, v'))$. v is a sink vertex and v' is a source vertex in G', so by Lemma 6.1.4 there are maximal paths p ending at v

and p' beginning at v', both of which are in G'. If v' is on p then p must begin at v', as v' is a source, so $(v, v'), p$ is a loop containing v, and is thus maximal by Lemma 6.1.1. Similarly, if v is on p' then $p', (v, v')$ is also a loop containing v. Otherwise, $p, (v, v'), p'$ is a valid path. Since p ends at the sink vertex v in G' it must begin at a source vertex v_1 in G'. Similarly, p' ends at a sink vertex v_2 in G'. Since v' is not on p and v is not on p', $v_1 \neq v'$ and so is still a source vertex in G and $v_2 \neq v$ and so is still a sink vertex in G. Thus $p, (v, v'), p'$ begins at a source vertex and ends at a sink vertex and hence is a maximal path in G by Lemma 6.1.1. □

Lemma 6.1.6. *If $G = (V, E)$ is path disjoint and p is a maximal path in G then every vertex on p is connected only to other vertices on p.*

Proof. For any $v_1, v_2 \in V$ with $(v_1, v_2) \in E$, if v_1 is on p then v_2 must also be on p. If not, then p must end at v_1, as otherwise there would be some edge $(v_1, v_2') \in E$, implying the out-degree of v_1 is at least 2. In addition, p cannot contain v_2, as v_2 is not on a maximal path. Thus $p, (v_1, v_2)$ is a valid path, contradicting the maximality of p. Similarly, if v_2 is on a maximal path then v_1 is as well. Hence, for any two vertices $v, v' \in V$ connected by some path p', if v is on p then, by induction on the length of p', so is v'. □

Theorem 6.1.1. *If $G = (V, E)$ is path disjoint then $v \in V$ is connected to only finitely many vertices in G if and only if v is on a maximal path.*

Proof. If v is on a maximal path p, then, by Lemma 6.1.6, v is connected only to other vertices on p. Since paths are finite, v is thus connected to only finitely many vertices in G. Conversely, if v is connected to only finitely many vertices in G, then by Lemma 6.1.5 it is on a maximal path. □

Lemma 6.1.7. *If G is path disjoint then every source vertex on a maximal path in G is connected to exactly one sink vertex and every sink vertex on a maximal path in G is connected to exactly one source vertex.*

Proof. If v is a source vertex on a maximal path then by Lemma 6.1.3 that maximal path is unique and by Lemma 6.1.1 that maximal path ends at a sink vertex. Similarly, if v is a sink vertex on a maximal path then by Lemma 6.1.3 that maximal path is unique and by Lemma 6.1.1 that maximal path begins at a source vertex. □

Definition 6.1.2 (Maximal Path Contraction). *The* maximal path contraction *of path disjoint G is the graph (V', E') constructed as follows. For each loop in the maximal paths modulo loops of G choose a canonical vertex on the loop. Let V_{loop} be the set of these canonical vertices. Define V' to be the set of all vertices in V that are either source vertices, sink vertices, vertices in V_{loop}, or vertices not on a maximal path. Define $E' = (E \cap (V' \times V')) \cup E_{\text{conn}}$, where $E_{\text{conn}} = \{(v_1, v_2) | \text{source vertex } v_1 \text{ and sink vertex } v_2 \text{ are connected in } G\}$.*

Theorem 6.1.2. *Let G be path disjoint and let G' be the maximal path contraction of G. Any vertex in G' is also in G and has in- and out-degree in G' at most what it has in G, with these being the same for sources and sinks. Thus G' is also path disjoint. Further, vertices are connected in G' if and only if they are connected in G and vertices are on a maximal path if and only if they are in G.*

Proof. If v is not on a maximal path in G then it cannot be a source connected to a sink or a sink connected to a source, as this connection would be a maximal path. Thus any edges to or from v in G' cannot be in E_{conn} and so are also present in G, and v has the same in- and out-degree in G' as in G. Further, any vertex connected to v in G must not be on a maximal path in G, by Lemma 6.1.6, and so will similarly retain all of its edges in G'. A straightforward induction then shows that any path containing v is in G if and only if it is in G'. Thus any connections to other vertices are retained from G to G'. In addition, v can thus not be on any maximal path in G' as it would be on this maximal path in G.

Now consider v on a maximal path in G. v is either in V_{loop} or is a strict source or a strict sink vertex in G. If v is in V_{loop} then there are no edges to or from it in E_{conn}. Thus, since v is connected in G only to other vertices on the loop that contains it, none of which are in G', it cannot be connected to any other vertices and hence is unused in G'. v thus has in- and out-degree 0 and is also on the empty maximal path beginning at it. If v is a strict source on a maximal path in G, then by Lemma 6.1.7 there is exactly one sink vertex $v' \in V$ to which v is connected in G, and so there is exactly one edge $(v, v') \in E_{\text{conn}}$. By Lemma 6.1.6 and Lemma 6.1.1 any other vertices connected to v in G are intermediate vertices on a maximal path but not in V_{loop} and so are not in G'. Thus (v, v') is the only edge to or from v in G', and v is a strict source in G'. A similar argument shows that any strict sink in G must have exactly

one edge to it in G', so v' also has only the one edge (v, v') to or from it. (v, v') is then a maximal path containing v and v', and v and v' are connected only to each other which are the only vertices in G' to which they are connected in G. □

Corollary 6.1.1. *If $G = (V, E)$ is path disjoint with all vertices on maximal paths, then (v_1, v_2) is in the maximal path contraction of G if and only if v_1 is a strict source in G, v_2 is a strict sink in G, and there is a path from v_1 to v_2 in G.*

Proof. Let G' be the maximal path contraction of G. If (v_1, v_2) is in G' then, since v_1 and v_2 must be on a maximal path in G and vertices on loops in G have no edges in G', v_1 must be a source and v_2 must be a sink in G. Since the two are connected in G' they must be connected in G so there must be a maximal path from v_1 to v_2. If, conversely, v_1 is a source and v_2 is a sink in G, then v_1 is a source and v_2 is a sink in G' and each is connected to exactly one other vertex. Since they are connected in G they must also be connected in G' and (v_1, v_2) must be in G'. □

Definition 6.1.3 (Overlapping Union). *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be any two graphs and let f_1 and f_2 be injective functions with domain V_1 and V_2, respectively. The pair (f_1, f_2) is called an* overlapping *of G_1 and G_2, with* $\mathrm{Ran}(f_1) \cap \mathrm{Ran}(f_2)$ *being the* overlap *of G_1 and G_2 with respect to (f_1, f_2). Further, any two vertices $v_1 \in V_1$ and $v_2 \in V_2$ with $f_1(v_1) = f_2(v_2)$ are said to overlap with respect to (f_1, f_2). The* overlapping union *of G_1 and G_2 with respect to (f_1, f_2) is the graph with vertices* $\mathrm{Ran}(f_1) \cup \mathrm{Ran}(f_2)$ *and whose edges are given by $\{(f_i(v_1), f_i(v_2)) | i \in \{1, 2\}$ and $(v_1, v_2) \in E_i\}$.*

Definition 6.1.4 (Respecting Polarity). *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be any two path disjoint graphs with overlapping (f_1, f_2). (f_1, f_2) is said to* respect the polarity *of G_1 and G_2 if and only if, for every $v_1 \in V_1$ and $v_2 \in V_2$ such that $f_1(v_1) = f_2(v_2)$, either v_1 is a source in G_1 and v_2 is a sink in G_2 or v_2 is a source in G_2 and v_1 is a sink in G_1.*

Theorem 6.1.3. *If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are any two path disjoint graphs and (f_1, f_2) is an overlapping for G_1 and G_2 that respects the polarity of G_1 and G_2, then the overlapping union of G_1 and G_2 with respect to (f_1, f_2) is also path disjoint. Further, the in- and out-degrees of any vertex v not mapped into the overlap are identical in the overlapping union and in the original graph.*

Proof. This is immediate, as any vertex in the overlap of G_2 and G_1 can only have in- and out-degree at most one by the definition of respecting polarity, while any vertex not in this overlap only has edges from one of the two graphs and so has in- and out-degree at most one by assumption. □

Theorem 6.1.4. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two path disjoint graphs with overlapping (f_1, f_2) that respects the polarity of G_1 and G_2. If G_1 and G_2 are finite then every vertex*

Proof. □

6.1.2 The Category $\mathbb{T}$

In this section, the category $\mathbb{T}$ is defined in terms of path disjoint graphs. The objects of this category will be the *tree types*, which specify a set of trees with the same form and with positive and negative leaves in the same positions. The morphisms from tree type T_1 to tree type T_2 will then be defined as trees of type $(T_1^*) \oslash T_2$, the tree type with right child T_2 and whose left child is the result of switching the polarities in T_1. These notions are formalized here in terms of path disjoint graphs over the set of positive and negative leaves in a tree. Composition in $\mathbb{T}$ is defined by taking an overlapping union of two mappings and then forming the maximal path contraction of the result. Some useful morphisms will then be introduced below.

Definition 6.1.5 (Tree Types). *The* tree types *are inductively defined as follows:*

1. *$\emptyset$ is a tree type, called the* empty *tree type;*
2. *$\mathcal{L}$ is a tree type, called the* positive leaf*, or just* leaf*, tree type;*
3. *$\mathcal{L}^*$ is a tree type, called the* negative leaf *tree type; and*
4. *if T_1 and T_2 are tree types, $T_1 \oslash T_2$ is a tree type, called a* node *tree type.*

Definition 6.1.6 (Duals of Tree Types). *The* dual *of tree type T, written T^*, is the result of replacing all occurrences of $\mathcal{L}$ with $\mathcal{L}^*$ and all occurrences of $\mathcal{L}^*$ with $\mathcal{L}$.*

Definition 6.1.7 (Tree Type Positions). *A* tree type position*, or just position, is a sequence of elements of the set* $\{1, 2\}$*. The empty sequence is denoted* ϵ*. If* q *is a position, then the notion of the* tree at position q in T *is defined inductively as follows:*

1. T *is the tree at position* ϵ *in* T*;*
2. *If* T *is the tree at position* q *in* T_i *for* $i \in \{1, 2\}$*, then* T *is the tree at position* i, q *in* $T_1 \oslash T_2$*.*

If $\mathcal{L}$ *or* $\mathcal{L}^*$*, respectively, is the tree at position* q *in* T*, then* q *is said to be a* positive *or* negative position *of* T*, respectively. The* leaf positions *of* T*, denoted* $\mathsf{lp}(T)$*, are the positions that are either positive or negative in* T*.*

Note that q is used here for positions to be distinct from p which is used for paths.

Definition 6.1.8 (Trees). *Given a tree type* T*, a* tree of type T *is a path disjoint directed graph whose vertices are the leaf positions of* T*, with negative positions being strict sources and positive positions being sinks or unused vertices.*

In the below, T is used for tree types and τ is used for trees. $\tau : T$ is used to denote that τ is a tree of type T. The notation $|\tau|$ is used to denote the tree type of τ.

Definition 6.1.9 (Tree Mappings). *A* tree mapping *from* T_1 *to* T_2 *is a tree of type* $T_1^* \oslash T_2$*.*

In the below, μ is used for tree mappings, and $T_1 \rightrightarrows T_2$ is used to denote the tree type $T_1^* \oslash T_2$ of tree mappings from T_1 to T_2.

Lemma 6.1.8. *Two tree mappings* $\mu_1, \mu_2 : T_1 \rightrightarrows T_2$ *are equal if and only if for each negative position* q *in* $T_1 \rightrightarrows T_2$ *there exists the same edge* (q, q') *in* μ_1 *and* μ_2*.*

Proof. The only if part is trivial, so let μ_1 and μ_2 satisfy the above condition. μ_1 and μ_2 have the same vertices by assumption. Further, since the only vertices in a tree mapping are sources and sinks, the edges of a tree mapping are exactly those edges (q, q') from some strict source to some strict sink, that is, from a negative to a positive position. These are equal in μ_1 and μ_2 by assumption, so they are equal. □

Definition 6.1.10 (Composition). *Let $\mu_1 : T_1 \rightrightarrows T_2$ and $\mu_2 : T_2 \rightrightarrows T_3$ be any two tree mappings. The* composition graph *of μ_1 and μ_2 is the overlapping union of μ_1 and μ_2 with respect to the functions*

$$\begin{aligned} f_1(1,q) &= 1,q \\ f_1(2,q) &= *,q \\ f_2(1,q) &= *,q \\ f_2(2,q) &= 2,q \end{aligned}$$

where $*$ is a new symbol prepended to the paths into T_2 and T_2^* to distinguish them from the other paths. The composition *of μ_2 with μ_1, written $\mu_2 \circ \mu_1$, is defined as the result of taking the maximal path contraction of the composition graph of μ_1 and mu_2 and removing all vertices of the form $*, q$ for some q.*

Lemma 6.1.9 (Well-Definedness of Composition). *For every $\mu_1 : T_1 \rightrightarrows T_2$ and $\mu_2 : T_2 \rightrightarrows T_3$ for all tree types T_1, T_2, and T_3, $\mu_2 \circ \mu_1$ is a tree mapping of type $T_1 \rightrightarrows T_3$. Further, the edge (q_1, q_2) exists in $\mu_2 \circ \mu_1$ if and only if $q_1 \neq q_2$ and there is a maximal path from q_1 to q_2 in the composition graph of μ_1 and μ_2.*

Proof. To see that (f_1, f_2) respects polarity, first note that if two vertices overlap then one is $(2, q)$ in μ_1 and the other is $(1, q)$ in μ_2. By the types of μ_1 and μ_2 q must then be a position in T_2. If q is a positive position in T_2 then $(2, q)$ is positive in $T_1 \rightrightarrows T_2$ and is thus a sink in μ_1 while $(1, q)$ is negative in $T_2 \rightrightarrows T_3$ and so is a strict source in μ_2. Otherwise q is a negative position in T_2 and $(2, q)$ is a strict source in μ_1 and a sink in μ_2. Thus the composition graph is path disjoint by Theorem 6.1.3. Note that this argument also shows that all vertices in the overlap of (f_1, f_2) have out-degree at least 1, so no maximal path in the composition graph ends at a vertex of the form $*, q$.

Next it is shown that, for any vertices q_1, q_2 in $\mu_2 \circ \mu_1$, the edge (q_1, q_2) is in $\mu_2 \circ \mu_1$ if and only if $q_1 \neq q_2$ and there is a maximal path from q_1 to q_2 in the composition graph of μ_1 and μ_2. Let G be this composition graph and let G' be its maximal path contraction. If there is a maximal path from q_1 to q_2 in G with $q_1 \neq q_2$, then this maximal path is non-empty, and an application of Theorem 6.1.2 yields that the edge (q_1, q_2) is in G'. Further, since q_1 and q_2 are assumed to be in $\mu_2 \circ \mu_1$, this edge is not removed when $\mu_2 \circ \mu_1$ is constructed from G'.

Conversely, if (q_1, q_2) is in $\mu_2 \circ \mu_1$ then it must be in G', as $\mu_2 \circ \mu_1$ is constructed from G' by removing vertices and edges. $q_1 \neq q_2$ follows by the definition of maximal path contraction. Since tree types are finite, both μ_1 and μ_2 must also be finite, and it is straightforward to see that G and G' must therefore be finite as well. An application of Lemma 6.1.5 yields that q_1 and q_2 must be on on maximal paths in G', and an application of Theorem 6.1.2 then shows that they must also be on maximal paths in G. Since G' is the maximal path contraction of G, by the definition of maximal path contraction G' can only contain vertices from G on maximal paths if those vertices are source, sink, or unused vertices from G, so by Theorem 6.1.2 and by the fact that (q_1, q_2) is in $\mu_2 \circ \mu_1$ it must be the case that q_1 is a source and q_2 is a sink in G. By Lemma 6.1.1 and Lemma 6.1.6, since q_1 and q_2 are connected and both are on maximal paths, therefore, a maximal path from q_1 to q_2 must exist in G.

The rest of the lemma follows from the above arguments. If $1, q$ is a positive or negative position, respectively, in $T_1 \rightrightarrows T_3$, then it is the same in $T_1 \rightrightarrows T_2$ and so must be a sink or strict source, respectively, in μ_1. Since $1, q$ overlaps with no vertex of μ_2, $1, q$ must also be a sink or strict source, respectively, in the composition graph and, since the composition graph is finite, there must be a maximal path ending at $1, q$, or, respectively, there must be a non-empty maximal path starting at $1, q$ in the composition graph. Thus, by the above arguments, if $1, q$ is a positive or negative position, respectively, in $T_1 \rightrightarrows T_3$, then it is a sink or strict source in $\mu_2 \circ \mu_1$. A similar argument holds for leaf positions $(2, q)$ in $T_1 \rightrightarrows T_3$. □

Lemma 6.1.10 (Associativity of Composition). *For every* $\mu_1 : T_1 \rightrightarrows T_2$, $\mu_2 : T_2 \rightrightarrows T_3$, *and* $\mu_3 : T_3 \rightrightarrows T_4$, $(\mu_3 \circ \mu_2) \circ \mu_1 = \mu_3 \circ (\mu_2 \circ \mu_1)$.

Proof. First note that $(\mu_3 \circ \mu_2) \circ \mu_1$ and $\mu_3 \circ (\mu_2 \circ \mu_1)$ have the same vertices. For the edges, let f_1, f_2, and f_3 be the following injective functions on the set of vertices V_1 of μ_1, V_2 of μ_2, and V_3 of μ_3, respectively:

$$\begin{aligned} f_1(1, q) &= 1, q \\ f_1(2, q) &= *_1, q \\ f_2(1, q) &= *_1, q \\ f_2(2, q) &= *_2, q \\ f_3(1, q) &= *_2, q \\ f_3(2, q) &= 2, q \end{aligned}$$

For $i \in \{1,2,3\}$, let $G_i = (\text{Ran}(f_i), \{(f_1(v_1), f_1(v_2))|(v_1,v_2) \in \mu_i\})$. It is straightforward to see this is an isomorphism. Further, letting $G_{i,i+1} = G_i \cup G_{i+1}$ for $i \in \{1,2\}$, it is straightforward to see that $G_{i,i+1}$ is isomorphic to the composition graph of μ_i and μ_{i+1}. Thus we have that the composition graph of μ_1 and $(\mu_3 \circ \mu_2)$ is isomorphic to $G_1 \cup G'_{2,3}$, where $G'_{2,3}$ is the result of taking the maximal path contraction of $G_{2,3}$ and removing vertices of the form $*_2, q$. We now show that, for any two vertices beginning with 1 or 2, the vertices are connected in $G_1 \cup G_{2,3}$ if and only if they are connected in $G_1 \cup G'_{2,3}$ if and only if they are connected in the composition graph of μ_1 and $(\mu_3 \circ \mu_2)$, by isomorphism. A similar proof applies to the composition graph of $(\mu_2 \circ \mu_1)$ and μ_3 and $G_{1,2} \cup G_3$, so, since $G_1 \cup G_{2,3} = G_{1,2} \cup G_3$, the paths of the two composition graphs are the same and $(\mu_3 \circ \mu_2) \circ \mu_1 = \mu_3 \circ (\mu_2 \circ \mu_1)$ by Lemma 6.1.9 and Lemma 6.1.8.

We actually show the slightly more general proposition that there exists a path p_1 in $G_1 \cup G_{2,3}$ starting and ending with vertices of the form $1, q$, $*_1, q$, or $2, q$ if and only if there is a path p_2 starting and ending at the same vertices in $G_1 \cup G'_{2,3}$. The only if is by induction on the number of vertices of the given form in p_1. If p_1 is empty, then p_2 can be empty. Otherwise $p_1 = p'_1, p''_1$, where p'_1 is the non-empty prefix of p_1 to the next vertex v of the given form. By the induction hypothesis there exists some p''_2 from v to the end of p_1 in $G_1 \cup G_{2,3}$, so we must only show there exists some p'_2 from the beginning of p_1 to v. p'_1 cannot have edges from both G_1 and $G_{2,3}$ as these graphs only share vertices of the given form. If p'_1 is entirely in G_1 then we can set $p'_2 = p'_1$. Otherwise p'_1 is a path in $G_{2,3}$ from vertices of the given form, and so is maximal by the previous paragraph and there is some edge (v', v) in $G'_{2,3}$ from the beginning of p_1 to v. Thus $(v', v), p'_2$ is a path in $G_1 \cup G'_{2,3}$ from the beginning to the end of p_1.

The if is proved by induction on the length of p_2. Again, if p_2 is empty then p_1 can be empty. Otherwise $p_2 = (v, v'), p'_2$. v' must be of the required form, as no vertices beginning with $*_2$ are in G_1 or $G'_{2,3}$. Thus the desired p'_1 exists by the induction hypothesis. If (v, v') is an edge of G_1 then $(v, v'), p'_1$ is the desired p_1. Otherwise (v, v') is an edge of $G'_{2,3}$, so by the above there is a path p''_1 from v to v' in $G_{2,3}$, and p''_1, p'_1 is the desired p_1.

□

Definition 6.1.11 (Identity Mapping). *Let* id_T *be the mapping of type* $T \rightrightarrows T$ *with edges* $((1,q),(2,q))$ *for every positive position* q *in* T *and edges* $((2,q),(1,q))$ *for every negative position* q *in* T*.*

Lemma 6.1.11. *For any mapping* $\mu : T_1 \rightrightarrows T_2$*,* $\mu \circ \mathsf{id}_{T_1} = \mu = \mathsf{id}_{T_2} \circ \mu$*.*

Proof. Let G be the composition graph of id_{T_1} and μ. For every negative or positive position $1, q$ in $T_1 \rightrightarrows T_2$ there exists an edge $((*,q),(1,q))$ or $((1,q),(*,q))$, respectively, in G resulting from id_{T_1}. For every negative position i, q_1 in $T_1 \rightrightarrows T_2$ there exists an edge $((i,q_1),(j,q_2))$ in μ, thus there is an edge $((i',q_1),(j',q_2))$, where i' and j' are the result of changing i or j, respectively, to $*$ if it is 1. This edge can be extended to a path from i, q_1 to j, q_2 in G by possibly using the edges mentioned above resulting from id_{T_1}. Thus $\mu \circ \mathsf{id}_{T_1} = \mu$ by Lemma 6.1.9 and Lemma 6.1.8. The proof for $\mathsf{id}_{T_2} \circ \mu = \mu$ is similar. □

Definition 6.1.12 (The Category $\mathbb{T}$). *The category whose objects are the tree types and whose morphisms from* T_1 *to* T_2 *are the tree mappings of type* $T_1 \rightrightarrows T_2$ *is a valid category.*

By Lemmas 6.1.10 and 6.1.11, $\mathbb{T}$ is indeed a well-defined category. In fact, $\mathbb{T}$ is a compact closed category, with $\emptyset$ as the identity object, $⍉$ as the tensor operation on objects, and T^* as the dual operation on objects. It is straightforward to define the required morphisms and show the required equations hold, but this shall not be important here.

Definition 6.1.13 (Tensor). *Let* $\mu_1 : T_1 \rightrightarrows T_1'$ *and* $\mu_2 : T_2 \rightrightarrows T_2'$ *be any two mappings. The* tensor *of* μ_1 *and* μ_2*, written* $\mu_1 \otimes \mu_2$*, is the mapping of type* $T_1 ⍉ T_2 \rightrightarrows T_1' ⍉ T_2'$ *containing on edge from* i, k, q *to* j, k, q' *for each edge from* i, q *to* j, q' *in* μ_k*, for all* $i, j, k \in \{1, 2\}$*.*

Lemma 6.1.12. *If* $\mu_1 : T_1 \rightrightarrows T_2$*,* $\mu_1' : T_1' \rightrightarrows T_2'$*,* $\mu_2 : T_2 \rightrightarrows T_3$*, and* $\mu_2' : T_2' \rightrightarrows T_3'$ *are any tree mappings of the given types, then* $(\mu_2 \otimes \mu_2') \circ (\mu_1 \otimes \mu_1') = (\mu_2 \circ \mu_1) \otimes (\mu_2' \circ \mu_1')$*.*

Proof. Let $f_i(i) = i$ and $f_i(3-i) = *$ for $i \in \{1,2\}$. The composition graph for $(\mu_1 \otimes \mu_1')$ and $(\mu_2 \otimes \mu_2')$ has an edge from $(f_k(i)), 1, q$ to $(f_k(j)), 1, q'$ for each edge from i, q to j, q in μ_k and one from $(f_k(i)), 2, q$ to $(f_k(j)), 2, q'$ for each edge from i, q

to j, q in μ'_k. Thus by induction and Lemma 6.1.9 there is an edge from $i, 1, q$ to $j, 1, q'$ in $(\mu_2 \otimes \mu'_2) \circ (\mu_1 \otimes \mu'_1)$ if and only if there is an edge from i, q to j, q' in $\mu_2 \circ \mu_1$. Similarly, there is an edge from $i, 2, q$ to $j, 2, q'$ in $(\mu_2 \otimes \mu'_2) \circ (\mu_1 \otimes \mu'_1)$ if and only if there is an edge from i, q to j, q' in $\mu'_2 \circ \mu'_1$. But this is exactly the definition of $(\mu_2 \circ \mu_1) \otimes (\mu'_2 \circ \mu'_1)$. □

Definition 6.1.14. *Let $\mu : T_1 \rightrightarrows T_2$ be any mapping.*

- add-l *is the mapping of type $T \rightrightarrows T \oslash \mathcal{L}$ with the following edges:*

$$\begin{array}{ll} ((1, q), (2, 1, q)) & \textit{for every positive position } q \textit{ in } T \\ ((2, 1, q), (1, q)) & \textit{for every negative position } q \textit{ in } T \end{array}$$

- add(μ) *is the mapping of type $T \rightrightarrows T \oslash (T_1 \rightrightarrows T_2)$ with the following edges:*

$$\begin{array}{ll} ((1, q), (2, 1, q)) & \textit{for every positive position } q \textit{ in } T \\ ((2, 1, q), (1, q)) & \textit{for every negative position } q \textit{ in } T \\ ((2, 2, q), (2, 2, q')) & \textit{for every edge } (q, q') \textit{ in } \mu \end{array}$$

- combine-l-l* *is the mapping of type $((T \oslash \mathcal{L}^*) \oslash \mathcal{L}) \rightrightarrows T$ with the following edges:*

$$\begin{array}{ll} ((1, 1, 1, q), (2, q)) & \textit{for every positive position } q \textit{ in } T \\ ((2, q), (1, 1, 1, q)) & \textit{for every negative position } q \textit{ in } T \\ ((1, 2), (1, 1, 2)) & \end{array}$$

- eval *is the mapping of type $(T_1 \oslash (T_1 \rightrightarrows T_2)) \rightrightarrows T_2$ with the following edges:*

$$\begin{array}{ll} ((1, 1, q), (1, 2, 1, q)) & \textit{for every positive position } q \textit{ in } T_1 \\ ((1, 2, 1, q), (1, 1, q)) & \textit{for every negative position } q \textit{ in } T_1 \\ ((1, 2, 2, q), (2, q)) & \textit{for every positive position } q \textit{ in } T_2 \\ ((2, q), (1, 2, 2, q)) & \textit{for every negative position } q \textit{ in } T_2 \end{array}$$

- eval_μ^{-1} *is the mapping of type* $T_2 \rightrightarrows (T_1 \oslash (T_1 \rightrightarrows T_2))$ *with the following edges:*

$((2,2,i,q),(2,2,j,q'))$	*for every edge* $((i,q),(j,q'))$ *in* μ *with* $i \neq j$
$(f(j,q),f(i,q'))$	*for every* $((i,q),(j,q'))$ *in* μ *with* $i \neq 2$ *or* $j \neq 2$
$((1,q),(2,2,2,q))$	*for every unused positive position* $2,q$ *in* μ
$((1,q'),(2,2,2,q'))$	*for every edge* $((2,q)(2,q'))$ *in* μ
$((2,2,2,q),(1,q))$	*for every edge* $((2,q)(2,q'))$ *in* μ

  where $f(1,q) = 2,1,q$ *and* $f(2,q) = 1,q$

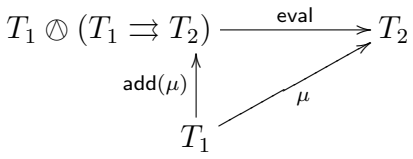
The intended meaning of these mappings is as follows. $\mathsf{add\text{-}l}$ adds an unused leaf as the right subtree of any tree. $\mathsf{add}(\mu)$ adds the mapping μ as the right subtree of a tree. $\mathsf{combine\text{-}l\text{-}l}^*$ takes any tree of type $(T \oslash \mathcal{L}^*) \oslash \mathcal{L}$, which intuitively is a tree whose right subtree is a leaf and whose left subtree is a tree with a hole, and combines the leaf into the hole, thus "filling" the hole. eval takes any tree of type $T_1 \oslash (T_1 \rightrightarrows T_2)$, which intuitively is a tree whose right subtree is a mapping including the left subtree in its domain, and applies the mapping to that left subtree. eval_μ^{-1} is intended to take the result $\mu(\tau)$ of applying μ to τ and returns the tree $\tau \oslash \mu$, though, as suggested in introduction to this chapter, this is not always quite achieved.

Lemma 6.1.13. *The following equalities hold for any tree mapping* μ*:*

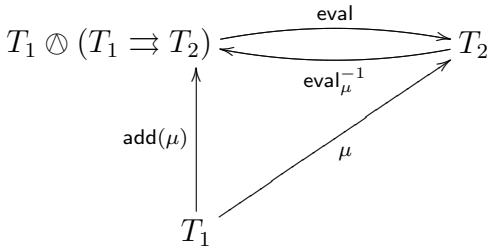
1. $\mathsf{eval} \circ \mathsf{eval}_\mu^{-1} = \mathsf{id}$
2. $\mathsf{eval} \circ \mathsf{add}(\mu) = \mu$
3. $\mathsf{eval}_\mu^{-1} \circ \mu = \mathsf{add}(\mu)$ *for any* $\mu : T_1 \rightrightarrows T_2$ *where* T_1 *has no negative leaves*
4. $(\mu \otimes \mathsf{id}) \circ \mathsf{add\text{-}l} = \mathsf{add\text{-}l} \circ \mu$
5. $(\mu \otimes \mathsf{id}) \circ \mathsf{add}(\mu') = \mathsf{add}(\mu') \circ \mu$
6. $\mathsf{combine\text{-}l\text{-}l}^* \circ ((\mu \otimes \mathsf{id}) \otimes \mathsf{id}) = \mu \circ \mathsf{combine\text{-}l\text{-}l}^*$

Proof. The proofs are routine but tedious, so are omitted here. □

The explanation of these equalities is as follows. The first two describe the following commutative diagram, repeated from the introduction of this chapter:



When the tree type T_1 contains no negative leaves, the first three of the equalities describe the more refined diagram



also repeated from the introduction of this chapter. The fourth and fifth equalities state that adding a right subtree and then performing an operation solely on the left subtree is equivalent to performing the operation on the tree before the adding and then adding the given right subtree. The last equality states that eliminating a hole and then performing an operation on the result is equivalent to performing the operation on the given subtree before doing the combining.

To finish this section, we briefly discuss names and how to compare them. This will be useful below in translating name-matching functions in CNIC. In $\mathbb{T}$, A world W can be identified with a mapping μ of type $\emptyset \rightrightarrows |W|$. This is because such a mapping has no structure in its left subtree, and so the mappings of this type are isomorphic to the worlds of type $|W|$. As discussed above, the names in W will be identified as those positive positions in W with no incident edges. Positive positions in W that do have incident edges are considered cancelled out. It is thus possible to identify the names in W with the mappings μ of type $(\emptyset \oslash \mathcal{L}) \rightrightarrows |W|$ such that $\mu \circ \mathsf{add\text{-}l} = \mu_W$, where μ_W is the mapping of type $\emptyset \rightrightarrows |W|$ corresponding to W. This is because such mappings can only differ in the position to which the leaf on the left is mapped, and it can only be mapped to positions in $|W|$ that do not already have an edge to them.

