Types and Verification for Infinite State Systems

Gavin Mendel-Gleason

Bachelor of Science in Physics and Pure Mathematics

A Dissertation submitted in fulfilment of the

requirements for the award of

Doctor of Philosophy (Ph.D.)

to the

Dublin City University

Faculty of Engineering and Computing, School of Computing

Supervisor: Geoff Hamilton

July 19, 2012

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: ______________________________

Student ID: ______________________________

Date: ______________________________

Contents

Abstract

Server-like or non-terminating programs are central to modern computing. It is a common requirement for these programs that they always be available to produce a behaviour. One method of showing such availability is by endowing a type-theory with constraints that demonstrate that a program will always produce some behaviour or halt. Such a constraint is often called *productivity*. We inroduce a type theory which can be used to type-check a polymorphic functional programming language similar to a fragment of the Haskell programming language. This allows placing constraints on program terms such that they will not type-check unless they are productive. We show that using program transformation techniques, one can restructure some programs which are not provably productive in our type theory into programs which are manifestly productive. This allows greater programmer flexibility in the specification of such programs. We have demonstrated a mechanisation of some of these important results in the proof-assistant Coq. We have also written a program transformation system for this term-language in the programming language Haskell.

Acknowledgments

I would like to thank my wife, Deirdre, for her long suffering patience. Without her contribution this thesis wold not exist. I also would like to thank Geoff Hamilton for his helpful suggestions and encouragement. I would like to thank James for assistance with some of the mathematical details and Paul Bowman for helping to make the mathematics more accessible. I would like to thank Rose Murphy for reading through the thesis despite not understanding it. I would also like to thank my examiners, specifically the helpful and detailed suggestions for revisions given by Conor McBride and the assistance of David Gray in making the section on transition systems readable.

List of Figures

Chapter 1

Introduction

Software plays an ever larger role in our lives. We see its impact in everything from automobiles to films. It is likely that this trend of growing importance will not slow any time soon. At the same time, demands on the quality and safety of programs have also increased, while methods of dealing with quality have not kept pace. Formal methods have not always been able to keep abreast of these increases in software complexity. If they are to be applicable, formal methods must be able to address real needs in software in ways that are usable by practitioners. Such a programme to improve the applicability of formal methods is necessarily ambitious. The present work aims to make a contribution to the programme by increasing the applicability of automatic methods of verification to a subset of the general problem.

Algorithms can be understood as an abstract description of a terminating process. When we say that an algorithm has been developed to compute some quantity, we mean that we have a finite procedure which is capable of producing an output *value* in finite time for any given inputs (where *value* is suitably defined).

Algorithms, however, do not subsume all possible programs of interest. It was realised fairly early on in the development of computers that certain programs, such as operating systems, should never terminate by design. In order to deal with this class of intentionally non-terminating programs we need a theory of another class of programs entirely.

The class of programs which are potentially non-terminating is sometimes called *reactive* systems [3]. This is an alternative view of programs and their behaviour, which seeks to provide constraints or proofs about behaviour of programs that we see in the real world, such as servers, which do not necessarily terminate.

When we say a program is *non-terminating* we may mean more than one thing. A server, such as the Apache web server, could be non-terminating in the sense that it repeatedly polls for new connections to serve and once requests are received, serves them. This is an example of a *productive* program. That is, we hope that it always produces some new behaviour when we provide it with input.

This idea of productivity is in contrast to a non-terminating program which potentially does nothing for an arbitrary period. If we view programs from the standpoint of reactive systems, a non-productive system is quite similar to a system which halts since it no longer has behaviour, a fact which is reflected in certain theories of reactive systems based on traces such as CCS (Calculus of Communicating Systems) [57]. The distinction between the two different types of non-termination (non-productive, and productive) will be central to our presentation.

As a formal method for demonstrating certain types of correctness, this thesis makes use of *types*. We view types, which will be described in more detail in the next section, as representing the specifications for the programs we write down. Types are a good way to demonstrate properties since they make their claims about correctness based on a *proof* which is a systematic way of describing how evidence should be laid out in order that we should believe it. This evidence based approach allows a *type checker*, that is, an algorithm which automatically checks if we should believe that the evidence provided does in fact demonstrate the property of interest. The method by which the proof is initially created can then be a quite free exercise which is performed by a theorem prover or even constructed by hand.

In the case of programs with potentially infinite behaviour it turns out that it can be a sticky problem how we might go about writing down the evidence that our program does what some specification claims. To represent infinite behaviour in a finite way, we naturally have to make use of some sort of circularity. Demonstrating circular arguments in a way that is *correct* will turn out to be central to this thesis.

We assume that the reader has some familiarity with functional programming so that we can present more concretely how our approach will work. First, we start with a simple program, written in the programming language Haskell [40].

$$\begin{aligned}
&\mathit{map} :: (a \to b) \to [a] \to [b] \\
&\mathit{map}\ f\ [\,] = [\,] \\
&\mathit{map}\ f\ (x : \mathit{xs}) = (f\ x) : (\mathit{map}\ f\ \mathit{xs}) \\
&\mathit{nats} :: [\mathit{Int}] \\
&\mathit{nats} = 0 : (\mathit{map}\ (1+)\ \mathit{nats})
\end{aligned}$$

The program $nats$ is well known to functional programmers. It computes an infinite list of numbers starting from 0. This program always computes a further value for any program which might consume an arbitrary portion as input. It is because of this behaviour that we call it a *productive* program. While a functional programmer can easily look at this program and deduce that it is productive, it turns out to be problematic when we wish to write down a formal proof that this is so. We shall sometimes call such productive computations, *coterminating*.

The type checker for Haskell will allow this program to pass as type-correct with no difficulty. However, Haskell will also allow the following program to type check.

$$loop :: a$$
$$loop = loop$$

This essentially states that the program $loop$ is of some arbitrary type a. In fact, we could have put any more specific type there, say $a \to b$ and loop would still pass our type checker. This may not be the worst thing if we are simply attempting to demonstrate that our programs won't produce bogus output. On the other hand, if we are interested in interpreting the type as some sort of specification of performed behaviour, then it is really quite a big problem. Programs may claim to have a type and do nothing, or claim to have a type and output only a portion of the claimed type and then hang. There are many conditions in which this would not be a suitable specification. Clearly a specification for a server which allows spinning off in non-productive computation for eternity is a fairly weak specification.

In the interest of demonstrating certain behaviours such as termination and productivity hold of our programs, our type system needs to be more careful than the one which is used by Haskell. A theorem prover which used something analogous to the Haskell approach would allow any theorem to be claimed as correct and this is clearly not a very useful theorem prover.

In theorem provers which use types to express theorems, such as Coq [8] and Agda [63] we generally introduce some limited way of demonstrating termination and co-termination behaviour. These (co)termination constraints ensure that our types are not plagued with non-terminating and non-productive computations. Termination is generally demonstrated by showing that some finite argument is always getting smaller. An example of such a program in Haskell might be as follows.

$$\mathbf{data}\ Nat = Z \mid S\ Nat$$
$$plus :: Nat \to Nat \to Nat$$
$$plus\ Z\ y = y$$

$plus\ (S\ x)\ y = S\ (plus\ x\ y)$

The function $plus$ in this program always calls itself on a structurally smaller argument. Provided that we insist that the construction of elements in Nat are only ever constructed themselves by type correct and terminating programs we can be confident that $plus$ terminates. It does what it says on the tin; when we get two Nats, we produce another Nat without fail.

There is a very similar program to $plus$ which we might call $coplus$. The $plus$ program above actually demonstrates productivity, a fact which was not important to us when we were worried that it should terminate. However, if we would like an addition which can cope with numbers which are potentially formed from an infinite number of S constructors we must show that they are productive.

$\mathbf{data}\ CoNat = CZ \mid CS\ CoNat$

$coplus :: CoNat \rightarrow CoNat \rightarrow CoNat$

$coplus\ CZ\ y = y$

$coplus\ (CS\ x)\ y = CS\ (coplus\ x\ y)$

This program is essentially identical to $plus$ aside from calling the type $CoNat$, which relates that we are interested in potentially infinite data. To see that this program is productive, we can note that there are no recursive calls which do not *emit* a constructor CS. This means that any *consumer* of the behaviour of *coplus* is guaranteed to get what it is looking for without yielding non-productive computations (provided that consumer is also a terminating or productive program). This guarantee that we will emit a constructor by checking that it occurs just prior to our recursive calls is generally known as a *guardedness* condition.

The method of demonstrating that programs terminate by using structural recursion, as in the $plus$ example, is quite attractive for a number of reasons. First, it is often quite natural to write programs which are structurally recursive. When it is not natural, there are often methods of introducing our reasoning about why the function terminates as arguments to the function. For instance, we can use some auxiliary relation which demonstrates the decreasing nature of the computation by providing a structure over which we can recurse. Finally, they demonstrate a kind of compositionality, which allows us to reason piece-wise about whether things terminate and compose them into terminating programs.

Things are a bit more complicated when attempting to show that programs are productive. Productivity using a *guard*, a constructor which demonstrates we will definitely have some behaviour prior to recursive calls, is often not a very natural way to write software. In addition it is not so clear how to get around this fact by providing auxiliary code (though Danielsson does demonstrate one approach [25]). The upshot is that there is a lot of room for improvement on proving *productivity*.

A concrete example of the problem can be seen with the function $sumlen$. This function takes a potentially infinite list, $CoList$, of potentially infinite numbers, $CoNat$, and sums them together with the length of the list. We call this operation $sumlen$.

$\mathbf{data}\ Colist\ a = CNil \mid CCons\ a\ (Colist\ a)$

$sumlen :: Colist\ CoNat \to CoNat$

$sumlen\ CNil = CZ$

$sumlen\ (CCons\ x\ xs) = CS\ (coplus\ x\ (sumlen\ xs))$

This program is definitely productive, however it fails to meet a guardedness condition as the recursive call to sumlen doesn't have a constructor immediately surrounding it. Neither can we simply move the CS constructor inside the coplus to avoid the problem.

$sumlen\_inner :: Colist\ CoNat \to CoNat$

$sumlen\_inner\ CNil = CZ$

$sumlen\_inner\ (CCons\ x\ xs) = coplus\ x\ (CS\ (sumlen\_inner\ xs))$

This definition also fails to be guarded because the call to $coplus$ may do something untoward with the CS. Of course we know that it does not, but how can we justify this? We can do so by inspecting $coplus$ itself and how it will treat its argument. Essentially we would like to make this program into a similar program which is itself guarded by careful *transformation* of the program into the following.

$sumlen' :: Colist\ CoNat \to CoNat$

$sumlen'\ CNil = CZ$

$sumlen'\ (CCons\ x\ xs) = CS\ (aux\ x\ xs)$

$aux\ Z\ xs = sumlen'\ xs$

$aux\ (CS\ x)\ xs = CS\ (aux\ x\ xs)$

This program now meets a guardedness condition as each call exhibits a constructor around each recursive call and hence is in a suitable form to be entered into Coq or Agda. Not only is it possible to transform the above program $sumlen$ into $sumlen'$ systematically we can also do so *automatically* using the supercompilation program transformation procedure. This suggests that we might attempt to type a larger class of programs automatically by using program transformation as a tool with which to widen the net of programs for which we can provide types.

Unfortunately by using program transformation we have created for ourselves another problem. If we would like to use type theory to generate proofs which are machine checkable we have now lost information about how the transformation took place. It is folklore that program transformation is hard to do properly while retaining the meaning of programs. How then can we trust that the program obtained after transformation is equivalent to the original?

The answer that we give in this thesis is that we should provide a *bisimulation* between the original term and the final result term. This *bisimulation* will give evidence which can be automatically verified, linking the original term to the transformed term. Because *bisimulation* preserves termination properties as well as other behaviours, it allows us to be confident that type checking the resultant term gives us information about the original term.

We now take a brief excursion into the type theory which will be required to understand this thesis, followed by an overview of the structure of the thesis.

1.1 Type Theory

There are many frameworks for describing properties of program terms. However, most techniques can be divided into two basic approaches. One approach is model-checking, where we make a model of some program and then systematically explore the available behaviours to ensure that some property holds. The method of exploring behaviours can vary widely, including state space enumeration, abstract interpretation [22] and the use of various temporal logics [94].

One other broad class of approaches is to use *type theory*. Type theory approaches the problem of describing properties of programs in a structural and syntactic way in contrast to the much more semantic model-checking approaches. No model is required as we describe proofs of our properties concretely in terms of the actual program syntax. This approach of assigning properties has a number of advantages in ease of implementation, and a tight coupling between the language and the properties we wish to prove. The approach makes use of the *Curry Howard Correspondence* to provide rich type theories such as those employed in theorem provers such as Agda [63], Coq [8] or Isabelle [62] or more modest type systems such as those employed by general purpose functional programming languages such as ML and Haskell.

In type theory as applied to programs we view the *type* of the program terms as representing their specification. The specification can then be said to correspond to a program when the program *type-checks*, that is, the program can be shown by an algorithm to be of the type supplied. Depending on how rich our type theory is, this provides us with a more or less general way of talking about the correctness of software. With very rich type theories, we should be able to capture a great number of properties of interest.

This work marries the use of types with the technique of exploring behaviours. We use type theory to ensure that the final answers are correct according to their specification, given as a type, using a relatively simple type-checking algorithm. We search the program space to look for programs which are equivalent to the original program by construction, but which additionally meet the syntactic conditions provided by our type theory. We avoid the problem of the correspondence of our model to the program by using semantic preserving transformations of the original program.

1.1.1 λ-calculus

The λ-calculus is a simple proto-logic which has served as an important basis for theoretical computer science. It is at once extremely simple, yet powerful. The language, together with its evaluation relation, is Turing complete, meaning that it provides access to the full range of computational power of a Turing machine. The Turing machine serves as an important yardstick of universality in computation [88].

The language as given in Figure 1.1 shows just how simple the rules are. The syntax is comprised of variables, variable abstraction and term application. The reduction system is comprised of little more than the replacement of variables with terms. The evaluation relation (which we write as $\rightsquigarrow$) couples our syntax with a notion of computation built from substitution. The term on the right hand side of the $\rightsquigarrow$ relation is a β*-reduct* and the process is known as β-reduction.

The convention of renaming variables is known as α-conversion. α-conversion leads to an important notion of equivalence. Two terms are said to be equivalent modulo α-conversion if we can find a one-to-one renaming of variables (being careful that we keep unlike variables distinct in our renaming) so that two terms are syntactically identical after the renaming. This α-conversion step can be thought of as representing the fact that the *name* of a formal parameter of a function is not consequential to the meaning. For instance, a function $f(x) = 1+x$ is functionally identical to one in which x is replaced with y, i.e. $f(y) = 1 + y$. We have opted here to use explicitly named

Syntax

$$\begin{array}{llll} \mathbf{Var} \ni x, y, z & & & \text{Variables} \\ \mathbf{Term} \ni r, s, t & ::= & x \mid \lambda x.\ r \mid r\ s & \text{Terms} \end{array}$$

Free Variables

$$\begin{array}{lcl} FV(x) & \equiv & \{x\} \\ FV(\lambda x.\ r) & \equiv & FV(r) \setminus \{x\} \\ FV(t\ s) & \equiv & FV(t) \cup FV(s) \end{array}$$

Substitution

$$\begin{array}{lcl} x[x := t] & \equiv & t \\ x[y := t] & \equiv & x \text{ if } x \neq y \\ (r\ s)[x := t] & \equiv & (r[x := t])\ (s[x := t]) \\ (\lambda y.\ r)[x := t] & \equiv & \lambda y.\ r[x := t] \\ & & \text{Provided that } (\lambda y.\ r) \text{ is } \alpha\text{-converted to use} \\ & & \text{a fresh variable if } y \in \{x\} \cup FV(t). \end{array}$$

Reduction

$$(\lambda x.\ s)\ t \rightsquigarrow s[x := t] \qquad \frac{r \rightsquigarrow r'}{\lambda x.\ r \rightsquigarrow \lambda x.\ r'}$$

$$\frac{s \rightsquigarrow s'}{s\ t \rightsquigarrow s'\ t} \qquad \frac{t \rightsquigarrow t'}{s\ t \rightsquigarrow s\ t'}$$

Figure 1.1: λ-Calculus

variables, as even though it complicates the technical machinery somewhat, it makes examples much easier to read.

There are some complications which arise due to variable naming and which lead to the testing of equality of variable names in our substitution rules. We need to avoid inadvertent *capture* of variables when performing substitutions. Avoiding capture is achieved with a systematic renaming of variables with a guaranteed fresh variable. Capture avoidance requires knowing the free variables of a term. This is done using the FV function which is defined in Figure 1.1. This function maps terms to sets of variables. The base case where FV is called on a variable, returns a singleton set (a set with one member) with the variable. When called with an application, it simply forms the union of the free variables of either term. When called on a λ abstraction it subtracts the bound variable from the free variables of the subterm.

This technical complication of dealing with renaming can be eliminated with the use of *de Bruijn indices* [26]. This involves the use of natural numbers to indicate which λ a given variable is associated with. In fact, this approach is taken in the implementation and mechanisation of the theory and we see this in more detail in Chapter 7.

The substitution function substitutes variables which are free in a term, with a new term. We can think of it as allowing a formal parameter to be replaced with a concrete representative. As we see in Figure 1.1, the function is writen as $r[x := t]$ and has the term r in which we would like to perform substitution on the left, the variable we wish to replace is x and the new term is t. We substitute whenever we encounter a variable of the same name, we distribute across application and we substitute in a λ abstraction only if it does not inadvertently refer to a bound variable. Again, this last problem is avoided by renaming bound variables appropriately.

When we read the rules for reduction we see there is a horizontal bar. We can read this as stating that if what is above the bar is derivable, then we can derive the statement below the bar. For example, supposing we have demonstrated that $r \rightsquigarrow r'$. From this it follows that $\lambda x.\ r \rightsquigarrow \lambda x.\ r'$. We might also view this operationally as stating that evaluation is allowed under a λ abstraction.

1. $(\lambda x.\ x\ y) =_\alpha (\lambda z.\ z\ y)$
2. $(\lambda x.\ x)\ y \rightsquigarrow x[x := y] = y$
3. $(\lambda y.\ (\lambda x.\ y))\ x \rightsquigarrow (\lambda x.\ y)[y := x] =_\alpha (\lambda z.\ y)[y := x]$
   $= (\lambda z. y[y := x]) = \lambda z. x$
4. $(\lambda x.\ (\lambda x.\ x))\ y \rightsquigarrow (\lambda x.\ x)[x := y] =_\alpha (\lambda z.\ z)[x := y]$
   $= (\lambda z.\ z[x := y]) = \lambda z.\ z$

Figure 1.2: Example Reductions

In Figure 1.2 we see some examples of various reductions and equivalences in the λ-calculus. Example 1 shows an α-renaming which renames every variable x with the variable z in the local scope of the λ-binder. In Example 2 we see a very simple reduction which involves the application of a λ to a term y. Example 3 demonstrates a reduction which requires renaming in order to avoid capture of a free variable within a local variable binding. Example 4 demonstrates a converse situation where a reduction requires renaming to avoid confusion between a local variable and one which is now free.

One of the important properties of the λ-calculus is the *Church Rosser* property [17]. Informally, this property allows us to ignore the order of reductions, and to know that various reduction paths will arrive at the same term.