By the same logic, ordered lists of names in W can be identified with mappings of type $(\ldots(\emptyset \oslash \mathcal{L})\ldots \oslash \mathcal{L}) \rightrightarrows |W|$. We denote this type as $\vec{\alpha} \rightrightarrows |W|$ below.

Name-matching functions in CNIC compare names with ordered lists of names. More specifically, if $\vec{\alpha}$ is a list of names, name-matching tests if some x of type Name is equal to α_i for some i or if x is disjoint from $\vec{\alpha}$. We now demonstrate how this may be done in $\mathbb{T}$:

Lemma 6.1.14. *If T is a tree type, $\mu : \vec{\alpha} \rightrightarrows T$ and $\mu_n : \mathcal{L} \rightrightarrows T$ such that $\mu_n \circ \mathsf{add\text{-}l} = \mu \circ \mathsf{add\text{-}l}^n$, then either $\mu_n = \mu \circ \mathsf{add\text{-}l}^j \circ (\mathsf{add\text{-}l}^i \otimes \mathsf{id})$ for some $i + j + 1 = n$, or there exists some $\mu' : \vec{\alpha}, \alpha \rightrightarrows T$ such that $\mu = \mu' \circ \mathsf{add\text{-}l}$ and $\mu_n = \mu' \circ (\mathsf{add\text{-}l}^n \otimes \mathsf{id})$.*

Proof. It is straightforward to see that $\mu_n \circ \mathsf{add\text{-}l}$ has all the same edges in the right subtree as μ_n, since $\mu_n \circ \mathsf{add\text{-}l}$ will simply remove the edge from the left subtree to the right that must be in any mapping of type $\mathcal{L} \rightrightarrows T$. Similarly, $\mu \circ \mathsf{add\text{-}l}^n$ has the same edges in the right subtree as μ. Thus the assumption that $\mu_n \circ \mathsf{add\text{-}l} = \mu \circ \mathsf{add\text{-}l}^n$ ensures that μ_n and and μ have the same edges in the right subtree.

Now consider the edge $((1,2),(2,q))$ in μ_n from the left subtree to the right. There cannot be an edge $((2,q'),(2,q))$ in μ because this would be in the right subtree and would thus be in μ_n, contradicting the fact that μ_n is path disjoint. Thus there is either no edge to $(2,q)$ in μ or there is an edge of the form $((1,1^i,2),(2,q))$, as all negative positions in the left subtree of $\vec{\alpha} \rightrightarrows T$ are of the form $(1,1^i,2)$. If there is no such edge, then it is straightforward to construct the morphism $\mu' : \vec{\alpha}, \alpha \rightrightarrows T$ that maps the $\vec{\alpha}$ as in μ but maps the new α to $(2,q)$. $\mu' \circ \mathsf{add\text{-}l} = \mu$ because composing with $\mathsf{add\text{-}l}$ simply removes the edge to $(2,q)$ in μ', and $\mu_n = \mu' \circ (\mathsf{add\text{-}l}^n \otimes \mathsf{id})$ since $(\mathsf{add\text{-}l}^n \otimes \mathsf{id})$ removes the other edges from the left subtree to the right in μ'.

If there is an edge of the form $((1,1^i,2),(2,q))$ in μ, then it is straightforward to see that $\mathsf{add\text{-}l}^j \circ (\mathsf{add\text{-}l}^i \otimes \mathsf{id})$ removes all other edges from the left subtree to the right, yielding a mapping of type $(\emptyset \oslash \mathcal{L}) \rightrightarrows T$ with the edge $((1,2),(2,q))$ and all edges in the right subtree the same. Thus $\mu \circ \mathsf{add\text{-}l}^j \circ (\mathsf{add\text{-}l}^i \otimes \mathsf{id}) = \mu_n$. □

6.1.3 Tree Mappings in Type Theory

Using the previous two sections, the category $\mathbb{T}$ can be defined in CIC. The tree types are straightforward to encode, as they form a simple inductive definition. The trees themselves can then be defined as a set of graphs that are path disjoint and that agree with a given tree type. Composition is more complex to define, as it involves overlapping unions and maximal path contraction. The proofs of Section 6.1.1 must also be fully encoded into CIC to show that composition is well-defined. This is tedious and does not significantly contribute to the current presentation, so is omitted here.

In the below, it will be useful to make the equalities of Lemma 6.1.13 hold on the type of mappings. As shown in the proof, these do hold when the given mappings are *ground*, meaning they have no variables. In addition, this proof can be encoded into CIC to produce a proof of type $\mathsf{eq}\ M_1\ M_2$ for the particular cases mentioned in the lemma. Unfortunately, this is only a *provable* equality, not a *definitional* equality. Using provable equality in CIC can be very tedious. Many uses require the principle of Uniqueness of Identity Proofs, meaning that any two proofs of $\mathsf{eq}\ M_1\ M_2$ are themselves equal. This principle is not itself provable in CIC without adding extra axioms [29]. Even with the extra axioms required, reasoning with provable equality can be very tedious.

Instead, the language CIC + $\mathbb{T}$ is defined here to include the equalities of Lemma 6.1.13 as definitional equalities. To do this, an extra relation $\sim$ is added to the theory of CIC. $\sim$ is defined to hold between two terms when one can be got from the other by replacing terms up to the equalities of Lemma 6.1.13. The subtyping relation of CIC is then extended to include the rule

$$\frac{A \sim B}{\vdash A \lesssim B}\ \textsf{st-sim}$$

for incorporating $\sim$. Note that $\sim$ is not added to the reduction relation, as this would require finding a convergent rewrite system that implies all the equalities of Lemma 6.1.13, and it is not known at this time whether this is possible.

We also add one final equality to the definition of $\sim$. Assuming that Lemma 6.1.14 is provable in CIC, it defines a function for determining which of the given equalities hold. This is equivalent to writing a function $\mathsf{find\text{-}name\text{-}case}_{\vec{\alpha}}$ of type

$$\begin{aligned}&\Pi T\,.\,\Pi\mu : (\vec{\alpha} \rightrightarrows T)\,.\,\Pi\mu_n : ((\emptyset \oslash \mathcal{L}) \rightrightarrows T)\,.\\ &\quad \mathsf{eq}\ (\mu \circ \mathsf{add\text{-}l}^n)\ (\mu_n \circ \mathsf{add\text{-}l}) \Rightarrow \mathsf{name\text{-}case}_{\vec{\alpha}}\ T\ \mu\ \mu_n\end{aligned}$$

where the type $\mathsf{name\text{-}case}_{\vec{\alpha}}$ is defined with the signature

$$\begin{aligned}\mathsf{name\text{-}case}_{\vec{\alpha}} \quad &: \quad \Pi T\,.\,\Pi\mu : (\vec{\alpha} \rightrightarrows T)\,.\,\Pi\mu_n : (\emptyset \oslash \mathcal{L}) \rightrightarrows T\,.\,\mathsf{Type}_0\\ \mathsf{name\text{-}case\text{-}i}_{\vec{\alpha},i} \quad &: \quad \Pi T\,.\,\Pi\mu : (\vec{\alpha} \rightrightarrows T)\,.\\ &\qquad \mathsf{name\text{-}case}_{\vec{\alpha}}\ T\ \mu\ (\mu \circ (\mathsf{add\text{-}l}^{i-1} \otimes \mathsf{id}) \circ \mathsf{add\text{-}l}^{n-i})\\ \mathsf{fresh\text{-}name\text{-}case}_{\vec{\alpha}} \quad &: \quad \Pi T\,.\,\Pi\mu : (\vec{\alpha}, \alpha \rightrightarrows T)\,.\\ &\qquad \mathsf{name\text{-}case}_{\vec{\alpha}}\ T\ (\mu \circ \mathsf{add\text{-}l})\ (\mu \circ (\mathsf{add\text{-}l}^n \otimes \mathsf{id}))\end{aligned}$$

The equality that is added to $\sim$ is

$$\begin{aligned}&\mathsf{find\text{-}name\text{-}case}_{\vec{\alpha}}\ T_2\ (\mu_2 \circ \mu)\ (\mu_2 \circ \mu_n)\\ &\quad \sim \mathsf{map\text{-}name\text{-}case}_{\vec{\alpha}}\ T\ \mu\ \mu_n\ T_2\ \mu_2\ (\mathsf{find\text{-}name\text{-}case}_{\vec{\alpha}}\ T\ \mu\ \mu_n)\end{aligned}$$

where $\mathsf{map\text{-}name\text{-}case}_{\vec{\alpha}}$ is the function

$$\begin{aligned}&\textbf{fun}\ (T, \mu, \mu_n, T_2, \mu_2)\ (\mathsf{name\text{-}case\text{-}i}_{\vec{\alpha},i}\ T\ \mu \setminus T, \mu \rightarrow \mathsf{name\text{-}case\text{-}i}_{\vec{\alpha},i}\ T_2\ (\mu_2 \circ \mu)\ \mid\\ &\qquad\qquad \mathsf{fresh\text{-}name\text{-}case}_{\vec{\alpha}}\ T\ \mu \setminus T, \mu \rightarrow \mathsf{fresh\text{-}name\text{-}case}_{\vec{\alpha}}\ T_2\ (\mu_2 \circ \mu)\\ &\qquad\qquad)\end{aligned}$$

with type $\mathsf{name\text{-}case}_{\vec{\alpha}}\ T\ \mu\ \mu_n \Rightarrow \mathsf{name\text{-}case}_{\vec{\alpha}}\ T_2\ (\mu_2 \circ \mu)\ (\mu_2 \circ \mu)$.

The purpose of the above equality is to simplify uses of the function $\mathsf{find\text{-}name\text{-}case}_{\vec{\alpha}}$. $\mathsf{find\text{-}name\text{-}case}_{\vec{\alpha}}$ is how name-matching will be implemented in the translation below. In general, however, names μ_n in the translation will be of the form $\mu \circ \mu_g$ where μ is a variable and μ_g is ground. Thus $\mathsf{find\text{-}name\text{-}case}_{\vec{\alpha}}$ cannot reduce. The above equality ensures that applying $\mathsf{find\text{-}name\text{-}case}_{\vec{\alpha}}$ to such a mapping is equal under $\sim$ to a term that does reduce. $\mathsf{map\text{-}name\text{-}case}_{\vec{\alpha}}$ does not inspect the given mappings, so it will always reduce if the $\mathsf{find\text{-}name\text{-}case}_{\vec{\alpha}}$ does.

In the below, CNIC will be proved strongly normalizing, and thus consistent, via a reduction to CIC + $\mathbb{T}$. This requires CIC + $\mathbb{T}$ itself to be strongly normalizing. To see this, we note that CIC + $\mathbb{T}$ is an instance of a formalism called the Calculus of Congruent Inductive Constructions, or CCIC [8]. CCIC allows equalities to be added to CIC in the above manner, as long as the equalities are in fact provable equalities in CIC. This is true of all equalities given for $\sim$ above.

Note that CCIC does has some technical side conditions disallowing the use of the above subtyping rule $\mathsf{st\text{-}sim}$ in some cases. Specifically, the $\sim$ relation can only be used to convert the weak terms in a type, where a *weak* term is a term in which all pattern-matching functions are fully applied to arguments and there are no applications of variables whose types are of the form $A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow \mathsf{Type}_i$ for some i. The reason for this restriction in CCIC is that that calculus allows the relation $\sim$ to depend on a local context of equality assumptions, which may be contradictory. For example, in body M of the term

$$\lambda\, e : \mathsf{eq}\ \mathsf{zero}\ (\mathsf{succ}\ \mathsf{zero})\,.\, M$$

the $\sim$ relation could use the assumption e that zero equals $\mathsf{succ}\ \mathsf{zero}$ to convert zero to $\mathsf{succ}\ \mathsf{zero}$ in types. Abbreviating the non-weak term

$$\mathbf{fun}\ (\mathsf{zero} \setminus \cdot \rightarrow \mathsf{nat} \mid \mathsf{succ}\ y \setminus y \rightarrow \mathsf{nat} \Rightarrow \mathsf{nat})$$

as F, we then have that $\mathsf{nat} = F\ \mathsf{zero} \sim F\ (\mathsf{succ}\ \mathsf{zero}) = \mathsf{nat} \Rightarrow \mathsf{nat}$, which then allows the non-terminating term $(\lambda\, x : F\ \mathsf{zero}\,.\, x\ x)\ (\lambda\, x : F\ \mathsf{zero}\,.\, x\ x)$ to be typed. The $\sim$ relation in CIC + $\mathbb{T}$, however, does not depend on any local assumptions, and thus weakness need not be enforced here. Private correspondence with one of the authors of the work on CCIC [34] has verified this claim.

6.2 Translating CNIC to CIC + $\mathbb{T}$

In this section, the translation from CNIC to CIC + $\mathbb{T}$ is given by the operations $[\![M]\!]^{\Delta}_{\mathrm{tm}}$ and $[\![M]\!]^{\Delta}_{\mathrm{pf}}$, discussed below. These operations define two translations of M with respect to a *translation context* Δ. Translation contexts give a specification of

the world of each name and variable used in M. It is also a tree itself, and specifies the world for M.

The remainder of this section is split into two pieces. Section 6.2.1 introduces translation contexts and gives some important properties. Section 6.2.2 then defines the translation and uses it to prove strong normalization of CNIC.

6.2.1 Translation Contexts

The translation contexts are used for three purposes in the translation below. First, they give a world for the translation. Second, they specify which free names in a term refer to which positive leaves in that world. Third, they specify a world for all of the free variables of a term.

The translation contexts are inductively defined along with the world they specify. This is as follows:

- $\cdot$ is a translation context, called the *empty* translation context, that specifies the unique world of type $\emptyset$.
- If Δ is a translation context that specifies a world of type $|\Delta|$, and Δ does not contain α, then Δ, α is a translation context that specifies the world of type $|\Delta| \oslash \mathcal{L}$ got from the world of Δ by adding a new leaf as a right subtree.
- If Δ is a translation context that specifies a world of type $|\Delta|$, and Δ contains α but not α^*, then Δ, α^* is a translation context that specifies the world of type $|\Delta| \oslash \mathcal{L}^*$ built by adding a negative leaf as a right subtree that cancels out the leaf already in Δ for α.
- If Δ is a translation context that specifies a world of type $|\Delta|$ and μ is a mapping with type $|\Delta| \rightrightarrows T$ for some T, then $\Delta, (\mu; \vec{x})$ is a translation context that specifies the world of type $|\Delta| \oslash |\Delta| \rightrightarrows T$ got from adding μ as a right subtree.

Note that, in these definitions and the below, $|\Delta|$ is used to denote the type of the world specified by Δ.

As suggested above, translation contexts are also used to specify to mappings, $\Delta(\vec{\alpha})$ and $\Delta(x)$. The first of these is used to "pick out" the names $\vec{\alpha}$ from the world specified by Δ. Specifically, $\Delta(\vec{\alpha})$ is defined to be the mapping of type $(\ldots(\emptyset \oslash \alpha_1)\ldots \oslash \alpha_n) \rightrightarrows |\Delta|$ that maps each α_i to the positive leaf created for it in the tree specificed by Δ. This means that this notation is undefined if, for any of the α_i, either α_i is not in Δ or α_i^* is in Δ.

The translation context $\Delta_1, (\mu; \vec{x}), \Delta_2$ specifies that the variables $\vec{x}$ are all in the world got by applying μ to the world specified by Δ_1. To bring them to the world specified by $\Delta_1, (\mu; \vec{x}), \Delta_2$, eval_μ^{-1} must first be applied to bring the variables to a world of the type $|\Delta_1| \oslash (|\Delta_1| \rightrightarrows \Delta_2)$. Then a mapping must be applied to add the Δ_2 part to the world. This mapping is denoted $\mathsf{tcadd}(\Delta_2)$, and is the mapping from $|\Delta_1, (\mu; \vec{x})|$ to $|\Delta_1, (\mu; \vec{x}), \Delta_2|$ that is constant on positions in the input and maps the tree specified by $\Delta_1, (\mu; \vec{x})$ to that specified by $\Delta_1, (\mu; \vec{x}), \Delta_2$.

Note that $\mathsf{tcadd}(\Delta_2)$ is not defined if Δ_2 contains some α^* but not the corresponding α. This would constitute a negative leaf with no corresponding positive leaf, a situation that is not allowed. This means that if we introduce a name α into Δ, then introduce x, and then cancel α, x cannot then be used. This corresponds exactly to the typing rule for name replacements, the only construct that cancels names. In the name replacement $M\ \langle\alpha\rangle$, the variable x cannot be used in M unless it occurs before α in the typing context.

Lemma 6.2.1. *For any Δ and any sequence $\vec{\alpha}$ of names in Δ,*

1. $(\Delta, \alpha)(\vec{\alpha}, \alpha) = (\Delta(\vec{\alpha}) \otimes \mathsf{id})$*;*
2. $(\Delta, \alpha^*)(\vec{\alpha}) = (\Delta(\vec{\alpha}, \alpha) \otimes \mathsf{id}) \circ (\vec{\alpha}, \alpha, \alpha^*)(\vec{\alpha})$*; and*
3. $(\Delta, (\mu; \Gamma))(\vec{\alpha}) = \mathsf{add}(\mu) \circ \Delta(\vec{\alpha})$.

Proof. Immediate by inspection of the definitions. □

An important form of translation context in the below will be the eval-ing translation contexts. It will turn out that these are the only translation contexts that can actually come about in the course of computation.

Definition 6.2.1 (Eval-ing Translation Context). *The* non-mapping length *of translation context Δ is* $\mathbf{len}(\Delta')$ *where Δ' is the longest suffix of Δ not containing any mapping elements $(\mu;\Gamma)$. An* eval-ing translation context *is one where every mapping element is of the form $(\mu \circ (\mathsf{eval} \otimes \mathsf{id}^n);\Gamma)$, n being the non-mapping length of the prefix of Δ before the given mapping element.*

Lemma 6.2.2. *The equivalences*

- $(\mathsf{eval} \otimes \mathsf{id}^n) \circ ((\Delta_1, (\mu;\Gamma), \Delta_2)(x)) \circ \mathsf{eval} \cong (\mathsf{eval} \otimes \mathsf{id}^n) \circ \mathsf{tcadd}(\Delta_2)$
- $(\mathsf{eval} \otimes \mathsf{id}^n) \circ ((\Delta_1, (\mu;\Gamma), \Delta_2)(x)) \circ \mu \cong (\mathsf{eval} \otimes \mathsf{id}^n) \circ \mathsf{tcadd}((\mu;\Gamma), \Delta_2)$

hold for any Δ_1, Δ_2, μ, Γ, and x with $x \in \Gamma$ and Δ_2 an eval-ing translation context with non-mapping length n.

Proof. Straightforward by the definitions. □

Definition 6.2.2 (Agreement of Translation Contexts). *The notion that translation context Δ_1 agrees under mapping μ_1 with translation context Δ_2 under mapping μ_2, written $\mu_1(\Delta_1) \approx \mu_2(\Delta_2)$, is defined inductively as follows:*

1. $\mathsf{tcadd}(\Delta_2)(\Delta_1) \approx \mathsf{id}(\Delta_1, \Delta_2)$;
2. $\mathsf{combine\text{-}l\text{-}l}^*(\Delta_1, \alpha_1, \Delta_2, \alpha_1^*, \alpha_2) \approx \mathsf{id}(\Delta_1, \alpha_2, \Delta_2)$;
3. $\mu_1 \otimes \mathsf{id}(\Delta_1, \alpha) \approx \mu_2 \otimes \mathsf{id}(\Delta_2, \alpha)$ *if* $\mu_1(\Delta_1) \approx \mu_2(\Delta_2)$;
4. $\mu_1 \otimes \mathsf{id}(\Delta_1, \alpha^*) \approx \mu_2 \otimes \mathsf{id}(\Delta_2, \alpha^*)$ *if* $\mu_1(\Delta_1) \approx \mu_2(\Delta_2)$; *and*
5. $\mathsf{eval}(\Delta_1, (\mu \circ \mu_1;\Gamma)) \approx \mathsf{eval}(\Delta_2, (\mu \circ \mu_2;\Gamma))$ *if* $\mu_1(\Delta_1) \approx \mu_2(\Delta_2)$.

Lemma 6.2.3. *If $\mu_1(\Delta_1) \approx \mu_2(\Delta_2)$ then:*

1. $\mu_1 \circ \Delta_1(\vec{\alpha}) = \mu_2 \circ \Delta_2(\vec{\alpha})$ *for any $\vec{\alpha}$ all of which have positive but no negative occurrences in Δ_1 and Δ_2; and*
2. $\mu_1 \circ \Delta_1(x) = \mu_2 \circ \Delta_2(x)$ *for any x occurring in both Δ_1 and Δ_2.*

Proof. By straightforwrd induction on the definition of $\mu_1(\Delta_1) \approx \mu_2(\Delta_2)$. □

6.2.2 The Translation

We turn now to the actual translation. As suggested above, the intent of the translation is to convert a term M in CNIC to a term of the form $\lambda T\,.\,\lambda\,\mu\,.\,N$, where μ has type $|\Delta| \rightrightarrows T$ and Δ is the translation context used to translate M. This in a sense denotes a set of terms, one for each tree type T and mapping μ to T. Intuitively, this means that, although M is translated into a world with names $\vec{\alpha}$ at particular positions, the translation includes a specification of what M would be at every possible world that is reachable (via a mapping) from the world of translation.

In the following translation, however, translations of terms are not often passed a T and a μ, because the result cannot be *re*-translated. Let M be a CNIC term and N be the translation of M with respect to Δ. Restating the previous sentence, if we pass a given T and μ to N then we get a result which is not a function on tree types and mappings. Thus we lose the ability to re-translate N. Instead, the translation below generally builds the new function $\lambda\,T_2\,.\,\lambda\,\mu_2\,.\,N\ T_2\ (\mu_2 \circ \mu)$, which takes in another mapping μ_2 from T to T_2 and passes the composition of this mapping to N. It is then possible to re-translate this new function by applying the same process again, and this may happen *ad infinitum*. This construction is written below as $\mu(N)$, and is called the *renaming* of N.

The main difficulty of the given approach concerns inductive types. Specifically, consider the CNIC term c for some constructor c. In CNIC, a pattern-matching function can match against c, and distinguish it from other constructors. If c is translated to a function, however, then it is no longer possible to pattern-match on it. Further, we cannot perform pattern-matching on the translation of c by simply passing it some particular T and μ, because that would just be pattern-matching on that particular *instance* of the translation of c. The reason this matters is that a function which takes in T and μ is not guaranteed to return the same constructor for each input. Thus, by matching just one particular instance of such a function agaist the c pattern it is not guaranteed that the given function is equal to the translation of c. But this is what is assumed by pattern-matching, that the input is equal to the given pattern. The same problem is apparent with name-matching functions.

Thus the translation is broken into two pieces. The first piece, $[\![M]\!]^{\Delta}_{\text{tm}}$, creates the actual translation of CNIC term M. The second piece, $[\![M]\!]^{\Delta}_{\text{pf}}$, creates the proof that $[\![M]\!]^{\Delta}_{\text{tm}}$ is a valid translation. Informally, a valid translation always returns the same constructor, no matter what mapping is passed to it, and always returns the same name modulo the renaming passed to it. Proofs must also be extended to functions and elements of a ∇ type. To be a valid translation at function type means that applying the function to a valid translation returns a valid translation. For a ∇ type, a valid translation is one that returns a valid translation for every name replacement.

This approach is very similar to proofs by logical relations [25, 12]. In a logical relations proof, some property is proved of all terms in the language by defining a set of terms by induction on the type such that all terms in the set satisfy the property. Induction on the terms is then used to show that all terms are in the given set. The set in question is called a logical relation. (Presumably it is called a relation because it defines a set for every type, which is thus a relation between terms and types.) For base types, like the inductive types, the logical relation simply includes all terms with the desired property. For composite types like function types, terms are in the logical relation if they have the desired property and if, additionally, all possible elimination forms for the term are in the set. For function types, the elimination form is function application. For ∇ types, the proper elimination form is name replacement.

The difference between logical relations proofs and the approach here is that here the logical relation is encoded as part of the translation. The general idea is that, if A is a CNIC type, then $[\![A]\!]^{\Delta}_{\text{pf}}$ defines a predicate on the elements of $[\![A]\!]^{\Delta}_{\text{tm}}$. If M is then a CNIC term of type A, then $[\![M]\!]^{\Delta}_{\text{pf}}$ will be a proof that $[\![M]\!]^{\Delta}_{\text{tm}}$ satisfies the predicate defined by $[\![A]\!]^{\Delta}_{\text{pf}}$. Technically speaking, $[\![A]\!]^{\Delta}_{\text{pf}}$ will have type

$$\lambda\, T\,.\,\lambda\, \mu : (|\Delta| \rightrightarrows T)\,.\,\Pi x : \mu([\![A]\!]^{\Delta}_{\text{tm-tp}})\,.\,\mathsf{Type}_i$$

where the notation $\mu([\![A]\!]^{\Delta}_{\text{tm-tp}})$ stands for $\Pi T_2\,.\,\Pi \mu_2\,.\,[\![A]\!]^{\Delta}_{\text{tm}}\ T_2\ (\mu_2 \circ \mu)$. $[\![M]\!]^{\Delta}_{\text{pf}}$ will then have type $\Pi T\,.\,\Pi \mu\,.\,[\![A]\!]^{\Delta}_{\text{pf-tp}}\ T\ \mu\ \mu([\![M]\!]^{\Delta}_{\text{tm}})$ specifying that any renaming of $[\![M]\!]^{\Delta}_{\text{tm}}$ will still be in the logical relation defined by A.

We turn now to the definition of the translation. This is given in Figures 6.1, 6.2, 6.3, and 6.4. Note that these figures follow the convention that the leading μ taken as argument in $\lambda\, T\,.\,\lambda\, \mu\,.$ for every term has type $|\Delta| \rightrightarrows T$. We now discuss the

translations for the term constructs. The term translation for variables x takes a tree type T and a mapping μ and applies x to T and $(\mu \circ \Delta(x))$. As discussed above, $\Delta(x)$ brings x to the world for Δ. μ is then composed with $\Delta(x)$ to move x to pass the input arguments on to x. For the proof translation of x, note that every variable x always has a corresponding proof variable x_{pf}. Otherwise it would not be possible to prove x is in the logical relation, as x could be any term. Note the symmetry between the term and proof translations.

For the translation of constructors, note that constructors are assumed to be fully applied to as many arguments as the arity of the constructor. This can be assured in the source language by η-expansion. Every constructor in CNIC gets translated to two constructors, one for the term translation and one for the proof translation. Similarly for the type constructors. More specifically, if a has type $\Pi\Gamma^{\text{x}} \,.\, \mathsf{Type}_i$ in CNIC, then the two types a and $\mathsf{valid}\text{-}a$ are created in the translation with the types

$$\begin{array}{lcl} a & : & \Pi T \,.\, \Pi \mu \,.\, \Pi [\![\Gamma^{\text{x}}]\!]^{\mu;\cdot}_{\text{ctxt}} \,.\, \mathsf{Type}_i \\ \mathsf{valid}\text{-}a & : & \Pi T \,.\, \Pi \mu \,.\, \Pi [\![\Gamma^{\text{x}}]\!]^{\mu;\cdot}_{\text{ctxt}} \,.\, \Pi x : a\ T\ \mu\ \Gamma^{\text{x}} \,.\, \mathsf{Type}_i \end{array}$$

where $[\![\Gamma^{\text{x}}]\!]^{\mu;\cdot}_{\text{ctxt}}$ translates Γ^{x} to take in a term and proof variable for every variable in Γ^{x}. This notion is defined in Figure 6.4.