Definition 1.1.1 (Reflexive Closure)**.** The reflexive closure R^r of a relation $R\colon S \times S$ is the extension of the relation R to include all pairs $(x, x) \in S \times S$. That is $R^r ::= R \cup \{(x, x) \in S \times S\}$

Definition 1.1.2 (Transitive Closure)**.** The transitive closure R^+ of a relation $R\colon S \times S$ is the

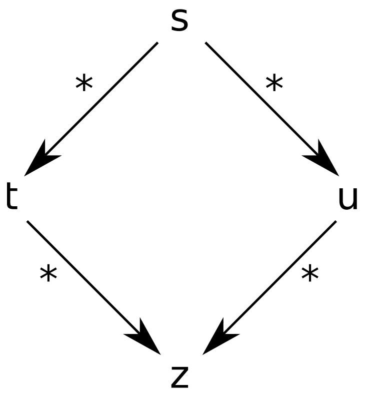
$$\begin{tikzcd} & s \arrow[dl, "*"'] \arrow[dr, "*"] & \\ t \arrow[dr, "*"'] & & u \arrow[dl, "*"] \\ & z & \end{tikzcd}$$

Figure 1.3: Church Rosser Property

extension of the relation R to include all pairs $(x, y) \in S \times S$ for all x, y such that there exists a z with $(x, z) \in R$ and $(z, y) \in R^+$. That is
$R^+ ::= \{(x, y)|(x, y) \in R \vee \exists z.(x, z) \in R \wedge (z, y) \in R^+\}$.

Definition 1.1.3 (Reflexive-Transitive Closure)**.** The reflexive and transitive closure R^* of a relation $R\colon S \times S$ is the extension of the relation R equal to $(R^+)^r$

Using the reflexive-transitive closure $\rightsquigarrow^*$ we can define the Church-Rosser property.

Theorem 1.1.4 (Church-Rosser Property)**.** *If $s \rightsquigarrow^* t$ and $s \rightsquigarrow^* u$ then there is a z such that $t \rightsquigarrow^* z$ and $u \rightsquigarrow^* z$.*

A pictorial representation of the Church Rosser property is given in Figure 1.3. The starred edges in this graph represent elements related by the $\rightsquigarrow^*$ relation. We see that though we can not be sure that t and u in this figure are related to each other, we know that they are related by $\rightsquigarrow^*$ to some third term z.

The λ-calculus without restriction exhibits terms which do not reduce to a value, under our evaluation relation. The term $(\lambda x.x\ x)(\lambda x.x\ x)$, sometimes called ω serves as an example. It turns out that our evaluation relation is not *well-founded*. This means that computations via the evaluation relation will potentially continue indefinitely.

Definition 1.1.5 (Well-Founded Relation)**.** A well-founded relation R on a domain $S \times S$ is one in which every non-empty set $C \subseteq S$ of the domain has a minimum element with respect to R. A minimum element s is an element for which $s\ R\ t$ does not hold for any t.

Definition 1.1.6 (Evaluation Sequence)**.** An Evaluation sequence is a sequence of terms $t_0\ t_1\ t_2\ \cdots$ such that each $t_i \rightsquigarrow t_{i+1}$.

Definition 1.1.7 (Strong Normalisation)**.** Strong normalisation for a language L is the property that for every term $r \in L$ there exists a term $s \in L$ such that $r \rightsquigarrow^* s$ and there is no term $t \in L$ with $s \rightsquigarrow t$. The subset of terms which do not reduce, are called *values*.

Type theory for the λ-calculus was originally developed by Church. He developed it as a means to ensure a minimum element for evaluation sequences[16]. The strong normalisation property, which can be proved for the simply typed λ-calculus [83], ensures that all evaluation sequences are finite, leading eventually to a minimum element.

The existence of such a minimum element for evaluation sequences, together with the Church-Rosser property, allows us to see these minimum elements as canonical representatives of the computation, a form which we will call a normal form. This representative gives us a concrete syntactic form which we can use to settle questions of equivalence. That is, two terms will be the same if their normal form is equivalent (modulo α-conversion). This form of equivalence, induced by the evaluation relation is known as α, β-equivalence.

Definition 1.1.8 (α-equivalence)**.** Given a term t, we take α-equivalence to be the least relation closed under the rule:

$$\frac{x \neq y \qquad y \notin FV(t)}{(\lambda x.\ t) =_\alpha (\lambda y.\ t[x := y])}$$

Definition 1.1.9 (β-equivalence)**.** Given terms t and s, we take β-equivalence to be the least relation closed under the rule:

$$\frac{}{((\lambda x.\ t)\ s) =_\beta t[x := s]}$$

It is also common to introduce a notion of *η-equivalence*, which corresponds with equivalence up to η-conversion and captures a notion of extensionality. That is, two functions which would arive at the same value are identified if one is simply the application of the same term to the formal parameter of its λ-abstraction. This equivalence identifies a larger number of proofs. While it is not necessary for our purposes we also give its definition below:

Definition 1.1.10 (η-equivalence)**.** Given a term t we say take η-equivalence to be the least relation closed under the rule:

$$\frac{x \notin FV(t)}{(\lambda x.\ t\ x) =_\eta t}$$

The concept of equivalence we work with is very much like the notion of equivalence used in the equation $3 = 1+2$. We have some understanding of a process by which $1+2$ can be converted to a canonical representative, namely 3, and the equation can then be shown to hold by the reflexive property of equality. This brings the notion of equivalence and computation (reduction) into a common framework.

1.1.2 Simple Theory of Types

In order to enrich the λ-calculus with a simple theory of types, we need to extend our exposition above with a simple type system, qualifying λ-abstraction with types, and building up terms in such a way that the term structure mirrors the type structure.

Types

$$\begin{array}{lll} \mathbf{TyBase} \ni B & & \text{Base Types} \\ \mathbf{Ty} \ni R, S, T & ::= & B \mid T \to S \end{array}$$

Terms

$$\begin{array}{llll} \mathbf{Term} \ni r, s, t & ::= & x \mid \lambda x\!:\!T.\ r \mid s\ t & \text{Terms} \\ \Gamma & ::= & \cdot \mid \Gamma \cup \{x\!:\!T\} & \end{array}$$

Context Formation

$$\frac{}{\cdot\ \texttt{Ctx}} \qquad \frac{\Gamma\ \texttt{Ctx} \quad x \notin dom(\Gamma)}{\Gamma \cup \{x\!:\!T\}\ \texttt{Ctx}}$$

Typing Rules

$$\frac{\Gamma \cup \{x\!:\!T\} \vdash s : S}{\Gamma \vdash (\lambda x\!:\!T.\ s) : T \to S}\, I^{\to} \qquad \frac{\Gamma \cup \{x\!:\!T\}\ \texttt{Ctx}}{\Gamma \cup \{x\!:\!T\} \vdash x : T}\, I^{Var}$$

$$\frac{\Gamma \vdash r : S \to T \quad \Gamma \vdash s : S}{\Gamma \vdash (r\ s) : T}\, E^{\to}$$

Figure 1.4: Simply Typed λ-Calculus

Curry realised that this type-system, given in Figure 1.4 represented a simple proof system [23]. In fact it is the propositional fragment of minimal logic.

Terms are either variables (x), abstractions ($\lambda x\!:\!T.\ r$) or applications ($r\ s$). We see that we have altered λ-expressions to contain the type of the variable over which they close.

We additionally have to add contexts of free variables, in order that we can do book-keeping on what type variables are when we encounter them. These variable contexts, represented by Γ can either be empty ($\cdot$) or extended with a variable of a given type ($\Gamma \cup \{x\!:\!T\}$). We use the notation $dom(\Gamma)$, to denote the collection of variables in Γ. We test inclusion in the context Γ using the relation ($\in$), such that when x has type T in context Γ, we write ($x\!:\!T \in \Gamma$). The formation of contexts is constrained using the context formation rules and a well formed context will be written ($\Gamma\ \ \texttt{Ctx}$). The context formation rules disallow formation of contexts with duplicate variables having different typings. This allows us to assume that any context dealt with in a proof is well formed and does not contain duplicate variables with different type assignments.

The $E^{\rightarrow}$ rule may be familiar as the *modus ponens* of logic, and represents the replacement of a hypothesis with a concrete representative. In functional programming languages, it is often called function application, and involves the replacement of a parameter with a term in the body of the function.

The $I^{\rightarrow}$ discharges a hypothesis, giving us a proof of a type which is parametric and dependent on some other proof yet to be supplied, but which must be of a given type. The I^{Var} rule states that given that some hypothesis is of a given type in our context, we may simply assume it.

The evaluation relation for the simply typed λ-calculus remains unchanged from those given in Figure 1.1. We can simply ignore the type annotations for the purpose of computation. However, restricting ourselves to well-typed terms (those terms which have a correct type derivation using the given formation rules, $E^{\rightarrow}$, $I^{\rightarrow}$ and I^{Var}) we have restricted terms in such a way that they will always normalise to a value. We can thereby decide equality between terms using this principle and have an assurance that all well-typed programs terminate.

The expression $\Gamma \vdash t : S$ is sometimes called a *sequent* and can be interpreted to mean that t has type S in the presence of the typed free variables in Γ. The proofs of our types are given by showing a *consequent* (that is, the sequent below the line) which can be derived from some number of *antecedents* (those sequents which occur above the line). A simple example using the ImpElim rule is as follows:

$$\dfrac{\dfrac{\{x\!:\!S \rightarrow T,\ y\!:\!S\}\ \mathtt{Ctx}}{\{x\!:\!S \rightarrow T,\ y\!:\!S\} \vdash x : S \rightarrow T}\, I^{Var} \qquad \dfrac{\{x\!:\!S \rightarrow T,\ y\!:\!S\}\ \mathtt{Ctx}}{\{x\!:\!S \rightarrow T,\ y\!:\!S\} \vdash y : S}\, I^{Var}}{\{x\!:\!S \rightarrow T,\ y\!:\!S\} \vdash x\ y : T}\, E^{\rightarrow}$$

Here we see a term $x\ y$ has type T using the ImpElim rule, together with two antecedents, one of which requires a derivation that x has type $S \rightarrow T$ the other that y has type S. These further derivations are each a consequent of the I^{Var} rule which requires no antecedents, but requires a validly formed context containing the variable. This results in a complete proof.

1.1.3 System F

The simple theory of types for the λ-calculus is, however, extremely restrictive. We must deal only with concrete types, restricting ourselves to the propositional fragment of minimal logic.

A classic example of the limitation of such a system can be seen in the identity function $(\lambda x : T.x)$. We would require a new identity function at every type T, despite the fact that structurally, they all have essentially the same form.

Girard[29] and Reynolds[72] developed a system known as System F which significantly enriches our type theory. This system allows quantification over types as well as terms. This extends our parametrisation of hypotheses to include hypotheses about the types themselves, rather than just about terms.

Because we have introduced variables which stand in for types, it is convenient to add an additional context with type variables, Δ In addition, since types will potentially contain variables themselves, it is useful to constrain the formation of types, in a manner analogous to the formation rules for terms. The extension to the λ-calculus necessary to include these new features is presented in Figure 1.5.

Since we now have type variables in addition to term variables, we need to be careful that types themselves are well formed. We do this by including a separate type variable context Δ, constrained by formation rules to avoid duplicate variables appearing in the context. If a context Δ is well formed (according to the formation rules) we are justified in assigning a tag ($\Delta\ \ \texttt{\textbf{TyCtx}}$). This ensures that if ($\Delta \cup \{X\}\ \ \texttt{\textbf{TyCtx}}$) then ($X \notin \Delta$).

Types will now also have to respect type variable contexts. Type formation rules are given which assign a tag **`type`** to a type T which has all free variables referring to variables in a context Δ when ($\Delta \vdash T\ \ \texttt{\textbf{type}}$).

In addition we introduce two new terms, ($\Lambda X.\ t$) which represents a term with an abstract type (X) which can be made concrete by application of the second newly introduced term ($t[T]$) which denotes type application. The type of such an abstraction is given by ($\forall X.\ T$). We can see how these two new terms interact by looking at the extended substitution rules.

We re-use the former formation rules, substitution functions and evaluation relations and extend them to deal with the newly introduced forms. We show the extensions required to obtain free-variables and substitution in Figure 1.6. Both substitution and the free type variable function FV^{ty} are overloaded to act on both terms and types.

This new substitution must work with both types and terms, as terms may contain references to type variables. Explicit use of the substitution of types over terms is given in the reduction of type applications using the $\rightsquigarrow$ relation. We also introduce two new typing rules (AllIntro and AllElim) which allow us to reason about types in abstract. These can be read in analogy to the ImpIntro and ImpElim rules of the simply typed λ-calculus.

In terms of establishing our connection between computation and logic, this system corre-

Types

$$\begin{array}{lll} \mathbf{TyVar} \ni X, Y, Z & & \text{Type Variables} \\ \mathbf{Ty} \ni R, S, T & ::= & X \mid R \to S \mid \forall X.\, S \end{array}$$

Terms

$$\mathbf{Term} \ni r, s, t \quad ::= \quad x \mid \Lambda X.\, t \mid \lambda x{:}T.\, r \mid s\; t \mid s\; T \quad \text{Terms}$$

Contexts

$$\Delta \quad ::= \quad \cdot \mid \Delta \cup \{X\} \qquad\qquad \Gamma \quad ::= \quad \cdot \mid \Gamma \cup \{x{:}T\}$$

Context Formation

$$\frac{}{\cdot\ \texttt{TyCtx}} \qquad \frac{\Delta\ \texttt{TyCtx} \quad X \notin \Delta}{\Delta \cup \{X\}\ \texttt{TyCtx}}$$

$$\frac{}{\Delta \vdash \cdot\ \texttt{Ctx}} \qquad \frac{\Delta \vdash \Gamma\ \texttt{Ctx} \quad x \notin dom(\Gamma) \quad \Delta \vdash T\ \texttt{type}}{\Delta \vdash \Gamma \cup \{x{:}T\}\ \texttt{Ctx}}$$

Type Formation

$$\frac{\Delta \vdash R\ \texttt{type} \quad \Delta \vdash S\ \texttt{type}}{\Delta \vdash R \to S\ \texttt{type}} \qquad \frac{\Delta \cup \{X\} \vdash R\ \texttt{type}}{\Delta \vdash \forall X.\, R\ \texttt{type}}$$

$$\frac{\Delta \cup \{X\}\ \texttt{TyCtx}}{\Delta \cup \{X\} \vdash X\ \texttt{type}}$$

Evaluation

$$(\Lambda X.\, r)[T] \rightsquigarrow r[X := T] \qquad \frac{s \rightsquigarrow t}{s[T] \rightsquigarrow t[T]}$$

Typing Rules

$$\frac{\Delta \cup \{X\}; \Gamma \vdash r : S}{\Delta; \Gamma \vdash \Lambda X.\, r : \forall X.\, S}\, I^{\forall} \qquad \frac{\Delta; \Gamma \vdash s : \forall X.\, S}{\Delta; \Gamma \vdash s[T] : S[X := T]}\, E^{\forall}$$

Figure 1.5: System F

Free Type Variables

$$
\begin{array}{lcl}
FV^{ty}(X) & \equiv & \{X\} \\
FV^{ty}(R \to S) & \equiv & FV^{ty}(R) \cup FV^{ty}(S) \\
FV^{ty}(\forall X.R) & \equiv & FV^{ty}(R) \setminus \{X\} \\
FV^{ty}(x) & \equiv & \emptyset \\
FV^{ty}(\mathtt{f}) & \equiv & \emptyset \\
FV^{ty}(\lambda x\!:\!T.\ r) & \equiv & FV^{ty}(T) \cup FV^{ty}(r) \\
FV^{ty}(\Lambda X.\ r) & \equiv & FV^{ty}(r) \setminus \{X\} \\
FV^{ty}(r\ s) & \equiv & FV^{ty}(r) \cup FV^{ty}(s) \\
FV^{ty}(r[S]) & \equiv & FV^{ty}(r) \cup FV^{ty}(S)
\end{array}
$$

Substitution

$$
\begin{array}{lcl}
X[X := T] & \equiv & T \\
X[Y := T] & \equiv & X \text{ if } X \neq Y \\
(R \to S)[X := T] & \equiv & R[X := T] \to S[X := T] \\
(\forall Y.\ R)[X := T] & \equiv & \forall Y.\ R[X := T] \\
 & & \text{Provided that } (\forall Y.R) \text{ is } \alpha\text{-converted to use} \\
 & & \text{a fresh type-variable if } Y \in \{X\} \cup FV^{ty}(R). \\
x[X := T] & \equiv & x \\
(\lambda x\!:\!S.\ r)[X := T] & \equiv & \lambda x\!:\!S.\ r[X := T].r[X := T] \\
(\Lambda Y.\ r)[X := T] & \equiv & \Lambda Y.\ r[X := T] \\
 & & \text{Provided that } (\Lambda Y.r) \text{ is } \alpha\text{-converted to use} \\
 & & \text{a fresh type-variable if } Y \in \{X\} \cup FV^{ty}(r). \\
(r\ s)[X := T] & \equiv & (r[X := T])\ (s[X := T]) \\
(r[S])[X := T] & \equiv & (r[X := T])[(S[X := T])]
\end{array}
$$

Figure 1.6: System F Substitution

sponds to the second-order intuitionistic logic that uses only universal quantification. This system is in fact strongly normalising [30], and so we can continue to view equality as being expressed by normalisation.

1.1.4 Curry Howard Correspondence

The programme of relating proof systems with computation does not stop at System-F. Eventually, Howard developed a full connection between natural deduction and type theory in what has become known as the Curry-Howard Isomorphism or Correspondence [79]. This broadened out type systems to encompass much more sophisticated logics, including higher order logics.

The basic programme is schematic and can be applied to many different logics and computational systems, hence why it is sometimes called a correspondence rather than an isomorphism. We identify proofs with programs, where the correspondence is given by relating each formation rule of our proofs and each syntactic method of combining type-correct programs. The type corresponds with some proposition to be proved and the program is the proof that this type is satisfied.

Types	$\cong$	Propositions
Programs	$\cong$	Proofs
Evaluation	$\cong$	Normalisation
Values	$\cong$	Normal-Form Proofs

Computation in the Curry-Howard sense arises more obliquely. Essentially computation comes from a notion of an equivalence relation over proofs. Proofs of the same proposition which are related through this equivalence relation are considered to be essentially the same proof, or program.

To tie our system together we construct a well-founded evaluation relation. This ordering allows us to talk of least elements over terms which gives us our canonical representatives, or *normal forms*.

We see in Figure 1.7 the analogy between equivalence of proofs under β-reduction and equivalence of proofs using modus ponens and those which remove the intermediate form and follow directly from the premise.

Generally speaking we will have a large number of potential methods of obtaining our canonical form computationally as the evaluation relation is not deterministic. That is, there is often more than one way to proceed.

Natural Deduction

$$\frac{A \quad A \to B}{B} \quad \cong \quad \begin{array}{c} A \\ \vdots \\ B \end{array}$$

Curry Howard

$$\frac{\Gamma \vdash t \,:\, A \qquad \Gamma \vdash (\lambda x\!:\!S.\ s) \,:\, S \to B}{\Gamma \vdash (\lambda x : S.s)\ t : B} \quad \cong \quad \begin{array}{c} \Gamma \vdash t : A \\ \vdots \\ \Gamma \vdash s[x := t] : B \end{array}$$

Figure 1.7: Substitution and Modus Ponens

The strength of the λ-calculus, System-F and other related systems is that the path used for the evaluation relation (from whence we derive our equivalence relation over proofs) is irrelevant, since all paths lead to the same normal forms, a fact which is assured by the Church-Rosser property.

Applicative Order Normal Order

$$\frac{t \rightsquigarrow t'}{s\ t \rightsquigarrow s\ t'} \qquad \frac{s \rightsquigarrow s'}{s\ t \rightsquigarrow s'\ t}$$

Figure 1.8: Choices for Evaluation Order

Since the choice is open, we can choose a deterministic strategy for following the evaluation relation to obtain a least element. Two common evaluation strategies for the λ-calculus are the *normal order* and *application order* as shown in Figure 1.8.

These developments in type theory have helped to clarify the relationship between two formal systems: programs and proofs. It has been the starting place for further developments including Per Martin-Löf's intuitionistic type-theory [50], Girard's System-F [30], the Calculus of Constructions [21] and a whole framework for understanding these various type systems in relation to each other which was presented by Barendregt[7].

We can see a visualisation of Barendregt's schema known as Barendregt's λ-cube in Figure 1.9. The coordinates of the cube correspond with having or not having the following properties:

- Terms depending on types as with System-F (also called $\lambda 2$ in the figure). This system allows terms which are parametric in some type. The types can be made concrete by application to a concrete type.

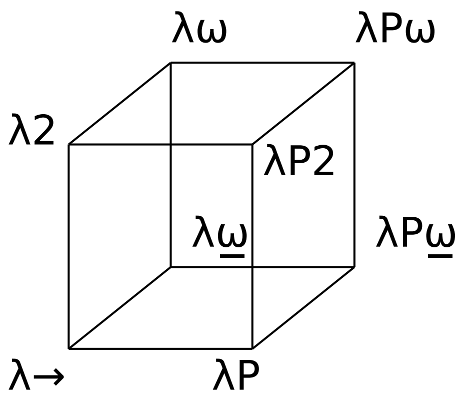

Figure 1.9: Barendregt's λ-cube

- Types depending on types as with $\lambda\underline{\omega}$. This system allows the use of type functions, functions which take a number of types and result in types. This allows type abstraction at the type level.

- Types depending on terms as with λP. This is sometimes what is meant by dependent types, as the types can *depend* on terms in the calculus. This system allows the use of functions which map some number of terms to a type. When combined with the above two properties we obtain the Calculus of Constructions (also called $\lambda P\omega$ in the figure).

1.2 Overview and Main Contributions

This work uses a fair amount of technical machinery from several different areas. Briefly we will review how these pieces fit together in order to establish the main important novel results.

The core contribution of this work is to demonstrate the use of supercompilation for demonstrating type correctness. We demonstrate that it is possible in some cases to check that a program meets its specification by using semantic preserving transformations until it manifestly type-checks using a type-checking algorithm. Since we use types as our specification of program correctness, we are demonstrating satisfaction of our specification by use of program transformation. Because we actually transform programs at the level of proofs and supercompilation uses cyclicity of structure, we find it natural to work with cyclic proof and so we introduce a cyclic type theory. In order to ensure that our program transformations are acceptable, we construct a bisimulation, giving evidence that our transformation is justified and retaining the evidential character which is the strength of type-checking.

We begin in Chapter 2 by giving a brief introduction to some of the technical machinery that is needed to understand the subsequent chapters. This includes an introduction to: least and greatest fixed points, the Coq proof assistant, the term language and type system used in this thesis and some notations.

We describe *cyclic proof* in Chapter 4 as a type system. Cyclic proof is the use of self-reference in proof structure. It turns out that such cycles are implicitly present in the type systems of most standard functional programming systems but not generally represented explicitly. We represent this cyclicity explicitly and use it as the foundation for our transformation systems.

We divide these cyclic proofs into two classes: pre-proofs and proofs, the former being a superset of the later. Proofs are those pre-proofs that we demonstrate are *sound*, a restriction discussed in Chapter 4. This soundness condition ensures termination for inductive types and co-termination for co-inductive types.

Our explicit representation of cyclic proof follows from ideas of Santocanale [76], Brotherston [13], Bradfield & Stirling [12] and Cockett [18]. The approach we give here enables us to represent recursion in a transparent fashion which works for both inductive and co-inductive types. Additionally, it provides simple descriptions of proof transformations which make use of recurrence structure, in our case, supercompilation.

The formulation of cyclic proofs given here is novel. We are unaware of any descriptions of cyclic proofs in a natural deduction style and none for System F. In addition, the term theory is simplified by the use of function constants which enables more fluidity in the use of cyclicity. A mechanisation of some important results such as progress, preservation and weak soundness have been constructed in the Coq proof assistant.

In Chapter 3, we describe (potentially infinite) transition systems and how to associate a transition system with any term. Transitions systems serve as the basis for our semantics. We develop a theory of equivalence with respect to these transition systems, showing two terms to be equivalent if their transition systems are *bisimilar*. This bisimilarity is described formally in Section 3.3. Important properties of this theory have been mechanised in the Coq proof assistant.

Once we have a suitable notion of equivalence we establish that our proof transformations are meaning preserving in Chapter 5. Specifically we show that the transition systems associated with a proof after proof transformation are bisimilar to the original. Thus we provide a new approach to understanding the formal basis for the *supercompilation* family of program transformations.

Because of this preservation of bisimilarity we can use proof transformation not only to obtain bisimilar terms, but to establish the soundness of terms by demonstrating bisimilarity with a term which is manifestly sound. This is done by describing a variant of the Modal μ-calculus and how it relates to our type theory. We show that the soundness criterion for our type systems implies that a modal logic formula related directly to the type of a term is satisfied for the transition system associated with a term.

Similar approaches have been used in local model checking by Bradfield, Stirling et al. in [12], [11]. The approach herein gives us an explicit bridge between the type theory of functional programming languages, program transformation of these languages and techniques in model checking. A similar approach to proving soundness for a functional language was demonstrated in [56], however no explicit use was made of transition systems or temporal logics, and the type system was very simple. Connections between type inhabitation and modal-logic have been noted before [18] and work has been done in establishing explicit type theories for temporal logics [80] [61], however, the present work gives a more complete description of the connection for a type system of a functional programming language.

The connection between types and formulas allows us to think of types as modal properties of transition systems, clarifying how we should interpret inductive and co-inductive types and endows our proofs with explicit demonstrations of equivalence to terms which syntactically manifest soundness.

In Chapter 7, we describe the mechanisation of some results, which is done with the proof assistant Coq[8] and a System F^+ supercompiler written in Haskell. Finally in Chapter 8 we give a final assessment of the contributions of this work and directions for further research.

Chapter 2

Preliminaries

2.1 Greatest and Least Fixed Points

The theory of greatest and least fixed points is central to both definitions of inductive and coinductive types in our theory and later is needed for understanding how this relates to behaviour. For this reason we present a brief introduction following from a more complete picture given by Gordon in [31].

Inductive and co-inductive types are least and greatest fixed point solutions respectively to equations of the form $X = F\,X$. It is called a fixed point because F leaves the solution fixed under application. They are least or greatest depending on whether the solution is the smallest or largest solution to the equation respectively.

There are restrictions on the form of F which require us to give some definitions. We take a universal set U and choose F to be an automorphism (a map with identical domain and codomain) on the power set of this set, $F : \mathcal{P}(U) \to \mathcal{P}(U)$. We assume that X and Y are drawn from this power set, or more formally, $X, Y \in \mathcal{P}(U)$.

Definition 2.1.1 (Monotone function)**.** A function is said to be *monotone* if whenever $X \subseteq Y$ it follows that $F\,X \subseteq F\,Y$.

With these definitions in hand we can proceed to obtain solutions to our equation by either of two methods which are *dual* to each other. The duality can be seen by the reversal of the direction of the inequalities and the interchange of union and intersection.

Definition 2.1.2 (Induction)**.** The set $\mu X.F\,X$ is $\bigcap\{X \mid F\,X \subseteq X\}$

Definition 2.1.3 (Co-induction)**.** The set $\nu X.F\,X$ is $\bigcup\{X \mid X \subseteq F\,X\}$

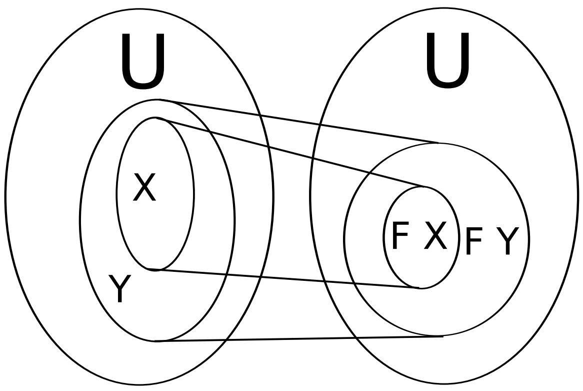

Figure 2.1: Monotone Function

In order to see the practical use of these definitions it is useful to take the familiar example of the natural numbers, $\mathbb{N}$. We can take $F\,X$ to be $\{0\} \cup \{S(x) \mid x \in X\}$ a set which is parametric in some set X. The natural numbers are then defined to be:

$$\mathbb{N} \equiv \mu X.\{0\} \cup \{S(x) \mid x \in X\}$$

We can think of this definition as *building up* our set, proceeding from a least set. To see how this works we can take an initial, empty-set $\emptyset$ and build up partial solutions, which we will index with an integer.

$$\mathbb{N}^0 \equiv F\ \emptyset = \{0\} \cup \{S(x) \mid x \in \emptyset\} = \{0\}$$

This set $\mathbb{N}^0$ is not yet a solution as $F\ \mathbb{N}^0 \not\subseteq \mathbb{N}^0$. We can however take a further iteration and get closer.

$$\mathbb{N}^1 \equiv F\ \mathbb{N}^0 = \{0\} \cup \{S(x) \mid x \in \{0\}\} = \{0, S(0)\}$$

As we take the limit of this process of partial constructions, we obtain a solution $\mathbb{N}$ which contains all of its predecessors (a fact which is ensured by the monotone property).

We can however obtain a solution which contains the point at infinity by taking the greatest-fixed point. Instead of *building-up* we will use a process of *restricting-down*. We will call this dual construction, the co-natural numbers, or $\overline{\mathbb{N}}$.

That $\mathbb{N} \subseteq \overline{\mathbb{N}}$ follows from the fact that the greatest fixed point is the union of solutions to the inequality $X \subseteq F\,X$ which holds for $\mathbb{N}$. The greatest solution also contains the point at infinity, provided that we understand ∞ to be an element such that $S(\infty) = \infty$. This is true since $\mathbb{N} \cup \{\infty\} \subseteq F\ (\mathbb{N} \cup \{\infty\})$ which can be simplified as follows:

$$F\ (\mathbb{N} \cup \{\infty\}) = \{0\} \cup \{S(x) \mid S(x) \in (\mathbb{N} \cup \{\infty\})\} = \mathbb{N} \cup \{S(\infty)\} = \mathbb{N} \cup \{\infty\}$$

This process of *building-up* elements of inductive sets allows us to deconstruct any element of such a set by a finite process of deconstruction. This process of deconstruction is *recursion*, and is crucial to ensuring termination of algorithms in functional languages.

Dually, the process of *restricting-down* to obtain elements requires us to provide elements through a potentially infinite process of construction. This leads us to the dual of recursion, namely *co-recursion*. Any element in a co-inductive set must be produced in a way that ensures that it has an acceptable behaviour (in the case of $\overline{\mathbb{N}}$, being one-more than some number) and is drawn from a set for which all elements have such a behaviour.

Our dual concept of co-induction is therefore about the behaviour of objects, as opposed to their finite construction. These infinite objects can be infinite streams, infinite trees or functions of an infinite (or indefinite) number of arguments.

2.2 Coq

In this work we make use of the proof assistant Coq for some examples and to *mechanise* some of the proofs. Coq makes extensive use of the Curry-Howard correspondence, identifying proofs and terms, and all propositions are a type which must be satisfied by a (co)terminating term. It allows proofs to be supplied as a term in a type theory: The Calculus of (Co)Inductive Constructions. The term is then checked by a type-checker to ensure that a derivation for the term, with the given type, is possible. The Coq proof assistant also provides a number of tools to help automate the creation of the terms meeting particular types, including a flexible tactic language for doing so.

The separation of the proof/term creation step from type checking allows quite a lot of freedom of implementation and an open architecture as it is not necessary to ensure under all conditions that a technique is correct since the eventual proof will have to be supplied to the type-checker in any event. The method does however have the disadvantage that enormous proofs can become infeasible as every step of the proof must be maintained for the type checker.

Coq has *sorts* which are composed of either *Set*, *Prop* or *Type*. *Prop* is a special sort which exhibits *proof irrelevance*, a feature which we discuss briefly in Section 2.3. For our purposes we can ignore the distinction.

Terms in Coq are composed of sorts, constants, variables, type abstraction and term abstraction. We can think of Coq as allowing terms to depend on types as first class objects and types to depend on terms, putting it in the far corner of the Barendregt Cube in Figure 1.9.

A typical Coq program will have some data-types and functions which act on the data-types. Theorems and Lemmas are represented by a program which *inhabits* or type checks for a type which is the Theorem or Lemma to be described.

2.2.1 Theorems and Properties

To demonstrate how this principle works we can look first at some simple propositions and their inhabitants. In Figure 2.2 we see a very simple theorem. This theorem demonstrates that conjunction of propositions is commutative.

```
Reserved Notation "x ∧ y" (at level 80, right associativity).
Inductive and (A B:Prop) : Prop :=
  conj : A → B → A ∧ B

where "A ∧ B" := (and A B) : type_scope.
Theorem and_is_commutative : ∀ A B, A ∧ B → B ∧ A.
Proof.
  refine
    (fun (A B : Prop) (p : A ∧ B) ⇒
      match p with
        | conj a b ⇒ conj B A b a
      end).
Defined.
```

Figure 2.2: Commutivity of Conjunction

The type $A \wedge B$ is the type of a pair of propositions and this data-type is introduced by way of an inductive data-type declaration. The declaration states that we have an inductive type, parametric in (or dependent on) two other types, A and B which are propositions. We can introduce this type by way of a single constructor, conj. This conj constructor will take a proof of A (which we can also read as a term of type A) and a proof of B.

We demonstrate the theorem we wish to prove using `Theorem` with the name and_is_commutative to which we attach a proposition, or type, which we wish to prove, which in this case is: $\forall\ A\ B, A \wedge B\ \to\ B \wedge A$. We are essentially creating a new constant with name and_is_commutative which we suggest has the type of $\forall\ A\ B, A \wedge B\ \to\ B \wedge A$. In order to show that we are justified in suggesting such a constant exists we must show that the type is inhabited by at least one term.

The term we supply is a function. The reason for this is that the proposition is parametric in two types, so we must have an abstraction over the two types so that we can work with arbitrary types. In addition, the implication is a statement that the proof is parametric in a term of type $A \wedge B$, that is a pair of terms each of type A and B respectively.

Consequently, our proof is a function, introduced with the keyword fun taking three parameters: A, B and a pair $A \wedge B$. We now make use of a principle of inductive types, namely the capacity to perform elimination on the type by making use of the fact that they can only be introduced by some finite number of constructors. Since in our case we have only a single constructor, we have a `match` clause with only one branch, the case where the type was introduced with the constructor conj. This constructor has two arguments, the two terms of type A and B, hence our elimination supplies us with these two terms, which we bind to the variables a and b.

Now that we have these two terms, we can reconstruct a term of type $A \wedge B$ by using the constructor conj again and simply swapping the terms. Since the type of conj is parametric in two other types, when creating a new term of this type we have to supply these. This is why we actually have four parameters to the constructor in the term (conj $B\ A\ b\ a$). This final term is of the appropriate type and we can then end our proof. The final line makes use of the `Defined` keyword, which invokes the type-checker to see if the term does in fact have the type we have stated. If it does not, then the entire theorem is rejected. If it does pass the type checker, then we are allowed to use the constant and_is_commutative as a representative of this type and our theorem can be considered to be true.

2.2.2 (Co)Recursion and (Co)Induction

There are a great number of mathematical objects and theorems which rely on a notion of *induction*. Induction is essentially the process of building up structures in a finite way starting with some number of *base cases* which are taken to hold by definition.

As we saw above, the conjunction was introduced as an inductive type. However this type in fact did not have any self-reference and therefore was in some sense degenerate. More interesting inductive types make use of self-reference.

Properties of such structures can be proved by using *recursion*, which demonstrates how to build up the property by demonstrating it on the base cases and that we can derive the property given we assume that it holds of everything *below*. In order to have a notion of *below* we require that we have a well-founded relation which gives us an ordering.

Since this is ubiquitous in the mathematical literature, in order to use type theory as a metamathematical language, it was useful to extend it to include inductively defined structures and principles of induction including finite objects which can naturally be expressed by *inductive types* [50].

Informally we can think of inductive types as being formed from a finite number of constructors with the possibility of some self-reference. A very common example of such an inductive type can be represented as follows:

$$\mu\mathbb{N} \equiv 0 : \mathbb{N} \mid S : \mathbb{N} \to \mathbb{N}$$

This definition of the natural numbers (or Peano numbers) gives a constant 0 and a successor function S which can be thought of as *one plus* some number which is already in $\mathbb{N}$ and gives us a way to get the *next* value.

We use the prefix μ to refer to the fact that we are interested in viewing the type $\mathbb{N}$ as all objects which are produced by some terminating process.

Coupled with these structures is a principle of elimination. This elimination principle allows us to deconstruct our inductive types in order to describe properties which hold over the structures. The elimination principle we need for the natural numbers corresponds closely with the well known *Axiom of induction*.

Definition 2.2.1 (Axiom of Induction)**.** $\forall P.P\ 0 \wedge \forall m.P\ m \to P\ (S\ m) \to \forall n.P\ n$

One can see the basic approach involved in the generation of this principle where each constructor is individually proved to meet the property in question (in this case our constructors are 0 and S) assuming the properties hold on the sub-cases. The generalisation of this approach to arbitrary inductively defined types is known as *structural induction*.

In fact we do not need to write down such a strict principle of induction for each data-type, we can (with certain technical restrictions) safely allow such principles to be generated on the fly.

We do this by using *recursion*. The Coq proof-assistant takes such an approach. We can therefore use Coq to help us get an intuition about how to perform inductive proofs in type theory.

The natural numbers in Coq can be written much as we did before, by describing the constructors. The syntax of the definition of the natural numbers is as follows:

```
Inductive Nat : Set :=
| O : Nat
| S : Nat → Nat.

Inductive lt : Nat → Nat → Prop :=
| lt_O : ∀ n, O < S n
| lt_S : ∀ n m, n < m → ((S n) < (S m))


where "n < m" := (lt n m) : nat_scope.
```

This definition of **Nat** states that there are two constructors, O and S. The O constructor is a natural number and the S constructor is a map from the natural numbers to themselves.

In addition we define a relation **lt**, which describes what it means for one natural number to be less than another. We do this with a base case, lt_O, which relates that a number plus one, is always greater than zero, or more formally $\forall\ n.O < S\ n$. The next case, lt_S states that we can always add one to both sides of the inequality, or $\forall\ n\ m.\ n < m \rightarrow (S\ n) < (S\ m)$. We introduce a notation $<$ for **lt** to make proofs easier to read.

Supposing we want to demonstrate a property over all natural numbers such as the following: $\forall n.\exists m.n < m$. That is, all natural numbers are bounded by some other natural number. In Coq we might write this proof as follows:

```
Require Import Nat.

Definition bounded : ∀ n, ∃ m, n < m :=
  (fix bounded (n : Nat) : ∃ m, n < m :=
    match n as n0 return ∃ m, n0 < m with
      | O ⇒ ex_intro (fun m : Nat ⇒ O < m) (S O) (lt_O O)
      | S n1 ⇒
        match (bounded n1) with
          | ex_intro m P ⇒
            ex_intro (fun m0 : Nat ⇒ S n1 < m0) (S m) (lt_S n1 m P)
        end
    end).
```

The form of the proof looks very much like a functional programming language, and indeed we are using the Curry-Howard correspondence to demonstrate that the type of the program is inhabited by a proof, or program term.

The universal quantification from our original statement is represented as a recursive function bounded, taking a parameter n and showing that in all cases, we can produce the result type of interest, $\exists m, n < m$. We know that bounded is recursive because of the *fix* keyword. It should be noted that the definition name bounded, is being punned with the fixpoint variable *bounded*. The definition name refers to the entire term, while the fixpoint variable is only in scope under the *fix* binder.