To translate constructor c of type $\Pi\Gamma^{\text{x}} \,.\, a\ \vec{M}$, we first partition Γ^{x} into the types that do not contain a and those that do contain a. This is because the type of c cannot mention the type $\mathsf{valid}\text{-}a$, since conceptually we must define $\mathsf{valid}\text{-}a$ after a. So we assume the type of c is $\Pi\Gamma^{\text{x}}_1, \Gamma^{\text{x}}_2 \,.\, a\ \vec{M}$ where Γ^{x}_1 contains the types without a and Γ^{x}_2 contains those with a. c then translates to the constructors c and $\mathsf{valid}\text{-}c$, with types

$$\begin{array}{lcl} c & : & \Pi T \,.\, \Pi \mu \,.\, \Pi [\![\Gamma^{\text{x}}_1]\!]^{\mu;\cdot}_{\text{ctxt}} \,. \quad (\mathsf{eval}([\![\Gamma^{\text{x}}_2]\!]^{\mu;\Gamma^{\text{x}}_1}_{\text{tm-tp}})) \Rightarrow a\ \mathsf{eval}([\![\vec{M}]\!]^{\mu;\Gamma^{\text{x}}_1}_{\text{args}}) \\ \mathsf{valid}\text{-}c & : & \Pi T \,.\, \Pi \mu \,.\, \Pi [\![\Gamma^{\text{x}}_1, \Gamma^{\text{x}}_2]\!]^{\mu;\cdot}_{\text{ctxt}} \,. \\ & & \quad \mathsf{valid}\text{-}a\ \mathsf{eval}([\![\vec{M}]\!]^{\mu;\Gamma^{\text{x}}_1,\Gamma^{\text{x}}_2}_{\text{args}})\ (\lambda T_2 \,.\, \lambda \mu_2 \,.\, c\ T_2\ (\mu_2 \circ \mu)\ \mu_2(\Gamma^{\text{x}}_1)) \end{array}$$

where $\mathsf{eval}([\![\Gamma^{\text{x}}_2]\!]^{\Delta}_{\text{tm-tp}})$ is used to define that only the term versions of the variables and types in Γ^{x}_2 are used in the type of c. The notation $\mu([\![\vec{M}]\!]^{\Delta}_{\text{args}})$ is simply the sequence of $\mu([\![M_i]\!]^{\Delta}_{\text{tm}})$ and $\mu([\![M_i]\!]^{\Delta}_{\text{pf}})$ for every M_i in $\vec{M}$. Given these definitions, the term translation of $c\ \vec{M}_1\ \vec{M}_2$ takes in T and μ and applies c to them, the term and proof

$$
\begin{array}{lcl}
\mu(x) & \equiv & \lambda\, T_2 \,.\, \lambda\, \mu_2 \,.\, x\ T_2\ (\mu_2 \circ \mu) \\
\mu([\![A]\!]^{\Delta}_{\mathrm{tp}}) & = & \Pi T_2 \,.\, \Pi \mu_2 \,.\, [\![A]\!]^{\Delta}_{\mathrm{tm}}\ T_2\ (\mu_2 \circ \mu) \\
\mu([\![A]\!]^{\Delta}_{\text{pf-tp}}(x)) & = & \Pi T_2 \,.\, \Pi \mu_2 \,.\, [\![A]\!]^{\Delta}_{\mathrm{pf}}\ T_2\ (\mu_2 \circ \mu)\ (\lambda\, T_3 \,.\, \lambda\, \mu_3 \,.\, x\ T_3\ (\mu_3 \circ \mu_2)) \\
\mu([\![\vec{M}]\!]^{\Delta}_{\mathrm{args}}) & = & \mu([\![M_1]\!]^{\Delta}_{\mathrm{tm}}), \mu([\![M_1]\!]^{\Delta}_{\mathrm{pf}}), \ldots, \mu([\![M_n]\!]^{\Delta}_{\mathrm{tm}}), \mu([\![M_n]\!]^{\Delta}_{\mathrm{pf}}) \\
 & & \\
[\![x]\!]^{\Delta}_{\mathrm{tm}} & = & \lambda\, T \,.\, \lambda\, \mu \,.\, x\ T\ (\mu \circ \Delta(x)) \\
[\![x]\!]^{\Delta}_{\mathrm{pf}} & = & \lambda\, T \,.\, \lambda\, \mu \,.\, x_{\mathrm{pf}}\ T\ (\mu \circ \Delta(x_{\mathrm{pf}})) \\
 & & \\
[\![c\ \vec{M}_1\ \vec{M}_2]\!]^{\Delta}_{\mathrm{tm}} & = & \lambda\, T \,.\, \lambda\, \mu \,.\, c\ T\ (\mu \circ \Delta(\emptyset))\ \mu([\![\vec{M}_1]\!]^{\Delta}_{\mathrm{args}})\ \mu([\![\vec{M}_2]\!]^{\Delta}_{\mathrm{tm}}) \\
[\![c\ \vec{M}]\!]^{\Delta}_{\mathrm{pf}} & = & \lambda\, T \,.\, \lambda\, \mu \,.\, \mathsf{valid}\text{-}c\ T\ (\mu \circ \Delta(\emptyset))\ \mu([\![\vec{M}]\!]^{\Delta}_{\mathrm{args}}) \\
 & & \\
[\![\alpha]\!]^{\Delta}_{\mathrm{tm}} & = & \lambda\, T \,.\, \lambda\, \mu \,.\, \mathsf{mk\text{-}name}\ T\ (\mu \circ \Delta(\alpha)) \\
[\![\alpha]\!]^{\Delta}_{\mathrm{pf}} & = & \lambda\, T \,.\, \lambda\, \mu \,.\, \mathsf{mk\text{-}valid\text{-}name}\ T\ (\mu \circ \Delta(\alpha)) \\
 & & \\
[\![\nu\, \alpha \,.\, M]\!]^{\Delta}_{\mathrm{tm}} & = & \lambda\, T \,.\, \lambda\, \mu \,.\, \lambda\, T_2 \,.\, \lambda\, \mu_2 : (T \rightrightarrows (T_2 \between \mathcal{L}^*)) \,. \\
 & & \quad [\![M]\!]^{\Delta,\alpha}_{\mathrm{tm}}\ T_2\ (\mathsf{combine\text{-}l\text{-}l}^* \circ ((\mu_2 \circ \mu) \otimes \mathsf{id})) \\
[\![\nu\, \alpha \,.\, M]\!]^{\Delta}_{\mathrm{pf}} & = & \lambda\, T \,.\, \lambda\, \mu \,.\, \lambda\, T_2 \,.\, \lambda\, \mu_2 : (T \rightrightarrows (T_2 \between \mathcal{L}^*)) \,. \\
 & & \quad [\![M]\!]^{\Delta,\alpha}_{\mathrm{pf}}\ T_2\ (\mathsf{combine\text{-}l\text{-}l}^* \circ ((\mu_2 \circ \mu) \otimes \mathsf{id})) \\
 & & \\
[\![M\ \langle\alpha\rangle]\!]^{\Delta}_{\mathrm{tm}} & = & \lambda\, T \,.\, \lambda\, \mu \,.\, [\![M]\!]^{\Delta,\alpha^*}_{\mathrm{tm}}\ (T \between \mathcal{L}^*)\ (\mu \otimes \mathsf{id})\ T\ \mathsf{id} \\
[\![M\ \langle\alpha\rangle]\!]^{\Delta}_{\mathrm{pf}} & = & \lambda\, T \,.\, \lambda\, \mu \,.\, [\![M]\!]^{\Delta,\alpha^*}_{\mathrm{pf}}\ (T \between \mathcal{L}^*)\ (\mu \otimes \mathsf{id})\ T\ \mathsf{id} \\
 & & \\
[\![\lambda\, x : A \,.\, M]\!]^{\Delta}_{\mathrm{tm}} & = & \lambda\, T \,.\, \lambda\, \mu \,.\, \lambda\, x : \mu([\![A]\!]^{\Delta}_{\text{tm-tp}}) \,.\, \lambda\, x_{\mathrm{pf}} : \mu([\![A]\!]^{\Delta}_{\text{pf-tp}}(x)) \,. \\
 & & \quad [\![M]\!]^{\Delta,(\mu;x,x_{\mathrm{pf}})}_{\mathrm{tm}}\ T\ \mathsf{eval} \\
[\![\lambda\, x : A \,.\, M]\!]^{\Delta}_{\mathrm{pf}} & = & \lambda\, T \,.\, \lambda\, \mu \,.\, \lambda\, x : \mu([\![A]\!]^{\Delta}_{\text{tm-tp}}) \,.\, \lambda\, x_{\mathrm{pf}} : \mu([\![A]\!]^{\Delta}_{\text{pf-tp}}(x)) \,. \\
 & & \quad [\![M]\!]^{\Delta,(\mu;x,x_{\mathrm{pf}})}_{\mathrm{pf}}\ T\ \mathsf{eval} \\
 & & \\
[\![M\ N]\!]^{\Delta}_{\mathrm{tm}} & = & \lambda\, T \,.\, \lambda\, \mu \,.\, [\![M]\!]^{\Delta}_{\mathrm{tm}}\ T\ \mu\ \mu([\![N]\!]^{\Delta}_{\mathrm{tm}})\ \mu([\![N]\!]^{\Delta}_{\mathrm{pf}}) \\
[\![M\ N]\!]^{\Delta}_{\mathrm{pf}} & = & \lambda\, T \,.\, \lambda\, \mu \,.\, [\![M]\!]^{\Delta}_{\mathrm{pf}}\ T\ \mu\ \mu([\![N]\!]^{\Delta}_{\mathrm{tm}})\ \mu([\![N]\!]^{\Delta}_{\mathrm{pf}})
\end{array}
$$

Figure 6.1: Translation of CNIC to CIC + $\mathbb{T}$: Terms

$$
\begin{array}{lcl}
[\![\mathsf{Type}_i]\!]^{\Delta}_{\mathrm{tm}} & = & \lambda\, T\,.\,\lambda\,\mu\,.\,\mathsf{Type}_i \\
[\![\mathsf{Type}_i]\!]^{\Delta}_{\mathrm{pf}} & = & \lambda\, T\,.\,\lambda\,\mu\,.\,\lambda\, A\!:\!\mu([\![\mathsf{Type}_i]\!]^{\Delta}_{\text{tm-tp}})\,.\,\Pi x\!:\!(\Pi T_2\,.\,\Pi\mu_2\,.\,A\ T_2\ \mu_2)\,.\,\mathsf{Type}_i \\
\\
[\![\Pi x\!:\!A\,.\,B]\!]^{\Delta}_{\mathrm{tm}} & = & \lambda\, T\,.\,\lambda\,\mu\,.\,\Pi x\!:\!\mu([\![A]\!]^{\Delta}_{\text{tm-tp}})\,.\,\Pi x_{\mathrm{pf}}\!:\!\mu([\![A]\!]^{\Delta}_{\text{pf-tp}}(x))\,. \\
 & & \quad [\![B]\!]^{\Delta,(\mu;x,x_{\mathrm{pf}})}_{\mathrm{tm}}\ T\ \mathsf{eval} \\
[\![\Pi x\!:\!A\,.\,B]\!]^{\Delta}_{\mathrm{pf}} & = & \lambda\, T\,.\,\lambda\,\mu\,.\,\lambda\, y\!:\!\mu([\![\Pi x\!:\!A\,.\,B]\!]^{\Delta}_{\text{tm-tp}})\,. \\
 & & \quad \Pi x\!:\!\mu([\![A]\!]^{\Delta}_{\text{tm-tp}})\,.\,\Pi x_{\mathrm{pf}}\!:\!\mu([\![A]\!]^{\Delta}_{\text{pf-tp}}(x))\,. \\
 & & \quad\quad [\![B]\!]^{\Delta,(\mu;x,x_{\mathrm{pf}})}_{\mathrm{pf}}\ T\ \mathsf{eval} \\
 & & \quad\quad\quad (\lambda\, T_2\,.\,\lambda\,\mu_2\,.\,y\ T_2\ \mu_2\ \mu_2(x)\ \mu_2(x_{\mathrm{pf}})) \\
\\
[\![\nabla\alpha\,.\,A]\!]^{\Delta}_{\mathrm{tm}} & = & \lambda\, T\,.\,\lambda\,\mu\,.\,\Pi T_2\,.\,\Pi\mu_2\!:\!(T \rightrightarrows (T_2 \oslash \mathcal{L}^*))\,. \\
 & & \quad [\![A]\!]^{\Delta,\alpha}_{\mathrm{tm}}\ T_2\ (\mathsf{combine\text{-}l\text{-}l}^* \circ ((\mu_2 \circ \mu) \otimes \mathsf{id})) \\
[\![\nabla\alpha\,.\,A]\!]^{\Delta}_{\mathrm{pf}} & = & \lambda\, T\,.\,\lambda\,\mu\,.\,\lambda\, x\!:\!\mu([\![\nabla\alpha\,.\,A]\!]^{\Delta}_{\text{tm-tp}})\,.\,\Pi T_2\,.\,\Pi\mu_2\!:\!(T \rightrightarrows T_2 \oslash \mathcal{L}^*)\,. \\
 & & \quad [\![A]\!]^{\Delta,\alpha}_{\mathrm{pf}}\ T_2\ (\mathsf{combine\text{-}l\text{-}l}^* \circ ((\mu_2 \circ \mu) \otimes \mathsf{id})) \\
 & & \quad\quad (\lambda\, T_3\,.\,\lambda\,\mu_3\!:\!(T_2 \rightrightarrows T_3)\,.\,x\ (T_3 \oslash \mathcal{L}^*)\ ((\mu_3 \otimes \mathsf{id}) \circ \mu_2)\ T_3\ \mathsf{id}) \\
\\
[\![a\ \vec{M}]\!]^{\Delta}_{\mathrm{tm}} & = & \lambda\, T\,.\,\lambda\,\mu\,.\,a\ T\ (\mu \circ \Delta(\emptyset))\ \mu([\![\vec{M}]\!]^{\Delta}_{\mathrm{args}}) \\
[\![a\ \vec{M}]\!]^{\Delta}_{\mathrm{pf}} & = & \lambda\, T\,.\,\lambda\,\mu\,.\,\lambda\, x\!:\!\mu([\![a\ \vec{M}]\!]^{\Delta}_{\text{tm-tp}})\,. \\
 & & \quad \mathsf{valid}\text{-}a\ T\ (\mu \circ \Delta(\emptyset))\ \mu([\![\vec{M}]\!]^{\Delta}_{\mathrm{args}})\ x \\
\\
[\![\mathsf{Name}]\!]^{\Delta}_{\mathrm{tm}} & = & \lambda\, T\,.\,\lambda\,\mu\,.\,\mathsf{name}\ T\ (\mu \circ \Delta(\emptyset)) \\
[\![\mathsf{Name}]\!]^{\Delta}_{\mathrm{pf}} & = & \lambda\, T\,.\,\lambda\,\mu\,.\,\lambda\, x\!:\!\mu([\![\mathsf{Name}]\!]^{\Delta}_{\text{tm-tp}})\,.\,\mathsf{valid\text{-}name}\ T\ (\mu \circ \Delta(\emptyset))\ x
\end{array}
$$

Figure 6.2: Translation of CNIC to CIC + $\mathbb{T}$: Types

$$
\begin{array}{l}
[\![u\ \langle\vec{\alpha}\rangle\ \vec{N}\ N]\!]^{\Delta}_{\mathrm{tm/pf}} = \\
\quad \lambda\, T\,.\,\lambda\, \mu\,. \\
\quad\quad (u\ (T \oslash \vec{\beta})\ (\mathsf{tcadd}(\vec{\beta}) \circ \mu \circ \Delta(\emptyset))\ (\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![\vec{N}]\!]^{\Delta}_{\mathrm{args}}))\ \\
\quad\quad\quad (\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![N]\!]^{\Delta}_{\mathrm{tm}}))\ ((\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![N]\!]^{\Delta}_{\mathrm{pf}}))\ (T \oslash \vec{\beta})\ \mathsf{id})\ \\
\quad\quad\quad (\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![N]\!]^{\Delta}_{\mathrm{pf}}))\ (\mathsf{mk\text{-}valid\text{-}eq\text{-}pf}\ (\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![N]\!]^{\Delta}_{\mathrm{pf}})))\ \\
\quad\quad\quad T\ (\mu \circ \Delta(\vec{\alpha}))\ (\mathsf{eq\text{-}refl}\ (\mathsf{mk\text{-}pair}\ (T \oslash \vec{\beta})\ (\mathsf{tcadd}(\vec{\beta}) \circ \mu \circ \Delta(\emptyset))))).1/2\ T\ \mathsf{id}
\end{array}
$$

$$
\begin{array}{l}
[\![(\mathbf{fun}\ u\ (\vec{\alpha}, \Gamma^{\mathrm{x}})\ (\nu\, \vec{\beta}\,.\, \vec{P} \to \vec{M}))\ \langle\vec{\alpha}\rangle\ \vec{N}\ N]\!]^{\Delta}_{\mathrm{tm/pf}} = \\
\quad \lambda\, T\,.\,\lambda\, \mu\,. \\
\quad\quad ([\![\mathbf{fun}\ u\ (\vec{\alpha}, \Gamma^{\mathrm{x}})\ (\nu\, \vec{\beta}\,.\, \vec{P} \to \vec{M})]\!]_{\mathrm{pmfun}}\ \\
\quad\quad\quad (T \oslash \vec{\beta})\ (\mathsf{tcadd}(\vec{\beta}) \circ \mu \circ \Delta(\emptyset))\ (\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![\vec{N}]\!]^{\Delta}_{\mathrm{args}}))\ \\
\quad\quad\quad (\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![N]\!]^{\Delta}_{\mathrm{tm}}))\ ((\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![N]\!]^{\Delta}_{\mathrm{pf}}))\ (T \oslash \vec{\beta})\ \mathsf{id})\ \\
\quad\quad\quad (\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![N]\!]^{\Delta}_{\mathrm{pf}}))\ (\mathsf{mk\text{-}valid\text{-}eq\text{-}pf}\ (\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![N]\!]^{\Delta}_{\mathrm{pf}})))\ \\
\quad\quad\quad T\ (\mu \circ \Delta(\vec{\alpha}))\ (\mathsf{eq\text{-}refl}\ (\mathsf{mk\text{-}pair}\ (T \oslash \vec{\beta})\ (\mathsf{tcadd}(\vec{\beta}) \circ \mu \circ \Delta(\emptyset))))).1/2\ T\ \mathsf{id}
\end{array}
$$

$$
\begin{array}{l}
[\![\mathbf{fun}\ u\ (\vec{\alpha}, \Gamma^{\mathrm{x}}{\uparrow}^{\vec{\beta}})\ (\nu\, \vec{\beta}\,.\, \vec{c}\ (\vec{\Gamma^{\mathrm{x}}}\ \langle\vec{\beta}\rangle) \setminus \vec{\Gamma^{\mathrm{x}}}{\uparrow}^{\vec{\beta}} \to \vec{M})]\!]_{\mathrm{pmfun}} = \\
\quad \mathbf{fun}\ u\ (T_a, \mu_a : (\emptyset \rightrightarrows T_a), [\![\Gamma^{\mathrm{x}}]\!]^{\mu_a;\cdot}_{\mathrm{ctxt}}) \\
\quad\quad\quad (\mathsf{valid}\text{-}c_1\ T_x\ \mu_x\ \Gamma^{\mathrm{x}}_1 \setminus T_x, \mu_x, [\![\Gamma^{\mathrm{x}}_1]\!]^{\mu;\cdot}_{\mathrm{ctxt}} \to \\
\quad\quad\quad\quad \lambda\, x_{\mathrm{pf}} : \Pi T_c\,.\,\Pi \mu_c\,.\,\mathsf{valid}\text{-}a\ T_c\ \mu_c\ (\mu_c \circ \mathsf{eval})([\![\vec{N_c}]\!]^{(\mu_x;\Gamma^{\mathrm{x}}_{c_1})}_{\mathrm{args}})\,. \\
\quad\quad\quad\quad \lambda\, e_{\mathrm{pf}} : \mathsf{eq}\ (\lambda\, T_c\,.\,\lambda\, \mu_c\,.\,\mathsf{valid}\text{-}c_i\ T_c\ \mu_c\ \mu_c(\Gamma^{\mathrm{x}}_1, x))\ x_{\mathrm{pf}}\,. \\
\quad\quad\quad\quad \lambda\, T_\beta\,.\,\lambda\, \mu_\beta : (\vec{\alpha} \rightrightarrows T_\beta)\,. \\
\quad\quad\quad\quad \lambda\, e : \mathsf{eq}\ (\mathsf{mk\text{-}pair}\ T_x\ \mu_x)\ (\mathsf{mk\text{-}pair}\ (T_\beta \oslash \vec{\beta})\ (\mathsf{tcadd}(\vec{\beta}) \circ \mu_\beta \circ \mathsf{tcadd}(\vec{\alpha})))\,. \\
\quad\quad\quad\quad\quad \mathsf{cast}\ e_{\mathrm{pf}}, (\mathsf{eq\text{-}symm}\ e) \\
\quad\quad\quad\quad\quad\quad (\mathsf{mk\text{-}pair} \\
\quad\quad\quad\quad\quad\quad\quad \mathsf{eval}([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T_\beta\ (\mathsf{cast}\ e\ \Gamma^{\mathrm{x}}_1)/\Gamma^{\mathrm{x}}_1][\![M_1]\!]^{\vec{\alpha},(\mu_\beta;\Gamma^{\mathrm{x}}_1)}_{\mathrm{tm}}) \\
\quad\quad\quad\quad\quad\quad\quad \mathsf{eval}([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T_\beta\ (\mathsf{cast}\ e\ \Gamma^{\mathrm{x}}_1)/\Gamma^{\mathrm{x}}_1][\![M_1]\!]^{\vec{\alpha},(\mu_\beta;\Gamma^{\mathrm{x}}_1)}_{\mathrm{pf}}))\ | \\
\quad\quad\quad \vdots \\
\quad\quad\quad | \ \mathsf{valid}\text{-}c_n\ \ldots)
\end{array}
$$

$$
\begin{array}{l}
[\![(\mathbf{nfun}\ (\vec{\alpha})\ (\vec{P^{\alpha}} \to \vec{M}))\ \langle\vec{\alpha}\rangle\ N]\!]^{\Delta}_{\mathrm{tm/pf}} = \\
\quad \lambda\, T\,.\,\lambda\, \mu\,. \\
\quad\quad (\mathbf{fun}\ (T_2, \mu_2, x, x_{\mathrm{pf}}) \\
\quad\quad\quad\quad (\mathsf{name\text{-}matching}_{\vec{\alpha},i}\ T_x\ \mu_x \setminus T_x, \mu_x : (\vec{\alpha} \rightrightarrows T_x) \to [\![M_{\alpha_i}]\!]^{\vec{\alpha},(\mu_x;\cdot)}_{\mathrm{tm/pf}}\ T_x\ \mathsf{eval} \\
\quad\quad\quad\quad\ | \ \ldots \\
\quad\quad\quad\quad\ | \ \mathsf{fresh\text{-}matching}_{\vec{\alpha}}\ T_x\ \mu_x \setminus T_x, \mu_x : (\vec{\alpha}, \alpha \rightrightarrows T_x) \to [\![M_\alpha]\!]^{\vec{\alpha},\alpha,(\mu_x;\cdot)}_{\mathrm{tm/pf}}\ T_x\ \mathsf{eval}) \\
\quad\quad)\ T\ (\mathsf{find\text{-}matching}_{\vec{\alpha}}\ T\ (\mu \circ \Delta(\vec{\alpha}))\ \mu([\![N]\!]^{\Delta}_{\mathrm{tm}})\ \mu([\![N]\!]^{\Delta}_{\mathrm{pf}}))
\end{array}
$$

Figure 6.3: Translation of CNIC to CIC + $\mathbb{T}$: Pattern-Matching Functions

$$[\![\cdot]\!]^{\mu;\vec{x}}_{\text{ctxt}} = \cdot$$
$$[\![x : A, \Gamma]\!]^{\mu;\vec{x}}_{\text{ctxt}} = x : \mathsf{eval}([\![A]\!]^{(\mu;\vec{x})}_{\text{tm-tp}}), x_{\text{pf}} : \mathsf{eval}([\![A]\!]^{(\mu;\vec{x})}_{\text{pf-tp}}(x)), [\![\Gamma]\!]^{\mu;\vec{x},x,x_{\text{pf}}}_{\text{ctxt}}$$

$$[\![\cdot]\!]_{\text{mctxt}} = \cdot$$

$$\begin{array}{l}
[\![\Sigma, a : \Pi\vec{x} : \vec{A} \,.\, \mathsf{Type}_i]\!]_{\text{mctxt}} = \\
\quad [\![\Sigma]\!]_{\text{mctxt}}, \\
\quad a \quad : \quad \Pi T \,.\, \Pi\mu : (\emptyset \rightrightarrows T) \,.\, \Pi [\![\vec{x} : \vec{A}]\!]^{\mu;\cdot}_{\text{ctxt}} \,.\, \mathsf{Type}_i, \\
\quad \mathsf{valid}\text{-}a \quad : \quad \Pi T \,.\, \Pi\mu : (\emptyset \rightrightarrows T) \,.\, \Pi [\![\vec{x} : \vec{A}]\!]^{\mu;\cdot}_{\text{ctxt}} \,.\, (a\ T\ \mu\ x_1\ x_{\text{pf-1}} \ldots x_n\ x_{\text{pf-}n}) \Rightarrow \mathsf{Type}_i
\end{array}$$

$$\begin{array}{l}
[\![\Sigma, c : \Pi\vec{x} : \vec{A} \,.\, \Pi\vec{y} : \vec{A}_a \,.\, a\ \vec{M}]\!]_{\text{mctxt}} = \\
\quad [\![\Sigma]\!]_{\text{mctxt}}, \\
\quad c \quad : \quad \Pi T \,.\, \Pi\mu : (\emptyset \rightrightarrows T) \,.\, \Pi [\![\vec{x} : \vec{A}]\!]^{\mu;\cdot}_{\text{ctxt}} \,. \\
\qquad\qquad (\mathsf{eval}[\![\vec{A}_a]\!]^{(\mu;x_1,x_{\text{pf-1}},\ldots,x_n,x_{\text{pf-}n})}_{\text{tm-tp}}) \Rightarrow a\ T\ \mu\ \mathsf{eval}([\![\vec{M}]\!]^{(\mu;x_1,x_{\text{pf-1}},\ldots,x_n,x_{\text{pf-}n})}_{\text{args}}) \\
\quad \mathsf{valid}\text{-}c \quad : \quad \Pi T \,.\, \Pi\mu : (\emptyset \rightrightarrows T) \,.\, \Pi [\![\vec{x} : \vec{A}, \vec{y} : \vec{A}_a]\!]^{\mu;\cdot}_{\text{ctxt}} \,. \\
\qquad\qquad \mathsf{valid}\text{-}a\ T\ \mu\ [\![\vec{M}]\!]^{(\mu;\overrightarrow{x,x_{\text{pf}}})}_{\text{args}}\ \mathsf{eval}([\![c\ \vec{x}]\!]^{(\mu;\overrightarrow{x,x_{\text{pf}}})}_{\text{tm}})
\end{array}$$

$$\begin{array}{l}
[\![\Sigma, u : \nabla\vec{\alpha} \,.\, \Pi(\vec{x} : \vec{A})\!\uparrow^{\vec{\beta}} \,.\, \Pi x : (\nabla\vec{\beta} \,.\, a\ (\vec{x}\ \langle\vec{\beta}\rangle)) \,.\, B]\!]_{\text{mctxt}} = \\
\quad [\![\Sigma]\!]_{\text{mctxt}}, \\
\quad u : \Pi T \,.\, \Pi\mu : (\emptyset \rightrightarrows T) \,.\, \Pi [\![\vec{x} : \vec{A}]\!]^{\mu;\cdot}_{\text{ctxt}} \,.\, \Pi x : \mathsf{eval}([\![a\ \vec{x}]\!]^{(\mu;\overrightarrow{x,x_{\text{pf}}})}_{\text{tm-tp}}) \,. \\
\qquad \Pi x_{\text{valid}} : \mathsf{valid}\text{-}a\ T\ \mu\ \overrightarrow{x\ x_{\text{pf}}} \,.\, \Pi x_{\text{pf}} : \mathsf{eval}([\![a\ \vec{x}]\!]^{(\mu;\overrightarrow{x,x_{\text{pf}}})}_{\text{pf-tp}}(x)) \,. \\
\qquad \Pi e_{\text{pf}} : \mathsf{eq}\ (\mathsf{re\text{-}move}\text{-}a\ T\ \mu\ \overrightarrow{x\ x_{\text{pf}}}\ x\ x_{\text{valid}})\ x_{\text{pf}} \,.\, \Pi T_\beta \,.\, \Pi\mu_\beta : (\vec{\alpha} \rightrightarrows T_\beta) \,. \\
\qquad \Pi e : \mathsf{eq}\ (\mathsf{mk\text{-}pair}\ T\ \mu)\ (\mathsf{mk\text{-}pair}\ (T_\beta \oslash \vec{\beta})\ (\mathsf{tcadd}(\vec{\beta}) \circ \mu_\beta \circ \mathsf{tcadd}(\vec{\alpha}))) \,. \\
\qquad\quad \mathsf{pair} \\
\qquad\qquad \mathsf{eval}([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T_\beta\ (\mathsf{cast}\ e\ (\overrightarrow{x, x_{\text{pf}}}, x, x_{\text{pf}}))/\overrightarrow{x, x_{\text{pf}}}, x, x_{\text{pf}}][\![B]\!]^{\vec{\alpha},(\mu;\overrightarrow{x,x_{\text{pf}}},x,x_{\text{pf}})}_{\text{tm-tp}}) \\
\qquad\qquad (\lambda z \,.\, \mathsf{eval}([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T_\beta\ (\mathsf{cast}\ e\ (\overrightarrow{x, x_{\text{pf}}}, x, x_{\text{pf}})) \\
\qquad\qquad\qquad\qquad /\overrightarrow{x, x_{\text{pf}}}, x, x_{\text{pf}}][\![B]\!]^{\vec{\alpha},(\mu;\overrightarrow{x,x_{\text{pf}}},x,x_{\text{pf}})}_{\text{pf-tp}}(z)))
\end{array}$$

Figure 6.4: Translation of CNIC to CIC + $\mathbb{T}$: Contexts

translations of $\vec{M}_1$, and just the term translations of $\vec{M}_2$. The proof translation of $c\ \vec{M}_1\ \vec{M}_2$ applied $\mathsf{valid}\text{-}c$ to T, μ, and the term and proof translations of the arguments.

The above type and object constructor declarations must define valid inductive types in CIC + $\mathbb{T}$. This makes two requirements on the types in the declarations. First, the types given to the type and object constructors must themselves be well-typed. This will be straightforward after the proof of Translation Typing, given below. Note that Translation Typing requires that translations of terms appear as $\mathsf{eval}([\![M]\!]^{\Delta}_{\mathrm{tm/pf}})$, which is satisfied here. The second requirement on the types in the above declarations is that these declarations satisfy the standard positivity and type universe constraints on inductive types as in, for example, the Coq system. These mostly follow directly from the fact that the declarations here mirror the declarations in CNIC in a manner that preserves type universes and strict positivity. The only complication here is that the type for $\mathsf{valid}\text{-}c$ itself contains c. Thus the type $\mathsf{valid}\text{-}a$ must intuitively be defined after a, as mentioned above, so no constructor for a can use the type $\mathsf{valid}\text{-}a$.

The fact that proving the well-typedness of the types for constructors requires a forward use of Translation Typing here, though it looks at first like a circular argument, is in fact a giant induction on the number of type and object constructors in the modal context Σ of a given typing derivation $\Sigma; \Gamma \vdash M : A$ of CNIC. This is because the type of a constructor in Σ can only contain constructors that occur earlier in Σ. Thus, if a is the first constructor in Σ, then the types for a and $\mathsf{valid}\text{-}a$ given above contain no other constructors. Therefore Translation Typing may be used directly to prove that these types are themselves well-formed. The next inductive step can then assume that the given types for a and $\mathsf{valid}\text{-}a$ are well-typed to prove that the types for the second constructor are well-formed, and this can be repeated for all of Σ.

Names are translated using the following inductive types:

$$
\begin{array}{lcl}
\mathsf{name} & : & \Pi T\,.\,\Pi \mu : \emptyset \rightrightarrows T\,.\,\mathsf{Type}_0 \\
\mathsf{mk\text{-}name} & : & \Pi T\,.\,\Pi \mu : (\emptyset \oslash \mathcal{L}) \rightrightarrows T\,.\,\mathsf{name}\ T\ (\mu \circ \mathsf{add\text{-}l}) \\
\mathsf{valid\text{-}name} & : & \Pi T\,.\,\Pi \mu : \emptyset \rightrightarrows T\,.\,\mathsf{name}\ T\ \mu \Rightarrow \mathsf{Type}_0 \\
\mathsf{mk\text{-}valid\text{-}name} & : & \Pi T\,.\,\Pi \mu : (\emptyset \oslash \mathcal{L}) \rightrightarrows T\,.\,\mathsf{valid\text{-}name}\ T\ (\mu \circ \mathsf{add\text{-}l}) \\
 & & \qquad (\lambda\, T_2\,.\,\lambda\, \mu_2\,.\,\mathsf{mk\text{-}name}\ T_2\ (\mu_2 \circ \mu))
\end{array}
$$

$\mathsf{valid\text{-}name}$ acts as the logical relation for names. Intuitively it states that a term is in the logical relation for names if it is of the form $\lambda\, T_2 \,.\, \lambda\, \mu_2 \,.\, \mathsf{mk\text{-}name}\ T_2\ (\mu_2 \circ \mu)$.

It turns out that $\mathsf{valid\text{-}name}$ and $\mathsf{valid\text{-}}a$ have a very useful property: if x_1 and x_2 both have the same $\mathsf{valid\text{-}name}$ or $\mathsf{valid\text{-}}a$ type then those proofs must be equal. This essentially shows why we do not need a logical relation for our logical relations proofs, as all logical relations proofs will be "valid translations" because they are all equal. This can be proved for $\mathsf{valid\text{-}name}$ with the folowing function:

$$
\begin{array}{l}
\textbf{fun}\ (T, \mu) \\
\quad (\mathsf{mk\text{-}valid\text{-}name}\ T\ \mu_n \setminus T, \mu_n \rightarrow \\
\qquad \lambda\, v_2 : (\mathsf{valid\text{-}name}\ T\ (\mu_n \circ \mathsf{add\text{-}l})\ (\lambda\, T_2 \,.\, \lambda\, \mu_2 \,.\, \mathsf{mk\text{-}name}\ T_2\ (\mu_2 \circ \mu_n)))\,. \\
\quad\ (\textbf{fun}\ (T, \mu') \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad) \\
\qquad\ (\mathsf{mk\text{-}valid\text{-}name}\ T\ \mu'_n \setminus T, \mu'_n \rightarrow \\
\qquad\qquad \lambda\, e : \mathsf{eq}\ (\lambda\, T_2 \,.\, \lambda\, \mu_2 \,.\, \mathsf{mk\text{-}name}\ T_2\ (\mu_2 \circ \mu_n))\,. \\
\qquad\qquad\qquad (\lambda\, T_2 \,.\, \lambda\, \mu_2 \,.\, \mathsf{mk\text{-}name}\ T_2\ (\mu_2 \circ \mu'_n)) \\
\qquad\qquad\quad \mathsf{cast}\ e\ (\mathsf{refl\text{-}equal}\ (\mathsf{mk\text{-}valid\text{-}name}\ T\ \mu_n))) \\
\quad\ )\ (\mathsf{refl\text{-}equal}\ (\lambda\, T_2 \,.\, \lambda\, \mu_2 \,.\, \mathsf{mk\text{-}name}\ T_2\ (\mu_2 \circ \mu_n)))
\end{array}
$$

This function has type

$$
\begin{array}{l}
\Pi T \,.\, \Pi \mu : (\emptyset \rightrightarrows T) \,.\, \Pi x : (\Pi T_2 \,.\, \Pi \mu_2 \,.\, \mathsf{name}\ T_2\ (\mu_2 \circ \mu))\,. \\
\quad \Pi v_1 : \mathsf{valid\text{-}name}\ T\ \mu\ x \,.\, \Pi v_2 : \mathsf{valid\text{-}name}\ T\ \mu\ x \,.\, \mathsf{eq}\ v_1\ v_2
\end{array}
$$

and is called $\mathsf{valid\text{-}name\text{-}eq}$ below. The function $\mathsf{valid\text{-}}a\mathsf{\text{-}eq}$ of type

$$
\begin{array}{l}
\Pi T \,.\, \Pi \mu : (\emptyset \rightrightarrows T) \,.\, \Pi \Gamma \,.\, \Pi x : (\Pi T_2 \,.\, \Pi \mu_2 \,.\, a\ T_2\ (\mu_2 \circ \mu)\ \mu_2(\Gamma))\,. \\
\quad \Pi v_1 : \mathsf{valid\text{-}}a\ T\ \mu\ \vec{M}\ x \,.\, \Pi v_2 : \mathsf{valid\text{-}}a\ T\ \mu\ \vec{M}\ x \,.\, \mathsf{eq}\ v_1\ v_2
\end{array}
$$

can be defined in a similar fashion.