The elimination is performed by a `match` statement which deconstructs n into two sub-cases, one where $n = 0$ and one where $n = S\ n1$. Existentials are represented as a *dependent pair*. That is, they are a pair of a term, and a proof that the term meets some property. We introduce them by using the existential constructor ex_intro. In this case we use ex_intro, together with the term that meets the property of interest, (S O), along with the proof, (lt_O O), that zero is less then a successor of a number and that it really does meet the property we are interested in (in this case $0 < m$).

We can think of the justification of the use of our *prior* case as being based on the fact that we have a recursive call which establishes this fact for the sub case, (*bounded n1*), which we can then reuse to provide our $n + 1$ step. That the process terminates is assured by the fact that we only work on sub-terms when we make recursive calls. We deconstruct the existential for the prior case in order to reuse its proof P and m to construct a new proof that covers the $m + 1$ case. We do this by using the Coq library again which contains a lemma lt_S with the type $\forall n\ m. n < m \rightarrow S\ n < S\ m$. Since our subcase proves the $n < m$ case, we simply apply this lemma to produce the proof we need.

The dual principle of *coinduction* has associated with it coinductively defined types and corecursion as a method of generating proofs of properties. However, instead of recursing on ever smaller arguments terminating at a base case, we will be adding a coinductive constructor at each corecursive step.

An example in Coq using the co-natural numbers demonstrates the method of construction for the point at infinity.

```
CoInductive conat : Set :=
| O : conat
| S : conat → conat.
Definition inf : conat := (cofix inf : conat := S inf).
```

Here we demonstrate the coinductive property by first using the constructor S and then we are able to safely assume that inf satisfies a coinductive property (namely, that it is a co-natural number). Essentially we are showing that we can always do the *next* thing, and so, despite being non-terminating or of infinite data-type, we know that we will always produce a constructor of the appropriate type. This program is productive and will not lead to non-termination without behaviour. Similarly with our bounded definition, we are punning the co-fixpoint variable, *inf*, with the name of the term.

The introduction of co-inductive types has given us the ability to work with *server-like* or *reactive* systems, since we can have infinite behaviours (some of which may be to react infinitely to stimuli).

However, this slight change leads to a host of complications. We now are in danger of losing many of the properties that we had previously with finite proofs. Normalisation can no longer be relied upon to give us a notion of equivalence that is very useful, and neither is it obvious how our computation should take place as the Church-Rosser property will not hold. The choice of evaluation order is now important in leading us to results.

In this work, coinduction plays a central role. It is used to provide infinite data types at the program level. However it is also used at the metalogical level to prove properties about the behaviour of our term calculus and its type system. Specifically we will be manipulating programs in such a way that we ensure that we retain their properties even if they are non-terminating.

2.3 Equality

At the core of many theories of mathematical reasoning is some notion of how to decide when one thing is like another. In fact the question of equality is not a simple question. Indeed, whether one thing is like another largely depends on what we would like to distinguish, and there are many ways to settle the question including canonical forms, equivalence relations and isomorphisms.

It is often the case that canonical forms are used to decide the problem of equality. In type theory and proof theory, *normalisation*, a process whereby we find *normal forms* or canonical referents is employed. This has the advantage of keeping the process of computation tightly coupled with proofs.

This approach however is not always fruitful. Normal forms can not be found in all formal systems, as we saw for the untyped λ-calculus. When normal forms *can always* be found, it proves to be a double edged sword. The very fact of a strong normalisation theorem precludes the possibility of Turing-completeness. This fact follows from a very simple application of the Halting problem. The Halting Problem is well known to be undecidable, and hence it follows that a Turing-complete system will not always exhibit normal forms.

In some cases, we may not be interested in full Turing completeness, but rather the fact that certain programs meet various important properties. However, we will find again that it is not always simple to represent equality as the result of a normalisation procedure, especially for infinite programs.

In type theory there are also cases in which the structure of the proof is unimportant as the information of consequence is completely constrained by the type, or the type is the only item of interest. This leads to a notion of *proof irrelevance*. Proof irrelevance identifies all proofs of a particular type. That is, we have an equivalence class of terms which does not distinguish between any two terms which have the same type.

This granularity however is far too coarse for our purposes. We are interested in programs which may be underspecified by their types. That is, we may have several programs which all are inhabitants of a type, but which we wish to distinguish. To see where we might run into problems with proof irrelevance, we can use the type $Bool$ as an example. It has two inhabitants, $true$ and $false$ and both are proofs of $Bool$. Identifying $true$ and $false$ in programs could lead to some obvious problems.

So what type of equivalence are we looking for? We would like to equate programs, and therefore proofs, which have the same *behaviour*, for a suitable notion of behaviour. Defining behaviour, it turns out, is slightly more involved than it might at first seem. In order to do so, we have to have some idea of what constitutes behaviour. Then we have to have some way of demonstrating an equivalence between these two types of behaviour.

Contextual equivalence was recognised fairly early on (see especially Morris [59]) to give a sensible answer to the question of what it means to have the same behaviour. Contextual equivalence answers the question by requiring that two programs must behave identically in all possible

```
CoInductive conat : Set :=
| O : conat
| S : conat → conat.
Definition inf : conat := (cofix inf : conat := S inf).
Definition inf2 : conat := (cofix inf2 : conat := S (S inf2)).
```

Figure 2.3: Program for inf and $inf2$

surrounding contexts to be considered behaviourally identical. In other words, no amount of programming in the term language will allow you to distinguish two terms if they are contextually equivalent.

In some sense we can understand these contexts to be predicates over the behaviours of terms. In the following definition we understand $\mathcal{B}$ to be the type of booleans, having two constants, $true$ and $false$ which inhabit the type.

This description of equivalence is powerful, but it is also technically challenging to prove. It requires quantification over contexts, which are relatively cumbersome. In addition, filling holes in contexts does not respect α-equivalence. This means we lose our first notion of equality modulo renaming of variables.

For this reason we will look to establish equality using an alternative method. It turns out that there is another method of demonstrating equivalence of *behaviour* over systems with even an infinite number of potential behaviours. This was discovered by Milner in the context of *transition systems*, which we will describe in more detail in Chapter. For now, we would like the reader to imagine the graphs in Figure 2.3 and Figure 2.4 as abstract depictions of the behaviour.

Going back to our example of the co-natural numbers, we can construct a program which contains two terms, *inf* and *inf2* which we have in Figure 2.3.

Visually, we can view these systems as being represented by the following S actions between states, as in Figure 2.4. One of the systems emits an S and returns to the initial state. The other system emits an S which leads to a new state, followed by an S returning to the initial state.

These two systems are, however, identical in some sense. We can make explicit the sense in which they are the same by means of a coinductive relation that demonstrates their equivalence which is given in Figure 2.5.

This relation is reflexive by construction, but it also has a rule that if we can demonstrate that a further pair of terms is **coeq** we can demonstrate that the case with successors is **coeq**. That

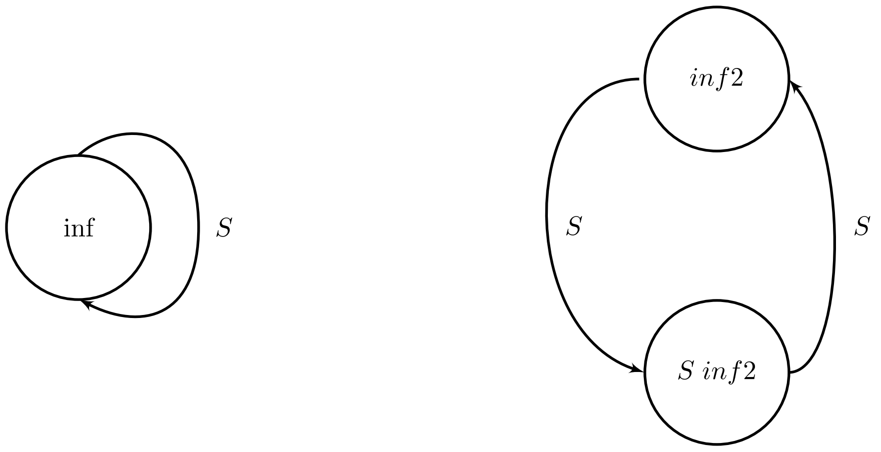

Figure 2.4: Two Bisimilar Systems

```
CoInductive conat : Set :=
| O : conat
| S : conat → conat.

Definition inf : conat := (cofix inf : conat := S inf).
Definition inf2 : conat := (cofix inf2 : conat := S (S inf2)).

CoInductive coeq : conat → conat → Prop :=
| Coeq_refl : ∀ (t:conat), coeq t t
| Coeq_next : ∀ (t s:conat), coeq t s → coeq (S t) (S s).

Definition decomp (x : conat) : conat :=
  match x with
    | O ⇒ O
    | S x' ⇒ S x'
  end.

Definition decomp_eql : ∀ x, x = decomp x :=
  fun x : conat ⇒
    match x as c return (c = decomp c) with
      | O ⇒ refl_equal O
      | S c ⇒ refl_equal (S c)
    end.

Definition inf_coeq_inf2 : coeq inf inf2 :=
  cofix inf_coeq_inf2 : coeq inf inf2 :=
    eq_ind_r (fun c : conat ⇒ coeq c inf2)
    (eq_ind_r (fun c : conat ⇒ coeq (decomp inf) c)
      (Coeq_next inf (S inf2)
        (eq_ind_r (fun c : conat ⇒ coeq c (S inf2))
          (Coeq_next inf inf2 inf_coeq_inf2) (decomp_eql inf)))
      (decomp_eql inf2)) (decomp_eql inf).
```

Figure 2.5: Coequivalence

this relation is *coinductive* will allow us to perform the following proof.

Theorem 2.3.1 (Behavioural equivalence of inf and inf2)**.** ***coeq*** *inf inf2*

Proof. The proof proceeds coinductively with the coinductive hypothesis that **coeq** inf inf2. First, we unfold the definitions of inf and inf2 to obtain S inf and S (S inf2) respectively. We can then apply the constructor Coeq_next of the **coeq** relation. This leaves us with **coeq** inf (S inf2) as our goal. We can unfold the definition of inf to S inf yielding the goal **coeq** (S inf) (S inf2). Once again we can apply the constructor coeq_next to obtain **coeq** inf inf2 which is our coinductive hypothesis. □

This proof demonstrates an equivalence between inf and inf2. The central concept of equivalence used in this work will be a coinductive equivalence relation using a similar technique to demonstrate such a behavioural equivalence of program terms known as a *bisimulation*.

It is important to notice that in the proof we make use of the property that we are trying to prove. It should be clear that this technique cannot be used in an unrestricted way or it would lead to an inconsistent logic as we could generate a proof for any type simply by assuming the thing we are trying to prove.

We ensure that this vacuous circularity cannot occur by insisting that we never make use of the coinductive hypothesis until after we have made use of a constructor from the coinductive type we wish to show has a proof. In the example above, we use Coeq_next prior to the use of the coinductive hypothesis. This principle of giving a constructor prior to use of the coinductive hypothesis is known as the *guardedness condition* which we will define more formally in Chapter 4.

Guardedness is one method of ensuring that proofs are *productive*. Informally, productivity means that when we examine the resulting coinductive proof by elimination, we are assured of getting a constructor. The degenerate case ($cofix\, f : A := f$) clearly does not satisfy this principle.

The approach of using coinduction to describe program behaviour has surprising consequences. We find fruitful connections can be drawn between type theories and modal logics with fixed points. Specifically we will find a fragment of the Modal μ-Calculus can be used to demonstrate that programs satisfy a type (that is, *type inhabitation*). This allows us to transform our problem of type inhabitation to the problem of determining the satisfaction of temporal formulae.

In previous work [37] we have shown how circularities can be recognised by the graph structure of behaviours. However, it is not necessary for vacuous circularity to be avoided in the case of program transformation of general recursive programs. There is merely a requirement that the programs be contextually equivalent.

2.4 Language

We use a variant of Girard's System F, which we call System F$^+$ extended with inductive and co-inductive types which is presented in Figure 2.6. We use this type system for several reasons. Firstly, it is less difficult to work with than its more sophisticated relatives on the Barendregt cube [7] (see Figure 1.9) yet it is still sophisticated enough to obtain interesting results.

While System-F is simpler than type theories which can be used for practical languages such as Haskell [81], it is rich enough that the difficult problems found in practical programming language type systems still arise and so it can inform the development of practical solutions. This allows us to retain an economy of presentation while still dealing with difficult questions.

Variables

$\mathbf{Var} \ni x, y, z$

Type Variables

$\mathbf{TyVar} \ni X, Y, Z$
$\mathbf{PosVar} \ni \hat{X}, \hat{Y}, \hat{Z}$

Function Symbols

$\mathbf{Fun} \ni \mathtt{f}, \mathtt{g}$

Types

$$\begin{array}{lll} \mathbf{Ty} \ni A, B, C & ::= & \mathbb{1} \mid X \mid \hat{X} \mid A \to B \mid \forall X.\ A \mid A + B \mid A \times B \\ & \mid & \nu \hat{X}.\ A \mid \mu \hat{X}.\ A \end{array}$$

Terms

$$\begin{array}{lll} \mathbf{Tr} \ni r, s, t, u, v & ::= & x \mid \mathtt{f} \mid () \mid \lambda x\colon A.\ t \mid \Lambda X.\ t \mid r\ s \mid r[A] \\ & \mid & \text{left}(t, A + B) \mid \text{right}(t, A + B) \mid (t, s) \\ & \mid & \text{in}_\nu(t, A) \mid \text{out}_\nu(t, A) \\ & \mid & \text{in}_\mu(t, A) \mid \text{out}_\mu(t, A) \\ & \mid & \text{case } r \text{ of } \{x_1 \Rightarrow s \mid x_2 \Rightarrow t\} \\ & \mid & \text{split } r \text{ as } (x_1, x_2) \text{ in } s \end{array}$$

Contexts

$$\begin{array}{llll} \Delta & ::= & \cdot \mid \Delta \cup \{X\} \mid \Delta \cup \{\hat{X}\} & \text{Type Variable Contexts} \\ \Gamma & ::= & \cdot \mid \Gamma \cup \{x\colon A\} & \text{Variable Contexts} \end{array}$$

Figure 2.6: System F$^+$ Syntax

Briefly, we include the usual variables x, constants f, lambda terms $\lambda x\colon A.\ t$, applications $r\ s$, type abstractions $\Lambda X.\, t$ and type applications $r[A]$.

We add to the usual System F language a unit type, $\mathbb{1}$, together with its inhabitant $()$. This will represent a program for which no *observations* can be made. By *observation* we mean, there is no reduction which can be performed on elements of unit type. This element is called a *terminal object* in terms of *Category theory* [69].

To this we can add sum types, $A + B$ together with two terms which provide right and left injections into those types, left and right respectively. In addition we have an elimination term case which allows us to map to some arbitrary type C.

We also include pairing (r, s) to inject into the product type $A \times B$ and an eliminator, split. These are all effectively just a short hand for the usual Church encoding of sums and products (see Appendix B).

To these algebraic constructions we add fixed points. We have two sets of (in) and (out) terms, one for inductive types (corresponding with least fixed points or *initial algebra semantics* $\mu\hat{X}.\, A$) and one for co-inductive types (corresponding with greatest fixed points, or *final co-algebraic semantics* $\nu\hat{X}.\, A$). We distinguish the type variables used in these types from those using universal quantification as we need to be more restrictive about what constitutes a well formed type. The restriction is that the types be *strictly positive* with respect to the use of (co)inductive type variables. Since we do not require this of universal quantification we need to segregate the two. The positivity is enforced by the formation rules in Figure 2.7. Essentially we must insist that (co)inductive type variables are not present in the context of the left hand side of the $(\rightarrow)$ type.

Unlike the other terms which we introduce (out) and (in) are not just syntactic sugar for Church encodings. While it is possible to encode greatest and least fixed points in System F directly (see Appendix B), without careful attention to the formation rules we would have introduced the possibility of non-well-founded terms. The reason for this is subtle but is due to the fact that least and greatest fixed points for a particular given functor must be constructed explicitly when using Church encodings and we cannot assume their existence *a priori* without some further restriction on the type.

Substitutions of a single variable will be written $[x := t]$ or $[X := A]$ for term and type substitutions respectively. We give the definition of substitution defined as a recursive function on terms and types in Figure 2.8 and Figure 7.6 respectively.

There also is the need to introduce recursive terms. Recursive terms in this presentation are represented using function constants. Function constants are drawn from a countably infinite set

Universally Quantified Type Variables

$$\begin{array}{lcl} UV(\Delta \cup \{\hat{X}\}) & := & UV(\Delta) \\ UV(\Delta \cup \{X\}) & := & \{X\} \cup UV(\Delta) \end{array}$$

Context Formation

$$\frac{}{\cdot \ \texttt{TyCtx}} \qquad \frac{\Delta \ \texttt{TyCtx} \quad X \notin \Delta}{\Delta \cup \{X\} \ \texttt{TyCtx}} \qquad \frac{\Delta \ \texttt{TyCtx} \quad \hat{X} \notin \Delta}{\Delta \cup \{\hat{X}\} \ \texttt{TyCtx}}$$

$$\frac{}{\Delta \vdash \cdot \ \texttt{Ctx}} \qquad \frac{\Delta \vdash \Gamma \ \texttt{Ctx} \quad x \notin dom(\Gamma) \quad \Delta \vdash A \ \texttt{type}}{\Delta \vdash \Gamma \cup \{x : A\} \ \texttt{Ctx}}$$

Type Formation

$$\frac{UV(\Delta) \vdash A \ \texttt{type} \quad \Delta \vdash B \ \texttt{type}}{\Delta \vdash A \to B \ \texttt{type}} \qquad \frac{\Delta \cup \{\hat{X}\} \vdash A \ \texttt{type} \quad \alpha \in \{\nu, \ \mu\}}{\Delta \vdash \alpha\hat{X}. \ A \ \texttt{type}}$$

$$\frac{\Delta \vdash A \ \texttt{type} \quad \Delta \vdash B \ \texttt{type} \quad \circ \in \{+, \ \times\}}{\Delta \vdash A \circ B \ \texttt{type}} \qquad \frac{}{\Delta \vdash 1 \ \texttt{type}}$$

$$\frac{\Delta \cup \{\hat{X}\} \ \texttt{TyCtx}}{\Delta \cup \{\hat{X}\} \vdash \hat{X} \ \texttt{type}} \qquad \frac{\Delta \cup \{X\} \ \texttt{TyCtx}}{\Delta \cup \{X\} \vdash X \ \texttt{type}} \qquad \frac{\Delta \cup \{X\} \vdash A \ \texttt{type}}{\Delta \vdash \forall X. \ A \ \texttt{type}}$$

Figure 2.7: Formation Rules

Term Substitution

$x[x := t]$	$\equiv$	t
$x[y := t]$	$\equiv$	x if $x \neq y$
$\mathtt{f}[x := t]$	$\equiv$	$\mathtt{f}$
$(r\ s)[x := t]$	$\equiv$	$(r[x := t])\ (s[x := t])$
$(\lambda y : A.\ r)[x := t]$	$\equiv$	$\lambda y : A.\ r[x := t]$ Provided that $\lambda y : A.\ r$ is α-converted to use a fresh variable if $y \in \{x\} \cup FV(t)$.
$(\Lambda X.\ r)[x := t]$	$\equiv$	$\Lambda X.\ r[x := t]$
$\text{in}_\alpha(s, A)[x := t]$	$\equiv$	$\text{in}_\alpha(s[x := t], A)$
$\text{out}_\alpha(s, A)[x := t]$	$\equiv$	$\text{out}_\alpha(s[x := t], A)$
$()[x := t]$	$\equiv$	$()$
$\text{right}(s, A)[x := t]$	$\equiv$	$\text{right}(s[x := t], A)$
$\text{left}(s, A)[x := t]$	$\equiv$	$\text{left}(s[x := t], A)$
$(s, u)[x := t]$	$\equiv$	$(s[x := t], u[x := t])$
case r of $\{y \Rightarrow s \mid z \Rightarrow u\}[x := t]$	$\equiv$	case $r[x := t]$ of $\{y \Rightarrow s[x := t] \mid z \Rightarrow u[x := t]\}$ Provided that $\lambda y : A.\ s$ or $\lambda z : A.\ u$ are α-converted to use a fresh variable if $y \in \{x\} \cup FV(t)$ or $z \in \{x\} \cup FV(t)$ respectively.
split r as (y, z) in $u[x := t]$	$\equiv$	split $r[x := t]$ as (y, z) in $u[x := t]$ Provided that $\lambda y : A.\ \lambda z : A.\ u$ is α-converted to use a fresh variable for y or z if $y \in \{x\} \cup FV(t)$ or $z \in \{x\} \cup FV(t)$ respectively.

Figure 2.8: Term Substitution for System F^{+}

Type Substitution on Terms

$x[X := A]$	$\equiv$	x
$\mathtt{f}[X := A]$	$\equiv$	$\mathtt{f}$
$()[X := A]$	$\equiv$	$()$
$(r\ s)[X := A]$	$\equiv$	$(r[X := A])\ (s[X := A])$
$(r[A])[X := A]$	$\equiv$	$(r[X := A])\ (A[X := A])$
$(\lambda x{:}\,A.\ r[X := A]$	$\equiv$	$\lambda x{:}\,A[X := A].\ r[X := A]$
$\mathrm{in}_\alpha(s, B)[X := A]$	$\equiv$	$\mathrm{in}_\alpha(s[X := A], B[X := A])$
$\mathrm{out}_\alpha(s, B)[X := A]$	$\equiv$	$\mathrm{out}_\alpha(s[X := A], B[X := A])$
$\mathrm{right}(s, B)[X := A]$	$\equiv$	$\mathrm{right}(s[X := A], B[X := A])$
$\mathrm{left}(s, B)[X := A]$	$\equiv$	$\mathrm{left}(s[X := A], B[X := A])$
$(s, u)[X := A]$	$\equiv$	$(s[X := A], u[X := A])$
(case r of $\{y \Rightarrow s \mid z \Rightarrow u\})[X := A]$	$\equiv$	case $r[X := A]$ of
		$\{\ y \Rightarrow s[X := A]$
		$\mid z \Rightarrow u[X := A]$
(split r as (y, z) in $u)[X := A]$	$\equiv$	split $r[X := A]$ as (y, z) in $u[X := A]$

Type Substitution on Types

$X[X := A]$	$\equiv$	A
$X[Y := A]$	$\equiv$	X if $X \neq Y$
$\mathbb{1}[X := A]$	$\equiv$	$\mathbb{1}$
$B + C[X := A]$	$\equiv$	$B[X := A] + C[X := A]$
$B \times C[X := A]$	$\equiv$	$B[X := A] \times C[X := A]$
$(B \to C)[X := A]$	$\equiv$	$B[X := A] \to C[X := A]$
$(\forall Y.\ B)[X := A]$	$\equiv$	$\forall Y.\ B[X := A]$
		Provided that $(\forall Y.\ B)$ is α-converted to use
		a fresh type-variable if $Y \in \{X\} \cup FV(A)$.
$(\Lambda Y.\ r)[X := A]$	$\equiv$	$\Lambda Y.\ r[X := A]$
		Provided that $(\Lambda Y.\ r)$ is α-converted to use
		a fresh type-variable if $Y \in \{X\} \cup FV(A)$.
$(\alpha Y.\ r)[X := A]$	$\equiv$	$\alpha Y.\ r[X := A]$, $\alpha \in \{\nu,\ \mu\}$
		Provided that $(\alpha Y.\ r)$ is α-converted to use
		a fresh type-variable if $Y \in \{X\} \cup FV(A)$.

Figure 2.9: Substitution for System F$^+$

Fun. With every term containing function constants is associated a function Ω which maps a function constant $\mathtt{f}$ to a term t, $\Omega(\mathtt{f}) = t$, where t may itself contain function constants in the domain of Ω. This allows us to deal with recursive and mutually recursive functions freely and without difficulty. All of our proofs will be parametric in this Ω as a constant and so it will be left implied. In addition to the constant Ω we also introduce a constant Ξ which associates each function symbol with a type. That is if $\Xi(\mathtt{f}) = A$ then there is a derivation of $\cdot\ ; \cdot \vdash \Omega(\mathtt{f})\ :\ A$ under the assumption that $cdot\ ; \cdot \vdash \mathtt{f}\ :\ A$. In other words, we are only interested in working with well typed programs in the usual functional programming sense.

For a term t with type A in a variable context Γ and type variable context Δ we write $\Delta\ ; \Gamma \vdash t\ :\ A$. A type derivation must be given using the rules given in Figure 2.10. Each elimination rule, which corresponds to some sort of *observation* is denoted with an E, with a superscript indexing the type which they eliminate. Introduction rules, which correspond with observable behaviour, are given by I, together with a superscript indexing the type that they introduce.

Static Semantics

$$\frac{\Delta \vdash \Gamma \cup \{x{:}A\}\ \mathtt{Ctx}}{\Delta\ ; \Gamma \cup \{x{:}A\} \vdash x\ :\ A}\ I^{Var} \qquad \frac{\Delta\ ; \Gamma \cup \{x{:}A\} \vdash t\ :\ B}{\Delta\ ; \Gamma \vdash (\lambda x{:}A.\ t)\ :\ A \to B}\ I^{\to}$$

$$\frac{\Delta\ ; \Gamma \vdash r\ :\ A \to B \qquad \Delta\ ; \Gamma \vdash s\ :\ A}{\Delta\ ; \Gamma \vdash (r\ s)\ :\ B}\ E^{\to} \qquad \frac{\Delta \cup \{X\}\ ; \Gamma \vdash t\ :\ A}{\Delta\ ; \Gamma \vdash (\Lambda X.\ t)\ :\ \forall X.\ A}\ I^{\forall}$$

$$\frac{\Delta\ ; \Gamma \vdash t\ :\ \forall X.\ A \qquad \Delta \vdash B\ \mathtt{type}}{\Delta\ ; \Gamma \vdash t[B]\ :\ A[X := B]}\ E^{\forall} \qquad \frac{}{\Delta\ ; \Gamma \vdash f\ :\ \Xi(\mathtt{f})}\ I^{\Omega}$$

$$\frac{}{\Delta\ ; \Gamma \vdash ()\ :\ \mathbb{1}}\ I^{1} \qquad \frac{\Delta\ ; \Gamma \vdash r\ :\ A \qquad \Delta\ ; \Gamma \vdash s\ :\ B}{\Delta\ ; \Gamma \vdash (r, s)\ :\ (A \times B)}\ I^{\times}$$

$$\frac{\Delta\ ; \Gamma \vdash t\ :\ A \qquad \Delta \vdash B\ \mathtt{type}}{\Delta\ ; \Gamma \vdash \mathrm{left}(t, A + B)\ :\ (A + B)}\ I^{+}_{L} \qquad \frac{\Delta\ ; \Gamma \vdash t\ :\ B \qquad \Delta \vdash A\ \mathtt{type}}{\Delta\ ; \Gamma \vdash \mathrm{right}(t, A + B)\ :\ (A + B)}\ I^{+}_{R}$$

$$\frac{\Delta\ ; \Gamma \vdash t\ :\ \alpha\hat{X}.\ A \qquad \alpha \in \{\mu, \nu\}}{\Delta\ ; \Gamma \vdash \mathrm{out}_{\alpha}(t, \alpha\hat{X}.\ A)\ :\ A[\hat{X} := \alpha\hat{X}.\ A]}\ E^{\alpha}$$

$$\frac{\Delta\ ; \Gamma \vdash t\ :\ A[\hat{X} := \alpha\hat{X}.\ A] \qquad \alpha \in \{\mu, \nu\}}{\Delta\ ; \Gamma \vdash \mathrm{in}_{\alpha}(t, \alpha\hat{X}.\ A)\ :\ \alpha\hat{X}.\ A}\ I^{\alpha}$$

$$\frac{\Delta\ ; \Gamma \vdash r\ :\ A + B \qquad \Delta\ ; \Gamma \cup \{x{:}A\} \vdash t\ :\ C \qquad \Delta\ ; \Gamma \cup \{y{:}B\} \vdash s\ :\ C}{\Delta\ ; \Gamma \vdash (\mathrm{case}\ r\ \mathrm{of}\ \{x \Rightarrow t \mid y \Rightarrow s\})\ :\ C}\ E^{+}$$

$$\frac{\Delta\ ; \Gamma \vdash s\ :\ A \times B \qquad \Delta\ ; \Gamma \cup \{x{:}A\} \cup \{y{:}B\} \vdash t\ :\ C}{\Delta\ ; \Gamma \vdash (\mathrm{split}\ s\ \mathrm{as}\ (x, y)\ \mathrm{in}\ t)\ :\ C}\ E^{\times}$$

Figure 2.10: System F^{+} Proof Rules

Inductive and co-inductive types are described using a least and greatest fixed point notation,

$\mu\hat{X}.\ A$ and $\nu\hat{X}.\ A$, respectively. Intuitively the least fixed point corresponds with finite data-types while the greatest fixed point corresponds with potentially infinite data-types.

The language makes use of iso-recursive types[70] (explicit folding and unfolding of recursive types) which are introduced with explicit type coercions (in) and (out). The use of iso-recursive types introduces the possibility of non-termination if the type is not given some additional restriction as was described previously. We will restrict the form of types which can be used with the (I^{α}/E^{α}) rule to meet an additional positivity condition on the structure of types following on from the presentation given in [68]. The positivity restriction requires that all uses of the α rules close over a variable which occurs *strictly positively*.

We introduce this positivity condition on types because an unrestricted version of inductive and coinductive types can lead to inconsistency. To see that this is the case we can use the example of the following type $\mathcal{D} \equiv \nu\hat{D}.\ \hat{D} \to \hat{D}$ (this is related to the domain equation for the λ-calculus, see [77]). With this type we can construct the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot \vdash \{x{:}\mathcal{D}\}\ \texttt{Ctx}}{\cdot\,;\{x{:}\mathcal{D}\} \vdash x : \mathcal{D}}}{\cdot\,;\{x{:}\mathcal{D}\} \vdash \mathrm{out}_{\nu}(x,\mathcal{D}) : \mathcal{D} \to \mathcal{D}} \qquad \dfrac{\cdot \vdash \{x{:}\mathcal{D}\}\ \texttt{Ctx}}{\cdot\,;\{x{:}\mathcal{D}\} \vdash x : \mathcal{D}}}{\cdot\,;\{x : \mathcal{D}\} \vdash \mathrm{out}_{\nu}(x,\mathcal{D})\ x : \mathcal{D}}}{\cdot\,;\cdot \vdash \lambda x{:}\mathcal{D}.\ \mathrm{out}_{\nu}(x,\mathcal{D})\ x : \mathcal{D} \to \mathcal{D}}$$

If we take the term at the base of the tree $\lambda x{:}\mathcal{D}.\ \mathrm{out}_{\nu}(x,\mathcal{D})\ x$ and call it ω we can form a proof:

$$\dfrac{\cdot\,;\cdot \vdash \omega : \mathcal{D} \to \mathcal{D} \qquad \dfrac{\cdot\,;\cdot \vdash \omega : \mathcal{D} \to \mathcal{D}}{\cdot\,;\cdot \vdash \mathrm{in}_{\nu}(\omega,\mathcal{D}) : \mathcal{D}}}{\cdot\,;\cdot \vdash \omega\ \mathrm{in}_{\nu}(\omega,\mathcal{D}) : \mathcal{D}}$$

This proof is clearly a problem, since despite having no cyclic structure and not making use of function constants, it will never normalise to a value. It is not a productive proof and should not be allowed in a coinductive type. The source of the problem stems from the non-monotonicity (as described in Section 2.1) of the functor $F\,X = X \to X$ and hence no guarantee of the existence of greatest or least fixed points. Positivity is not the only method of ensuring that such fixed-points exist, it is possible to show that other restrictions will also lead to consistent proofs. In System F, by means of Church encodings (as in Appendix B), all terms which have the form of a constructor must be demonstrated explicitly. This obviates the problem as System F is strongly normalising. In fact, it is possible to exhibit Church encodings for recursive types which violate the positivity condition, but which do not lead to inconsistency, a fact guaranteed by the strong normalisation of System F (see AppendixC).

With our terms and the static description of the language we also need to present our dynamic rules. We will first describe a series of *experiments*, or *atomic contexts* which correspond to *elimination rules* and which will induce a deterministic reduction strategy. This is in contrast to our former description of the evaluation relation for System F which was not deterministic as it included both call-by-name and call-by-value reduction. The experiments make use of a privileged variable $(-)$ which is the redex. The Experiments are presented in Figure 2.11.

Experiments

$$\begin{array}{lll} E[-] := & - \, s & \text{case} - \text{of}\ \{x \Rightarrow r \mid y \Rightarrow s\} \\ & - \, A & \text{split} - \text{as}\ (x, y)\ \text{in}\ r \\ & \text{out}_\alpha(-, A) & \end{array}$$

Figure 2.11: System F$^+$ Experiments

These atomic experiments induce the deterministic (functional) reduction relation given in Figure 2.12. The left hand side of the reductions are *induced* in the sense that inversion on the typing relation allows only the terms provided as redexes to the experiments. Inversion of the typing relation is simply using the formation rules for typing to determine what restrictions are present on the antecedents. It is therefore a property of the well-typedness of terms. We give a more comprehensive description in Section 2.5.

We write down the structural rules for evaluation which allows us to decend into the redex position over a general relation R as $(s\ \mathcal{R}^s\ t)$. We will then form the evaluation relations as the closure under this structural rule.

Using the above provided atomic experiments we can then combine experiments to form more general contexts by composition of experiments.

$$E^*[-] := E_0[\cdots E_n[-]]$$

Here we have some context $E^*[-]$ composed of a number of experiments. This description of contexts in terms of atomic experiments will allow us to characterise the *behaviour* of terms and will lead to our notion of equivalence.

In order to talk about contextual equivalence, which we will need later and discuss in Section 2.3, we will need a definition for values. This definition is *lazy* in the sense that we do not look further than the top level term constructor.

Definition 2.4.1 ($\mathcal{V}$)**.** A term v is a *value*, which we write $v \in \mathcal{V}$ if v is in the grammar: $\lambda x\!:\!A.\ r \mid \Lambda A.\ r \mid \text{in}_\alpha(r, U) \mid (r, s) \mid \text{left}(r, A + B) \mid \text{right}(r, A + B)$

Reduction Rules

$$(\lambda x : A.r)\ s \rightsquigarrow_1 r[x := s] \qquad (\Lambda X.r)\ A \rightsquigarrow_1 r[X := A]$$

$$\text{out}_\alpha(\text{in}_\alpha(r, U), U) \rightsquigarrow_1 r \qquad \texttt{f} \rightsquigarrow_\delta \Omega(\texttt{f})$$

$$\text{case left}(r, A + B) \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\} \rightsquigarrow_1 s[x := r]$$

$$\text{case right}(r, A + B) \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\} \rightsquigarrow_1 t[y := r]$$

$$\text{split } (r, s) \text{ as } (x, y) \text{ in } t \rightsquigarrow_1 t[x := r][y := s]$$

Structural Rules

$$\frac{r\ \mathcal{R}\ r'}{r\ \mathcal{R}^s\ r'} \qquad \frac{r\ \mathcal{R}^s\ r'}{r\ s\ \mathcal{R}^s\ r'\ s} \qquad \frac{r\ \mathcal{R}^s\ r'}{r\ A\ \mathcal{R}^s\ r'\ A} \qquad \frac{r\ \mathcal{R}^s\ r'}{\text{out}_\alpha(r, U)\ \mathcal{R}^s\ \text{out}_\alpha(r', U)}$$

$$\frac{r\ \mathcal{R}^s\ r'}{\text{case } r \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\}\ \mathcal{R}^s \text{ case } r' \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\}}$$

$$\frac{r\ \mathcal{R}^s\ r'}{\text{split } r \text{ as } (x, y) \text{ in } t\ \mathcal{R}^s \text{ split } r' \text{ as } (x, y) \text{ in } t}$$

Figure 2.12: System F$^+$ Evaluation

Evaluation Relations

$$\begin{array}{lll}
r \rightsquigarrow_n s & ::= & r \rightsquigarrow_1^s s \\
r \rightsquigarrow s & ::= & r \rightsquigarrow_\delta^s s \lor r \rightsquigarrow_n s \\
r\ R^+\ s & ::= & r\ R\ s \lor (\exists r'.r\ R\ r' \land r'\ R^+\ s) \\
r\ R^*\ s & ::= & r = s \lor r\ R^+\ s \\
r \Downarrow s & ::= & r \rightsquigarrow^* s \land s \in \mathcal{V} \\
r \Downarrow & ::= & \exists s.r \Downarrow s \\
r \Uparrow & ::= & \exists s.r \rightsquigarrow s \land s \Uparrow
\end{array}$$

Figure 2.13: Relations Related to Evaluation

In Figure 2.13 we write $\rightsquigarrow^+$ as the transitive closure and $\rightsquigarrow^*$ as the reflexive transitive closure of the one-step evaluation relation. We provide a short hand for reduction to a particular value s as $r \Downarrow s$, and reduction to some value as $r \Downarrow$. Divergence is written as $r \Uparrow$ and it can be established that $(r \Uparrow) \leftrightarrow \neg(r \Downarrow)$.

2.5 Inversion

We make extensive use of *inversion* on the typing relation. The basic idea is that a given type can only be introduced using particular constructors from the typing relation. This corresponds to a case analysis on which formation rules could produce a given type, with subsequent elimination of cases which would arrive at a contradiction. The inversion rules are given in Figure 2.14. The proof of each rule follows directly from the definition of the typing relation.

If $\Delta\,;\Gamma \vdash x\,:\,A$, then $\Delta \vdash \Gamma \cup \{x\!:\!A\}$ $\mathtt{Ctx}$.
If $\Delta\,;\Gamma \vdash \lambda x\!:\!A.\,t\,:\,C$ then $C = A \to B$ for some B and $\Delta\,;\Gamma \cup \{x\!:\!A\} \vdash t\,:\,B$.
If $\Delta\,;\Gamma \vdash s\;t\,:\,C$ then $\Delta\,;\Gamma \vdash s\,:\,A \to C$ and $\Delta\,;\Gamma \vdash t\,:\,A$ for some A.
If $\Delta\,;\Gamma \vdash \Lambda X.\,t\,:\,C$ then $C = \forall X.\,B$ for some B and $\Delta \cup \{X\}\,;\Gamma \vdash t\,:\,C$.
If $\Delta\,;\Gamma \vdash s[A]\,:\,C$ then $\Delta\,;\Gamma \vdash s\,:\,\forall X.\,B$ and $B = C[X := A]$ for some B.
If $\Delta\,;\Gamma \vdash (s,t)\,:\,C$ then $C = A \times B$ and $\Delta\,;\Gamma \vdash s\,:\,A$ and $\Delta\,;\Gamma \vdash t\,:\,B$ for some A, B.
If $\Delta\,;\Gamma \vdash$ split s as (x,y) in $t\,:\,C$ then $\Delta\,;\Gamma \vdash s\,:\,A \times B$ and
    $\Delta\,;\Gamma \cup \{x\!:\!A\} \cup \{y\!:\!B\} \vdash t\,:\,C$ for some A, B.
If $\Delta\,;\Gamma \vdash$ left$(t, A + B)\,:\,C$ then $C = A + B$ and $\Delta\,;\Gamma \vdash t\,:\,A$.
If $\Delta\,;\Gamma \vdash$ left$(t, A + B)\,:\,C$ then $C = A + B$ and $\Delta\,;\Gamma \vdash t\,:\,A$.
If $\Delta\,;\Gamma \vdash$ case r of $\{x \Rightarrow s \mid y \Rightarrow t\}\,:\,C$ then $\Delta\,;\Gamma \vdash r\,:\,A + B$ and
    $\Delta\,;\Gamma \cup \{x\!:\!A\} \vdash s\,:\,C$ and $\Delta\,;\Gamma \cup \{y\!:\!B\} \vdash s\,:\,C$.
If $\Delta\,;\Gamma \vdash \mathtt{f}\,:\,C$ then $\Xi(\mathtt{f}) = C$.