These equalities allow us to safely "re-move" $\mathsf{valid\text{-}name}$ and $\mathsf{valid\text{-}}a$ proofs. For example, if M has type Name in CNIC, then $[\![M]\!]^{\Delta}_{\text{pf}}$ is a term set of $\mathsf{valid\text{-}name}$ proofs. Applying $[\![M]\!]^{\Delta}_{\text{pf}}$ to a particular T and μ yields a particular $\mathsf{valid\text{-}name}$ proof in a particular world. Specifically, $[\![M]\!]^{\Delta}_{\text{pf}}\ T\ \mu$ has type $\mathsf{valid\text{-}name}\ T\ (\mu \circ \Delta(\emptyset))\ \mu([\![M]\!]^{\Delta}_{\text{tm}})$. This proof is no longer a term set, and so cannot be moved again directly using the renaming operation $\mu(\cdot)$. The proof can be re-moved, however, using the the function

re-move-name, defined as follows:

$$\begin{aligned}&\textbf{fun}\ \mathsf{re\text{-}move\text{-}name}\ (T, \mu, x)\\ &\qquad (\mathsf{mk\text{-}valid\text{-}name}\ T\ \mu \setminus T, \mu \rightarrow\\ &\qquad\quad \lambda\, T_2\,.\,\lambda\, \mu_2\,.\,\mathsf{mk\text{-}valid\text{-}name}\ T_2\ (\mu_2 \circ \mu))\end{aligned}$$

The re-move-name function has type

$$\begin{aligned}&\Pi T\,.\,\Pi \mu : (\emptyset \rightrightarrows T)\,.\,\Pi x : (\Pi T_2\,.\,\Pi \mu_2\,.\,\mathsf{name}\ T_2\ (\mu_2 \circ \mu))\,.\,\mathsf{valid\text{-}name}\ T\ \mu\ x \Rightarrow\\ &\quad \Pi T_2\,.\,\Pi \mu_2\,.\,\mathsf{valid\text{-}name}\ T_2\ (\mu_2 \circ \mu)\ \mu_2(x)\end{aligned}$$

meaning that re-move-name $T\ \mu\ x$ takes any proof of type valid-name $T\ \mu\ x$ and creates a term set function of type $\Pi T_2\,.\,\Pi \mu_2\,.\,\mathsf{valid\text{-}name}\ T_2\ (\mu_2 \circ \mu)\ \mu_2(x)$. Further, the function valid-name-eq above demonstrates that if a valid-name proof $[\![M]\!]^{\Delta}_{\mathrm{pf}}$ is applied to T and μ and then re-moved, the result is equal to $\mu([\![M]\!]^{\Delta}_{\mathrm{pf}})$. The function witnessing this equality is called mk-valid-eq-pf below. A similar function to re-move-name can be defined for inductive types a as follows:

$$\begin{aligned}&\textbf{fun}\ \mathsf{re\text{-}move\text{-}}a\ (T, \mu, \Gamma, x)\\ &\qquad (\mathsf{valid\text{-}}c_1\ T\ \mu\ \Gamma_1 \setminus T, \mu, \Gamma_1 \rightarrow\\ &\qquad\quad \lambda\, T_2\,.\,\lambda\, \mu_2\,.\,\mathsf{valid\text{-}}c_1\ T_2\ (\mu_2 \circ \mu)\ \mu_2(\Gamma_1)\ \mu_2(x)\\ &\qquad\ \vdots\\ &\qquad \mathsf{valid\text{-}}c_n\ T\ \mu\ \Gamma_n \setminus T, \mu, \Gamma_n \rightarrow\\ &\qquad\quad \lambda\, T_2\,.\,\lambda\, \mu_2\,.\,\mathsf{valid\text{-}}c_n\ T_2\ (\mu_2 \circ \mu)\ \mu_2(\Gamma_n)\ \mu_2(x))\end{aligned}$$

Again, the valid-a-eq function above demonstrates that

$$\mathsf{re\text{-}move\text{-}}a\ T\ \mu\ \overrightarrow{x\ x_{\mathrm{pf}}}\ M_{\mathrm{tm}}\ (M_{\mathrm{pf}}\ T\ \mu)$$

is equal to M_{pf} for any type indices $\vec{x}$ and $\vec{x_{\mathrm{pf}}}$ and any M_{tm} and M_{pf}. The function witnessing this fact is also called mk-valid-eq-pf below. It will be clear from context which version of mk-valid-eq-pf is intended.

The translations of ν-abstractions take in two pairs of tree types and mappings. The first mapping, μ, maps the world Δ, without the new name α, to T. The second mapping, μ_2, is used in conjunction with the extended world that does contain α. μ_2 has return type $T_2 \between \mathcal{L}^*$, which conceptually specifies a world of type T_2 with a "hole"

in it. This hole specifies where the new name α should go in the world. The body of the ν-abstraction is translated in world Δ, α with the new name, and the result is passed the mapping $(\mathsf{combine\text{-}l\text{-}l}^* \circ ((\mu_2 \circ \mu) \otimes \mathsf{id}))$. The mapping $((\mu_2 \circ \mu) \otimes \mathsf{id})$ leaves α in the same place (because of the tensor with id) and yields a world with tree type $(T_2 \mathbin{⅋} \mathcal{L}^*) \mathbin{⅋} \mathcal{L}$, where the right-most leaf is the new name α. $\mathsf{combine\text{-}l\text{-}l}^*$ then "fills the hole" by mapping α into the negative leaf. This yields a world of type T_2.

To translate the name replacement $M\ \langle\alpha\rangle$, M is first translated in the translation context Δ, α^*. This cancels out the name α in Δ, or, under the hole interpretation in the previous paragraph, Δ, α^* creates a hole attached to α in Δ. The translation of M will be of the type as that of ν-abstractions, so M will take two pairs of tree types and mappings. For the first mapping, the mapping $(\mu \otimes \mathsf{id})$ is used. This applies μ to the Δ part of Δ, α^* but leaves α^* alone, yielding a tree of type $T \mathbin{⅋} \mathcal{L}^*$. The second mapping to ν-abstractions is a mapping to a tree of type $T_2 \mathbin{⅋} \mathcal{L}^*$. Since we already have such a tree type, the identity mapping id can be passed for the second mapping, leaving the hole intact. If M is a ν-abstraction, these mappings are then combined with $\mathsf{combine\text{-}l\text{-}l}^*$, which maps a new name into the hole represented by $\mathcal{L}^*$ in $T \mathbin{⅋} \mathcal{L}^*$.

The difficulty in translating λ-abstractions is in defining how to rename them. This is because renaming a λ-abstraction means the argument is already going to be renamed. Thus a renaming in a λ-abstraction should rename everything but the variables, as these will already be in the new world. To achieve this behavior, the input mapping μ is put into the translation context, along with the variable. Recall that this specifies the variable as already having been mapped by μ. The body of the λ-abstraction is the passed the mapping eval. When eval reaches the variable in question, eval just cancels out the eval_μ^{-1} applied to the variable. Names and other variables, however, are already in Δ and so, when they occur, will have $\mathsf{add}(\mu)$ applied to them. When eval reaches these, it turns $\mathsf{add}(\mu)$ into μ, and the given construct does have μ applied to it.

Translations of applications $M\ N$ are straightforward. The input μ is passed to the translation of M, telling it to expect an argument that has been renamed by μ. Then the argument is renamed by μ as promised.

The $[\![A]\!]^{\Delta}_{\mathrm{tm}}$ translations of the types, given in Figure 6.2, are straightforward. They directly reflect the above discussion on how terms are translated. The proof translations are more interesting, as they define the logical relations for the types. The proof translation for Name specifies that the logical relation includes only those terms for which $\mathsf{valid\text{-}name}$ holds. For constructors, the logical relation states that valid-a should likewise hold. The logical relation for ∇ types essentially states that a term M is in the logical relation for $\nabla\alpha\,.\,A$ if applying a name replacement to it is in the logical relation for A. If a name replacement is used on M then it will pass $(\mu' \otimes \mathsf{id})$ for the μ_2 argument, and thus the term to which $[\![A]\!]^{\Delta}_{\mathrm{pf}}$ is applied is

$$(\lambda\, T_3\,.\,\lambda\,\mu_3\,.\,x\ (T_3 \between \mathcal{L}^*)\ ((\mu_3 \circ \mu') \otimes \mathsf{id})\ T_3\ \mathsf{id})$$

which is the renaming by μ' of the name replacement of x. This reflects the notion described above that the logical relation for non-base types should state that all elimination forms of the term result in terms that are in the logical relation for the resulting type. In the logical relation for function types this is even more apparent, as the term in the predicate is exactly the translation of an application. Finally, the proof translation for Type_i essentially states that the proof translations of types should be predicates.

To define the translation of pattern-matching it will be necessary to add and remove ν-abstractions from a term. This is defined with the operations $\mathbf{add\text{-}nus}_{\vec{\beta}}$ and $\mathbf{remove\text{-}nus}_{\vec{\beta}}$:

$$\begin{array}{lcl}
\mathbf{remove\text{-}nus}_{\alpha,\vec{\alpha}}\ T\ M & = & \mathbf{remove\text{-}nus}_{\vec{\alpha}}\ (\mathcal{L} \between T) \\
 & & \qquad (\lambda\, T_2\,.\,\lambda\,\mu_2 : ((T \between \mathcal{L}) \rightrightarrows (T_2))\,. \\
 & & \qquad\ M\ (T_2 \between \mathcal{L}^*)\ ((\mu_2 \otimes \mathsf{id}) \circ \mathsf{tcadd}(\alpha, \alpha^*))\ T_2\ \mathsf{id}) \\
\mathbf{remove\text{-}nus}.\ T\ M & = & M \\
\mathbf{add\text{-}nus}_{\alpha,\vec{\alpha}}\ T\ M & = & \lambda\, T_2\,.\,\lambda\,\mu_2\,.\,\lambda\, T_3\,.\,\lambda\,\mu_3 : (T_2 \rightrightarrows (T_3 \between \mathcal{L}^*))\,. \\
 & & \qquad (\mathbf{add\text{-}nus}_{\vec{\alpha}}\ (T \between \mathcal{L})\ M)\ T_3\ (\mathsf{combine\text{-}l\text{-}l}^* \circ ((\mu_3 \circ \mu_2) \otimes \mathsf{id})) \\
\mathbf{add\text{-}nus}.\ T\ M & = & M
\end{array}$$

$\mathbf{remove\text{-}nus}_{\vec{\beta}}$ follows the definition of the translation of name replacement, so it is essentially assuming a new world of type $T \between \vec{\beta}$ and name replacing all the new names $\vec{\beta}$ into its argument. Conversely, $\mathbf{add\text{-}nus}_{\vec{\beta}}$ follows the definition of the translation

of ν-abstractions. Thus it is essentially taking a term in a world of type $T \mathbin{\oslash} \vec{\beta}$ and binding the $\vec{\beta}$ with ν-abstractions, yielding a term in a world of type T.

The translation of pattern-matching functions is complex. Similar to constructors, pattern-matching functions are assumed to be fully applied up to their scrutinee. This can be achieved through η-expansion. Pattern-matching functions work here by matching over the proof translation of the scrutinee, that is, over the valid-a proof. To match inside ν-abstractions, any leading ν-abstractions are stripped with $\textbf{remove-nus}_{\vec{\beta}}$. The pattern-matching thus happens in an extended world with names $\vec{\beta}$ added. Inside the pattern-matching function, these ν-abstractions are re-added with $\textbf{add-nus}_{\vec{\beta}}$.

The translation of pattern-matching functions is broken into two pieces, the translation of the pattern-matching function itself and the translation of its application. The translation of the pattern-matching function, denoted $[\![\cdot]\!]_{\text{pmfun}}$, does not have a term and a proof piece, but is instead all one piece. This translation matches over the type valid-a that the term input satisfies its logical relation. The parameters are the parameters to this type, which include a tree type T_a, a mapping μ_a of type $\emptyset \rightrightarrows T_a$, and the term and proof translations of the arguments.

A number of arguments are also taken inside the pattern match. The first argument, x_{pf}, is a valid-a proof for the same term as the pattern, except that the pattern can no longer be mapped, as it is not a function. The second argument, e_{pf}, is a proof that the pattern is in fact an instance of x_{pf}. The third and fourth arguments are a tree type T_β and a mapping from $\vec{\alpha}$ to T_β. The next argument, e, gives the purpose for T_β and μ_β, which is to show that the world of the pattern variables is actually equal to $\vec{\alpha}, (\mu_\beta; \vec{x}), \vec{\beta}$.

The return value of the pattern-matching function is translated twice, once for the term translation and once for the proof translation. These are translated in the world $\vec{\alpha}, (\mu_\beta; \Gamma_1^{\text{x}})$, corresponding to the typing judgment in CNIC that they be well-typed in context $\vec{\alpha}, \vec{x}$. Each copy must have the correct versions of the pattern variables, which are cast with e, to show they in a world of the form $\vec{\beta}$, and then their ν-abstractions are re-added. Finally, mk-pair makes a pair of the term and type translations of the return value.

The second piece of pattern-matching functions, the application, is performed as follows. First note that the notation $[\![M]\!]^{\Delta}_{\mathrm{tm/pf}}$ is intended to define both the term and proof translations at once. The only difference between the two is the projection .1 is taken in the term translation while the projection .2 is taken in the proof translation. This is denoted .1/2.

Since the arguments all have type $\nabla\vec{\beta}\,.\,A$ for some A, we must first remove the leading ν-abstractions using $\mathbf{remove\text{-}nus}_{\vec{\beta}}$. This yields terms in the world $\Delta', \vec{\beta}$ where Δ' is the world resulting from applying μ to Δ. This world can also be represented by the mapping $\mathsf{tcadd}(\vec{\beta}) \circ \mu \circ \Delta(\emptyset)$ of type $T \oslash \vec{\beta}$, so this type and world are passed as the first two arguments to the translation of the pattern-matching function. Next, the arguments themselves are passed, with the proof argument for the scrutinee being passed tree $T \oslash \vec{\beta}$ and mapping id so that we may get a term of type $\mathsf{valid}\text{-}a$. Next, the actual proof translation of the scrutinee is passed, along with the proof created by $\mathsf{mk\text{-}valid\text{-}eq\text{-}pf}$ that $\mathsf{re\text{-}move}\text{-}a$ of the scrutinee is equal to this actual proof translation of the scrutinee. Next, we pass the desired tree type T and mapping μ for the result along with a proof that the tree mapping we passed before is equal to $\mathsf{tcadd}(\vec{\beta}) \circ \mu$. Finally, we take the first projection .1 to get the term translation returned by the pattern-matching function, or take the second projection .2 to get the proof translation returned by the pattern-matching function. Since the result of the pattern-matching function is already in the appropriate world, it is then just passed the same tree type T and the identity mapping id.

For the name-matching functions, we here only translate those that do not traverse ν-abstractions, meaning those that have type $\nabla\vec{\alpha}\,.\,\Pi x : \mathsf{Name}\,.\,B$. It is straightforward to model name-matching functions that do traverse ν-abstractions using ones that do not by performing a name-matching inside the ν-abstraction, having that name-matching produce an element of an inductive type describing which case holds, and then traversing the ν-abstraction with a normal pattern-matching function. The possible results of name-matching in this case can be described by the following

inductive type:

$$\begin{array}{lcl}
\mathsf{name\text{-}matching}_{\vec{\alpha}} & : & \Pi T\,.\,\Pi\mu : (\vec{\alpha} \rightrightarrows T)\,. \\
& & \quad \Pi x : \mu([\![\mathsf{Name}]\!]^{\vec{\alpha}}_{\text{tm-tp}})\,.\,\mu([\![\mathsf{Name}]\!]^{\vec{\alpha}}_{\text{pf-tp}}) \Rightarrow \mathsf{Type}_0 \\
\mathsf{name\text{-}matching}_{\vec{\alpha},i} & : & \Pi T\,.\,\Pi\mu : (\vec{\alpha} \rightrightarrows T)\,.\,\mathsf{name\text{-}matching}_{\vec{\alpha}}\ T\ \mu\ \mu([\![\alpha_i]\!]^{\vec{\alpha}}_{\text{tm}})\ \mu([\![\alpha_i]\!]^{\vec{\alpha}}_{\text{pf}}) \\
\mathsf{fresh\text{-}matching}_{\vec{\alpha}} & : & \Pi T\,.\,\Pi\mu : (\vec{\alpha}, \alpha \rightrightarrows T)\,. \\
& & \quad \mathsf{name\text{-}matching}_{\vec{\alpha}}\ T\ \mu\ \mu([\![\alpha]\!]^{\vec{\alpha},\alpha}_{\text{tm}})\ \mu([\![\alpha]\!]^{\vec{\alpha},\alpha}_{\text{pf}})
\end{array}$$

To determine which of these holds, the following function can be used:

$$\begin{array}{l}
\textbf{fun}\ (T, \mu, x)\ (\mathsf{mk\text{-}valid\text{-}name}\ T\ \mu_n \setminus T, \mu_n \rightarrow \\
\qquad (\textbf{fun}\ (T, \mu) \\
\qquad\quad (\mathsf{name\text{-}case\text{-}i}_{\vec{\alpha},i}\ T\ \mu_i \setminus T, \mu_i \rightarrow \mathsf{name\text{-}matching}_{\vec{\alpha},i}\ T\ \mu_i \mid \\
\qquad\quad\ \mathsf{fresh\text{-}name\text{-}case}_{\vec{\alpha}}\ T\ \mu_f \setminus T, \mu_f \rightarrow \mathsf{fresh\text{-}matching}_{\vec{\alpha}}\ T\ \mu_f) \\
\qquad)\ T\ \mu_n \\
\quad)
\end{array}$$

This function has type

$$\begin{array}{l}
\Pi T\,.\,\Pi\mu : \emptyset \rightrightarrows T\,.\,\Pi x : (\Pi T_2\,.\,\Pi\mu_2\,.\,\mathsf{name}\ T_2\ (\mu_2 \circ \mu))\,. \\
\quad \Pi v : \mathsf{valid\text{-}name}\ T\ \mu\ x\,.\,\mathsf{name\text{-}matching}_{\vec{\alpha}}\ T\ \mu\ x\ (\mathsf{re\text{-}move\text{-}name}\ T\ \mu\ x\ v)
\end{array}$$

We denote by $\mathsf{find\text{-}matching}_{\vec{\alpha}}$ the function

$$\begin{array}{c}
\lambda T\,.\,\lambda\mu : (\emptyset \rightrightarrows T)\,.\,\lambda x : (\Pi T_2\,.\,\Pi\mu_2\,.\,\mathsf{name}\ T_2\ (\mu_2 \circ \mu))\,. \\
\lambda x_{\text{pf}} : (\Pi T_2\,.\,\Pi\mu_2\,.\,\mathsf{valid\text{-}name}\ T_2\ (\mu_2 \circ \mu)\ \mu_2(x))\,. \\
\quad \mathsf{cast}\ (\mathsf{mk\text{-}valid\text{-}eq\text{-}pf}\ x_{\text{pf}})\ (F\ T\ \mu\ x\ (x_{\text{pf}}\ T\ \mathsf{id}))
\end{array}$$

where F is the above function. This function returns a $\mathsf{name\text{-}matching}_{\vec{\alpha}}\ T\ \mu\ x\ x_{\text{pf}}$ for any input. Note that the cast by the result of mk-valid-eq-pf changes the re-move-name type above to that of x_{pf}. The translation of name-matching functions then simply calls $\mathsf{find\text{-}matching}_{\vec{\alpha}}$ and pattern-matches on the result.

Lemma 6.2.4. *For any M and any μ_1, μ_2:*

1. $[\![M]\!]^{\Delta}_{\text{tm/pf}} =_{\text{admin}} \lambda T\,.\,\lambda\mu\,.\,[\![M]\!]^{\Delta}_{\text{tm/pf}}\ T\ \mu \cong \mathsf{id}([\![M]\!]^{\Delta}_{\text{tm/pf}})$;

2. $\mu_1([\![M]\!]^{\Delta}_{\text{tm/pf}}) \cong \mu_2([\![M]\!]^{\Delta}_{\text{tm/pf}})$ *holds if and only if the equality* $[\![M]\!]^{\Delta}_{\text{tm/pf}}\ T\ (\mu \circ \mu_1) \cong [\![M]\!]^{\Delta}_{\text{tm/pf}}\ T\ (\mu \circ \mu_2)$ *holds for some variable* μ*, if and only if* $[(\mu \circ \mu_1)/\mu]M' \cong [(\mu \circ \mu_2)/\mu]M'$*, where* M' *is the part of* $[\![M]\!]^{\Delta}_{\text{tm/pf}}$ *after the initial* $\lambda T.\,\lambda \mu\,.$*;*

3. $\mu_2(\mu_1([\![M]\!]^{\Delta}_{\text{tm/pf}})) \cong (\mu_2 \circ \mu_1)([\![M]\!]^{\Delta}_{\text{tm/pf}})$*;*

4. $\mu_1([\![M]\!]^{\Delta}_{\text{tm-tp}}) \cong \mu_2([\![M]\!]^{\Delta}_{\text{tm-tp}})$ *if and only if* $\mu_1([\![M]\!]^{\Delta}_{\text{tm}}) \cong \mu_2([\![M]\!]^{\Delta}_{\text{tm}})$*; and*

5. $\mu_1([\![M]\!]^{\Delta}_{\text{pf-tp}}(x)) \cong \mu_2([\![M]\!]^{\Delta}_{\text{pf-tp}}(x))$ *if and only if* $\mu_1([\![M]\!]^{\Delta}_{\text{pf}}) \cong \mu_2([\![M]\!]^{\Delta}_{\text{pf}})$.

Proof. For item 1, every case in Figures 6.2, 6.1, and 6.3 starts with $\lambda T.\,\lambda \mu\,.$, so $[\![M]\!]^{\Delta}_{\text{tm/pf}} =_{\text{admin}} \lambda T.\,\lambda \mu\,.\,[\![M]\!]^{\Delta}_{\text{tm/pf}}\ T\ \mu$. Further, $\mathsf{id}([\![M]\!]^{\Delta}_{\text{tm/pf}})$ is defined as the term $\lambda T.\,\lambda \mu\,.\,[\![M]\!]^{\Delta}_{\text{tm/pf}}\ T\ (\mu \circ \mathsf{id})$, which is equivalent by $\sim$ to $\lambda T.\,\lambda \mu\,.\,[\![M]\!]^{\Delta}_{\text{tm/pf}}\ T\ \mu$. For item 2, note that $\mu_i([\![M]\!]^{\Delta}_{\text{tm/pf}}) = \lambda T.\,\lambda \mu\,.\,[\![M]\!]^{\Delta}_{\text{tm/pf}}\ T\ (\mu \circ \mu_i)$. The term $[\![M]\!]^{\Delta}_{\text{tm/pf}}\ T\ (\mu \circ \mu_i)$ reduces by two administrative reduction steps to $[(\mu \circ \mu_i)/\mu]M'$, with M' being that part of $[\![M]\!]^{\Delta}_{\text{tm/pf}}$ after the initial $\lambda T.\,\lambda \mu\,.$, whereby the desired result follows. For item 3,

$$\begin{array}{lll}
\mu_2(\mu_1([\![M]\!]^{\Delta}_{\text{tm/pf}})) & = & \lambda T.\,\lambda \mu\,.\,(\lambda T.\,\lambda \mu\,.\,[\![M]\!]^{\Delta}_{\text{tm/pf}}\ T\ (\mu \circ \mu_1))\ T\ (\mu \circ \mu_2) \\
& =_{\text{admin}} & \lambda T.\,\lambda \mu\,.\,[\![M]\!]^{\Delta}_{\text{tm/pf}}\ T\ ((\mu \circ \mu_2) \circ \mu_1) \\
& \sim & \lambda T.\,\lambda \mu\,.\,[\![M]\!]^{\Delta}_{\text{tm/pf}}\ T\ (\mu \circ (\mu_2 \circ \mu_1)) \\
& = & (\mu_2 \circ \mu_1)([\![M]\!]^{\Delta}_{\text{tm/pf}}).
\end{array}$$

Item 4 follows from item 2 by the fact that $\mu([\![M]\!]^{\Delta}_{\text{tm-tp}})$ is defined to be the term $\Pi T_2\,.\,\Pi \mu_2\,.\,[\![M]\!]^{\Delta}_{\text{tm}}\ T_2\ (\mu_2 \circ \mu)$, while item 5 follows from item 2 by the fact that $\mu([\![M]\!]^{\Delta}_{\text{pf-tp}})$ is $\Pi T_2\,.\,\Pi \mu_2\,.\,[\![A]\!]^{\Delta}_{\text{pf}}\ T_2\ (\mu_2 \circ \mu)\ (\lambda T_3\,.\,\lambda \mu_3\,.\,x\ T_3\ (\mu_3 \circ \mu_2))$. □

Lemma 6.2.5 (Re-Translation). *For any* Δ_1*,* Δ_2*,* μ_1*,* μ_2*, and* M*, if* $\mu_1(\Delta_1) \approx \mu_2(\Delta_2)$ *then* $\mu_1([\![M]\!]^{\Delta_1}_{\text{tm/pf}}) \cong \mu_2([\![M]\!]^{\Delta_2}_{\text{tm/pf}})$ *if both of these are defined.*

Proof. These are proved simultaneously by induction on the structure of M, where the tm case is considered to be before the pf case, so the pf case can use the result of the tm case. By Lemma 6.2.4 it is sufficient to prove $[(\mu \circ \mu_1)/\mu]M_1 \cong [(\mu \circ \mu_2)/\mu]M_2$, where M_1 and M_2 are the parts after the initial $\lambda T.\,\lambda \mu\,.$ of $[\![M]\!]^{\Delta_1}_{\text{tm/pf}}$ and $[\![M]\!]^{\Delta_2}_{\text{tm/pf}}$, respectively. Further, by inspection of the rules for the translation, M_1 and M_2 are

syntactically equal up to occurrences of Δ_1 in M_1 being replaced by occurrences of Δ_2 in M_2. Thus we simply consider all possible subterms of $[\![M]\!]^{\Delta}_{\text{tm/pf}}$ containing μ or Δ, and show that replacing μ by μ_1 and Δ by Δ_1 in the subterm yields a term that is equivalent under $\cong$ to the result of replacing μ by μ_1 and Δ by Δ_1 in the same subterm.

If the term $\mu([\![N]\!]^{\Delta}_{\text{tm/pf}})$ occurs in $[\![M]\!]^{\Delta}_{\text{tm/pf}}$ for N a strict subterm of M, then the equality $(\mu \circ \mu_1)([\![N]\!]^{\Delta_1}_{\text{tm/pf}}) \cong (\mu \circ \mu_2)([\![N]\!]^{\Delta_2}_{\text{tm/pf}})$ holds by the induction hypothesis. Lemma 6.2.4 then yields the desired result for $\mu([\![N]\!]^{\Delta}_{\text{tm-tp}})$, $\mu([\![N]\!]^{\Delta}_{\text{pf-tp}}(x))$, and any occurrences of $\mu([\![M]\!]^{\Delta}_{\text{tm-tp}})$ in $[\![M]\!]^{\Delta}_{\text{pf}}$. The same argument covers $\mu([\![\vec{N}]\!]^{\Delta}_{\text{args}})$, as this term includes only terms $\mu([\![N]\!]^{\Delta}_{\text{tm/pf}})$ for subterms N of M.

If $\mu \circ \Delta(\vec{\alpha})$ occurs in $[\![M]\!]^{\Delta}_{\text{tm/pf}}$, then, since $[\![M]\!]^{\Delta_1}_{\text{tm/pf}}$ and $[\![M]\!]^{\Delta_2}_{\text{tm/pf}}$ are both defined, $(\mu \circ \mu_1) \circ \Delta(\vec{\alpha}) \cong (\mu \circ \mu_2) \circ \Delta(\vec{\alpha})$ by Lemma 6.2.3. Similarly, if $\mu \circ \Delta(x)$ occurs in $[\![M]\!]^{\Delta}_{\text{tm/pf}}$, then, since $[\![M]\!]^{\Delta_1}_{\text{tm/pf}}$ and $[\![M]\!]^{\Delta_2}_{\text{tm/pf}}$ are both defined, $(\mu \circ \mu_1) \circ \Delta(x) \cong (\mu \circ \mu_2) \circ \Delta(x)$ by Lemma 6.2.3.

If $[\![N]\!]^{\Delta,\alpha}_{\text{tm/pf}}\ T_2\ (\mu' \circ (\mu \otimes \mathsf{id}))$ occurs in $[\![M]\!]^{\Delta}_{\text{tm/pf}}$ for N a strict subterm of M and μ' not containing μ, then, substituting $\mu \circ \mu_i$ for μ and applying Lemma 6.2.4, we get the terms $((\mu_i \otimes \mathsf{id})([\![N]\!]^{\Delta,\alpha}_{\text{tm/pf}}))\ T_2\ (\mu' \circ (\mu \otimes \mathsf{id}))$. These are equivalent under $\cong$ by the induction hypothesis, using the fact that $\mu_1(\Delta_1) \approx \mu_2(\Delta_2)$ implies $(\mu_1 \otimes \mathsf{id})(\Delta_1, \alpha) \approx (\mu_2 \otimes \mathsf{id})(\Delta_2, \alpha)$. Note that this case also applies to $M = \nu\,\alpha\,.\,N$ and $M = \nabla\alpha\,.\,N$, as $\mathsf{combine\text{-}l\text{-}l}^* \circ ((\mu_2 \circ \mu) \otimes \mathsf{id}) \sim \mathsf{combine\text{-}l\text{-}l}^* \circ (\mu_2 \otimes \mathsf{id}) \circ (\mu \otimes \mathsf{id})$. A similar argument holds if $[\![N]\!]^{\Delta,\alpha^*}_{\text{tm/pf}}\ (T \oslash \mathcal{L}^*)\ (\mu' \circ (\mu \otimes \mathsf{id}))$ or $[\![N]\!]^{\Delta,(\mu;\Gamma)}_{\text{tm/pf}}\ T\ \mathsf{eval}$ occurs in $[\![M]\!]^{\Delta}_{\text{tm/pf}}$, as $\mu_1(\Delta_1) \approx \mu_2(\Delta_2)$ implies both $(\mu_1 \otimes \mathsf{id})(\Delta_1, \alpha^*) \approx (\mu_2 \otimes \mathsf{id})(\Delta_2, \alpha^*)$ and $\mathsf{eval}(\Delta_1, (\mu \circ \mu_1; \Gamma)) \approx \mathsf{eval}(\Delta_2, (\mu \circ \mu_2; \Gamma))$.