Figure 2.14: Inversion

2.6 Explicit Substitution

In the following chapter we need substitutions in order to make precise our notion of a cyclic proof, and to that end we produce a typing sequent for explicit substitutions. This typing will demonstrate not only that every term of the type is correct but also demonstrates explicitly the source and target contexts. We show the formation rules for typed explicit substitutions in Figure 2.15. Our presentation follows on work done by Pfenning [67] and Chapman [15].

The empty substitution, $(\cdot)$ leaves a term unchanged and if a term is well typed in a context, then the application of the identity substitution will likewise leave the variable context unaffected. For this reason the type of the substitution is the same as the variable context in which it can be applied.

A term substitution is a pair of a variable and a term, composed with some substitution θ. It takes us from a context in which the variable is present in the context at type A and the term has a typing derivation with type A in a context which the free variable is removed, but some further number of free variables are introduced as Γ'. It additionally takes another substitution having the same context for application ($\Gamma \cup \Gamma'$) as is needed after substitution of the term t and having a target context consonant with the final substitution type (Γ'').

The situation is essentially the same for type substitutions, excepting that we must demonstrate the well formedness of the type A rather than a typing derivation.

In order to apply an explicit substitution we use the function $\Leftarrow$: $\mathbf{Term} \to \mathbf{Sub} \to \mathbf{Term}$. This will be needed when we want to establish definitional equality of a term with another term after the application of a substitution. We define this application over both terms and types as well as sequents. The application of a substitution over a sequent will have the type and term variable

Explicit Substitutions

$\mathbf{Sub} \ni \sigma, \theta$ Substitutions

Substitutions Formation

$$\Delta\,;\Gamma \vdash \cdot : «\Delta\,;\Gamma» \ \mathtt{sub}$$

$$\frac{\Delta\,;\Gamma \cup \Gamma' \vdash \theta : «\Delta\,;\Gamma''» \ \mathtt{sub} \qquad \Delta\,;\Gamma \cup \Gamma' \vdash t\,:\,A}{\Delta\,;\Gamma \cup \{x\!:\!A\} \vdash (x,t)\ \circ\ \theta : «\Delta\,;\Gamma''» \ \mathtt{sub}}$$

$$\frac{\Delta \cup \Delta'\,;\Gamma \vdash \theta : «\Delta''\,;\Gamma» \ \mathtt{sub} \qquad \Delta \cup \Delta' \vdash A \ \mathtt{type}}{\Delta \cup \{X\}\,;\Gamma \vdash (X,A)\ \circ\ \theta : «\Delta''\,;\Gamma» \ \mathtt{sub}}$$

Substitution Application

$$\begin{array}{lcl}
t \Leftarrow \cdot & := & t \\
t \Leftarrow (x,u)\ \circ\ \theta & := & (t[x := u]) \Leftarrow \theta \\
t \Leftarrow (X,A)\ \circ\ \theta & := & (t[X := A]) \Leftarrow \theta \\
B \Leftarrow \cdot & := & B \\
B \Leftarrow (x,u)\ \circ\ \theta & := & B \Leftarrow \theta \\
B \Leftarrow (X,A)\ \circ\ \theta & := & B[X := A] \Leftarrow \theta
\end{array}$$

Substitution on Typed Terms

$$\Delta\,;\Gamma \vdash (t\,:\,A) \Leftarrow \theta \quad := \quad \Delta\,;\Gamma \vdash t \Leftarrow \theta\,:\,A \Leftarrow \theta$$

Substitution on Sequents

$$(\Delta\,;\Gamma \vdash t\,:\,A) \Leftarrow \theta \quad := \quad \Delta'\,;\Gamma' \vdash t \Leftarrow \theta\,:\,A \Leftarrow \theta$$

where $\Delta\,;\Gamma \vdash \theta : «\Delta'\,;\Gamma'» \ \mathtt{sub}$

Figure 2.15: Explicit Substitution

contexts for the range of the substitution as specified by the formation rules for substitutions. It is important to remember that this ($\Leftarrow$) is not syntactic, it is a total function at the meta-level and therefore if we give the substitution explicitly, we can verify definitional equivalence under the application of the function.

If one sequent is equivalent to another under substitution then we call the second a substitution instance of the first. We define this formally as follows.

Definition 2.6.1 (Substitution Instance)**.** A sequent $(\Gamma' \,; \Delta' \vdash t \,:\, A)$ is said to be a substitution instance of a sequent $(\Gamma \,; \Delta \vdash s \,:\, B)$
iff $\quad ((\Gamma \,; \Delta \vdash s \,:\, B) \Leftarrow \sigma) = \Gamma' \,; \Delta' \vdash t \,:\, A$
and $\quad \Delta \,; \Gamma \vdash \sigma :$ «$\Delta' \,; \Gamma'$» **`sub`**.

The fact that the resulting sequents are well typed is a result of the preservation of types under substitution, a fact which has a mechanised proof in Chapter 7.

Chapter 3

Transition Systems

3.1 Introduction

Transition systems have proved to be a very flexible tool for representing processes. The model checking community has used them extensively for representing software and for determining if software meets some specification. They are very flexible and cope well with representing concurrency among other things.

We describe how a labelled transition system can be associated with any term in our calculus. This opens the door to describing equivalence as a bisimilarity of transition systems. In addition we relate type inhabitation to the satisfaction of properties in a temporal logic. This helps to bridge the gap between inductive and coinductive types and greatest and least fixed points for transition systems.

Formally, a transition system is a structure which consists of a collection of states and actions and a relation which associates states via some action. Formally such a system is described by a tuple as follows:

$$\mathcal{T} = (\mathcal{S}, \mathcal{A}, \delta \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}),$$

Where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions and $\delta(s, \alpha, s')$ is a relation representing potential transitions from a state s to some state s' by way of some action $\alpha \in A$.

For our purposes, sets of states will be represented by terms in the context of their associated programs, and transitions will be generated according to the operational behaviour of the program. Intuitively, we mean that the behaviour of a program makes *choices* for a calling program.

Transitions resulting from evaluation are not *observable* in the sense that they do not have any visible operational behaviour to the caller and are not distinguishable by the application of any experiment. This might seem odd in that a non-terminating program is certainly different than one which terminates. However, an infinite number of one-step evaluations can be seen as a failure to make a choice and is equivalent (that is, bisimilar) to a transition system with no further edges.

3.2 System F$^+$ Transition Systems

We first develop an alphabet of actions, given in Figure 3.1 which is directly based on the list of experiments given previously in Figure 2.11. The labelled transition relation is given by describing a relation, $\mapsto$. This relation will describe *external* transitions only, that is, those which are visible to the caller. The alphabet Act will form the edge labels of our transition systems. An arbitrary action is described with γ which may range over any of the actions.

Meta Variables

$\mathbf{MetaTerm} \ni a, b, c, d$

$\mathbf{MetaType} \ni M, N$

Action Labels

$\mathbf{ActVar} \ni \gamma$
$\mathbf{Act} \quad ::= \quad a\!:\!M \mid M \mid \textit{left} \mid \textit{right} \mid \textit{fst} \mid \textit{snd} \mid \textit{in}$

Transition Edge Formation Rules

$$\frac{\cdot\,;\cdot \vdash s : A \to B}{s \xmapsto{c:A} s\ c} \qquad \frac{\cdot\,;\cdot \vdash s : \forall X.\,A}{s \xmapsto{C} s\ C}$$

$$\text{left}(s, A+B) \xmapsto{\textit{left}} s \qquad \text{right}(s, A+B) \xmapsto{\textit{right}} s$$

$$(s,t) \xmapsto{\textit{fst}} s \qquad (s,t) \xmapsto{\textit{snd}} t$$

$$\text{in}(s, U) \xmapsto{\textit{in}} s$$

$$\frac{s \rightsquigarrow^* t \qquad t \xmapsto{\gamma} u}{s \xmapsto{\gamma} u}$$

Figure 3.1: Actions and Labelled Transitions

The application ($s\ c$) supplying a test term (c) to a term (s), manifests itself in our transition systems as the assumption that we have some term ($c\!:A$). Essentially we are describing the transition system parametrically in terms of some other transition system of appropriate type.

This transition system only displays the *external* behaviour of terms. By *external* we mean that some transition edge does not depend on data (in this case free variables). This is because our notion is formulated in terms of *closed* terms. We might introduce *internal* transition edges to describe reduction for instance, but these are not necessary for our purposes.

It should be noted that when we write down our transition systems with terms and meta-terms, we are writing down *sets* of terms rather than individual terms. This is due to the fact that the meta-terms are assumed to be closed, and therefore contain no free variables, but are not concrete terms themselves but exist in the meta-logic. They therefore stand in for any term of the appropriate type and may exhibit any behaviour which is consonant with their type.

For reasoning about transition systems it turns out to be useful that the $\rightsquigarrow$ relation is *deterministic*. By this we mean that we can order the sequence of reductions.

Theorem 3.2.1 (Evaluation is Deterministic)**.** *If* $\Delta\ ;\Gamma \vdash t\ :\ A$*, and* $t \rightsquigarrow s$ *and* $t \rightsquigarrow u$*, then* $u = s$*.*

Proof. By induction on the term t and inversion on subsequent type derivations of the sub-terms.

- Case $\mathtt{f}$: Since Ω is assumed to be a function.
- Case $(t\ s)$: Since t is unique by the inductive hypothesis and is a λ-term (or non-terminating), there is only one reduct for $t\ s$ if t terminates (the substitution of the bound variable) which is deterministic because substitution is a function (by its definition).
- Case $(t[A])$: Similarly as with $t\ s$.
- Case $(\text{case}\ t\ \text{of}\ \{x \Rightarrow r \mid y \Rightarrow s\})$: By the induction hypothesis t is either left, right or non-terminating. If t it is non-terminating, then there is no reduction, and if it is terminating we have the usual case evaluation rule leading to a substitution in either the right or left branch. In either case it is deterministic as substitution is deterministic and by the inductive hypothesis.
- Case split: Similarly as with case.
- Case out: Similarly as with case.

□

3.3 Bisimilarity

With the $\mapsto$ relation defined, we can define a notion of bisimilarity coinductively. In essence we will relate two terms if all of their actions are the same and all subterms are related. This is an application of Parks' principle [74] to co-inductively defined relations over transition systems.

In order to define bisimilarity we first define a *pre-order* $\lesssim$, which we call a *simulation* and then take $\sim$ to be the symmetrised version of the relation.

Definition 3.3.1 (Simulation)**.** A term s is said to *simulate* a term r, which we write $r \lesssim s$, if whenever there is an r' and γ such that $r \overset{\gamma}{\mapsto} r'$ then there exists an s' such that $s \overset{\gamma}{\mapsto} s'$ and $r' \lesssim s'$.

Definition 3.3.2 (Bisimilarity)**.** A term r is said to be a bisimilar to a term s, which we write $r \sim s$, if $r \lesssim s \wedge s \lesssim r$.

This gives us a definition of a coinductive equivalence relation between terms based solely on observable behaviour. We will use this as the basis of our notions of proof equivalence, which gives us great flexibility because we can manipulate pre-proofs which may in fact demonstrate non-termination without fear that we alter termination properties. This is imperative for demonstrating that our technique of showing soundness after transformation is correct.

3.4 Examples

We give a number of examples which motivate the above rules. The first example shows how case destructuring and recombination results in a bisimilar term.

Example 3.4.1. $x \sim \text{case } x \text{ of } \{y \Rightarrow \text{left}(y, A + B) \mid z \Rightarrow \text{right}(z, A + B)\}$

Proof of Example 3.4.1. Suppose that $x \Uparrow$, then case x of $\cdots \Uparrow$. Suppose that $x \Downarrow$, then $x \Downarrow \text{left}(y, A + B) \vee x \Downarrow \text{right}(z, A + B)$. If $x \Downarrow \text{left}(y, A + B)$ then under the evaluation relation, and using the associated transition rule, both the left and right sides have the same transition, namely $x \xmapsto{\textit{left}} y$. Since bisimilarity is reflexive, $y \sim y$. The argument follows symmetrically for $x \Downarrow \text{right}(z, A + B)$ and therefore they are bisimilar. $\square$

In this proof we see how we might systematically use case analysis on the metavariables of the transition system. All of this is taken care of automatically in the supercompilation algorithm of Section 5.3.

Example 3.4.2. $t \sim \lambda x\!:\!A.\, t\ x$ assuming that $\cdot\, ; \cdot \vdash t\ :\ A \to B$

In the above example we see a *principle of extensionality*. That is, we see the equivalence of a function with an abstraction of that function applied to the abstracted variable. We demonstrate bisimilarity in the following proof.

Proof of Example 3.4.2. Both terms exhibit the transition $\xmapsto{c:A}$. In the first case this leads to $t\ c$ and in the second to $(\lambda x\!:\!A.\ t\ x)\ c$. Since $(\lambda x\!:\!A.\ t\ x)\ c \rightsquigarrow t\ c$, any further transition that either exhibits will be matched by the other using the rule for forming transitions from evaluations. $\square$

3.5 Bismulation is Contextual Equivalence

We still have not shown that bisimulation is a suitable notion of equivalence over proofs. In order to do so we will relate it back to a well accepted notion of operational equivalence called *contextual equivalence*. This work is essentially the same as that given by Gordon in [32], aside from the inclusion of universal type quantification.

Definition 3.5.1 (Contextual Order and Equivalence)**.** Suppose $(\Delta\ ;\Gamma \vdash r\ :\ A)$ and $(\Delta\ ;\Gamma \vdash s\ :\ A)$ we say that r and s are contextually ordered or $(r \sqsubseteq s)$ if $(\forall C.\cdot\ ;\cdot \vdash C[r]\ :\ \mathcal{B})$ and $(\cdot\ ;\cdot \vdash C[s]\ :\ \mathcal{B})$ and $(C[r] \Downarrow t)$ then $(C[s] \Downarrow t)$. We say that the terms are contextually equivalent, or $(r \cong s)$ if the converse also holds.

Because bisimulation is concerned with behaviour, it deals with closed expressions, or expressions in the empty context. We will find it necessary to be able to think of extensions of these relations to open expressions.

First we need to be able to talk about relations over *open expressions*, or expressions with variables in a context. We do this by defining *proved expressions* and the Γ-closure of a relation. This allows us to exchange typed variables from a context with meta-variables and their typing derivations. The meta-variables will then allow us to perform inversion on the typing derivation.

Definition 3.5.2 (Proved expression)**.** A relation over proved expressions is a relation $\mathcal{R}$ which is a subset of the tuples of the form $(\Delta, \Gamma, r, s, A) = \mathbf{Rel}$ which we will write $(\Delta\ ;\Gamma \vdash r\ \mathcal{R}\ s\ :\ A)$.

Definition 3.5.3 (Γ-Closure)**.** Given a proved expression $(\Delta\ ;\Gamma \vdash r\ \mathcal{R}\ s\ :\ A)$, if $(\Delta = X_0 \cdots X_j)$ and $(\Gamma = x_0\ :\ A_0 \cdots x_i\ :\ A_i)$ with $(\Delta \vdash \Gamma\ \ \mathtt{Ctx})$ then the Γ-*closure* is the substitution $(r[\vec{x} := \vec{c}][\vec{X} := \vec{C}]\ \mathcal{R}\ s[\vec{x} := \vec{c}][\vec{X} := \vec{C}])$ for arbitrary well typed meta-terms $\vec{c}$, $\vec{C}$, with typing derivations $(\cdot\ ;\cdot \vdash c_i\ :\ A_i)$ and $(\cdot \vdash C_i\ \ \mathtt{type})$.

Definition 3.5.4 (Open Extension)**.** The open extension $\mathcal{R}^o$ of a relation $\mathcal{R}$ is defined such that $\Delta\ ;\Gamma \vdash r\ \mathcal{R}^o\ s\ :\ A$ iff for all $\vec{t}$, $r[\vec{x} := \vec{c}]\ \mathcal{R}\ s[\vec{x} := \vec{c}]$ where $\vec{x}$ are the variables of Γ for arbitrary well typed meta-terms $\vec{c}$, $\vec{C}$, with typing derivations $(\cdot\ ;\cdot \vdash c_i\ :\ A_i)$ and $(\cdot \vdash C_i\ \ \mathtt{type})$.

$$\frac{\Delta\,;\Gamma \vdash r\;\mathcal{R}\;s \;:\; A \to B \qquad \Delta\,;\Gamma \vdash t\;\mathcal{R}\;u \;:\; A}{\Delta\,;\Gamma \vdash r\;t\;\widehat{\mathcal{R}}\;s\;u \;:\; B}\,CE^{\to} \qquad \frac{\Delta \vdash \Gamma \cup \{x\!:\!A\}\ \texttt{Ctx}}{\Delta\,;\Gamma \vdash x\;\widehat{\mathcal{R}}\;x \;:\; A}\,CI^{Var}$$

$$\frac{\Delta\,;\Gamma \cup \{x\!:\!A\} \vdash t\;\mathcal{R}\;s \;:\; B}{\Delta\,;\Gamma \vdash (\lambda x\!:\!A.\,t)\;\widehat{\mathcal{R}}\;(\lambda x\!:\!A.\,s) \;:\; B}\,CI^{\to} \qquad \frac{\Delta \cup \{X\}\,;\Gamma \vdash t\;\mathcal{R}\;s \;:\; A}{\Delta\,;\Gamma \vdash (\Lambda X.\,t)\;\widehat{\mathcal{R}}\;(\Lambda X.\,s) \;:\; \forall X.\,A}\,CI^{\forall}$$

$$\frac{\Delta\,;\Gamma \vdash t\;\mathcal{R}\;s \;:\; \forall A.\,T}{\Delta\,;\Gamma \vdash t[B]\;\widehat{\mathcal{R}}\;s[B] \;:\; A[X := B]}\,CE^{\forall} \qquad \frac{}{\Delta\,;\Gamma \vdash ()\;\widehat{\mathcal{R}}\;() \;:\; \mathbb{1}}\,CI^{1}$$

$$\frac{\Delta\,;\Gamma \vdash r\;\mathcal{R}\;s \;:\; A \qquad \Delta\,;\Gamma \vdash t\;\mathcal{R}\;u \;:\; B}{\Delta\,;\Gamma \vdash (r,t)\;\widehat{\mathcal{R}}\;(s,u) \;:\; A \times B}\,CI^{\times}$$

$$\frac{\Delta\,;\Gamma \vdash r\;\mathcal{R}\;s \;:\; A}{\Delta\,;\Gamma \vdash \text{left}(r, A+B)\;\widehat{\mathcal{R}}\;\text{left}(s, A+B) \;:\; A+B}\,CI^{+}_{L}$$

$$\frac{\Delta\,;\Gamma \vdash r\;\mathcal{R}\;s \;:\; B}{\Delta\,;\Gamma \vdash \text{right}(r, A+B)\;\widehat{\mathcal{R}}\;\text{right}(s, A+B) \;:\; A+B}\,CI^{+}_{R}$$

$$\frac{\Delta\,;\Gamma \vdash t\;\mathcal{R}\;s \;:\; A[X := \alpha\hat{X}.\,A] \qquad \alpha \in \{\mu,\nu\}}{\Delta\,;\Gamma \vdash \text{in}_{\alpha}(t, C)\;\widehat{\mathcal{R}}\;\text{in}_{\alpha}(s, C) \;:\; \alpha\hat{X}.\,A}\,CI^{\alpha}$$

$$\frac{\Delta\,;\Gamma \vdash t\;\mathcal{R}\;s \;:\; \alpha\hat{X}.\,A \qquad \alpha \in \{\mu,\nu\}}{\Delta\,;\Gamma \vdash \text{out}_{\alpha}(t, C)\;\widehat{\mathcal{R}}\;\text{out}_{\alpha}(s, C) \;:\; T[X := \alpha\hat{X}.\,A]}\,CE^{\alpha}$$

$$\frac{\Delta\,;\Gamma \vdash r\;\mathcal{R}\;s \;:\; A+B \qquad \Delta\,;\Gamma \cup \{x\!:\!A\} \vdash t\;\mathcal{R}\;u \;:\; C \qquad \Delta\,;\Gamma \cup \{y\!:\!B\} \vdash v\;\mathcal{R}\;w \;:\; C}{\Delta\,;\Gamma \vdash (\text{case } r \text{ of } \{x \Rightarrow t \mid y \Rightarrow s\})\;\widehat{\mathcal{R}}\;(\text{case } s \text{ of } \{x \Rightarrow u \mid y \Rightarrow w\}) \;:\; C}\,CE^{+}$$

$$\frac{\Delta\,;\Gamma \vdash r\;\mathcal{R}\;s \;:\; A \times B \qquad \Delta\,;\Gamma \cup \{x\!:\!A\} \cup \{y\!:\!B\} \vdash t\;\mathcal{R}\;u \;:\; C}{\Delta\,;\Gamma \vdash (\text{split } r \text{ as } (x,y) \text{ in } t)\;\widehat{\mathcal{R}}\;(\text{split } s \text{ as } (x,y) \text{ in } u) \;:\; C}\,CE^{\times}$$

Figure 3.2: Compatible Refinement

Definition 3.5.5 (Compatible Refinement)**.** The *Compatible Refinement* $\widehat{\mathcal{R}}$ of a relation $\mathcal{R}$ is defined inductively in Figure 3.2. The compatible refinement ensures that the formation rules for derivations are mirrored by the relation.

Definition 3.5.6 (Precongruence)**.** A relation $\mathcal{R}^o$ is said to be a *precongruence* if it contains its compatible refinement, that is $\widehat{\mathcal{R}} \subseteq \mathcal{R}^o$.

Definition 3.5.7 (Congruence)**.** A *congruence* is a precongruence which is also an equivalence relation.

We will introduce a Lemma which will allow us to describe relations which are pre-congruences using an alternative but more natural formulation. Because we need to describe variables which may be relevant locally in a context, we use the notation $\Delta'; \Gamma' -_A$ to denote the whole with locally relevant contexts from Δ' and Γ'.

Lemma 3.5.8 (Congruence Lemma)**.** *Assuming that $\mathcal{R}^o$ is a preorder then $\mathcal{R}^o$ is a precongruence iff:*

$$\frac{\Delta\,;\Gamma \vdash C[\Delta';\Gamma'-_A] \,:\, B \qquad \Delta \cup \Delta'\,;\Gamma \cup \Gamma' \vdash r\;\mathcal{R}^o\;s \,:\, A}{\Gamma \vdash C[r]\;\mathcal{R}^o\;C[s] : B}\;\textit{Cong}$$

Where Δ' and Γ' are the bound variables in C whose scope includes the hole $-_A$.

Proof. We proceed by induction on the structure of $C[-_A]$ followed by inversion on its derivation. First we prove $(\rightarrow)$:

- $C[-_A] = -_A$: $\Delta' = \cdot$ and $\Gamma' = \cdot$ and therefore the conclusion $\Delta\,;\Gamma \vdash C[r]\;\mathcal{R}^o\;C[s] \,:\, A$ is simply a restatement of the hypothesis.

- $C[-_A] = C'\;t$: By the inductive hypothesis we have that $\Delta\,;\Gamma \vdash C'[r]\;\mathcal{R}^o\;C'[s] \,:\, D \rightarrow B$. By compatible refinement we have that $\Delta\,;\Gamma \vdash C'[r]\;t\;\mathcal{R}^o\;C'[s]\;t \,:\, B$ and hence $\Delta\,;\Gamma \vdash C[r]\;\mathcal{R}^o\;C[s] \,:\, B$

- $C[-_A] = t[C']$: Similar to the previous argument.

- $C[-_A] = \lambda x\!:\!D.\;C'[-_A]$: This decomposition constrains the variables of Γ' to have the form $\Gamma' = \{x\!:\!D\} \cup \Gamma''$ and by inversion we obtain the derivation $\Delta\;;\{x\!:\!D\} \cup \Gamma'' \vdash C'[-_A] \,:\, B$. By the inductive hypothesis we have $\Delta\,;\Gamma \cup \{x\!:\!D\} \cup \Gamma'' \vdash C'[r]\;\mathcal{R}^o\;C'[s] \,:\, B$ and by compatible refinement we obtain the goal.

- $C[-_A] = \Lambda X.\ C'[-_A]$: This decomposition constrains the variables of Δ' to have the form $\Delta' = \{X\} \cup \Delta''$ and by inversion we obtain the derivation $\{X\} \cup \Delta''\ ; \Gamma \vdash C'[-_A]\ :\ B$. By the inductive hypothesis we have $\Delta \cup \{X\} \cup \Delta''\ ; \Gamma\ \vdash\ C'[r]\ \mathcal{R}^o\ C'[s]\ :\ B$ and by compatible refinement we obtain the goal.

- The remaining forms are similar.

For the reverse direction ($\leftarrow$):

- CI^{Var}: $\mathcal{R}^o$ is reflexive as $\mathcal{R}^o$ is a pre-order.

- CI^1: $\mathcal{R}^o$ is reflexive as $\mathcal{R}^o$ is a pre-order.

- $CE^{\rightarrow}$: Supposing we have $\Delta\ ; \Gamma \vdash s\ \mathcal{R}^o\ s'\ :\ D \rightarrow A$ and $\Delta\ ; \Gamma \vdash t\ \mathcal{R}^o\ t'\ :\ D$ then we can derive $\Delta\ ; \Gamma \vdash s\ t\ \mathcal{R}^o\ s'\ t\ :\ B$ and $\Delta\ ; \Gamma \vdash s'\ t\ \mathcal{R}^o\ s'\ t'\ :\ B$ and since $\mathcal{R}^o$ is a pre-order, by transitivity, we can derive $\Delta\ ; \Gamma \vdash s\ t\ \mathcal{R}^o\ s'\ t'\ :\ B$.

- $CE^{\forall}$: Supposing we have $\Delta\ ; \Gamma \vdash s\ \mathcal{R}^o\ t\ :\ \forall X.\ A$ and a well-formed type D. Taking the context $C[-_A] = -_A[D]$ we have $\Delta\ ; \Gamma \vdash s[D]\ \mathcal{R}^o\ s'[D]\ :\ A[X := D]$.

- $CI^{\rightarrow}$: Suppose that $\Delta\ ; \Gamma \cup \{x\!:D\} \vdash s\ \mathcal{R}^o\ t\ :\ B$. Using the context $C[-_A] = \lambda x\!:D.\ -_A$ we have that $\Delta\ ; \Gamma \vdash \lambda x\!:D.\ s\ \mathcal{R}^o\ \lambda x\!:D.\ t\ :\ B$ by the assumption Cong.

- $CI^{\forall}$: Suppose that $\Delta \cup \{X\}\ ; \Gamma \vdash s\ \mathcal{R}^o\ t\ :\ B$. Using the context $C[-_A] = \Lambda X.\ -_A$ we have that $\Delta\ ; \Gamma \vdash \Lambda X.\ s\ \mathcal{R}^o\ \Lambda X.\ t\ :\ B$ by the assumption Cong.

- The remaining forms are similar.

$\square$

Theorem 3.5.9 (Substitution preserves similarity)**.** *Given* $\Delta\ ; \Gamma \vdash u \lesssim^o u'\ :\ A$ *and* $\Delta\ ; \Gamma \cup \{x\!:A\} \vdash s \lesssim^o s'\ :\ B$ *we have* $\Delta\ ; \Gamma \vdash s[x := u] \lesssim^o s'[x := u']\ :\ B$.

Proof. We proceed coinductively by constructing the simulation for $s[x := u] \lesssim^o s'[x := u'] : B$ by induction on s and inversion on $\Delta\ ; \Gamma \cup \{x\!:A\} \vdash s \lesssim^o s'\ :\ B$, mirroring each proof-rule, save for the single case where we have a CI^{Var} with variable x. In this case build the simulation from the simulation of $u \lesssim^o u'$. $\square$

Theorem 3.5.10 (Similarity is a precongruence)**.** *Given that* $\Delta\ ; \Gamma\ \vdash\ s \lesssim^o t\ :\ A$ *it follows that Cong holds.*

Proof. Due to the congruence lemma, we can prove this by showing that $\widehat{\lesssim^o} \subseteq \lesssim^o$. We proceed coinductively, with an inversion on the formation rules for $\lesssim^o$.

- CI^{Var}: By reflexivity.
- CI^1: By reflexivity.
- $CE^{\to}$: Supposing that $\Delta\,;\Gamma \vdash s \lesssim^o s' \;:\; A \to B$ and $\Delta\,;\Gamma \vdash t \lesssim^o t' \;:\; A$ we need to show that $\Delta\,;\Gamma \vdash s\;t \lesssim^o s'\;t' \;:\; B$. If $s' \Uparrow$ then $s \Uparrow$ and both applications diverge. If s does not diverge, $s \Downarrow \lambda x\!:\!A.\; r$ for some r then $s' \Downarrow \lambda x\!:\!A.\; r'$ for some r' with $r \lesssim^o r'$ as a property of the $\lesssim^o$ relation. By the substitution lemma, we have the conclusion.
- Similarly for the other rules.

□

Theorem 3.5.11 (Similarity is a Contextual Order). $\lesssim \subseteq \sqsubseteq$

Proof. Suppose we have $s \lesssim t$ with s and t at type A and an arbitrary expression $- : A \vdash r : 1+1$. We need to show that $r[s] \Downarrow$ implies $r[t] \Downarrow$. By the congruence property it follows that $\cdot \vdash r[s] \lesssim^o r[t] : 1+1$. Hence, if $r[s] \Downarrow$ then $r[s] \xmapsto{inr} u$ or $r[s] \xmapsto{inl} u$ and by the $\lesssim^o$ relation $r[t]$ must have the same edge. Hence $r[t] \Downarrow$, since either right or left is in $\mathcal{V}$. □

Theorem 3.5.12 (Bisimulation is Contextual Equivalence). $\sim \subseteq \cong$

Proof. This follows from the fact that similarity is a contextual order by symmetry. □

Unfortunately a mechanisation of this proof has not yet been obtained, but some of the infrastructure has been provided including Γ-closure. It is hoped that a mechanisation will be completed in future work.

We suspect that it requires an axiom of excluded middle between convergent and divergent terms (a fact used in the proof here and in Gordon's proof for contextual equivalence of FPC+[32]) and is not directly possible in an intuitionistic type theory. This excluded middle could, however, be added as an axiom. We intend to fully mechanise this proof in the future. Though the unconstructive nature of the proofs is somewhat unsatisfying with such an axiom, it is unlikely to lead to problems. We don't actually require knowledge of which event occurs, only that divergence will be preserved when it occurs.

3.6 Properties of Bisimulations

The important properties of bisimulations are related to its ability to capture pre-proof equivalence as well as proof equivalence.

Lemma 3.6.1 (Simulation is Reflexive)**.** *For all terms s we have that $s \lesssim s$.*

Proof. A term, by definition, has the identical associated transition system, and therefore the same edges. □

Lemma 3.6.2 (Simulation is Transitive)**.** *Given $s \lesssim t$ and $t \lesssim r$ we have that $s \lesssim r$.*

Proof. We have as an hypothesis that $s \lesssim t$. We know then that given an edge α leaving s there is a corresponding edge leaving t and similarly, for each edge α leaving t there is corresponding edge leaving r. Hence there must be a corresponding edge α leaving r for each α leaving s. □

Lemma 3.6.3 (Bisimulation is Reflexive)**.** *For all s we have that $s \sim s$.*

Proof. By the definition of bisimulation and reflexivity of $\lesssim$. □

Lemma 3.6.4 (Bisimulation is Symmetric)**.** *Given any terms s and t such that $s \sim t$ then $t \sim s$.*

Proof. Since $s \sim t$ is composed of $s \lesssim t$ and $t \lesssim s$ it is symmetric by construction. □

Lemma 3.6.5 (Bisimulation is Transitive)**.** *Given any terms s,t and r such that $s \sim t$ and $t \sim r$ we have that $s \sim r$.*

Proof. Since $s \sim t$ and $s \sim r$ implies that $s \lesssim t$ and $s \lesssim r$ and $\lesssim$ is transitive, we can deduce that $s \lesssim r$. Similarly, since $r \lesssim t$ and $t \lesssim s$ and we have that $\lesssim$ is transitive, we can deduce that $r \lesssim s$. Finally $s \lesssim r$ and $r \lesssim s$ implies $s \sim r$. □

Theorem 3.6.6 (Bisimulation is an Equivalence Relation)**.** *Bisimulation is reflexive, symmetric and transitive.*

Proof. This follows from the above lemmata. □

Theorem 3.6.7 (Bisimulation preserves termination)**.** *Given a derivation $\cdot\,;\cdot \vdash t\,:\,A$ and $t \sim s$ then if $t \Uparrow$ then we have that $s \Uparrow$ and if $t \Downarrow$ then $s \Downarrow$*

Sketch. The proof of this theorem is quite straightforward, though somewhat detailed, as convergence leads to a behaviour by definition. Conversely, divergent terms exhibit no behaviours. A mechanisation of this proof has been done in Coq. □

3.7 Related Work

For the purposes of reasoning about functional programs, and indeed the meaning of types themselves it is useful to use transition systems as a semantic domain. This approach is related to the approach taken in process calculi such as CCS [55] and CSP [38]. The work for various process calculi generally give the transition systems directly rather than viewing them as results of an operational semantics of a programming language.

Early work on applicative bisimulation as a suitable equivalence for the lazy λ-calculus is given by Abramsky in [2]. Further work is given by Howe in [39].

The framework given here is based on the one given by Gordon in [32]. We modify Gordon's work to cope with a polymorphic functional programming language. Gordon has also presented bisimulation over other programming languages including an object calculus with sub-typing [34]. An unpublished work by Gordon deals with universal polymorphism [33].

Sands demonstrates a general framework for showing bisimulations between various term languages by means of a technique entitled Generalised Deterministic Structural Operational Semantics (GDSOS). He demonstrates a number of sound reasoning techniques which can be applied to any system which fits into this framework.

Further work involving extensions which deal with existential types have been developed by Peirce and Sumii [82]. They note problems attempting to capture existential types when using a traditional bisimulation relation. In their work bisimulations are extended to sets of relations coupled with a context. The problems noted do not appear to effect the use of coinductive types in our presentation.

The use of bisimulation in the context of programming languages with dependent types is explored by McBride in [51]. This work demonstrates the impossibility of obtaining canonical referents for term calculi which admit infinite objects.

Sangiorgi, Kobayashi and Sumii describe *environmental bisimulations* for a higher-order language in [75]. This approach does not use the simple applicative bisimulations presented here. In languages with side-effects, resource use or existential types, environmental approaches are likely to be of interest. However, in a completely pure term language, they introduce unnecessary complexity. For a further discussion of the relative merits of either approach see [44].

Chapter 4

Cyclic Proof

4.1 Introduction

Cyclic proof is a method of describing proofs in a finite way using recurrences. The technique of cyclic proof turns out to be a way of presenting inductive and coinductive arguments and a convenient method of generically describing recursion. It has a number of advantages over the use of standard inductive definitions in constructive type theory.

Firstly, we are able to represent inductive arguments and coinductive arguments in much the same way and using the same framework. In addition, instead of seeing induction rules as connected directly with a particular recursive term we can be much more fluid in our understanding of the relationship between the type and the eventual principle of induction used by the term. We give a simple example of this in Section 4.7.

The use of cyclic proof also allows us to defer arguments about the soundness of our (co)induction principle until after we have performed rewrites. That is, if we establish that two proofs are the same up to bisimilarity, the notion of equivalence defined in Chapter 3. If one proof is sound, then so is the other.

We separate cyclic proofs into two classes: pre-proofs and proofs. The former is a superset of the latter. The proofs will be defined by a syntactic condition on the form of the proof which we give in Section 4.6. The pragmatic value in separating pre-proof from proof is that general programming constructs naturally fall into the classification of pre-proof whereas *total*[89] programs are described with proofs. This could be a boon in software engineering practice where it might not be possible to express certain sections of a program in a *total* fashion.

The approach described here does not even require soundness of individual sections of a proof which might be composed. The composition may in fact prove sound, though the components are not. To see how this can be true, consider for example the following program (in Haskell):

```
data Nat = Zero | Succ Nat deriving Show

filter :: (a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x : xs) =
  case f x of
    True -> x : (filter f xs)
    False -> filter f xs

from :: Nat -> [Nat]
from n = n : (from (Succ n))

nats :: [Nat]
nats = from Zero

odd :: Nat -> Bool
odd Zero = False
odd (Succ Zero) = True
odd (Succ (Succ n)) = odd n

test = filter odd nats
```

The *filter* function is not productive on colists for arbitrary functions f. We can provide the example of the constantly false function to demonstrate that this is the case. However there is absolutely nothing wrong with *test*, which in fact is productive, due to the specific character of the combination of *odd* and *nats*. This demonstrates how soundness can be a fact which is only true in composition.

By contrast, in a language such as Coq, we are forbidden from even writing a pre-proof, since we might *infect* the proof system with unsound arguments. We cannot make use of auxiliary functions which may be sound in a context but which are not sound in general and some compositions of productive functions will not be possible even though they are demonstrably productive.

Cyclic proof additionally gives us new ways of connecting our understanding of proof with practical programming languages. Typically, in functional programming languages, type checking for defined functions is done by use of a typing rule that assumes the type of the function and proceeds to check the body. This is the familiar rule from programming languages such as Haskell

[32] [70] [65]. An example of such a typing rule, which we call here $FunRec$ is as follows:

$$\frac{\Gamma, f : A \to B \vdash \Omega(f) : A \to B}{\Gamma \vdash f : A \to B}\, FunRec$$

Coupled with restrictions on the form of recursive terms to ensure (co)termination (where cotermination is understood to mean productivity), this rule can in fact be sound. However, it is also *opaque* in the sense that any transformation of this proof tree will be rigidly expressed in terms of the original function declarations.