□

Lemma 6.2.6 (Translation Substitution). *The equivalence*

$$(\mathsf{eval} \otimes \mathsf{id}^n)([\mu([\![\vec{M}]\!]^{\Delta_1}_{\text{args}})/\Gamma'][\![N]\!]^{\Delta_1,(\mu;\Gamma'),\Delta_2}_{\text{tm/pf}}) \cong (\mathsf{eval} \otimes \mathsf{id}^n)([\![[\vec{M}/\Gamma]N]\!]^{\Delta_1,(\mu;\cdot),\Delta_2}_{\text{tm/pf}})$$

holds for any Δ_1, Δ_2, μ, $\vec{x}$, $\vec{M}$, N, Γ, and Γ', where Δ_2 is an eval-ing translation context with non-mapping length n and Γ' has a term and proof variable for each variable of Γ. Thus $\mathsf{eval}([\mu([\![\vec{M}]\!]^{\Delta}_{\text{args}})/\Gamma'][\![N]\!]^{\Delta,(\mu;\Gamma')}) \cong \mu([\![[\vec{M}/\Gamma]N]\!]^{\Delta}_{\text{tm/pf}})$ holds by Re-Translation.

Proof. Proof is by induction on the structure of N. The only interesting case is where $N \equiv x_i$ for some $x_i \in \Gamma$, as the other cases are immediate by the induction hypothesis. The following equivalences hold in this case:

$$\begin{aligned}
&(\mathsf{eval} \otimes \mathsf{id}^n)([\mu([\![\vec{M}]\!]^{\Delta_1}_{\text{args}})/\Gamma'][\![N]\!]^{\Delta_1,(\mu;\Gamma'),\Delta_2}_{\text{tm}}) \\
\equiv\ & (\mathsf{eval} \otimes \mathsf{id}^n)([\mu([\![\vec{M}]\!]^{\Delta_1}_{\text{args}})/\Gamma'](\Delta_1, (\mu;\Gamma'), \Delta_2(x_i))(x_i)) \\
\cong\ & (\mathsf{eval} \otimes \mathsf{id}^n) \circ (\Delta_1, (\mu;\Gamma'), \Delta_2(x_i)) \circ \mu)([\![M_i]\!]^{\Delta_1}_{\text{tm}}) \\
\cong\ & (\mathsf{eval} \otimes \mathsf{id}^n) \circ (\mathsf{tcadd}((\mu;\cdot), \Delta_2)))([\![M_i]\!]^{\Delta_1}_{\text{tm}}) \\
\cong\ & (\mathsf{eval} \otimes \mathsf{id}^n)([\![M_i]\!]^{\Delta_1,(\mu;\Gamma'),\Delta_2}_{\text{tm}}) \\
\equiv\ & (\mathsf{eval} \otimes \mathsf{id}^n)([\![[\vec{M}/\Gamma]x_i]\!]^{\Delta_1,(\mu;\cdot),\Delta_2}_{\text{tm}})
\end{aligned}$$

where the fourth line holds by Lemma 6.2.2 and the fifth holds by Re-Translation and by the fact that no x in Γ' can be free in M'_i. The case for $[\![N]\!]^{\Delta}_{\text{pf}}$ is similar. □

Lemma 6.2.7 (Modal Translation Substitution). *The equivalence*

$$[[\![F]\!]_{\text{pmfun}}/u][\![M]\!]^{\Delta}_{\text{tm/pf}} \equiv [\![[F/u]M]\!]^{\Delta}_{\text{tm/pf}}$$

holds for any translation contesxt Δ, pattern-matching function F, term M, and modal variable u.

Proof. Proof is by straightforward induction on M. □

Lemma 6.2.8. *For any Δ, T, μ, $\vec{\alpha}$ with length n,* CNIC *term M, and* CIC $+$ $\mathbb{T}$ *term M', the following hold:*

1. $\textbf{remove-nus}_{\vec{\alpha}}\ T\ \mu([\![\nu\,\vec{\alpha}\,.\,M]\!]^{\Delta}_{\text{tm/pf}}) \cong (\mu \otimes \mathsf{id}^n)([\![M]\!]^{\Delta,\vec{\alpha}}_{\text{tm/pf}})$
2. $\textbf{add-nus}_{\vec{\alpha}}\ T\ (\mu \otimes \mathsf{id}^n)([\![M]\!]^{\Delta,\vec{\alpha}}_{\text{tm/pf}}) \cong \mu([\![\nu\,\vec{\alpha}\,.\,M]\!]^{\Delta}_{\text{tm/pf}})$
3. $\textbf{add-nus}_{\vec{\alpha}}\ T\ (\textbf{remove-nus}_{\vec{\alpha}}\ T\ M') \cong M'$

Proof. By straightforward induction on the definitions of $\textbf{remove-nus}_{\vec{\alpha}}\ T\ M$ and $\textbf{add-nus}_{\vec{\alpha}}\ T\ M$. □

Lemma 6.2.9 (Translation Reduction). *If $M \rightsquigarrow N$ then $[\![M]\!]^{\Delta}_{\text{tm}} \cong [\![N]\!]^{\Delta}_{\text{tm}}$ and $[\![M]\!]^{\Delta} \cong [\![N]\!]^{\Delta}_{\text{pf}}$.*

Proof. Proof is by induction on the structure of M, where cases other than redexes follow directly by the induction hypothesis. So we consider the case where M is a redex.

Case: $(\nu\,\alpha_1\,.\,M_1)\ \langle\alpha_2\rangle \rightsquigarrow [\alpha_2/\alpha_1]M_1$

For the tm case, the left-hand side is

$$\begin{array}{l}\lambda\,T\,.\,\lambda\,\mu\,.\,(\lambda\,T_2\,.\,\lambda\,\mu_2\,.\,\lambda\,T_3\,.\,\lambda\,\mu_3\,.\,[\![M_1]\!]_{\mathrm{tm}}^{\Delta,\alpha_2^*,\alpha_1}\ T_3\ (\mathsf{combine\text{-}l\text{-}l}^*\circ((\mu_3\circ\mu_2)\otimes\mathsf{id})))\\ \qquad (T\mathbin{⍉}\mathcal{L}^*)\ (\mu\otimes\mathsf{id})\ T\ \mathsf{id}\end{array}$$

By two reductions, this becomes

$$\lambda\,T\,.\,\lambda\,\mu\,.\,[\![M_1]\!]_{\mathrm{tm}}^{\Delta,\alpha_2^*,\alpha_1}\ (\mathsf{combine\text{-}l\text{-}l}^*\circ(((\mu\otimes\mathsf{id})\circ\mathsf{id})\otimes\mathsf{id}))$$

By some steps of $\sim$ we then have

$$\lambda\,T\,.\,\lambda\,\mu\,.\,[\![M_1]\!]_{\mathrm{tm}}^{\Delta,\alpha_2^*,\alpha_1}\ (\mu\circ\mathsf{combine\text{-}l\text{-}l}^*)$$

and, by Re-Translation, we thus get

$$\lambda\,T\,.\,\lambda\,\mu\,.\,[\![M_1]\!]_{\mathrm{tm}}^{\Delta}$$

which is equivalent to the right-hand side above. The pf case is similar.

Case: $(\lambda\,x\!:\!A\,.\,M_1)\ M_2 \rightsquigarrow [M_2/x]M_1$

For the tm case, the left-hand side is

$$\begin{array}{l}\lambda\,T\,.\,\lambda\,\mu\,.\,(\lambda\,T_2\,.\,\lambda\,\mu_2\,.\,\lambda\,x\!:\!\mu([\![A]\!]_{\mathrm{tm\text{-}tp}}^{\Delta})\,.\,\lambda\,x_{\mathrm{pf}}\!:\!\mu([\![A]\!]_{\mathrm{tm/pf}}^{\Delta})\,.\,[\![M_1]\!]_{\mathrm{tm}}^{\Delta}\ T_2\ \mathsf{eval})\\ \qquad T\ \mu\ \mu([\![M_2]\!]_{\mathrm{tm}}^{\Delta})\ \mu([\![M_2]\!]_{\mathrm{pf}}^{\Delta})\end{array}$$

which reduces to

$$[\mu([\![M_2]\!]_{\mathrm{tm}}^{\Delta})/x, \mu([\![M_2]\!]_{\mathrm{pf}}^{\Delta})/x_{\mathrm{pf}}]([\![M_1]\!]_{\mathrm{tm}}^{\Delta}\ T\ \mathsf{eval})$$

and the result follows by Translation Substitution. The pf case is similar.

Case:

$$(\mathbf{fun}\ u\ (\vec{\alpha}, \Gamma_a^{\mathrm{x}})\ (\nu\,\vec{\beta}\,.\,\vec{P} \to \vec{M}))\ \langle\vec{\alpha}\rangle\ \vec{N}\ (\nu\,\vec{\beta}\,.\,c_i\ \vec{Q})$$
$$\rightsquigarrow [(\nu\,\vec{\beta}\,.\,\vec{Q})/\Gamma_i^{\mathrm{x}}, \mathbf{fun}\ u\ (\vec{\alpha}, \Gamma_a^{\mathrm{x}})\ (\nu\,\vec{\beta}\,.\,\vec{P} \to \vec{M})/u]M_i$$

We have that $(\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![(\nu\,\vec{\beta}\,.\,c_i\ \vec{Q})]\!]^{\Delta}_{\mathrm{pf}}))$ applied to $(T \oslash \vec{\beta})$ and id equals $\mathsf{valid}\text{-}c_i\ T\ (\mu \circ \Delta(\emptyset))\ \mu([\![Q]\!]^{\Delta}_{\mathrm{args}})$. Thus this matches the ith pattern in the translation of the pattern-matching function, and a pattern-matching reduction step can take place, along with some β-reductions to substitute the arguments after the scrutinee into the result. In particular, the equalities in the cast become ground, and so can reduce. Further, the projection can then occur on the $\mathsf{mk\text{-}pair}$, yielding

$$\lambda\,T\,.\,\lambda\,\mu\,.\,\mathsf{eval}([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T\ (\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![\vec{Q}]\!]^{\Delta}_{\mathrm{args}}))/\Gamma_1^{\mathrm{x}}][\![M_1]\!]^{\vec{\alpha},(\mu;\Gamma_i^{\mathrm{x}})}_{\mathrm{tm}})\ T\ id$$

in the tm case. By Translation Substitution, Modal Translation Substitution, and Lemma 6.2.4, this is then equal to

$$[\![[\vec{Q}/\Gamma_{c_i}, \mathbf{fun}\ u\ (\vec{\alpha}, \Gamma_a^{\mathrm{x}})\ (\nu\,\vec{\beta}\,.\,\vec{P} \to \vec{M})/u]M_i]\!]^{\Delta}_{\mathrm{tm}}.$$

The pf case is similar.

Case: $(\mathbf{nfun}\ (\vec{\alpha})\ (\vec{P} \to \vec{M}))\ \langle\vec{\alpha}\rangle\ \alpha_i \rightsquigarrow M_{\alpha_i}$

For the tm case we have that $\mu([\![\alpha_i]\!]^{\Delta}_{\mathrm{pf}})\ T\ \mathsf{id}$ reduces to $\mathsf{mk\text{-}valid\text{-}name}\ T\ (\mu \circ \Delta(\alpha_i))$. The term

$$\mathsf{find\text{-}matching}_{\vec{\alpha}}\ T\ (\mu \circ \Delta(\vec{\alpha}))\ \mu([\![N]\!]^{\Delta}_{\mathrm{tm}})\ \mu([\![N]\!]^{\Delta}_{\mathrm{pf}})$$

thus reduces to a term containing

$$\mathsf{find\text{-}name\text{-}case}_{\vec{\alpha}}\ T\ (\mu \circ \Delta(\vec{\alpha_i}))\ (\mu \circ \Delta(\alpha_i))$$

This is equal under $\sim$ to

$$\mathsf{map\text{-}name\text{-}case}_{\vec{\alpha}}\ |\Delta|\ (\Delta(\vec{\alpha}))\ (\Delta(\alpha_i))\ T\ \mu\ (\mathsf{find\text{-}name\text{-}case}_{\vec{\alpha}}\ |\Delta|\ (\Delta(\vec{\alpha}))\ (\Delta(\alpha_i)))$$

which will reduce to a $\mathsf{name\text{-}case\text{-}i}_{\vec{\alpha},i}$, causing the whole $\mathsf{find\text{-}matching}_{\vec{\alpha}}$ term to reduce to $\mathsf{name\text{-}matching}_{\vec{\alpha},i}\ T\ \mu$. The whole term then redues to

$$\lambda\,T\,.\,\lambda\,\mu\,.\,[\![M_{\alpha_i}]\!]^{\vec{\alpha},\mu\circ\Delta(\vec{\alpha})}_{\mathrm{tm}}\ T_x\ \mathsf{eval}$$

which by Re-Translation is equivalent under $\cong$ to $[\![M_{\alpha_i}]\!]^{\Delta}_{\text{tm}}$.

Case: $(\textbf{nfun}\ (\vec{\alpha})\ (\vec{P} \to \vec{M}))\ \langle\vec{\alpha}\rangle\ \alpha \rightsquigarrow M_\alpha$

Similar to the previous case.

□

Corollary 6.2.1 (Translation Equality). *For any M, N, and Δ, $\vdash M = N$ implies $[\![M]\!]^{\Delta}_{\text{tm/pf}} \cong [\![N]\!]^{\Delta}_{\text{tm/pf}}$.*

Corollary 6.2.2 (Translation Subtyping). *For any A, B, and Δ, $\vdash A \lesssim B$ implies $\vdash [\![A]\!]^{\Delta}_{\text{tm/pf}} \lesssim [\![B]\!]^{\Delta}_{\text{tm/pf}}$.*

Proof. This is immediate by Translation Equality and by the fact that, for any $i \leq j$, $\vdash [\![\mathsf{Type}_i]\!]^{\Delta}_{\text{tm/pf}} \lesssim [\![\mathsf{Type}_j]\!]^{\Delta}_{\text{tm/pf}}$. □

Definition 6.2.3. *The translation context Δ and context Γ' of* CIC $+\ \mathbb{T}$ *are said to* agree with *modal context Σ and normal context Γ of* CNIC *if and only if:*

- Γ' *contains* $[\![\Sigma]\!]_{\text{mctxt}}$ *as a sub-context;*
- $\Delta(\alpha)$ *and* $\Delta(x)$ *are defined for all names α and variables x in Γ;*
- Γ' *contains* $x : \mathsf{eval}([\![A]\!]^{\Delta_1,(\mu;\vec{x})}_{\text{tm-tp}})$ *and* $x_{\text{pf}} : \mathsf{eval}([\![A]\!]^{\Delta_1,(\mu;\vec{x})}_{\text{pf-tp}}(x))$ *for each $x : A \in \Gamma$, in the same order as in Γ; and*
- Δ *has the form* $\vec{\alpha}, (\mu; \vec{x}); \Delta'$ *for eval-ing translation context Δ'.*

Lemma 6.2.10. *For any* CNIC *type A, tree types T and T_2, tree mapping $\mu : T \rightrightarrows T_2$,* CIC $+\ \mathbb{T}$ *context Γ, and* CIC $+\ \mathbb{T}$ *terms M and N, the following hold:*

1. *If* $\Gamma \vdash_{\text{CIC}\ +\mathbb{T}} M : (\mu \otimes \vec{\alpha})([\![A]\!]^{\Delta,\vec{\alpha}}_{\text{tm-tp}})$ *then*
   $\Gamma \vdash_{\text{CIC}\ +\mathbb{T}} (\textbf{add-nus}_{\vec{\alpha}}\ T\ M) : \mu([\![\nabla\vec{\alpha}\,.\,A]\!]^{\Delta}_{\text{tm-tp}})$*;*
2. *If* $\Gamma \vdash_{\text{CIC}\ +\mathbb{T}} M : (\mu \otimes \vec{\alpha})([\![A]\!]^{\Delta,\vec{\alpha}}_{\text{pf-tp}}(N))$ *then*
   $\Gamma \vdash_{\text{CIC}\ +\mathbb{T}} (\textbf{add-nus}_{\vec{\alpha}}\ T\ M) : \mu([\![\nabla\vec{\alpha}\,.\,A]\!]^{\Delta}_{\text{pf-tp}}(\textbf{add-nus}_{\vec{\alpha}\ T\ N}))$*;*
3. *If* $\Gamma \vdash_{\text{CIC}\ +\mathbb{T}} M : \mu([\![\nabla\vec{\alpha}\,.\,A]\!]^{\Delta}_{\text{tm-tp}})$ *then*
   $\Gamma \vdash_{\text{CIC}\ +\mathbb{T}} (\textbf{remove-nus}_{\vec{\alpha}}\ T\ M) : (\mu \otimes \vec{\alpha})([\![A]\!]^{\Delta,\vec{\alpha}}_{\text{tm-tp}})$*; and*

4. *If* $\Gamma \vdash_{\mathrm{CIC}\ +\mathbb{T}} M : \mu([\![\nabla\vec{\alpha}\,.\,A]\!]^{\Delta}_{\text{pf-tp}}(N))$ *then*
$\Gamma \vdash_{\mathrm{CIC}\ +\mathbb{T}} (\mathbf{remove\text{-}nus}_{\vec{\alpha}}\ T\ M) : (\mu \otimes \vec{\alpha})([\![A]\!]^{\Delta,\vec{\alpha}}_{\text{pf-tp}}(\mathbf{remove\text{-}nus}_{\vec{\alpha}}\ T\ N))$.

Proof. By induction on $\vec{\alpha}$. □

Lemma 6.2.11. *The equivalence*

$$\begin{aligned}&\mathsf{eval}([\![M]\!]^{\Delta,(\mu;\vec{x}),\vec{\beta},(\mathsf{eval}\otimes\vec{\beta};x,x_{\mathrm{pf}})}_{\mathrm{tm/pf}})\\ &\quad\cong (\mathsf{eval}\otimes\vec{\beta})([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T\ x/y][\![[y\ \langle\vec{\beta}\rangle/x]M]\!]^{\Delta,(\mu;\vec{x},y,y_{\mathrm{pf}}),\vec{\beta}}_{\mathrm{tm/pf}})\end{aligned}$$

holds for any M.

Proof. By Lemma 6.2.8 we have $\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ (\mathbf{add\text{-}nus}_{\vec{\beta}}\ T\ x) \cong x$, yielding

$$\begin{aligned}&\mathsf{eval}([\![M]\!]^{\Delta,(\mu;\vec{x}),\vec{\beta},(\mathsf{eval}\otimes\vec{\beta};x,x_{\mathrm{pf}})}_{\mathrm{tm/pf}})\\ \cong\ &\mathsf{eval}([\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ (\mathbf{add\text{-}nus}_{\vec{\beta}}\ T\ (x,x_{\mathrm{pf}}))/(x,x_{\mathrm{pf}})][\![M]\!]^{\Delta,(\mu;\vec{x}),\vec{\beta},(\mathsf{eval}\otimes\vec{\beta};x,x_{\mathrm{pf}})}_{\mathrm{tm/pf}})\\ \equiv\ &\mathsf{eval}([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T\ (x,x_{\mathrm{pf}})/(y,y_{\mathrm{pf}})]\qquad\qquad\qquad\qquad\qquad\qquad)\\ &\qquad\qquad[\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ (y,y_{\mathrm{pf}})/(x,x_{\mathrm{pf}})][\![M]\!]^{\Delta,(\mu;\vec{x}),\vec{\beta},(\mathsf{eval}\otimes\vec{\beta};x,x_{\mathrm{pf}})}_{\mathrm{tm/pf}}\\ \cong\ &\mathsf{eval}([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T\ x/y]\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad)\\ &\qquad\qquad[(\mathsf{eval}\otimes\vec{\beta})([\![y\ \langle\vec{\beta}\rangle]\!]^{\Delta,(\mu;\vec{x},y,y_{\mathrm{pf}}),\vec{\beta}}_{\mathrm{tm/pf}})/x][\![M]\!]^{\Delta,(\mu;\vec{x},y,y_{\mathrm{pf}}),\vec{\beta},(\mathsf{eval}\otimes\vec{\beta};x,x_{\mathrm{pf}})}_{\mathrm{tm/pf}}\\ \cong\ &(\mathsf{eval}\otimes\vec{\beta})([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T\ x/y][\![[y\ \langle\vec{\beta}\rangle/x]M]\!]^{\Delta,(\mu;\vec{x},y,y_{\mathrm{pf}}),\vec{\beta}}_{\mathrm{tm/pf}})\end{aligned}$$

□

Lemma 6.2.12 (Translation Typing). *If* $\Sigma;\Gamma \vdash_{\mathrm{CNIC}} M : A$ *then*

- $\Gamma' \vdash_{\mathrm{CIC}\ +\mathbb{T}} [\![M]\!]^{\Delta}_{\mathrm{tm}}\ T\ \mu' : [\![A]\!]^{\Delta}_{\mathrm{tm}}\ T\ \mu'$ *and*
- $\Gamma' \vdash_{\mathrm{CIC}\ +\mathbb{T}} [\![M]\!]^{\Delta}_{\mathrm{pf}}\ T\ \mu' : [\![A]\!]^{\Delta}_{\mathrm{pf}}\ T\ \mu'\ (\mu'([\![M]\!]^{\Delta}_{\mathrm{tm}}))$

hold for any Δ, Γ', *and* μ' *such that:*

- *The suffix of* Δ *after its first mapping is an eval-ing translation context, i.e. ,* $\Delta = \Delta_1, (\mu;\vec{x}), \Delta_2$ *for* Δ_1 *containing no mappings and* Δ_2 *being an eval-ing translation context;*

- Δ *has non-mapping length* n*;*
- Δ *and* Γ' *agree with* Γ*;*
- $[\![M]\!]^{\Delta}_{\mathrm{tm}}$ *and* $[\![M]\!]^{\Delta}_{\mathrm{pf}}$ *are defined; and*
- $\mu' \cong (\mu \circ (\mathsf{eval} \otimes \mathsf{id}^n))$ *for some* μ*.*

Proof. The proof is by induction on the typing proof of M. Many of the cases are repetitive, so we show only some of the more interesting and illustrative ones.

Case:

$$\frac{\Gamma \vdash M : B \quad \Gamma \vdash A : \mathsf{Type}_i \quad \vdash B \lesssim A}{\Gamma \vdash M : A}\ \textsf{t-subt}$$

Immediate by the induction hypothesis and Translation Subtyping.

Case:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsf{t-var}$$

$$[\![x]\!]^{\Delta}_{\mathrm{tm}} = \lambda\, T\,.\,\lambda\, \mu\,.\, x\ T\ (\mu \circ \Delta(x))$$
$$[\![x]\!]^{\Delta}_{\mathrm{pf}} = \lambda\, T\,.\,\lambda\, \mu\,.\, x_{\mathrm{pf}}\ T\ (\mu \circ \Delta(x))$$

For the tm case we have that $[\![x]\!]^{\Delta}_{\mathrm{tm}}$ being well-defined implies $\Delta = \Delta_1, (\mu; \vec{x}), \Delta_2$ for some $\Delta_1, (\mu; \vec{x}), \Delta_2$ with x in $\vec{x}$. Further, since Δ and Γ' agree with Γ it follows that $x : \mathsf{eval}([\![A]\!]^{\Delta_1,(\mu;\vec{x})}_{\text{tm-tp}}) \in \Gamma'$. Hence $[\![x]\!]^{\Delta}_{\mathrm{tm}}\ T\ \mu' \equiv x\ T\ (\mu' \circ \Delta(x))$ has the following type:

$$\begin{aligned}
&[\![A]\!]^{\Delta_1,(\mu;\vec{x})}_{\text{tm-tp}}\ T\ (\mu' \circ \Delta(x) \circ \mathsf{eval}) \\
&\cong [\![A]\!]^{\Delta_1,(\mu;\vec{x})}_{\text{tm-tp}}\ T\ (\mu \circ (\mathsf{eval} \otimes \mathsf{id}^n) \circ \Delta(x) \circ \mathsf{eval}) \\
&\cong [\![A]\!]^{\Delta_1,(\mu;\vec{x})}_{\text{tm-tp}}\ T\ (\mu \circ (\mathsf{eval} \otimes \mathsf{id}^n) \circ \mathsf{tcadd}(\Delta_2)) \\
&\cong [\![A]\!]^{\Delta_1,(\mu;\vec{x}),\Delta_2}_{\text{tm-tp}}\ T\ (\mu \circ (\mathsf{eval} \otimes \mathsf{id}^n)) \\
&\cong [\![A]\!]^{\Delta_1,(\mu;\vec{x}),\Delta_2}_{\text{tm-tp}}\ T\ \mu'
\end{aligned}$$

where the first line is just the application of a renaming, the second line is by the assumption that $\mu' \cong \mu \circ (\mathsf{eval} \otimes \mathsf{id}^n)$, the third line is by Lemma 6.2.2, the fourth line is by Re-Translation, and the fifth is again by the assumption that $\mu' \cong \mu \circ (\mathsf{eval} \otimes \mathsf{id}^n)$. The pf case is similar.

Case:

$$\frac{\Gamma, \alpha \vdash M : A}{\Gamma \vdash \nu\, \alpha\, .\, M : \nabla \alpha\, .\, A}\ \textsf{t-nu}$$

$$\begin{aligned}
[\![\nu\, \alpha\, .\, M]\!]^{\Delta}_{\mathrm{tm}} &= \lambda\, T\, .\, \lambda\, \mu\, .\, \lambda\, T_2\, .\, \lambda\, \mu_2 : (T \rightrightarrows (T_2 \oslash \mathcal{L}^*))\, . \\
&\qquad [\![M]\!]^{\Delta,\alpha}_{\mathrm{tm}}\ T_2\ (\textsf{combine-l-l}^* \circ ((\mu_2 \circ \mu) \otimes \textsf{id})) \\
[\![\nu\, \alpha\, .\, M]\!]^{\Delta}_{\mathrm{pf}} &= \lambda\, T\, .\, \lambda\, \mu\, .\, \lambda\, T_2\, .\, \lambda\, \mu_2 : (T \rightrightarrows (T_2 \oslash \mathcal{L}^*))\, . \\
&\qquad [\![M]\!]^{\Delta,\alpha}_{\mathrm{pf}}\ T_2\ (\textsf{combine-l-l}^* \circ ((\mu_2 \circ \mu) \otimes \textsf{id}))
\end{aligned}$$

Since Δ and Γ' agree with Γ it follows that Δ, α and Γ' agree with Γ, α. Further, by assumption, $\mu' \cong \mu \circ (\textsf{eval} \otimes \textsf{id}^n)$ where n is the non-mapping length of Δ, and thus $(\textsf{combine-l-l}^* \circ ((\mu_2 \circ \mu') \otimes \textsf{id})) \cong \textsf{combine-l-l}^* \circ (\mu_2 \otimes \textsf{id}) \circ (\mu \otimes \textsf{id}) \circ (\textsf{eval} \otimes \textsf{id}^{n+1})$. Since $n+1$ is the non-mapping length of Δ, α, the inductive hypothesis yields

$$\Gamma' \vdash [\![M]\!]^{\Delta,\alpha}_{\mathrm{tm}}\ T_2\ (\textsf{combine-l-l}^* \circ ((\mu_2 \circ \mu') \otimes \textsf{id})) : [\![A]\!]^{\Delta,\alpha}_{\mathrm{tm}}\ T_2\ (\textsf{combine-l-l}^* \circ ((\mu_2 \circ \mu') \otimes \textsf{id}))$$

whereby $\Gamma' \vdash [\![\nu\, \alpha\, .\, M]\!]^{\Delta}_{\mathrm{tm}}\ T\ \mu' : \Pi T\, .\, \Pi \mu\, .\, [\![A]\!]^{\Delta}\ T\ \mu'$ follows. The pf case is similar.

Case:

$$\frac{\mathbf{remove}_{\alpha}(\Gamma) \vdash M : \nabla \alpha\, .\, A}{\Gamma \vdash M\ \langle \alpha \rangle : A}\ \textsf{t-namerepl}$$

$$\begin{aligned}
[\![M\ \langle \alpha \rangle]\!]^{\Delta}_{\mathrm{tm}} &= \lambda\, T\, .\, \lambda\, \mu\, .\, [\![M]\!]^{\Delta,\alpha^*}_{\mathrm{tm}}\ (T \oslash \mathcal{L}^*)\ (\mu \otimes \textsf{id})\ T\ \textsf{id} \\
[\![M\ \langle \alpha \rangle]\!]^{\Delta}_{\mathrm{pf}} &= \lambda\, T\, .\, \lambda\, \mu\, .\, [\![M]\!]^{\Delta,\alpha^*}_{\mathrm{pf}}\ (T \oslash \mathcal{L}^*)\ (\mu \otimes \textsf{id})\ T\ \textsf{id}
\end{aligned}$$

Since Δ and Γ' agree with Γ, Δ, α^* and Γ' agree with $\mathbf{remove}_{\alpha}(\Gamma)$. Further, by assumption, $\mu' \cong \mu \circ (\textsf{eval} \otimes \textsf{id}^n)$ where n is the non-mapping length of Δ, and thus $\mu' \otimes \textsf{id})) \cong (\mu \otimes \textsf{id}) \circ (\textsf{eval} \otimes \textsf{id}^{n+1})$. Since $n+1$ is the non-mapping length of Δ, α^*, the inductive hypothesis yields

$$\Gamma' \vdash [\![M]\!]^{\Delta,\alpha^*}_{\mathrm{tm}}\ (T \oslash \mathcal{L}^*)\ (\mu' \otimes \textsf{id}) : [\![(\nabla \alpha\, .\, A)]\!]^{\Delta,\alpha^*}_{\mathrm{tm}}\ (T \oslash \mathcal{L}^*)\ (\mu' \otimes \textsf{id})$$

where the given type is equivalent under $\cong$ to

$$\begin{aligned}
&\Pi T_2\, .\, \Pi \mu_2 : (T \oslash \mathcal{L}^* \rightrightarrows T_2 \oslash \mathcal{L}^*)\, . \\
&\quad [\![A]\!]^{\Delta,\alpha^*,\alpha}_{\mathrm{tm}}\ T_2\ (\textsf{combine-l-l}^* \circ ((\mu_2 \otimes \textsf{id}) \circ ((\mu' \otimes \textsf{id}) \otimes \textsf{id})))
\end{aligned}$$

by expanding the definition of $[\![\nabla\alpha\,.\,A]\!]^{\Delta}_{\text{tm}}$. Including the additional tree type and tree mapping arguments given in $[\![M\ \langle\alpha\rangle]\!]^{\Delta}_{\text{tm}}$ yields

$$\begin{aligned}
&\Gamma \vdash [\![M]\!]^{\Delta,\alpha^*}_{\text{tm}}\ (T \between \mathcal{L}^*)\ (\mu \otimes \mathsf{id})\ T\ \mathsf{id} \\
&\quad : [\![A]\!]^{\Delta,\alpha^*,\alpha}_{\text{tm}}\ T\ (\mathsf{combine\text{-}l\text{-}l}^* \circ ((\mathsf{id} \otimes \mathsf{id}) \circ ((\mu' \otimes \mathsf{id}) \otimes \mathsf{id}))) \\
&\quad \sim [\![A]\!]^{\Delta,\alpha^*,\alpha}_{\text{tm}}\ T\ (\mu' \circ \mathsf{combine\text{-}l\text{-}l}^*) \\
&\quad \cong [\![A]\!]^{\Delta}_{\text{tm}}\ T\ \mu'
\end{aligned}$$

where the first line follows by the typing rule for applications, the second from the rules for $\sim$, and the third by Re-Translation. The pf case is similar.

Case:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda\, x{:}A\,.\,M : \Pi x{:}A\,.\,B}\ \mathsf{t\text{-}lambda}$$

$$\begin{aligned}
[\![\lambda\, x{:}A\,.\,M]\!]^{\Delta}_{\text{tm}} &= \lambda\, T\,.\,\lambda\,\mu\,.\,\lambda\, x{:}\mu([\![A]\!]^{\Delta}_{\text{tm-tp}})\,.\,\lambda\, x_{\text{pf}}{:}\mu([\![A]\!]^{\Delta}_{\text{pf-tp}}(x))\,.\,[\![M]\!]^{\Delta,(\mu;x,x_{\text{pf}})}_{\text{tm}}\ T\ \mathsf{eval} \\
[\![\lambda\, x{:}A\,.\,M]\!]^{\Delta}_{\text{pf}} &= \lambda\, T\,.\,\lambda\,\mu\,.\,\lambda\, x{:}\mu([\![A]\!]^{\Delta}_{\text{tm-tp}})\,.\,\lambda\, x_{\text{pf}}{:}\mu([\![A]\!]^{\Delta}_{\text{pf-tp}}(x))\,.\,[\![M]\!]^{\Delta,(\mu;x,x_{\text{pf}})}_{\text{pf}}\ T\ \mathsf{eval}
\end{aligned}$$

Since Δ and Γ' agree with Γ it follows that $\Delta, (\mu'; x, x_{\text{pf}})$ and

$$\Gamma', x : \mathsf{eval}([\![A]\!]^{\Delta,(\mu';\cdot)}_{\text{tm-tp}}), x : \mathsf{eval}([\![A]\!]^{\Delta,(\mu';\cdot)}_{\text{pf-tp}}(x))$$

agree with $\Gamma, x : A$. Further, by assumption, $\mu' \cong \mu \circ (\mathsf{eval} \otimes \mathsf{id}^n)$ where n is the non-mapping length of Δ, and thus $\Delta, (\mu'; x, x_{\text{pf}})$ is an eval-ing translation context with non-mapping length 0. It follows by the inductive hypothesis that

$$\Gamma' \vdash [\![M]\!]^{\Delta,(\mu';x,x_{\text{pf}})}_{\text{tm}}\ T\ \mathsf{eval} : [\![B]\!]^{\Delta,(\mu';x,x_{\text{pf}})}_{\text{tm}}\ T\ \mu'$$

which then yields $\Gamma \vdash [\![\lambda\, x : A\,.\,M]\!]^{\Delta}_{\text{tm}} : \Pi T\,.\,\Pi\mu\,.\,[\![\Pi x : A\,.\,B]\!]^{\Delta}_{\text{tm}}\ T\ \mu$. The pf case is similar.

Case:

$$\frac{\Gamma \vdash M : \Pi x{:}A\,.\,B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : [N/x]B}\ \mathsf{t\text{-}app}$$

$$\begin{aligned}
[\![M\ N]\!]^{\Delta}_{\text{tm}} &= \lambda\, T\,.\,\lambda\,\mu\,.\,[\![M]\!]^{\Delta}_{\text{tm}}\ T\ \mu\ \mu([\![N]\!]^{\Delta}_{\text{tm}})\ \mu([\![N]\!]^{\Delta}_{\text{pf}}) \\
[\![M\ N]\!]^{\Delta}_{\text{pf}} &= \lambda\, T\,.\,\lambda\,\mu\,.\,[\![M]\!]^{\Delta}_{\text{pf}}\ T\ \mu\ \mu([\![N]\!]^{\Delta}_{\text{tm}})\ \mu([\![N]\!]^{\Delta}_{\text{pf}})
\end{aligned}$$

The induction hypothesis yields

$$\begin{array}{l}
\Gamma' \vdash [\![M]\!]^{\Delta}_{\mathrm{tm}}\ T\ \mu' :\ \ \Pi x{:}\mu'([\![A]\!]^{\Delta}_{\text{tm-tp}})\,.\,\Pi x_{\mathrm{pf}}{:}\mu'([\![A]\!]^{\Delta}_{\text{pf-tp}}(x))\,.\,[\![B]\!]^{\Delta,(\mu';x,x_{\mathrm{pf}})}_{\mathrm{tm}}\ T\ \mathsf{eval}\\
\Gamma' \vdash [\![M]\!]^{\Delta}_{\mathrm{pf}}\ T\ \mu' :\ \ \Pi x{:}\mu'([\![A]\!]^{\Delta}_{\text{tm-tp}})\,.\,\Pi x_{\mathrm{pf}}{:}\mu'([\![A]\!]^{\Delta}_{\text{pf-tp}}(x))\,.\\
\qquad\qquad [\![B]\!]^{\Delta,(\mu';x,x_{\mathrm{pf}})}_{\mathrm{pf}}\ T\ \mathsf{eval}\ \lambda T_2\,.\,\lambda\,\mu_2\,.\,[\![M]\!]^{\Delta}_{\mathrm{tm}}\ T_2\ \mu_2\ \mu_2(x)\ \mu_2(x_{\mathrm{pf}})\\
\Gamma' \vdash [\![N]\!]^{\Delta}_{\mathrm{tm}}\ T_N\ (\mu_N \circ \mu') : [\![N]\!]^{\Delta}_{\mathrm{pf}}\ T_N\ (\mu_N \circ \mu')\\
\Gamma' \vdash [\![N]\!]^{\Delta}_{\mathrm{pf}}\ T_N\ (\mu_N \circ \mu') : [\![N]\!]^{\Delta}_{\mathrm{pf}}\ T_N\ (\mu_N \circ \mu')\ (\mu_N \circ \mu')([\![N]\!]^{\Delta}_{\mathrm{tm}})
\end{array}$$

for any tree type T_N and tree mapping $\mu_N : T \rightrightarrows T_N$. Thus it follows that

$$\begin{array}{c}
\Gamma' \vdash [\![M\ N]\!]^{\Delta}_{\mathrm{tm}} : [\mu'([\![N]\!]^{\Delta}_{\mathrm{tm}})/x, \mu'([\![N]\!]^{\Delta}_{\mathrm{pf}})/x_{\mathrm{pf}}][\![B]\!]^{\Delta,(\mu';x,x_{\mathrm{pf}})}_{\mathrm{tm}}\ T\ \mathsf{eval}\\
\Gamma' \vdash [\![M\ N]\!]^{\Delta}_{\mathrm{pf}} : [\mu'([\![N]\!]^{\Delta}_{\mathrm{tm}})/x, \mu'([\![N]\!]^{\Delta}_{\mathrm{pf}})/x_{\mathrm{pf}}][\![B]\!]^{\Delta,(\mu';x,x_{\mathrm{pf}})}_{\mathrm{pf}}\ T\ \mathsf{eval}\\
(\lambda T_2\,.\,\lambda\,\mu_2\,.\,[\![M]\!]^{\Delta}_{\mathrm{tm}}\ T_2\ \mu_2\ \mu_2([\![N]\!]^{\Delta}_{\mathrm{tm}})\ \mu_2([\![N]\!]^{\Delta}_{\mathrm{pf}}))
\end{array}$$

where the required typings then hold by Translation Substitution, Re-Translation, and the fact that $(\lambda T_2\,.\,\lambda\,\mu_2\,.\,[\![M]\!]^{\Delta}_{\mathrm{tm}}\ T_2\ \mu_2\ \mu_2([\![N]\!]^{\Delta}_{\mathrm{tm}})\ \mu_2([\![N]\!]^{\Delta}_{\mathrm{pf}})) \equiv [\![M\ N]\!]^{\Delta}_{\mathrm{tm}}$.

Case:

$$\frac{\begin{array}{c}
\Sigma;\cdot \vdash \nabla\vec{\alpha}\,.\,\Pi\Gamma^{\mathrm{x}}{\uparrow}^{\vec{\beta}}\ .\,\Pi x{:}(\nabla\vec{\beta}\,.\,a\ (\Gamma^{\mathrm{x}}\ \langle\vec{\beta}\rangle))\,.\,B : \mathsf{Type}_i \qquad \forall i(\Sigma;\cdot \vdash c_i : \Pi\Gamma^{\mathrm{x}}_i\,.\,a\ \vec{M}_i)\\
\forall i(\Sigma, u : (\nabla\vec{\alpha}\,.\,\Pi\Gamma^{\mathrm{x}}{\uparrow}^{\vec{\beta}}\ .\,\Pi x{:}(\nabla\vec{\beta}\,.\,a\ (\Gamma^{\mathrm{x}}\ \langle\vec{\beta}\rangle))\,.\,B)\ ;\ \vec{\alpha}, \Gamma^{\mathrm{x}}_i{\uparrow}^{\vec{\beta}} \vdash\\
\qquad\qquad N_i : [(\nu\,\vec{\beta}\,.\,[\Gamma^{\mathrm{x}}_i\ \langle\vec{\beta}\rangle/\Gamma^{\mathrm{x}}_i]\vec{M}_i)/\Gamma^{\mathrm{x}}\ ,\ \nu\,\vec{\beta}\,.\,c_i\ (\Gamma^{\mathrm{x}}_i\ \langle\vec{\beta}\rangle)\ /x]B)\\
\Gamma^{\mathrm{x}}, x \text{ fully applied w.r.t. } \vec{\beta} \text{ in } B \qquad \forall i(\vdash \mathbf{app\text{-}check}_u(\Gamma^{\mathrm{x}}_i; N_i)) \qquad \Gamma \vdash \vec{c}\ \mathbf{covers}\ a
\end{array}}{\Sigma;\Gamma \vdash \mathbf{fun}\ u\ (\vec{\alpha}, \Gamma^{\mathrm{x}}{\uparrow}^{\vec{\beta}})\ (\nu\,\vec{\beta}\,.\,\vec{c}\,(\Gamma^{\mathrm{x}}\ \langle\vec{\beta}\rangle) \setminus \vec{\Gamma^{\mathrm{x}}}{\uparrow}^{\vec{\beta}}{\rightarrow}\ \vec{N}) : \nabla\vec{\alpha}\,.\,\Pi\Gamma^{\mathrm{x}}{\uparrow}^{\vec{\beta}}\,.\,\Pi x{:}(\nabla\vec{\beta}\,.\,a\ (\Gamma^{\mathrm{x}}\ \langle\vec{\beta}\rangle))\,.\,B}\ \mathsf{t\text{-}pmfun}$$

$$\frac{\begin{array}{c}
\vdots\\
\forall i(\Sigma;\Gamma \vdash Q_i : \nabla\vec{\beta}\,.\,[\vec{Q}\ \langle\vec{\beta}\rangle/\Gamma^{\mathrm{x}}]A_i) \quad \Sigma;\Gamma \vdash Q : \nabla\vec{\beta}\,.\,a\ (\Gamma^{\mathrm{x}}\ \langle\vec{\beta}\rangle)
\end{array}}{\Sigma;\Gamma \vdash \mathbf{fun}\ u\ (\vec{\alpha}, \Gamma^{\mathrm{x}}{\uparrow}^{\vec{\beta}})\ (\nu\,\vec{\beta}\,.\,\vec{c}\,(\Gamma^{\mathrm{x}}\ \langle\vec{\beta}\rangle) \setminus \vec{\Gamma^{\mathrm{x}}}{\uparrow}^{\vec{\beta}}{\rightarrow}\ \vec{N})\ \langle\vec{\alpha}\rangle\ \vec{Q}\ Q : [\vec{Q}/\Gamma^{\mathrm{x}}, Q/x]B}\ \mathsf{t\text{-}app}$$

$$\begin{array}{l}
[\![(\mathbf{fun}\ u\ (\vec{\alpha}, \Gamma^{\mathrm{x}}{\uparrow}^{\vec{\beta}})\ (\nu\,\vec{\beta}\,.\,\vec{c}\,(\Gamma^{\mathrm{x}}\ \langle\vec{\beta}\rangle) \setminus \vec{\Gamma^{\mathrm{x}}}{\uparrow}^{\vec{\beta}}{\rightarrow}\ \vec{N}))\ \langle\vec{\alpha}\rangle\ \vec{Q}\ Q]\!]^{\Delta}_{\mathrm{tm/pf}} =\\
\quad \lambda T\,.\,\lambda\,\mu\,.\\
\quad\ ([\![\mathbf{fun}\ u\ (\vec{\alpha}, \Gamma^{\mathrm{x}}{\uparrow}^{\vec{\beta}})\ (\nu\,\vec{\beta}\,.\,\vec{c}\,(\Gamma^{\mathrm{x}}\ \langle\vec{\beta}\rangle) \setminus \vec{\Gamma^{\mathrm{x}}}{\uparrow}^{\vec{\beta}}{\rightarrow}\ \vec{N})]\!]_{\mathrm{pmfun}}\\
\qquad (T \oslash \vec{\beta})\ (\mathsf{tcadd}(\vec{\beta}) \circ \mu \circ \Delta(\emptyset))\ (\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![\vec{Q}]\!]^{\Delta}_{\mathrm{args}}))\\
\qquad (\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![Q]\!]^{\Delta}_{\mathrm{tm}}))\ ((\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![Q]\!]^{\Delta}_{\mathrm{pf}}))\ (T \oslash \vec{\beta})\ \mathsf{id})\\
\qquad (\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![Q]\!]^{\Delta}_{\mathrm{pf}}))\ (\mathsf{mk\text{-}valid\text{-}eq\text{-}pf}\ (\mathbf{remove\text{-}nus}_{\vec{\beta}}\ T\ \mu([\![Q]\!]^{\Delta}_{\mathrm{pf}})))\\
\qquad T\ (\mu \circ \Delta(\vec{\alpha}))\ (\mathsf{eq\text{-}refl}\ (\mathsf{mk\text{-}pair}\ (T \oslash \vec{\beta})\ (\mathsf{tcadd}(\vec{\beta}) \circ \mu \circ \Delta(\emptyset))))).1/2\ T\ \mathsf{id}
\end{array}$$

We first prove that $\Gamma' \vdash_{\mathrm{CIC}\ +\mathbb{T}} [\![\mathbf{fun}\ u\ (\vec{\alpha}, \Gamma^{\mathrm{x}}{\uparrow}^{\vec{\beta}})\ (\nu\,\vec{\beta}\,.\,\vec{c}\,(\Gamma^{\mathrm{x}}\ \langle\vec{\beta}\rangle) \setminus \vec{\Gamma^{\mathrm{x}}}{\uparrow}^{\vec{\beta}}{\to}\ \vec{N})]\!]_{\mathrm{pmfun}} : A_u$, where A_u is

$$
\begin{array}{l}
\Pi T\,.\,\Pi\mu : (\emptyset \rightrightarrows T)\,.\,\Pi[\![\vec{x} : \vec{A}]\!]^{\mu;\cdot}_{\mathrm{ctxt}}\,.\,\Pi x : \mathsf{eval}([\![a\ \vec{x}]\!]^{(\mu;\overrightarrow{x,x_{\mathrm{pf}}})}_{\text{tm-tp}})\,. \\
\quad \Pi x_{\mathrm{valid}} : \mathsf{valid}\text{-}a\ T\ \mu\ \overrightarrow{x\ x_{\mathrm{pf}}}\,.\,\Pi x_{\mathrm{pf}} : \mathsf{eval}([\![a\ \vec{x}]\!]^{(\mu;\overrightarrow{x,x_{\mathrm{pf}}})}_{\text{pf-tp}}(x))\,. \\
\quad \Pi e_{\mathrm{pf}} : \mathsf{eq}\ (\mathsf{re\text{-}move}\text{-}a\ T\ \mu\ \overrightarrow{x\ x_{\mathrm{pf}}}\ x\ x_{\mathrm{valid}})\ x_{\mathrm{pf}}\,.\,\Pi T_\beta\,.\,\Pi\mu_\beta : (\vec{\alpha} \rightrightarrows T_\beta)\,. \\
\quad \Pi e : \mathsf{eq}\ (\mathsf{mk\text{-}pair}\ T\ \mu)\ (\mathsf{mk\text{-}pair}\ (T_\beta \between \vec{\beta})\ (\mathsf{tcadd}(\vec{\beta}) \circ \mu_\beta \circ \mathsf{tcadd}(\vec{\alpha})))\,. \\
\quad\ \mathsf{pair} \\
\quad\quad \mathsf{eval}([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T_\beta\ (\mathsf{cast}\ e\ (\overrightarrow{x, x_{\mathrm{pf}}}, x, x_{\mathrm{pf}}))/\overrightarrow{x, x_{\mathrm{pf}}}, x, x_{\mathrm{pf}}][\![B]\!]^{\vec{\alpha},(\mu;\overrightarrow{x,x_{\mathrm{pf}}},x,x_{\mathrm{pf}})}_{\text{tm-tp}}) \\
\quad\quad (\lambda\,z\,.\,\mathsf{eval}([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T_\beta\ (\mathsf{cast}\ e\ (\overrightarrow{x, x_{\mathrm{pf}}}, x, x_{\mathrm{pf}}))\ /\overrightarrow{x, x_{\mathrm{pf}}}, x, x_{\mathrm{pf}}][\![B]\!]^{\vec{\alpha},(\mu;\overrightarrow{x,x_{\mathrm{pf}}},x,x_{\mathrm{pf}})}_{\text{pf-tp}}(z)))\,.
\end{array}
$$

The desired result then follows by the induction hypothesis, Lemma 6.2.10, Translation Substitution, and Re-Translation. To see that the above typing judgment holds, first note that

$$
\begin{array}{l}
\Gamma', [\![\Gamma^{\mathrm{x}}_i{\uparrow}^{\vec{\beta}}]\!]^{\mu_\beta;\cdot}_{\mathrm{ctxt}} \vdash \mathsf{eval}([\![N_i]\!]^{\vec{\alpha},(\mu_\beta;\Gamma^{\mathrm{x}}_i)}_{\mathrm{tm}}) \\
\qquad\qquad : \mathsf{eval}([\![[(\nu\,\vec{\beta}\,.\,[\Gamma^{\mathrm{x}}_i\ \langle\vec{\beta}\rangle/\Gamma^{\mathrm{x}}_i]\vec{M}_i)/\Gamma^{\mathrm{x}}\ ,\ \nu\,\vec{\beta}\,.\,c_i\ (\Gamma^{\mathrm{x}}_i\ \langle\vec{\beta}\rangle)\ /x]B]\!]^{\vec{\alpha},(\mu_\beta;\Gamma^{\mathrm{x}}_i)}_{\text{tm-tp}})
\end{array}
$$

The following equivalences hold on this type:

$$
\begin{array}{ll}
 & \mathsf{eval}([\![[(\nu\,\vec{\beta}\,.\,[\Gamma^{\mathrm{x}}_i\ \langle\vec{\beta}\rangle/\Gamma^{\mathrm{x}}_i]\vec{M}_i)/\Gamma^{\mathrm{x}}\ ,\ \nu\,\vec{\beta}\,.\,c_i\ (\Gamma^{\mathrm{x}}_i\ \langle\vec{\beta}\rangle)\ /x]B]\!]^{\vec{\alpha},(\mu_\beta;\Gamma^{\mathrm{x}}_i)}_{\text{tm-tp}}) \\
\cong & \mathsf{eval}([\mathsf{eval}([\![\nu\,\vec{\beta}\,.\,[\Gamma^{\mathrm{x}}_i\ \langle\vec{\beta}\rangle/\Gamma^{\mathrm{x}}_i]\vec{M}_i]\!]^{\vec{\alpha},(\mu_\beta;\Gamma^{\mathrm{x}}_i)}_{\text{tm/pf}})/(\overrightarrow{x, x_{\mathrm{pf}}}), \\
 & \qquad \mathsf{eval}([\![\nu\,\vec{\beta}\,.\,c_i\ (\Gamma^{\mathrm{x}}_i\ \langle\vec{\beta}\rangle)]\!]^{\vec{\alpha},(\mu_\beta;\Gamma^{\mathrm{x}}_i)}_{\text{tm/pf}})/(x, x_{\mathrm{pf}})][\![B]\!]^{\vec{\alpha},(\mu_\beta;\Gamma^{\mathrm{x}}_i)}_{\text{tm-tp}}) \\
\cong & \mathsf{eval}([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T\ (\mathsf{eval} \otimes \vec{\beta})([\![[\Gamma^{\mathrm{x}}_i\ \langle\vec{\beta}\rangle/\Gamma^{\mathrm{x}}_i]\vec{M}_i]\!]^{\vec{\alpha},(\mu_\beta;\Gamma^{\mathrm{x}}_i),\vec{\beta}}_{\text{tm/pf}})/(\overrightarrow{x, x_{\mathrm{pf}}}), \\
 & \qquad \mathbf{add\text{-}nus}_{\vec{\beta}}\ T\ (\mathsf{eval} \otimes \vec{\beta})([\![c_i\ (\Gamma^{\mathrm{x}}_i\ \langle\vec{\beta}\rangle)]\!]^{\vec{\alpha},(\mu_\beta;\Gamma^{\mathrm{x}}_i),\vec{\beta}}_{\text{tm/pf}})/(x, x_{\mathrm{pf}})][\![B]\!]^{\vec{\alpha},(\mu_\beta;\Gamma^{\mathrm{x}}_i)}_{\text{tm-tp}})
\end{array}
$$

Using $\vec{y}$ for the variables in Γ^{x}_i, Lemma 6.2.11 gives

$$
\begin{array}{l}
(\mathsf{eval} \otimes \vec{\beta})([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T\ \overrightarrow{y, y_{\mathrm{pf}}}/\overrightarrow{y, y_{\mathrm{pf}}}][\![[\Gamma^{\mathrm{x}}_i\ \langle\vec{\beta}\rangle/\Gamma^{\mathrm{x}}_i]\vec{M}_i]\!]^{\vec{\alpha},(\mu_\beta;\overrightarrow{y,y_{\mathrm{pf}}}),\vec{\beta}}_{\text{tm/pf}}) \\
\cong \mathsf{eval}([\![\vec{M}_i]\!]^{\vec{\alpha},(\mu_\beta;\cdot),\vec{\beta},(\mathsf{eval}\otimes\vec{\beta};\overrightarrow{y,y_{\mathrm{pf}}})}_{\text{tm/pf}}) \\
\cong \mathsf{eval}([\![\vec{M}_i]\!]^{(\mathsf{tcadd}(\vec{\beta})\circ\mu_\beta\circ\mathsf{tcadd}(\vec{\alpha});\overrightarrow{y,y_{\mathrm{pf}}})}_{\text{tm/pf}}) \\
= \mathsf{eval}([\![\vec{M}_i]\!]^{(\mu_x;\overrightarrow{y,y_{\mathrm{pf}}})}_{\text{tm/pf}})
\end{array}
$$

and

$$
\begin{aligned}
&(\mathsf{eval} \otimes \vec{\beta})([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T\ \overrightarrow{y, y_{\mathrm{pf}}}/\overrightarrow{y, y_{\mathrm{pf}}}][\![c_i\ (\overrightarrow{y, y_{\mathrm{pf}}}\ \langle\vec{\beta}\rangle)]\!]^{\vec{\alpha},(\mu_\beta;\overrightarrow{y,y_{\mathrm{pf}}}),\vec{\beta}}_{\mathrm{tm/pf}}) \\
&\cong \mathsf{eval}([\![c_i\ \overrightarrow{y, y_{\mathrm{pf}}}]\!]^{\vec{\alpha},(\mu_\beta;\cdot),\vec{\beta},(\mathsf{eval}\otimes\vec{\beta};\overrightarrow{y,y_{\mathrm{pf}}})}_{\mathrm{tm/pf}}) \\
&\cong \mathsf{eval}([\![c_i\ \overrightarrow{y, y_{\mathrm{pf}}}]\!]^{(\mathsf{tcadd}(\vec{\beta})\circ\mu_\beta\circ\mathsf{tcadd}(\vec{\alpha});\overrightarrow{y,y_{\mathrm{pf}}})}_{\mathrm{tm/pf}}) \\
&= \mathsf{eval}([\![c_i\ \overrightarrow{y, y_{\mathrm{pf}}}]\!]^{(\mu_x;\overrightarrow{y,y_{\mathrm{pf}}})}_{\mathrm{tm/pf}})
\end{aligned}
$$

where the final equality in each case comes from the equality e taken as an argument in each branch of $[\![\mathbf{fun}\ u\ (\vec{\alpha}, \Gamma^{\mathrm{x}}{\uparrow^{\vec{\beta}}})\ (\nu\,\vec{\beta}\,.\,\vec{c}\ (\Gamma^{\mathrm{x}}\ \langle\vec{\beta}\rangle)\setminus \vec{\Gamma^{\mathrm{x}}}{\uparrow^{\vec{\beta}}}\to \vec{N})]\!]_{\mathrm{pmfun}}$. As a final point, we also have

$$
\begin{aligned}
& \mathsf{eval}([\![c_i\ \overrightarrow{y, y_{\mathrm{pf}}}]\!]^{(\mu_x;\overrightarrow{y,y_{\mathrm{pf}}})}_{\mathrm{pf}}) \\
\equiv\ & \lambda\, T\,.\,\lambda\,\mu\,.\,\mathsf{valid}\text{-}c_i\ T\ (\mu\circ\mu_x)\ \mu(\overrightarrow{y, y_{\mathrm{pf}}}) \\
=\ & \mathsf{re\text{-}move}\text{-}a\ T_x\ \mu_x\ \Gamma^{\mathrm{x}}_i\ x\ (\mathsf{valid}\text{-}c_i\ T\ \mu_x\ \overrightarrow{y, y_{\mathrm{pf}}}) \\
=\ & x_{\mathrm{pf}}
\end{aligned}
$$

where this last equality holds by e_{pf}. Thus we have that the term

$$
\begin{aligned}
&\lambda\, x_{\mathrm{pf}} : \Pi T_c\,.\,\Pi\mu_c\,.\,\mathsf{valid}\text{-}a\ T_c\ \mu_c\ (\mu_c\circ\mathsf{eval})([\![\vec{N}_c]\!]^{(\mu_x;\Gamma^{\mathrm{x}}_{c_1})}_{\mathrm{args}})\,. \\
&\lambda\, e_{\mathrm{pf}} : \mathsf{eq}\ (\lambda\, T_c\,.\,\lambda\,\mu_c\,.\,\mathsf{valid}\text{-}c_i\ T_c\ \mu_c\ \mu_c(\Gamma^{\mathrm{x}}_1, x))\ x_{\mathrm{pf}}\,. \\
&\lambda\, T_\beta\,.\,\lambda\,\mu_\beta : (\vec{\alpha} \rightrightarrows T_\beta)\,. \\
&\lambda\, e : \mathsf{eq}\ (\mathsf{mk\text{-}pair}\ T_x\ \mu_x)\ (\mathsf{mk\text{-}pair}\ (T_\beta \oslash \vec{\beta})\ (\mathsf{tcadd}(\vec{\beta})\circ\mu_\beta\circ\mathsf{tcadd}(\vec{\alpha})))\,. \\
&\quad \mathsf{cast}\ e_{\mathrm{pf}}, (\mathsf{eq\text{-}symm}\ e) \\
&\quad\ (\mathsf{mk\text{-}pair} \\
&\qquad \mathsf{eval}([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T_\beta\ (\mathsf{cast}\ e\ \Gamma^{\mathrm{x}}_i)/\Gamma^{\mathrm{x}}_i][\![\vec{M}_i]\!]^{\vec{\alpha},(\mu_\beta;\Gamma^{\mathrm{x}}_i)}_{\mathrm{tm}}) \\
&\qquad \mathsf{eval}([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T_\beta\ (\mathsf{cast}\ e\ \Gamma^{\mathrm{x}}_i)/\Gamma^{\mathrm{x}}_i][\![\vec{M}_i]\!]^{\vec{\alpha},(\mu_\beta;\Gamma^{\mathrm{x}}_i)}_{\mathrm{pf}}))
\end{aligned}
$$

has the proper instance of type

$$
\begin{aligned}
&\Pi x_{\mathrm{pf}} : \mathsf{eval}([\![a\ \vec{x}]\!]^{(\mu;\overrightarrow{x,x_{\mathrm{pf}}})}_{\mathrm{pf\text{-}tp}}(x))\,. \\
&\quad \Pi e_{\mathrm{pf}} : \mathsf{eq}\ (\mathsf{re\text{-}move}\text{-}a\ T\ \mu\ \overrightarrow{x\ x_{\mathrm{pf}}}\ x\ x_{\mathrm{valid}})\ x_{\mathrm{pf}}\,.\,\Pi T_\beta\,.\,\Pi\mu_\beta : (\vec{\alpha} \rightrightarrows T_\beta)\,. \\
&\quad \Pi e : \mathsf{eq}\ (\mathsf{mk\text{-}pair}\ T\ \mu)\ (\mathsf{mk\text{-}pair}\ (T_\beta \oslash \vec{\beta})\ (\mathsf{tcadd}(\vec{\beta})\circ\mu_\beta\circ\mathsf{tcadd}(\vec{\alpha})))\,. \\
&\qquad \mathsf{pair} \\
&\qquad\ \mathsf{eval}([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T_\beta\ (\mathsf{cast}\ e\ (\overrightarrow{x, x_{\mathrm{pf}}}, x, x_{\mathrm{pf}}))/\overrightarrow{x, x_{\mathrm{pf}}}, x, x_{\mathrm{pf}}][\![B]\!]^{\vec{\alpha},(\mu;\overrightarrow{x,x_{\mathrm{pf}}},x,x_{\mathrm{pf}})}_{\mathrm{tm\text{-}tp}}) \\
&\qquad\ (\lambda\, z\,.\,\mathsf{eval}([\mathbf{add\text{-}nus}_{\vec{\beta}}\ T_\beta\ (\mathsf{cast}\ e\ (\overrightarrow{x, x_{\mathrm{pf}}}, x, x_{\mathrm{pf}})) \\
&\qquad\qquad\qquad\qquad /\overrightarrow{x, x_{\mathrm{pf}}}, x, x_{\mathrm{pf}}][\![B]\!]^{\vec{\alpha},(\mu;\overrightarrow{x,x_{\mathrm{pf}}},x,x_{\mathrm{pf}})}_{\mathrm{pf\text{-}tp}}(z)))
\end{aligned}
$$

in context $\Gamma', T_x, \mu_x : \emptyset \rightrightarrows T_x, \Gamma_i^{\mathrm{x}}$, as required.

Case:

$$\frac{\begin{array}{c}\Sigma; \cdot \vdash \nabla\vec{\alpha}\,.\,\Pi x\!:\!(\nabla\vec{\beta}\,.\,\mathsf{Name})\,.\,B : \mathsf{Type}_i \\ \forall i(\Sigma; \vec{\alpha}, \Gamma_i^{\alpha} \vdash M_i : \nabla\vec{\beta}\,.\,\mathsf{Name}) \quad \forall i(\Sigma; \vec{\alpha}, \Gamma_i^{\alpha} \vdash N_i : [M_i/x]B)\end{array}}{\Sigma; \Gamma \vdash \mathbf{nfun}\ (\vec{\alpha})\ (\nu\,\vec{\beta}\,.\,\vec{M} \setminus \vec{\Gamma^{\alpha}} \to \vec{N}) : \nabla\vec{\alpha}\,.\,\Pi x\!:\!(\nabla\vec{\beta}\,.\,\mathsf{Name})\,.\,B}\ \textsf{t-nfun}$$

$$\frac{\begin{array}{c}\vdots \\ \Sigma; \Gamma \vdash Q : \nabla\vec{\beta}\,.\,\mathsf{Name}\end{array}}{\Sigma; \Gamma \vdash \mathbf{nfun}\ (\vec{\alpha})\ (\nu\,\vec{\beta}\,.\,\vec{M} \setminus \vec{\Gamma^{\alpha}} \to \vec{Q})\ \langle\vec{\alpha}\rangle\ Q : [Q/x]B}$$

$$\begin{array}{l}[\![(\mathbf{nfun}\ (\vec{\alpha})\ (\vec{M} \to \vec{N}))\ \langle\vec{\alpha}\rangle\ Q]\!]^{\Delta}_{\mathrm{tm/pf}} = \\ \quad \lambda\,T\,.\,\lambda\,\mu\,. \\ \quad\ (\mathbf{fun}\ (T_2, \mu_2, x, x_{\mathrm{pf}}) \\ \qquad\ (\mathsf{name\text{-}matching}_{\vec{\alpha},i} 1\ T_x\ \mu_x \setminus T_x, \mu_x : (\vec{\alpha} \rightrightarrows T_x) \to [\![N_{\alpha_1}]\!]^{\vec{\alpha},(\mu_x;\cdot)}_{\mathrm{tm/pf}}\ T_x\ \mathsf{eval} \\ \qquad\ \mid \ldots \\ \qquad\ \mid \mathsf{fresh\text{-}matching}_{\vec{\alpha}}\ T_x\ \mu_x \setminus T_x, \mu_x : (\vec{\alpha}, \alpha \rightrightarrows T_x) \to [\![N_{\alpha}]\!]^{\vec{\alpha},\alpha,(\mu_x;\cdot)}_{\mathrm{tm/pf}}\ T_x\ \mathsf{eval}) \\ \quad\ )\ T\ (\mathsf{find\text{-}matching}_{\vec{\alpha}}\ T\ (\mu \circ \Delta(\vec{\alpha}))\ \mu([\![Q]\!]^{\Delta}_{\mathrm{tm}})\ \mu([\![Q]\!]^{\Delta}_{\mathrm{pf}}))\end{array}$$

The induction hypothesis yields

$$\Gamma' \vdash_{\mathrm{CIC}\ +\mathbb{T}} \mu([\![Q]\!]^{\Delta}_{\mathrm{tm/pf}}) : \mu([\![\nabla\vec{\beta}\,.\,\mathsf{Name}]\!]^{\Delta}_{\mathrm{tm/pf}})$$

and thus

$$\begin{array}{r}\Gamma' \vdash_{\mathrm{CIC}\ +\mathbb{T}} \mathsf{find\text{-}matching}_{\vec{\alpha}}\ T\ (\mu \circ \Delta(\vec{\alpha}))\ \mu([\![Q]\!]^{\Delta}_{\mathrm{tm}})\ \mu([\![Q]\!]^{\Delta}_{\mathrm{pf}}) \\ : \mathsf{name\text{-}matching}_{\vec{\alpha}}\ T\ (\mu \circ \Delta(\vec{\alpha}))\ \mu([\![Q]\!]^{\Delta}_{\mathrm{tm}})\ \mu([\![Q]\!]^{\Delta}_{\mathrm{pf}})\end{array}$$

The induction hypothesis also yields the following:

$$\begin{aligned}\Gamma', T_x, \mu_x : \vec{\alpha} \rightrightarrows T_x \vdash_{\mathrm{CIC}\ +\mathbb{T}} & [\![N_{\alpha_i}]\!]_{\mathrm{tm}}^{\vec{\alpha},(\mu_x;\cdot)}\ T_x\ \mathsf{eval} \\ & : \mathsf{eval}([\![[\nu\,\vec{\beta}\,.\,\alpha_i/x]B]\!]_{\text{tm-tp}}^{\vec{\alpha},(\mu_x;\cdot)}) \\ & \cong [\mathsf{eval}([\![\nu\,\vec{\beta}\,.\,\alpha_i]\!]_{\text{tm/pf}}^{\vec{\alpha},(\mu_x;\cdot)})/(x, x_{\mathrm{pf}})][\![B]\!]_{\mathrm{tm}}^{\vec{\alpha},(\mu_x;x,x_{\mathrm{pf}})}\end{aligned}$$

$$\begin{aligned}\Gamma', T_x, \mu_x : \vec{\alpha} \rightrightarrows T_x \vdash_{\mathrm{CIC}\ +\mathbb{T}} & [\![N_{\beta_i}]\!]_{\mathrm{tm}}^{\vec{\alpha},(\mu_x;\cdot)}\ T_x\ \mathsf{eval} \\ & : \mathsf{eval}([\![[\nu\,\vec{\beta}\,.\,\beta_i/x]B]\!]_{\text{tm-tp}}^{\vec{\alpha},(\mu_x;\cdot)}) \\ & \cong [\mathsf{eval}([\![\nu\,\vec{\beta}\,.\,\beta_i]\!]_{\text{tm/pf}}^{\vec{\alpha},(\mu_x;\cdot)})/(x, x_{\mathrm{pf}})][\![B]\!]_{\mathrm{tm}}^{\vec{\alpha},(\mu_x;x,x_{\mathrm{pf}})}\end{aligned}$$

$$\begin{aligned}\Gamma', T_x, \mu_x : \vec{\alpha}, \alpha \rightrightarrows T_x \vdash_{\mathrm{CIC}\ +\mathbb{T}} & [\![N_{\alpha_i}]\!]_{\mathrm{tm}}^{\vec{\alpha},\alpha,(\mu_x;\cdot)}\ T_x\ \mathsf{eval} \\ & : \mathsf{eval}([\![[\nu\,\vec{\beta}\,.\,\alpha/x]B]\!]_{\text{tm-tp}}^{\vec{\alpha},\alpha,(\mu_x;\cdot)}) \\ & \cong [(\mathsf{eval} \circ \mathsf{eval}^{-1}_{\mu\circ\mathsf{add}(\alpha)} \circ \mathsf{eval})([\![\nu\,\vec{\beta}\,.\,\alpha]\!]_{\text{tm/pf}}^{\vec{\alpha},(\mu_x\circ\mathsf{add}(\alpha);\cdot)}) \\ & \qquad\qquad /(x, x_{\mathrm{pf}})][\![B]\!]_{\mathrm{tm}}^{\vec{\alpha},(\mu_x\circ\mathsf{add}(\alpha);x,x_{\mathrm{pf}})}\end{aligned}$$

Thus the pattern-matching function in $[\![(\mathbf{nfun}\ (\vec{\alpha})\ (\nu\,\vec{\beta}\,.\,\vec{M} \to \vec{N}))\ \langle\vec{\alpha}\rangle\ Q]\!]_{\mathrm{tm}}^{\Delta}$ has type

$$\begin{aligned}&\Pi T_2\,.\,\Pi\mu_2\,.\,\Pi x : [\![\nabla\vec{\beta}\,.\,\mathsf{Name}]\!]_{\text{tm-tp}}^{\vec{\alpha},\mu}\,.\,\Pi x_{\mathrm{pf}} : [\![\nabla\vec{\beta}\,.\,\mathsf{Name}]\!]_{\text{tm-tp}}^{\vec{\alpha},\mu}\,. \\ &\quad \mathsf{name\text{-}matching}_{\vec{\alpha}}\ T_2\ \mu_2\ x\ x_{\mathrm{pf}} \Rightarrow [\![B]\!]_{\mathrm{tm}}^{\vec{\alpha},(\mu_2;x,x_{\mathrm{pf}})}\end{aligned}$$

and the entire term has the desired type. The pf case is similar.

□

Translation typing already gives consistency of CNIC, since if there were a term of type $\Pi A : \mathsf{Type}_i\,.\,A$, proving that every type is inhabited, then there would be a translated term of type

$$\Pi T\,.\,\Pi\mu\,.\,\Pi A : (\Pi T_2\,.\,\Pi\mu_2\,.\,\mathsf{Type}_i)\,.\,\Pi A_{\mathrm{pf}} : (\Pi T_2\,.\,\Pi\mu_2\,.\,\Pi x : \mu_2(A)\,.\,\mathsf{Type}_i)\,.\,A\ T\ \mathsf{id}$$

in CIC + $\mathbb{T}$, from which it is then possible to construct a term of type $\Pi A : \mathsf{Type}_i\,.\,A$. It is useful independently, however, to have strong normaliztion hold, so that equality is decidable in the theory.

To show strong normalization by a translation it is required that a step of reduction in the source language induces at least one step of reduction in the target language.

This is not so here, however; Translation Reduction only guarantees that if $M \rightsquigarrow N$ in CNIC then $[\![M]\!]^{\Delta}_{\mathrm{tm}} \cong [\![N]\!]^{\Delta}_{\mathrm{tm}}$. Re-inspecting the proof of Translation Reduction, the only case that does not result in a reduction in CIC + $\mathbb{T}$ is in reducing name-matching functions, since this requires reducing a term of the form

$$\mathsf{find\text{-}name\text{-}case}_{\vec{\alpha}}\ T\ (\mu \circ \Delta(\emptyset))\ (\mu \circ \Delta(\alpha))$$

where μ is the variable for the argument to the term. Translation Reduction proceeds from this case by a step of $\sim$, but this will not help us here as $\sim$ is not reduction. What is needed is for μ itself to be ground so that $\mathsf{find\text{-}name\text{-}case}_{\vec{\alpha}}$ can reduce. This can only be accomplished by passing a ground mapping to the term in question, which in turn requires passing a ground mapping to every term in the translation. This, however, would sacrifice Re-Translation and Translation Substitution.

It is possible, though, to keep two copies of the translation of a term, one that is passed a ground mapping and one that satisfies Re-Translation and Translation Substitution. This is done with the function forget, defined by the trivial projection function $\lambda\, x\,.\,\lambda\, y\,.\, y$ that forgets one of its arguments. We can thus form $\mathsf{forget}\ (M\ T\ \mu_g)\ M$ where μ_g is ground, and we have a term that is equal to M but that keeps a separate copy of M that gets passed a ground mapping.

Note that forget here is similar to the R function in Hofmann's dissertation [28]. This is not surprising, as the R function there was used to map reductions in extensional type theory, which are any provable equalities, to reductions in intensional type theory. This is similar to the problem here, where we have $\sim$ steps that do not correlate to actual intensional reductions.

Theorem 6.2.1 (Strong Normalization). *The language* CNIC *is strongly normalizing.*

Proof. Let $\{\!|M|\!\}^{\Delta}_{\mathrm{tm}}$ and $\{\!|M|\!\}^{\Delta}_{\mathrm{tm}}$ be the set of all terms that can be got from $[\![M]\!]^{\Delta}_{\mathrm{tm}}$ and $[\![M]\!]^{\Delta}_{\mathrm{pf}}$, respectively, by recursively replacing every subterm of the form $[\![N]\!]^{\Delta}_{\mathrm{tm}}$ with a term of the form

$$\lambda\, T\,.\,\lambda\, \mu\,.\,\mathsf{forget}\ ([\![N]\!]^{\Delta}_{\mathrm{tm}}\ T_g\ \mu_g)\ ([\![N']\!]^{\Delta}_{\mathrm{tm}}\ T\ \mu)$$

where N' is a CNIC term such that $[\![N']\!]^{\Delta}_{\mathrm{tm}}\ T'\ \mu'$ reduces to $[\![N]\!]^{\Delta}_{\mathrm{tm}}\ T'\ \mu'$ for any ground T' and μ'. By Translation Reduction we immediately have that any element of $\{\!|M|\!\}^{\Delta}_{\mathrm{tm}}$ or $\{\!|M|\!\}^{\Delta}_{\mathrm{pf}}$ is equal by reductions of forget and steps of $\cong$ to $[\![M]\!]^{\Delta}_{\mathrm{tm}}$ or $[\![M]\!]^{\Delta}_{\mathrm{pf}}$, respectively. Further, it is apparent by Translation Typing that every element of $\{\!|M|\!\}^{\Delta}_{\mathrm{tm}}$ and $\{\!|M|\!\}^{\Delta}_{\mathrm{pf}}$ is well-typed.

Any reduction $M \rightsquigarrow N$ in CNIC induces an equivalent reduction from elements of $\{\!|M|\!\}^{\Delta}_{\mathrm{tm}}$ to $\{\!|N|\!\}^{\Delta}_{\mathrm{tm}}$ and from elements of $\{\!|M|\!\}^{\Delta}_{\mathrm{pf}}$ to $\{\!|N|\!\}^{\Delta}_{\mathrm{pf}}$. This is because the only way a forget terms are only in the tm subterms, so they do not interfere with reductions for pattern-matching and name-matching functions. Thus, any infinite reduction sequence in CNIC would lead to an infinite reduction sequence in CIC + $\mathbb{T}$, which is impossible. $\square$

Chapter 7

Constructor Predicate Type Theory

Constructor Predicate Type Theory, or CPTT, is a type theory for encoding and manipulating name binding. Using the HOEC approach introduced above, CPTT makes name binding a straightforward notion to define. This in turn makes programming languages easier to define, implement, and prove correct.

As above, name binding is encoded with ν-abstractions. In contrast with CNIC, however, CPTT allows ν-abstractions to bind constructors of any type. In this way, CPTT supports typing, the fourth property of name binding discussed above. This in turn makes it easy to encode typed programming languages, as the type associated with a name can be encoded into its type. As in CNIC, names can be compared and name bindings can be travesed by recursive functions. These features are not possible with existing formalisms.

The fact that ν-abstractions can introduce constructors at arbitrary types leads to a difficulty with totality of pattern-matching functions. Specifically, for a pattern-matching function to be total it must have a pattern for each possible input. But, if arbitrary constructors can be introduced then a pattern-matching function would need infinitely many cases to remain total. This is not possible.

To alleviate this difficulty, CPTT introduces a concept called *constructor predicates*. Constructor predicates specify what constructors could possibly be in a term. This allows a pattern-matching function to specify what constructors it expects. It is then an error to apply a function to a term with constructors it does not expect. Totality is

then regained, as a pattern-matching function need only be total on the constructors that satisfy its predicate.

The remainder of this Chapter is organized as follows. Section 7.1 briefly introduces CPTT through an example. Section 7.2 then formalizes the operational and static semantics of CPTT.

7.1 Informal Introduction and Examples

In this section we introduce CPTT with some examples. One of the key benefits of CPTT is that it enables name binding with typing. Thus we focus on a typed programming language, the simply-typed λ-calculus, to make the most use of this feature.

The types of the simply-typed λ-calculus include some set of base types, along with the type $A \Rightarrow B$ for any types A and B. We here just use one base type, b. The λ-calculus types can then be encoded as follows:

$$\begin{array}{lcl} \mathsf{tp} & : & \mathsf{IndType}_0 \\ \mathsf{b} & : & \mathsf{tp} \\ \mathsf{arrow} & : & \mathsf{tp} \Rightarrow \mathsf{tp} \Rightarrow \mathsf{tp} \end{array}$$

Note that tp is not given type Type_0. Instead, inductive types have type $\mathsf{IndType}_i$ for some i. This is because, in CPTT, inductive types are not types by themselves. Instead, they become types when paired with a constructor predicate, or CPs. Thus the second declaration above does not give the type tp to the b constructor, as tp is not a type. Instead, tp here is syntactic sugar for $\lceil\top\rceil\ \mathsf{tp}$. Similarly, arrow is actually given the type $\lceil\top\rceil\ \mathsf{tp} \Rightarrow \lceil\top\rceil\ \mathsf{tp} \Rightarrow \lceil\top\rceil\ \mathsf{tp}$. This is type is abbreviated as in the above to increase clarity.

The types given to constructors are actually type templates in CPTT. A constructor thus does not have just one type but many types, one for each output CP. The type template can be instantiated for any output type that the constructor satisfies. For example, b can be given the type $\lceil\top\rceil\ \mathsf{tp}$, as $\top$ is the vacuously true CP that all

constructors satisfy b can also be given the type $\lceil \mathsf{b} \rceil\ \mathsf{tp}$, specifying that b is the only allowed constructor. b cannot be given the type $\lceil \mathsf{arrow} \rceil\ \mathsf{tp}$, as b is not the constructor arrow. It can be given the type $\lceil \mathsf{arrow} \vee \mathsf{b} \rceil\ \mathsf{tp}$, however.

Given the λ-calculus types defined as the type tp, we now turn to defining the λ-calculus terms. To incorporate typing into the definition, the inductive type for terms is indexed by elements of tp. Specifically, $\mathsf{term}\ t$ is the type of (encodings of) λ-terms with type t. The terms can be defined as follows:

$$\begin{array}{lcl} \mathsf{term} & : & \mathsf{tp} \Rightarrow \mathsf{IndType}_0 \\ \mathsf{app} & : & \Pi t_1 : \mathsf{tp}\,.\,\Pi t_2 : \mathsf{tp}\,.\,\mathsf{term}\ (\mathsf{arrow}\ t_1\ t_2) \Rightarrow \mathsf{term}\ t_1 \Rightarrow \mathsf{term}\ t_2 \\ \mathsf{lam} & : & \Pi t_1 : \mathsf{tp}\,.\,\Pi t_2 : \mathsf{tp}\,.\,(\nabla c : \mathsf{term}\ t_1\,.\,\mathsf{term}\ t_2) \Rightarrow \mathsf{term}\ (\mathsf{arrow}\ t_1\ t_2) \end{array}$$

Again, term is an inductive type indexed by elements of tp. app takes any λ-calculus types t_1 and t_2 and build an application from a λ-calculus term of type $\mathsf{arrow}\ t_1\ t_2$ and one of type t_1. The result type is t_2. lam takes any t_1 and t_2 and builds a λ-abstraction from a term of type $\nabla c : \mathsf{term}\ t_1\,.\,\mathsf{term}\ t_2$. This is the type of name bindings, where the name has type $\mathsf{term}\ t_1$ and the body of the name binding has type $\mathsf{term}\ t_2$. Note that there need be no constructor for variables, as variables are introduced by the lam constructor with name bindings.

As a simple function over this type, we consider again a function for counting variable occurrences. This is defined as follows:

$$\begin{array}{l} \textbf{fun}\ \mathsf{countvars}\ (\xi, t : \mathsf{tp}) \\ \qquad (c \setminus c : \mathsf{term}\ t \rightarrow \mathsf{succ}\ \mathsf{zero}\ \mid \\ \qquad\ \mathsf{app}\ t_1\ t_2\ x\ y \setminus t_1, t_2, x : \mathsf{term}\ (\mathsf{arrow}\ t_1\ t_2), y : \mathsf{term}\ t_1 \rightarrow \\ \qquad\qquad \mathsf{add}\ (\mathsf{countvars}\ \xi\ (\mathsf{arrow}\ t_1\ t_2)\ x)\ (\mathsf{countvars}\ \xi\ t_1\ y)\ \mid \\ \qquad\ \mathsf{lam}\ t_1\ t_2\ x \setminus t_1, t_2, x : \nabla c : \mathsf{term}\ t_1\,.\,\mathsf{term}\ t_2 \rightarrow \\ \qquad\qquad \mathsf{lift\text{-}nat}\ (\nu\, c : \mathsf{term}\ t_1\,.\,\mathsf{countvars}\ t_2\ x\ \langle c \rangle)) \end{array}$$

The first case is for variables. This case matches against any arbitrary constructor of type $\mathsf{term}\ t$, and returns $\mathsf{succ}\ \mathsf{zero}$. The second case matches an application and recurses on the two arguments, adding the results. The third case recurses inside the name binding for a λ-abstraction, by binding a new constructor, recursing on the constructor replacement, and calling $\mathsf{lift\text{-}nat}$. Note that we have not discussed the

first parameter, ξ. ξ is a CP variable specifying the CP for the arguments. The type of countvars is

$$\Pi\xi : \mathsf{CP} \,.\, \Pi t : \left[\psi_{\mathsf{term}} \wedge \xi\right] \mathsf{tp} \,.\, \Pi x : \left[\psi_{\mathsf{term}} \wedge \xi\right] \,.\, \left[\psi_{\mathsf{nat}}\right] \mathsf{nat}$$

where ψ_{nat} is the CP $\mathsf{nat} \vee \mathsf{zero} \vee \mathsf{succ}$ and ψ_{term} is the CP

$$\mathsf{tp} \vee \mathsf{b} \vee \mathsf{arrow} \vee \mathsf{term} \vee \mathsf{app} \vee \mathsf{lam} \vee (\mathsf{term}\ x \setminus x) \vee (\neg\Pi{*}.\mathsf{term}\ x \setminus x)$$

This CP matches any of the given constructors, along with any constructor of type term x for some x and any constructor that does not have return type term x for some x. The definition of countvars is thus total, as the type ensures that it will only "see" constructors of the given forms.

Capture-avoiding substitution can be defined by the following function:

$$\begin{array}{l}
\textbf{fun}\ \mathsf{subst}\ (\xi, t_M, t_N, M) \\
\quad (\nu\, c : \mathsf{term} t_M \,.\, c \setminus \cdot \to M \mid \\
\quad \nu\, c : \mathsf{term} t_M \,.\, d \setminus d : \mathsf{term}\ t_N \to d \mid \\
\quad \nu\, c : \mathsf{term} t_M \,.\, \mathsf{app}\ (t_1\ \langle c \rangle)\ (t_2\ \langle c \rangle)\ (x\ \langle c \rangle)\ (y\ \langle c \rangle) \\
\qquad \setminus t_1 : \nabla c \,.\, \mathsf{tp}, t_2 : \nabla c \,.\, \mathsf{tp}, x : \nabla c \,.\, \mathsf{term}\ (\mathsf{arrow}\ t_1\ \langle c \rangle\ t_2\ \langle c \rangle), \\
\qquad y : \nabla c \,.\, \mathsf{term}\ t_1\ \langle c \rangle \to \\
\quad \mathsf{app}(\mathsf{lift\text{-}tp}\ t_1)\ (\mathsf{lift\text{-}tp}\ t_2) \\
\qquad (\mathsf{subst}\ \xi\ t_M\ (\nu\, c \,.\, \mathsf{arrow}\ t_1\ \langle c \rangle\ t_2\ \langle c \rangle)\ M\ x) \\
\qquad (\mathsf{subst}\ \xi\ \nu\, c \,.\, t_1\ t_M\ M\ y) \mid \\
\quad \nu\, c : \mathsf{term} t_M \,.\, \mathsf{lam}\ (t_1\ \langle c \rangle)\ (t_2\ \langle c \rangle)\ (x\ \langle c \rangle) \\
\qquad \setminus t_1 :, t_2, x : \nabla c : (\mathsf{term}\ t_M) \,.\, \nabla d : (\mathsf{term}\ t_1\ \langle d \rangle) \,.\, \mathsf{term}\ t_2\ \langle d \rangle \to \\
\quad \mathsf{lam}\ (\mathsf{lift\text{-}tp}\ t_1)\ (\mathsf{lift\text{-}tp}\ t_2) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad) \\
\qquad (\nu\, d : (\mathsf{term}\ (\mathsf{lift\text{-}tp}\ t_1)) \,.\, \mathsf{subst}\ (\xi \vee d)\ t_2\ \nu\, c \,.\, x\ \langle c, d \rangle)
\end{array}$$

Note that lift-tp is the lifting function for tp, similar to lift-nat for nat. subst substitutes the term M of type t_M into the binding $\nu\, c \,.\, N$. The body N has type t_N, but, since t_N is associated with N, it is easier to write the function if we use a ν-abstraction for

t_N as well. Thus we have the type

$$\begin{array}{l}\Pi\xi : \mathsf{CP} \,.\, \Pi t_M : \left[\psi_{\mathsf{term}} \wedge \xi\right] \mathsf{tp} \,.\, \Pi t_N : \nabla c : \mathsf{term}\ t_M \,.\, \left[(\psi_{\mathsf{term}} \vee c) \wedge \xi\right] \mathsf{tp} \,. \\ \quad \Pi M : \left[\psi_{\mathsf{term}} \wedge \xi\right] (\mathsf{term}\ t_M) \,. \\ \quad \Pi N : \nabla c : \mathsf{term}\ t_M \,.\, \left[(\psi_{\mathsf{term}} \vee c) \wedge \xi\right] (\mathsf{term}\ t_N\ \langle c \rangle) \,. \\ \quad\quad \left[\psi_{\mathsf{term}} \wedge \xi\right] \mathsf{term}\ (\mathsf{lift\text{-}tp}\ t_N)\end{array}$$

for subst. This specifies that if all arguments satisfy ξ, then so does the output. Note that the bodies of t_N and N are allowed to contain the constructor c in addition to satisfying ξ, as c will be removed in the output. This type demonstrates one way in which CPs are useful. Consider the CP $\psi_{\mathsf{term}} \wedge (\neg(\mathsf{term}\ t \setminus t))$. This indicates that there are no free constructors of type term t for some t. A term satisfying this CP is thus the encoding of a closed λ-term, meaning it has no free variables. If this CP is used as the input CP to subst, then the output type also is guaranteed to satisfy this CP. Thus the type of subst states that it brings closed λ-terms to closed λ-terms. This would be useful in writing an evaluator for λ-terms, as it guarantees such an evaluator will never have to specify a value for free variables.

7.2 CPTT Formalized

The syntax of CPTT is given in Figure 7.1. This is primarily for the sake of preciseness, as most of these constructs have been introduced above or in a previous chapter. The one new construct here is the CP $\phi \oplus \phi$ which represents the exclusive-or of two CPs. This will play an important role in the operational semantics below. Note that, although the CPs are given as a separate syntactic category in this figure, they are actually considered as terms here and in the below. We now state some conventions. M, N, and Q are used for terms, A and B are used for terms meant to be types, and I is used for terms meant to be inductive types. x, y, and z are used for variables, c, d, e, and f are used for constructors, ϕ and ψ are used for terms meant to be CPs, Γ is used for typing contexts, and σ is used for substitutions.

$$
\begin{array}{lll}
\text{Terms} & M ::= & \mathsf{Type}_i \parallel \Pi x{:}A\,.\,B \parallel \nabla c{:}A\,.\,B \parallel \left[\phi\right] I \\
& & \parallel \mathsf{IndType}_i \parallel c \parallel x \parallel M\ \langle c\rangle \parallel \nu\, c{:}A\,.\,M \\
& & \parallel M\ M \parallel \lambda\, x{:}A\,.\,M \parallel \mathbf{fun}\ x\ (\Gamma)\ (\vec{P} \setminus \vec{\Gamma} \to \vec{M}) \\
\text{Constructor Predicates} & \phi ::= & \top \parallel \bot \parallel c \parallel \xi \parallel A \setminus \Gamma^{\mathrm{x}} \parallel \Pi{*}.I \setminus \Gamma^{\mathrm{x}} \\
& & \parallel \phi \wedge \phi \parallel \phi \vee \phi \parallel \phi \oplus \phi \parallel \neg\phi \\
\text{Contexts} & \Gamma ::= & \Gamma, c : A \parallel \Gamma, x : A \parallel \cdot \\
\text{Substitutions} & \sigma ::= & [M/x, \sigma] \parallel \cdot
\end{array}
$$

Figure 7.1: Syntax of CPTT

$$
\begin{array}{l}
(\mathbf{fun}\ u\ (\vec{c}, \Gamma^{\mathrm{x}})\ (\ldots \mid \nu\, \vec{d}\,.\, c_i\ \vec{x}\ \langle \vec{d}\rangle \setminus \vec{x} \to M_i \mid \ldots))\ \langle \vec{c}\rangle\ \vec{N'}\ (\nu\, \vec{d}\,.\, c_i\ \vec{N}) \\
\qquad \rightsquigarrow \\
\qquad [\mathbf{fun}\ u\ (\vec{c}, \Gamma^{\mathrm{x}})\ (\ldots \mid \nu\, \vec{d}\,.\, c_i\ \vec{x}\ \langle \vec{d}\rangle \setminus \vec{x} \to M_i \mid \ldots)/u, (\nu\, \vec{d}\,.\, \vec{N})/\vec{x}]M_i
\end{array}
$$

$$
\begin{array}{lcl}
(\lambda\, x{:}A\,.\,M)\ N & \rightsquigarrow & [N/x]M \\
(\nu\, c{:}A\,.\,M)\ \langle c\rangle & \rightsquigarrow & M
\end{array}
$$

Figure 7.2: Operational Semantics of CPTT: Terms

7.2.1 Operational Semantics

The operational semantics of CPTT is given by two rewrite systems, one for the terms and one for the CPs. For the terms, an HNRS is used. This is system is defined in Figure 7.2, and is similar to the one given for CNIC in Chapter 5. It is straightforward to see that this system, like the other, is orthogonal and thus confluent.

To rewrite the CPs, note that the CPs form a boolean algebra. It is known that a convergent rewrite system cannot be defined for boolean algebras [66]. However, it is possible to define an AC-rewrite system for boolean rings, which are defined in terms of conjunction and exclusive-or [31]. The $\wedge$ and $\oplus$ operators are both defined to be AC. Disjunction and negation are then rewritten to equivalent formulas that use conjunction and exclusive-or.

The rewrite system for CPs, based on the standard AC-rewrite system for boolean rings, is given in Figure 7.3. The first eight rules are the standard rewrite system of Hsiang and Dershowitz [31]. These are known to be convergent. The remaining rules in a sense define the meaning of the specific CPs of CPTT. In the CP $\phi_1 \wedge \phi_2$, if ϕ_1 is a more specific CP than ϕ_2, meaning ϕ_1 matches the appropriate pattern for ϕ_2,

$$
\begin{array}{rcll}
x \vee y & \rightsquigarrow & (x \wedge y) \oplus x \oplus y & \\
\neg x & \rightsquigarrow & x \oplus \top & \\
x \oplus \bot & \rightsquigarrow & x & \\
x \oplus x & \rightsquigarrow & \bot & \\
x \wedge \top & \rightsquigarrow & x & \\
x \wedge x & \rightsquigarrow & x & \\
x \wedge \bot & \rightsquigarrow & \bot & \\
x \wedge (y \oplus z) & \rightsquigarrow & (x \wedge y) \oplus (x \wedge z) & \\
c \wedge d & \rightsquigarrow & \bot & \text{if } c \neq d \\
c^A \wedge B \setminus \Gamma^{\mathrm{x}} & \rightsquigarrow & c^A & \text{if } A = \sigma B \\
c^A \wedge B \setminus \Gamma^{\mathrm{x}} & \rightsquigarrow & \bot & \text{if } A \not\downarrow B \\
c^{\Pi\Gamma \,.\, I} \wedge \Pi *.I' \setminus \Gamma^{\mathrm{x}} & \rightsquigarrow & c^{\Pi\Gamma \,.\, I} & \text{if } I = \sigma I' \\
c^{\Pi\Gamma \,.\, I} \wedge \Pi *.I' \setminus \Gamma^{\mathrm{x}} & \rightsquigarrow & \bot & \text{if } I \not\downarrow I' \\
A \setminus \Gamma^{\mathrm{x}}_A \wedge B \setminus \Gamma^{\mathrm{x}}_B & \rightsquigarrow & A \setminus \Gamma^{\mathrm{x}}_A & \text{if } A = \sigma B \\
A \setminus \Gamma^{\mathrm{x}}_A \wedge B \setminus \Gamma^{\mathrm{x}}_B & \rightsquigarrow & \bot & \text{if } A \not\downarrow B \\
\Pi\Gamma \,.\, I_1 \setminus \Gamma^{\mathrm{x}}_1 \wedge \Pi *.I_2 \setminus \Gamma^{\mathrm{x}}_2 & \rightsquigarrow & \Pi\Gamma \,.\, I_1 \setminus \Gamma^{\mathrm{x}}_1 & \text{if } I_1 = \sigma I_2 \\
\Pi\Gamma \,.\, I_1 \setminus \Gamma^{\mathrm{x}}_1 \wedge \Pi *.I_2 \setminus \Gamma^{\mathrm{x}}_2 & \rightsquigarrow & \bot & \text{if } I_1 \not\downarrow I_2 \\
\Pi *.I_1 \setminus \Gamma^{\mathrm{x}}_1 \wedge \Pi *.I_2 \setminus \Gamma^{\mathrm{x}}_2 & \rightsquigarrow & \Pi *.I_1 \setminus \Gamma^{\mathrm{x}}_1 & \text{if } I_1 = \sigma I_2 \\
\Pi *.I_1 \setminus \Gamma^{\mathrm{x}}_1 \wedge \Pi *.I_2 \setminus \Gamma^{\mathrm{x}}_2 & \rightsquigarrow & \bot & \text{if } I_1 \not\downarrow I_2
\end{array}
$$

Figure 7.3: Operational Semantics of CPTT: CPs

then $\phi_1 \wedge \phi_2$ rewrites to ϕ_1, the more specific CP. If these cannot be unified (specified using the notation $M_1 \not\downarrow M_2$) then $\phi_1 \wedge \phi_2$ rewrites to the false CP $\bot$.

It is straightforward to see this rewrite system is terminating [61]. To see that it is confluent requires checking that all the critical pairs are joinable, as per Section 2.3.3. In fact, only critical pairs with the new rules need be checked, as the old rules are already known to be convergent. Each new rule is either of the form $\phi_1 \wedge \phi_2 \rightsquigarrow \phi_1$ or $\phi_1 \wedge \phi_2 \rightsquigarrow \bot$, so only critical pairs with such rules need be searched. It turns out that all these critical pairs are in fact joinable, so we spare the reader the details here.

As a final point about the operational semantics, it was shown in Section 4.4 that both termination and confluence are modular between a left-linear HNRS and an ACRS such as the two rewrite systems here if the set of terms is restricted. This restriction states that no term-redex should be a subterm of a CP-redex. To ensure this is the case, the types in CPs are required to be normal forms. Given this, Section 4.4 shows that the combined system is confluent, as each individual system is. This result also shows that any strong normalization proof need only consider rewriting for terms, as termination of that rewrite system will extend to the full operational semantics.

7.2.2 Static Semantics

The static semantics of CPTT is given in Figure 7.4. We note differences from CNIC. The most notable difference is the addition of $\mathsf{IndType}_i$. The $\mathsf{IndType}_i$ mirror the type universes Type_i as a chain of inductive types. We have $\mathsf{IndType}_i$ is a constructor for the inductive type specified by $\mathsf{IndType}_{i+1}$. Since inductive types do not become types without the addition of a CP, we have $\mathsf{IndType}_i$ thus has type $\left[\phi\right]\ \mathsf{IndType}_{i+1}$. The theory makes the $\mathsf{IndType}_i$ special, as they satisfy any CP. Thus ϕ in the type of $\mathsf{IndType}_i$ can be any CP. This is the meaning of the rule $\mathsf{t\text{-}indtype}$.

To form an inductive type requires an element of the type $\left[\phi\right]\ \mathsf{IndType}_i$ for some ϕ. Thus the inductive types themselves are elements of the inductive type defined by $\mathsf{IndType}_i$. This puts type constructors on the same level as term constructors, and both of these can be introduced locally. The rule $\mathsf{t\text{-}cp}$ specifies that $\left[\phi\right]\ A$ is a Type_i is A is an element of the type $\left[\phi\right]\ \mathsf{IndType}_i$.

Constructors in CPTT have infinitely many types, one for each output CP. To form these types, the operation $\mathbf{reify}^i_c(\phi, A)$ is defined. This takes the type scheme A associated with c and constructs the instance of this type scheme whose return type uses the CP ϕ. i is the level of A, meaning that the return type of A has type Type_i. $\mathbf{reify}^i_c(\phi, A)$ is defined as follows:

$$
\begin{array}{lcl}
\mathbf{reify}^i_c(\phi, \left[\top\right]\ I) & = & \left[\phi \vee c\right]\ I \\
\mathbf{reify}^i_c(\phi, \Pi x : A\,.\,B) & = & \Pi x : \mathbf{arg\text{-}reify}^i_c(\phi, A)\,.\,\mathbf{reify}^i_c(\phi, B) \\
\mathbf{arg\text{-}reify}^i_c(\phi, \left[\psi\right]\ I) & = & \left[\phi \wedge \psi\right]\ I \\
\mathbf{arg\text{-}reify}^i_c(\phi, \nabla d : A\,.\,B) & = & \nabla d : A\,.\,\mathbf{arg\text{-}reify}^i_c(\phi \vee d, B) \\
\mathbf{arg\text{-}reify}^i_c(\phi, \Pi x : A\,.\,B) & = & \Pi x : \mathbf{neg\text{-}reify}^i_c(\phi, A)\,.\,\mathbf{arg\text{-}reify}^i_c(\phi, B) \\
\mathbf{arg\text{-}reify}^i_c(\phi, \mathsf{Type}_j) & = & \mathsf{Type}_j \text{ if } j \leq i \\
\mathbf{neg\text{-}reify}^i_c(\phi, \left[\psi\right]\ I) & = & \left[\neg c \wedge \psi\right]\ I \\
\mathbf{neg\text{-}reify}^i_c(\phi, \Pi x : A\,.\,B) & = & \Pi x : \mathbf{neg\text{-}reify}^i_c(\phi, A)\,.\,\mathbf{neg\text{-}reify}^i_c(\phi, B) \\
\mathbf{neg\text{-}reify}^i_c(\phi, \nabla d : A\,.\,B) & = & \nabla c : A\,.\,\mathbf{neg\text{-}reify}^i_c(\phi \vee d, B) \\
\mathbf{arg\text{-}reify}^i_c(\phi, \mathsf{Type}_j) & = & \mathsf{Type}_j \text{ if } j < i
\end{array}
$$

We note a few points about this definition. First, the return type for c is ensured to always use a CP that c satisfies. This is done by setting this CP to $\phi \vee c$. For

arguments, $\mathbf{reify}^i_c(\phi, \Pi x\!:\!A\,.\,B)$ uses the operation $\mathbf{arg\text{-}reify}^i_c(\phi, A)$ to reify argument types. Any inductive types of the form $\left[\psi\right] I$ in the argument types of A are reified to use the CP $\phi \wedge \psi$, specifying that ψ acts as an upper bound to the constructors in allowed in the argument. This allows the type A to specify negative constraints about constructors in arguments. Reification of Type_j simply ensures that j is no greater than the level of the whole of A. Reification of the argument type $\nabla d\!:\!A\,.\,B$ reifies B in the new CP $\phi \vee d$, indicating that the body of a ν-abstraction of this type might contain d. Function types are reified by reifying their return types and negatively reifying their input types. Negative reification is done with $\mathbf{neg\text{-}reify}^i_c(\phi, A)$, and ensures that type universes in a non-positive position are below the level of A and that no inductive types can possibly contain c.

The last typing rule to examine in Figure 7.4 is t-pmfun. This behaves similarly to the corresponding rule in CNIC. Pattern-matching functions will in general take in a sequence of constructors $\vec{c}$, a CP ξ, a sequence of arguments Γ^{x}, and a scrutinee x. The scrutinee have type $\nabla\Gamma^{\mathrm{c}}_{\nu}\,.$ for an arbitrary list of constructors and type $\Gamma^{\mathrm{c}}_{\nu}$. This is expressed with the same raising operator as in CNIC. The input CP ξ specifies the CP for the scrutinee. The scrutinee must also satisfy some CP ϕ that includes all the patterns in the pattern-matching function. This is expressed by $\Gamma \vdash (\Gamma_1, c_1), \ldots, (\Gamma_n, c_n)$ **covers** $\left[\phi \wedge \xi\right] a$.

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : \mathsf{Type}_i \quad \vdash A \lesssim B}{\Gamma \vdash M : B}\ \textsf{t-subt} \qquad \frac{}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}}\ \textsf{t-type}$$

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \quad \Gamma, x : A \vdash B : \mathsf{Type}_i}{\Gamma \vdash \Pi x{:}A\,.\,B : \mathsf{Type}_i}\ \textsf{t-pi-p} \qquad \frac{\Gamma \vdash A : \mathsf{Type}_i \quad \Gamma, x : A \vdash B : \mathsf{Type}_0}{\Gamma \vdash \Pi x{:}A\,.\,B : \mathsf{Type}_0}\ \textsf{t-pi-i}$$

$$\frac{\Gamma, \xi : \mathsf{CP} \vdash A : \mathsf{Type}_i}{\Gamma \vdash \Pi \xi{:}\mathsf{CP}\,.\,A : \mathsf{Type}_i}\ \textsf{t-pi-cp} \qquad \frac{\Gamma \vdash A : \mathsf{Type}_i \quad \Gamma \vdash \mathbf{ctype}(A) \quad \Gamma, c : A \vdash B : \mathsf{Type}_i}{\Gamma \vdash \nabla c{:}A\,.\,B : \mathsf{Type}_i}\ \textsf{t-nabla}$$

$$\frac{\Gamma \vdash A : \left[\phi\right] \mathsf{IndType}_i}{\Gamma \vdash \left[\phi\right] A : \mathsf{Type}_i}\ \textsf{t-cp} \qquad \frac{}{\Gamma \vdash \mathsf{IndType}_i : \left[\phi\right] \mathsf{IndType}_{i+1}}\ \textsf{t-indtype} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsf{t-var}$$

$$\frac{c : A \in \Gamma \quad \mathrm{level}(A) = i}{\Gamma \vdash c : \mathbf{reify}^i_c(\phi, A)}\ \textsf{t-ctor} \qquad \frac{\Gamma \vdash c : A \quad \mathbf{remove}_c(\Gamma) \vdash M : \nabla c{:}A\,.\,B}{\Gamma \vdash M\ \langle c \rangle : B}\ \textsf{t-constrepl}$$

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \quad \Gamma \vdash \mathbf{ctype}(A) \quad \Gamma, c : A \vdash M : B}{\Gamma \vdash \nu c : A.M : \nabla c{:}A\,.\,B}\ \textsf{t-nu} \qquad \frac{\Gamma \vdash M : \Pi x{:}A\,.\,B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : [N/x]B}\ \textsf{t-app}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda\, x{:}A\,.\,M : \Pi x{:}A\,.\,B}\ \textsf{t-lambda}$$

$$\frac{\begin{array}{c}\cdot \vdash \nabla \Gamma^{\mathrm{c}}\,.\,\Pi \xi{:}\mathsf{CP}\,.\,\Pi \Gamma^{\mathrm{x}}{\uparrow}^{\Gamma^{\mathrm{c}}_\nu}\,.\,\Pi x{:}\left[\phi \wedge \xi\right] (a\ \Gamma^{\mathrm{x}}){\uparrow}^{\Gamma^{\mathrm{c}}_\nu}_{\nabla}\,.\,B : \mathsf{Type}_i \\ \forall i (\Gamma \vdash \Gamma^{\mathrm{c}}, \Gamma^{\mathrm{c}}_i : c_i \Pi \Gamma^{\mathrm{x}}_{c_i}\,.\,\left[\phi \wedge \xi\right] a\ \vec{M}_i) \\ \forall i (\Gamma^{\mathrm{c}}, \Gamma^{\mathrm{c}}_i, \xi : \mathsf{CP}, u : (\nabla \Gamma^{\mathrm{c}}\,.\,\Pi \Gamma^{\mathrm{x}}{\uparrow}^{\Gamma^{\mathrm{c}}_\nu}\,.\,\Pi x{:}(a\ \Gamma^{\mathrm{x}}){\uparrow}^{\Gamma^{\mathrm{c}}_\nu}_{\nabla}\,.\,B), \Gamma^{\mathrm{x}}_i{\uparrow}^{\Gamma^{\mathrm{c}}_\nu} \\ \qquad \vdash N_i : [(M_i){\uparrow}^{\Gamma^{\mathrm{c}}_\nu}\ /\Gamma^{\mathrm{x}}\ ,\ (c_i\ \Gamma^{\mathrm{x}}_i){\uparrow}^{\Gamma^{\mathrm{c}}_\nu}\ /x]B) \\ \forall i (\vdash \mathbf{app\text{-}check}_u(\Gamma^{\mathrm{x}}_i; N_i)) \qquad \Gamma \vdash (\Gamma_1, c_1), \ldots, (\Gamma_n, c_n)\ \mathbf{covers}\ \left[\phi \wedge \xi\right] a\end{array}}{\begin{array}{l}\Gamma \vdash \mathbf{fun}\ u\ (\Gamma^{\mathrm{c}}, \xi, \Gamma^{\mathrm{x}}{\uparrow}^{\Gamma^{\mathrm{c}}_\nu})\ ((c\ \Gamma^{\mathrm{x}}_c){\uparrow}^{\Gamma^{\mathrm{c}}_\nu}\ \setminus \vec{\Gamma}_c{\uparrow}^{\Gamma^{\mathrm{c}}_\nu} \to \vec{N}) \\ \qquad : \nabla \Gamma^{\mathrm{c}}\,.\,\Pi \phi \wedge \xi{:}\mathsf{CP}\,.\,\Pi \Gamma^{\mathrm{x}}{\uparrow}^{\Gamma^{\mathrm{c}}_\nu}\,.\,\Pi x{:}(\left[\phi \wedge \xi\right] (a\ \Gamma^{\mathrm{x}}){\uparrow}^{\Gamma^{\mathrm{c}}_\nu}_{\nabla})\,.\,B\end{array}}\ \textsf{t-pmfun}$$

$$\frac{\Gamma \vdash \phi : \mathsf{CP} \quad \Gamma \vdash \psi : \mathsf{CP}}{\Gamma \vdash \phi \wedge \psi : \mathsf{CP}}\ \textsf{tcp-and} \qquad \frac{\Gamma \vdash \phi : \mathsf{CP} \quad \Gamma \vdash \psi : \mathsf{CP}}{\Gamma \vdash \phi \vee \psi : \mathsf{CP}}\ \textsf{tcp-or}$$

$$\frac{\Gamma \vdash \phi : \mathsf{CP} \quad \Gamma \vdash \psi : \mathsf{CP}}{\Gamma \vdash \phi \oplus \psi : \mathsf{CP}}\ \textsf{tcp-xor} \qquad \frac{\Gamma \vdash \phi : \mathsf{CP}}{\Gamma \vdash \neg \phi : \mathsf{CP}}\ \textsf{tcp-neg} \qquad \frac{c : A \in \Gamma}{\Gamma \vdash c : \mathsf{CP}}\ \textsf{tcp-ctor}$$

$$\frac{\Gamma' \vdash A : \mathsf{Type}_i}{\Gamma \vdash A \setminus \Gamma' : \mathsf{CP}}\ \textsf{tcp-patt} \qquad \frac{\Gamma' \vdash A : \mathsf{Type}_i}{\Gamma \vdash \Pi *.A \setminus \Gamma' : \mathsf{CP}}\ \textsf{tcp-star-patt}$$

Figure 7.4: Typing for CPTT

Chapter 8

Conclusion

This dissertation has presented a technique, called Higher-Order Encodings with Constructors or HOEC, for encoding name binding. The basic principle of HOEC is that a special-purpose construct should be added to a language to encode name binding. This special-purpose construct should intrinsically satisfy the required properties of name binding, so that name binding can be got "for free" and need not be formalized explicitly. This also coincides with standard mathematical practice, which is to assume the properties of name binding without formalizing the concepts precisely.

The construct used here is called a ν-abstraction. ν-abstractions introduce fresh, locally scoped constructors. The ν-abstractions can be used to encode name bindings, while the locally scoped constructors are used to encode names themselves. The benefit of this approach over others is that constructors can be easily examined and compared for equality. This also leads to the fact that recursion can easily extend inside ν-abstractions.

This dissertation brought HOEC to two different formalisms. First, the HOEC approach was brought to term rewriting. The resulting formalism is called Higher-Order Name-Binding Rewriting. This formalism allows rewrite systems to be defined that can encode name bindings and compute with them as data. It also allows the definition of programming languages which manipulate bindings.

The second formalism to which HOEC was applied is Intensional Constructive Type Theory, or ICTT. ICTT is a mathematical theory that is also a programming language. Thus programs can be written and properties of them proved, both in ICTT.

Adding HOEC facilities to ICTT means that data with name bindings can be represented. An important class of data with name bindings is programming languages themselves. ICTT with HOEC is a convenient theory theory for defining, implementing, and proving properties about programming languages.

8.1 Related Work

There has been much research into encoding name binding. Perhaps the most well-known paradigm is Higher-Order Abstract Syntax, or HOAS [52]. Instead of using a construct like a ν-abstraction, HOAS uses meta-language functions, where the meta-language variables bound be these functions become the encodings of names and variables. For example, in HOAS, the λ-term $\lambda x.\lambda y.x\ y$ would be encoded as

$$\mathsf{lam}\ (\mathbf{fun}\ (x : \mathsf{lam}\ (\mathbf{fun}\ (y : \mathsf{app}\ x\ y \to)) \to)).$$

HOAS achieves α-equivalence for free, as functions are generally equal modulo α-equivalence.

HOAS has the benefit that functions are already present in most languages, and thus special-purpose constructs like the ν-abstraction need not be added. There are several drawbacks, however. First, only parametric functions represent variable binding. For example, the term

$$\mathsf{lam}\ (\mathbf{fun}\ (\mathsf{lam}\ F \to F\mathsf{lam}\ F \mid \mathsf{app}\ x\ y \to x, yx))$$

does not encode any valid λ-term. Thus languages with HOAS must have a separate type for parametric functions, so that only these will be used in encodings. This is the so-called "exotic terms" problem. Second, pattern-matching over functions is notoriously complex [21, 74]. Both of these problems greatly increase the complexity of any language that wishes to support HOAS.

The are languages in the literature that do support HOAS. The first such language was Twelf [53]. Twelf is based on logic programming, however, a paradigm in which programs themselves are not objects in the language. Thus it is impossible to prove

properties directly about programs in Twelf. Other more recent attempts [60, 56, 19] define functional programming languages that operate on LF [27], a language of parametric functions. Though functional programming does allow programs as objects in the language, these languages all enforce a strict separation between programs and proofs, again disallowing proofs about programs.

Another interesting approach to reasoning about variable binding is that of Miller and Tiu [44]. This work is the originator of the ∇-type used in Heifer. The work starts from a sequent-style formulation of first-order logic and adds the quantifier $\nabla c{:}T\,.\,P$. This construct has the same meaning as the propositional reading in Heifer, that "if new constructor c of type T were added, then P would be true." Miller and Tiu also add a form of logic programming, allowing the user to define programs in the theory. As in Twelf, however, these programs are not themselves objects in the theory, so proofs cannot be directly about programs.

The final approach to encoding variable binding considered here is Nominal Logic [59, 23]. Nominal Logic, based on FM set theory (set theory with atoms), uses a set of *atoms* to encode variables. It then defines an *atom-abstraction* operation to encode variable binding. This operation is defined in such a way that the binding of x in a set using x is equal to the binding of y in the same set that uses y instead. User-defined types can then use atom-abstraction to encode variable binding, attaining α-equivalence for free.

Nominal Logic, however, does not satisfy typing. Instead, all names are in a single type of names. This is similar to the nominal calculus CNIC defined here. This makes it harder to encode typed programming languages, as extra typing information must be provided. Nominal Logic has also not been fully incorporated into Intensional Constructive Type Theory. There are nominal datatype packages for the theorem provers Isabelle [71] and Coq [5], but these require some complex machinery. In addition, these implementations are not quite complete: both require the user to define a complex induction scheme for any type that uses variable binding. Thus variable binding is not quite for free, as the user must understand and be comfortable with how the particular implementation works.

8.2 Future Work

The main direction for future work is in proving the consistency of CPTT. This should be possible by a translation to CNIC, using the names and name binding of that calculus to encode the locally bound constructors of CPTT. The main difficulty is in modeling the action of the CPs. Specifically, if the user introduces a constructor c of type $a\ A$ for some term A of type Type_0, it is impossible to determine the normal form of the CP $(a\ \mathsf{nat} \setminus \cdot) \wedge c$ *in the theory*, because this requires testing if A is equal to nat.

There are two possible solutions to this problem. One is to somehow encode not just the CPs themselves but also the reductions that occur on them in a typing derivation. Given a proof that the CP $(a\ \mathsf{nat} \setminus \cdot) \wedge c$ reduces to c, then, it is possible to see that A equals nat.

A different approach would be to restrict the free variables in constructors so that the question of CP normalization is decidable in the the theory. For example, the types of constructors could be restricted so that only free variables with an inductive type are permitted. The above problem of determining the normal form of $(a\ \mathsf{nat} \setminus \cdot) \wedge c$ then goes away, because if c is introduced with type $a\ A$, then it must be known at the time c is introduced whether A equals nat or not, as A can only have free variables of inductive type. For any given CPTT term, a *basis set* could be formed of all atomic CPs (without $\wedge$ and $\oplus$) occurring in the term. This could then be encoded as an inductive type itself in CNIC, along with functions for computing the $\wedge$ and $\oplus$ of any two elements of this type. More experience with CPTT is needed to know if this restriction would be burdensome or if most useful programs in CPTT already adhere to this restriction.

Other directions of research are opened up by the fact that unification on CPs is decidable. This is because the CPs form a boolean algebra (actually, a boolean ring), a problem which has been heavily studied. See Martin and Nipkow [39] for a summary of results. This opens up the possibility of a type inference algorithm to infer the CPs in a term. This would increase usability of CPTT, as CPs would essentially go "behind the scenes" and a programmer writing in CPTT would not have to be concerned directly with CPs at all. Another feature enabled by a decidable unification

problem on CPs is subtyping. Specifically, the type $\left[\phi\right]\ I$ could be made a subtype of $\left[\phi \vee \psi\right]\ I$ for any ψ. Deciding if $\left[\phi_1\right]\ I$ is a subtype of $\left[\phi_2\right]\ I$ thus would require searching for a solution to the equation $\phi_1 \vee X = \phi_2$, a unification problem.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] S. Abramsky, D. Gabbay, and T. Maibaum, editors. *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[3] T. Altenkirch. Integrated verification in Type Theory. Lecture notes for a course at ESSLLI 96, Prague, 1996. Available from the author's website.

[4] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark Challenge. In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, 2005.

[5] Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in Coq (extended abstract). In Alberto Momigliano and Brigitte Pientka, editors, *Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2006)*, pages 69–77. Elsevier, 2007.

[6] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[7] Franz Baader and Wayne Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 8. Elsevier and MIT Press, 2001.

[8] Frédéric Blanqui, Jean-Pierre Jouannaud, and Pierre-Yves Strub. From formal proofs to mathematical proofs: a safe, incremental way for building in first-order decision procedures. In *IFIP International Conference on Theoretical Computer Science*, 2008.

[9] C. Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[10] James Cheney. Simple nominal type theory. In *Proceedings of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.

[11] T. Coquand. An analysis of Girard's paradox. In *1st Symposium on Logic in Computer Science*, pages 227–236, 1986.

[12] Thierry Coquand and Jean Gallier. A proof of strong normalization for the theory of constructions using a kripke-like interpretation. In *Proceedings of the Workshop on Logical Frameworks*, 1990.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2 edition, 2001.

[14] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, 1996.

[15] N. Dershowitz and D.A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 9, pages 535–610. Elsevier Science, 2001.

[16] Nachum Dershowitz. Hierarchical termination. In *Proceedings of the International Workshop on Conditional and Typed Rewriting Systems*, pages 89–105, 1995.

[17] Nachum Dershowitz. Innocuous constructor-sharing combinations. In *Proceedings of the Eighth International Conference on Rewriting Techniques and Applications*, pages 202–216, 1997.

[18] Nachum Dershowitz. When are two rewrite systems more than none? In *Proceedings of the 22nd International Symposium on Mathematical Foundations of Computer Science*, pages 37–43, 1997.

[19] Kevin Donnelly and Hongwei Xi. Combining higher-order abstract syntax with first-order abstract syntax in ATS. In *Workshop on Mechanized Reasoning about Languages with Variable Binding (MERλIN'05)*, Tallinn, Estonia, September 2005.

[20] Gilles Dowek. Higher-order unification and matching. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 16. Elsevier and MIT Press, 2001.

[21] Leonidas Fegaras and Tim Sheard. Revisiting Catamorphisms Over Datatypes with Embedded Functions (or, Programs from Outer Space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 284–294. ACM Press, 1996.

[22] Maribel Fernández, Murdoch J. Gabbay, and Ian Mackie. Nominal rewriting systems. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 108–119, 2004.

[23] Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.

[24] Jürgen Giesl, editor. *16th International Conference on Rewriting Techniques*, volume 3467 of *Lecture Notes in Computer Science*, Nara, Japan, April 2005. Springer-Verlag.

[25] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[26] Bernhard Gramlich. Relating innermost, weak, uniform and modular termination of term rewriting systems. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, pages 285–296, 1992.

[27] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[28] M. Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, University of Edinburgh, 1995.

[29] M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In G. Sambin, editor, *Twenty-five years of constructive type theory*, pages 83–111. Oxford: Clarendon Press, 1998.

[30] W. Howard. The formulae-as-types notion of construction. In Seldin and Hindley [64], pages 479–490.

[31] Jieh Hsiang and Nachum Dershowitz. Rewrite methods for clausal and non-clausal theorem proving. In *Proceedings of the 10th Colloquium on Automata, Languages and Programming*, pages 331–346, London, UK, 1983. Springer-Verlag.

[32] G. Huet and G. Plotkin, editors. *Logical Frameworks*. Cambridge University Press, 1991.

[33] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[34] Jean-Pierre Jouannaud. Private correspondence, 2008.

[35] M. Kaufmann, P. Manolios, and J Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic, 2000.

[36] Max Kelly and M. L. Laplaza. Coherence for compact closed categories. *Journal of Pure and Applied Algebra*, 19:193–213, 1980.

[37] Z. Khasidashvili. Expression reduction systems. In *Proceedings of I.Vekua Institute of Applied Mathematics*, volume 36, pages 200–220, 1990.

[38] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1-2):279–308, 1993.

[39] Ursula Martin and Tobias Nipkow. Boolean unification—the story so far. *Journal of Symbolic Computation*, 7(3-4):275–293, 1989.

[40] Per Martin-Löf. An intuitionistic theory of types. In G Sambin and J Smith, editors, *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, 1998.

[41] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192, 1998.

[42] Erik Meijer and Graham Hutton. Bananas in Space: Extending fold and unfold to Exponential Types. In *Proceedings of the 7th SIGPLAN-SIGARCH-WG2.8 International Conference on Functional Programming and Computer Architecture*. ACM Press, La Jolla, California, June 1995.

[43] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[44] Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. Comput. Logic*, 6(4):749–783, 2005.

[45] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[46] Alexandre Miquel and Benjamin Werner. The not so simple proof-irrelevant model of cc. In *Proceedings of the International Workshop of Types for Proofs and Programs (TYPES '03)*, 2003.

[47] J. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[48] Tobias Nipkow. Higher-order critical pairs. In *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 342–349, 1991.

[49] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[50] Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In *TLCA '93: Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, London, UK, 1993. Springer-Verlag.

[51] Gerald E. Peterson and Mark E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, 1981.

[52] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation*, 1988.

[53] F. Pfenning and C. Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

[54] Brigitte Pientka. International workshop on logical frameworks and meta-languages: Theory and practice (LFMTP'07). `http://www.cs.mcgill.ca/~{}bpientka/lfmtp07/`.

[55] Brigitte Pientka. Functional programming with higher-order abstract syntax and explicit substitutions. In *PLPV '06: Proceedings of the First Workshop on Programming Languages meets Program Verification*, 2006.

[56] Brigitte Pientka. Functional programming with higher-order abstract syntax and explicit substitutions. In *Programming Languages meets Program Verification (PLPV '06)*, Seattle, WA, USA, August 2006.

[57] Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[58] Benjamin Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[59] A. M. Pitts. Nominal logic: A first order theory of names and binding. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001. Proceedings*, volume 2215 of *Lecture Notes in Computer Science*, pages 219–242. Springer-Verlag, Berlin, 2001.

[60] Adam Poswolsky and Carsten Schürmann. Extended report on delphin: A functional programming language with higher-order encodings and dependent types. Technical Report YALEU/DCS/TR-1375, Yale University, 2007.

[61] Albert Rubio. A fully syntactic ac-rpo. In *Information and Computation*, pages 133–147, 1999.

[62] C. Schurmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266(1-2):1–57, 2001.

[63] C. Schürmann, A. Poswolsky, and J. Sarnat. The ∇-Calculus. Functional Programming with Higher-Order Encodings. In *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications*, pages 339–353. Springer-Verlag, 2005.

[64] J. Seldin and J. Hindley, editors. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, 1980.

[65] Masaru Shirahata. A sequent calculus for compact closed categories, 1996.

[66] Rolf Socher-Ambrosius. Boolean algebra admits no convergent term rewriting system. In *Proceedings of the Fourth International Conference on Rewriting Techniques and Applications (RTA '91)*, pages 264–274, Como, Italy, 1991.

[67] TeReSe, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[68] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version V8.0*, 2004. http://coq.inria.fr.

[69] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.6.1*, 2007. `http://www.haskell.org/ghc/docs/latest/html/users_guide/`.

[70] A. Troelstra. A history of constructivism in the twentieth century. Technical Report ITLI Prepublication Series ML-91-05, University of Amsterdam, 1991.

[71] C. Urban and C. Tasson. Nominal Techniques in Isabelle/HOL. In *Proc. of the 20th International Conference on Automated Deduction (CADE)*, volume 3632 of *LNCS*, pages 38–53, 2005.

[72] Vincent van Oostrom. Counterexamples to higher-order modularity. Available at http://www.phil.uu.nl/ oostrom/publication/rewriting.html, 2005.

[73] Vincent van Oostrom and Femke van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. In *Proceedings of the First International Workshop on Higher-Order Algebra, Logic, and Term Rewriting*, pages 276–304, 1993.

[74] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: encoding higher-order abstract syntax with parametric polymorphism. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 249–262. ACM Press, September 2003.

[75] Stephanie Weirich. Second Informal ACM SIGPLAN Workshop on Mechanizing Metatheory. http://www.cis.upenn.edu/~{}sweirich/wmm/.

Vita

Edwin M. Westbrook

Date of Birth November 9, 1979

Place of Birth Chicago, IL

Degrees B.S. Computer Science, May 2001
University of California, Berkeley, Berkeley, California

Honors National Merit Scholar, 1997
University of California, Berkeley

Publications

Ian Wehrman, Aaron Stump and Edwin Westbrook. *Slothrop: Knuth-Bendix Completion with a Modern Termination Checker.* 17th International Conference on Rewriting Techniques and Applications.

Edwin Westbrook. *Pattern Solutions to Higher-Order Unification Problems.* 20th International Workshop on Unification (UNIF 06)

Edwin Westbrook. *Free Variable Types.* Seventh Symposium on Trends in Functional Programming (TFP 06)

Edwin Westbrook, Aaron Stump and Ian Wehrman. *A Language-based Approach to Functionally Correct Imperative Programming.* 10th ACM SIGPLAN International Conference on Functional Programming, pp. 268-279.

Technical Reports

Edwin Westbrook, Aaron Stump and Ian Wehrman. *A Language-based Approach to Functionally Correct Imperative Programming.* WUCSE-2005-32. July, 2005.

Professional Activities

Reviewer for Rewriting Techniques and Applications (RTA '08), 2008

Reviewer for Transactions on Programming Languages and Systems (TOPLAS), 2007

Reviewer for Rewriting Techniques and Applications (RTA '07), 2007

Reviewer for Programming Languages Meets Program Verification (PLPV '06), 2006

Reviewer for Principles of Programming Languages (POPL '06), 2006

Departmental Service

Doctoral Student Seminar co-coordinator, 2007 – 2008

Hot Topics Seminar coordinator, 2007 – 2008

Work Experience

Engineer/Scientist II, 2001 – 2003
Apple Computer, Cupertino, California

December 2008

Higher-Order Encodings with Constructors, Westbrook, Ph.D. 2008