A cyclic pre-proof essentially mirrors the type checking method of assuming recursive function types given above. Intuitively, the method given above is to assume the type of a recursive function, place it in the context, begin to check its body, and then unify with its subsequent occurrence. Our cyclic pre-proof makes use of three contexts, the type variable context, the term variable context and a new third context which keeps a type sequent holding the type term and variable contexts. The same rule as given above can be presented schematically (we can only do so schematically since we do not know the structure of $\Omega(f)$) as follows:

$$\cfrac{\cfrac{\cfrac{\Delta\,;\Gamma \vdash \cdot : «\Delta\,;\Gamma» \;\; \texttt{sub}}{\zeta \cup \{\Delta\,;\Gamma \vdash f \,:\, A \to B\} \cup \zeta'\,;\Delta\,;\Gamma \vdash (f \,:\, A \to B) \Leftarrow \cdot}\,\text{Cyc}}{\vdots}}{\cfrac{(\zeta \cup \{\Delta\,;\Gamma \vdash f \,:\, A \to B\})\,;\Delta\,;\Gamma \vdash \Omega(f) \,:\, A \to B}{\zeta\,;\Delta\,;\Gamma \vdash f \,:\, A \to B}\, I^{\Omega}}$$

In this case the substitution $\cdot$ is the empty substitution as we are simply directly re-expressing the sequent from the former term. In general the resulting term and contexts can change under the application of the substitution. We will introduce this rule in more detail in the next section, but this example demonstrates that anything currently well typed using the usual methods will have at least a pre-proof in our system.

Gentzen showed that it was possible in some logical systems to remove cuts by providing a rewriting of the proof [28]. If cuts can be entirely eliminated in all cases, the logical system is said to admit a cut elimination procedure. We take this observation and the notion of cyclic proof and couple it with proof rewriting using our evaluation relation. This allows us to remove intermediate implication elimination steps. While we cannot expect there to be a general implication-elimination procedure we can remove some intermediate computations. Perhaps more importantly, it is possible to use this fact to rework pre-proofs into forms in which soundness is syntactically apparent.

Our presentation of cyclic proof is related to the one given by Brotherston [13]. We however present our system as a type theory using the rules from Figure 2.10. Cycles in the proof are presented as *substitution instances* of prior sequents, which are kept in a context. The present method follows on work presented by us in [52]. Since in Brotherston's system the terms are not explcit the technical details are somewhat different. We want to preserve the behaviour of the orginal computation. Proof irrelevance is not available to us and so the structure of the cycles are more constrainted. Simply obtaining a correct type under substitution is insufficient.

In order to construct a proof, we initially create a *pre-proof*, that is, a cyclic derivation which is not guaranteed to be sound, but is instead *weakly sound*, a condition described in Section 4.3. In order to show that this pre-proof is in fact a proof we will require additional evidence. The additional evidence is given as a structural or syntactic condition on the pre-proof structure. We then take the pre-proof as being a proof. That this in fact leads to soundness is not demonstrated until Chapter 6 as it requires the development of a semantic theory.

4.2 Cyclic Proof Rules

The presentation of cyclic proof is given by reuse of the proof rules from Figure 2.10. We extend each of these rules with a context which stores the history of sequents encountered. We represent the cyclic proofs with a relatively simple repeating structure with subsequent terms which are *instances* of previous sequents. Instances are sequents which are definitionally equal after the application of a well-typed substitution. We will call the rule that introduces this cyclicity a Cyc rule.

Sequent Contexts

$$\mathbf{SeqCtx} \ni \zeta \quad := \quad \cdot \mid (\Delta\,;\Gamma \vdash t\,:\,A) \cup \zeta$$

Cyclic Proof Rules

$$\frac{\Delta\,;\Gamma \vdash \sigma : «\Delta'\,;\Gamma'» \ \texttt{sub}}{\zeta \cup \{\Delta\,;\Gamma \vdash t\,:\,A\} \cup \zeta'\,;\Delta'\,;\Gamma' \vdash (t\,:\,A) \Leftarrow \sigma}\ \mathrm{Cyc}(\Delta\,;\Gamma \vdash t\,:\,A)$$

$$\frac{\zeta \cup \{\star\}\,;\Delta'\,;\Gamma' \vdash s\,:\,B}{\zeta\,;\Delta\,;\Gamma \vdash t\,:\,A\ \texttt{label}\ \star}\ \mathrm{Label} \qquad \text{Where } \star = \Delta\,;\Gamma \vdash t\,:\,A$$

Figure 4.1: Cyclic Proofs

A Cyc Rule is a rule formed by a well typed substitution $\Delta\,;\Gamma \vdash \sigma : «\Delta'\,;\Gamma'»$ **sub** and a sequent which is a substitution instance in ζ.

A Label Rule is a shorthand convenience that allows to present proofs without keeping every sequent in the sequent context. As such, the rule simply extends the sequent context of some already present inference step. The rule allows us to introduce a cycle by placing the current sequent into the context such that it is associated with a name. This cycle may in fact be vacuous, we will not prohibit the construction of such cycles but instead they will have to be global restrictions on the proof tree to be discussed later.

$$\begin{array}{c}\dfrac{\Delta \,;\Gamma \vdash \sigma : «\Delta' \,;\Gamma'» \;\; \mathtt{\textbf{sub}}}{\zeta \cup \{\star\} \cup \zeta' \,;\Delta' \,;\Gamma' \vdash (s \,:\, A) \Leftarrow \sigma}\,\mathrm{Cyc}(\star) \\ \vdots \\ \hline \zeta \,;\Delta \,;\Gamma \vdash s \,:\, A \;\mathtt{\textbf{label}}\;\star\end{array}$$

Figure 4.2: Cyc Rule

An example of the form of a proof which makes use of the Cyc Rule is presented in Figure 4.2. We notate the sequent which is added to the context with the label (in this case $\star$) which helps us to keep track of the sequent without having to write it out long hand each time.

The advantage of the cyclic proof form is that it gives us more freedom about the way in which we would like to create recurrences and the function constants act as mere labels in the terms, rather than dictating the form of the proof. The Cyc rule allows us to have parametric circularities.

We could have used a form in which only *renamings* of variables were used, instead of the more general instantiation. This can be easily introduced by allowing abstraction to terms equivalent modulo β-reduction. However, the form as it is presented here simplifies the syntactic conditions for demonstrating that the pre-proof is in fact a proof.

It is important to understand that in our discussion of terms, we do not eliminate function constants from the term under scrutiny. Instead, we want to create cycles which are not *dictated* by the function constant definitions at our disposal. The reason for doing so is that we want to be freed from a certain level of the bureaucracy of syntax imposed by these definitions, but do not yet know what will actually free us. It is only through the process of unfolding the definitions that we become liberated from this bureaucracy. This is in some sense the *essence* of the approach of supercompilation.

We can also transform the general proof rule into the more common form using function constants as is done in standard functional programming languages. This is done by generating a new function Ω' with a new term and type which allows us to represent the instance. That this rule is justified as a replacement for instances is demonstrated by a systematic process of constructing functions from cyclic proofs, given by *reification* (Section 5.4).

4.2.1 Cyclic Type-Checking

Our implementation assumes that we have a term under scrutiny together with functions Ω and Ξ, all of which allow us to ensure that the term is type-correct. In the implementation this is done by using the I^{Ω} rule. This version requires the use of the function Ξ since type-inference would otherwise be undecidable for our proof-system. The form of the rule is given as:

$$\frac{\Xi(f) = A}{\Delta\ ; \Gamma \vdash f\ :\ A}\ I^{\Omega}$$

Of course, as it stands it merely asserts the type correctness of function constants. For this to be useful we have to know that all of the terms in Ω type-check and are consistent (syntactically equal) to the type given by Ξ. This is expressed in Definition 4.2.1. This rule is however, formally equivalent to using a cyclic proof as we can see in Theorem 4.2.2.

We briefly describe how to produce a correct proof using the FunRec rule. The function Ω is type-checked first. This involves type-checking each term of Ω. If the type of the proof of Ω is the same as the type given by Ξ for every term in Ω than the program type-checks. We now check the term under scrutiny, t, using the FunRec rule as well. If this type-checks, the program is considered to be type-correct, and will have a valid cyclic pre-proof.

In Figure 4.3 and Figure 4.4 we use the notation D_{type} to denote the type of a given derivation D. We assume that the partial function $\mathcal{T}$ exists which follows from the fact that type-checking is decidable for System F and inference is decidable given suitable annotations. We annotate type folding and unfolding, injection into disjunctions as well as function constants.

More formally, we present the type-checking constraint on programs in Definition 4.2.1.

Definition 4.2.1 (Program Law)**.** A program is said to be type-correct if

$$\forall (f, s) \in \Omega.\ \cdot\ ; \cdot \vdash s\ :\ \Xi(f)$$

and the term under scrutiny t has a derivation $\cdot\ ; \cdot\ \vdash\ t\ :\ A$. We introduce the partial function $\mathcal{T}[\Delta, \Gamma, t]$ which produces the derivation $\Delta\ ; \Gamma \vdash\ t\ :\ A$ when it exists, as given in Figure 4.3 and Figure 4.4.

This implies that t has a valid cyclic pre-proof, which we describe formally in Theorem 4.2.2.

$$\mathcal{T}[\Delta,\Gamma,x] \quad := \quad \text{if } \Delta \vdash \Gamma \cup \{x\!:\!A\} \ \texttt{Ctx}$$

$$\text{then } \frac{\Delta \vdash \Gamma \cup \{x\!:\!A\} \ \texttt{Ctx}}{\Delta\,;\Gamma \cup \{x\!:\!A\} \vdash x \,:\, A} I^{Var} \text{ else } Fail$$

$$\mathcal{T}[\Delta,\Gamma,\mathtt{f}] \quad := \quad \frac{\Xi(f) = A}{\Delta\,;\Gamma \vdash \mathtt{f} \,:\, A} I^{\Omega}$$

$$\mathcal{T}[\Delta,\Gamma,()] \quad := \quad \frac{}{\Delta\,;\Gamma \vdash () \,:\, \mathbb{1}} I^{1}$$

$$\mathcal{T}[\Delta,\Gamma,(t,s)] \quad := \quad \text{let } D^1 = \mathcal{T}[\Delta,\Gamma,t] \ D^2 = \mathcal{T}[\Delta,\Gamma,s]$$

$$\text{in } \frac{D^1 \quad D^2}{\Delta\,;\Gamma \vdash (t,s) \,:\, D^1_{type} \times D^2_{type}} I^{\times}$$

$$\mathcal{T}[\Delta,\Gamma,\text{left}(t,A+B)] \quad := \quad \text{let } D = \mathcal{T}[\Delta,\Gamma,t]$$

$$\text{in if } D_{type} = A \text{ then } \frac{D}{\Delta\,;\Gamma \vdash \text{left}(t,A+B) \,:\, A+B} I^{+}_{L}$$

$$\text{else } Fail$$

$$\mathcal{T}[\Delta,\Gamma,\text{right}(t,A+B)] \quad := \quad \text{let } D = \mathcal{T}[\Delta,\Gamma,t]$$

$$\text{in if } D_{type} = B \text{ then } \frac{D}{\Delta\,;\Gamma \vdash \text{right}(t,A+B) \,:\, A+B} I^{+}_{R}$$

$$\text{else } Fail$$

$$\mathcal{T}[\Delta,\Gamma,\text{in}_{\alpha}(t,\alpha\hat{X}.\ A)] \quad := \quad \text{let } D = \mathcal{T}[\Delta,\Gamma,t]$$

$$\text{in if } D_{type} = A[X := \alpha\hat{X}.\ A]$$

$$\text{then } \frac{D}{\Delta\,;\Gamma \vdash \text{in}_{\alpha}(t,\alpha\hat{X}.\ A) \,:\, \alpha\hat{X}.\ A} I^{\alpha}$$

$$\text{else } Fail$$

Figure 4.3: FunRec Pre-Proof

$$\begin{array}{lcl}
\mathcal{T}[\Delta,\Gamma,\Lambda X.t] & := & \text{let } D = \mathcal{T}[\Delta\cup,\Gamma,t] \\
 & & \text{in } \dfrac{D}{\Gamma\vdash \Lambda X.t : \forall X.D_{type}} I^{\forall}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{T}[\Delta,\Gamma,\lambda x\!:\!A.\ t] & := & \text{let } D = \mathcal{T}[\Delta,\{(x,A)\}\cup\Gamma,t] \\
 & & \text{in } \dfrac{D}{\Delta\,;\Gamma\vdash \lambda x\!:\!A.\ t\ :\ A\to D_{type}} I^{\to}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{T}[\Delta,\Gamma,\text{split } t \text{ as } (x,y) \text{ in } s] & := & \text{let } D^1 = \mathcal{T}[\Delta,\Gamma,t] \\
 & & \quad (A\times B) = D^1_{type} \text{ or } Fail \\
 & & \quad D^2 = \mathcal{T}[\Delta,\{(x,A),(y,B)\}\cup\Gamma,s] \\
 & & \text{in } \dfrac{D^1\ D^2}{\Delta\,;\Gamma\vdash \text{split } t \text{ as } (x,y) \text{ in } s\ :\ D^2_{type}} E^{\times}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{T}[\Delta,\Gamma,\text{case } t \text{ of } \{x\Rightarrow r\mid y\Rightarrow s\}] & := & \text{let } D^1 = \mathcal{T}[\Delta,\Gamma,t]\ (A+B) = D_{type} \text{ or } Fail \\
 & & \quad D^2 = \mathcal{T}[\Delta,\{(x:A)\}\cup\Gamma,r] \\
 & & \quad D^3 = \mathcal{T}[\Delta,\{(y:B)\}\cup\Gamma,s] \\
 & & \text{in if } D^2_{type} = D^3_{type} \\
 & & \text{then } \dfrac{D^1\quad D^2\quad D^3}{\Delta\,;\Gamma\vdash \text{case } t \text{ of } \{x\Rightarrow r\mid y\Rightarrow s\}\ :\ D^2_{type}} E^{+} \\
 & & \quad \text{else } Fail
\end{array}$$

$$\begin{array}{lcl}
\mathcal{T}[\Delta,\Gamma,t\ s] & := & \text{let } D^1 = \mathcal{T}[\Delta,\Gamma,t]\ (A\to B) = D^1_{type} \text{ or } Fail \\
 & & \quad D^2 = \mathcal{T}[\Delta,\Gamma,s] \\
 & & \text{in if } D^2_{type} = A \text{ then } \dfrac{D^1\quad D^2}{\Delta\,;\Gamma\vdash t\ s\ :\ B} E^{\to} \\
 & & \quad \text{else } Fail
\end{array}$$

$$\begin{array}{lcl}
\mathcal{T}[\Delta,\Gamma,t[A]] & := & \text{let } D = \mathcal{T}[\Delta,\Gamma,t]\ \forall X.\ B = D_{type} \text{ or } Fail \\
 & & \text{in } \dfrac{D}{\Delta\,;\Gamma\vdash t[A]\ :\ B[X:=A]} E^{\forall}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{T}[\Delta,\Gamma,\text{out}_\alpha(t,\alpha\hat{X}.\ A)] & := & \text{let } D = \mathcal{T}[\Delta,\Gamma,t]\ (\alpha\hat{X}.\ A) = D_{type} \\
 & & \text{in } \dfrac{D}{\Delta\,;\Gamma\vdash \text{out}_\alpha(t,\alpha\hat{X}.\ A)\ :\ A[A:=\alpha\hat{X}.\ A]} E^{\alpha}
\end{array}$$

Figure 4.4: FunRec Pre-Proof 2

$$\begin{array}{lcl}
\mathcal{P}\left[\xi,\dfrac{\Xi(f)=A}{\Delta\,;\Gamma\vdash f\ :\ A}\text{Rec}\right] & := & \text{let } \star = \Delta\,;\Gamma\vdash f\ :\ A \text{ in} \\
 & & \text{if } \exists\zeta,\zeta'.\ \xi = (\zeta\cup\{\star\}\cup\zeta') \\
 & & \text{then } \dfrac{\Delta\,;\Gamma\vdash\cdot : «\Delta\,;\Gamma»\ \texttt{sub}}{\zeta\cup\{\star\}\cup\zeta'\,;\Delta\,;\Gamma\vdash (f\ :\ A)\Leftarrow\cdot}\text{Cyc}(\star) \\
 & & \text{else } \dfrac{\mathcal{P}[\zeta\cup\{\star\}\,;\Delta\,;\Gamma\vdash\Omega(f)\ :\ A]}{\zeta\,;\Delta\,;\Gamma\vdash\Omega(f)\ :\ A\ \texttt{label}\ \star} I^{\Omega}
\end{array}$$

$$\mathcal{P}\left[\zeta,\dfrac{D^1\cdots D^n}{S}\text{Rule}\right] \quad := \quad \dfrac{\mathcal{P}[\mathcal{F},D^1]\cdots\mathcal{P}[\mathcal{F},D^n]}{\zeta\,;S}\text{Rule}$$

Figure 4.5: FunRec Transformation to Cyclic Pre-Proof

Theorem 4.2.2. *If a term t has a proof $\cdot\ ; \cdot \vdash t\ :\ C$ using the I^{Ω} rule, it has a cyclic pre-proof of the same sequent.*

Proof. First we will type-check the function Ω constructing sub-proofs that we will then be able to reuse. For each $(f, s) \in \Omega$ we will create a pre-proof for s, namely $\cdot\ ; \cdot \vdash s\ :\ \Xi(f)$.

The algorithm is manifestly partially correct as we can construct each pre-proof using the function in Figure 4.5, starting with ζ as the empty set, to transform the current Rec proof into one of cyclic form.

The fact that this terminates, and is therefore totally correct, follows from the fact that there are only a finite number of function constants and that terms themselves must be finite.

Given that we have Ω type-checks, we can simply reuse the proofs from Ω in the construction of the proof of t replacing every occurrence of a FunRec rule with the pre-proof of the corresponding term in Ω after doing a single function constant unfold using the Delta rule. $\square$

This proof has also been mechanised in Coq. We assume in this work that all programs under consideration type-check, and therefore have a corresponding cyclic pre-proof.

4.3 Preservation and Progress

Based on the system with pre-proofs, we may already arrive at a number of results which are common for functional programming languages. Type preservation states that if we find a reduction, it is of the same type as the original term. We can combine this with a notion of progress, which states that there is always some reduction possible, with preservation to get a weak soundness result.

In order to proceed we need a basic lemma governing substitution which is utilised by the elimination rules. However in order to prove this lemma we need context *weakening* and *strengthening* lemmas. These are used to alter the form of the contexts in ways which preserve meaning. We have not included them explicitly in our proof rules for the sake of brevity.

Lemma 4.3.1 (Weakening)**.** *Given a derivation $\Delta\ ; \Gamma \cup \Gamma' \vdash t\ :\ B$ we can produce a derivation $\Delta\ ; \Gamma \cup \{x\!:\!A\} \cup \Gamma' \vdash t\ :\ B$ when $x \notin \Gamma$*

Lemma 4.3.2 (Strengthening)**.** *Given a derivation $\Delta\ ; \Gamma \cup \{x\!:\!A\} \cup \Gamma' \vdash t\ :\ B$ we can produce a derivation $\Delta\ ; \Gamma \cup \Gamma' \vdash t\ :\ B$ when $x \notin FV(t)$*

The proof of preservation requires that we can *invert* typing derivations. Essentially since types can only come from a limited number of proof rules we can prove properties by looking only at these limited proof rules for a given type.

Lemma 4.3.3 (Substitution Preserves Types)**.** *Given derivations*
$\Delta\,;\Gamma \cup \{x : A\} \cup \Gamma' \vdash r\ :\ B$ *and*
$\Delta\,;\Gamma \cup \Gamma' \vdash s\ :\ A$ *we can produce a derivation*
$\Delta\,;\Gamma \cup \Gamma' \vdash r[x := s]\ :\ B$

Sketch. The proof proceeds by induction on the structure of r.

- (I^{Var}) In the variable case we perform case analysis on the variable to determine if it is bound or free. If it is free, then we must check to see if it is equal to the substiution variable. If it is, we will return s which has type A, the same type as x by construction.

  If it is not equal, then we do not change the variable, and hence the type does not change.

  If the variable is bound then we cannot replace it, and hence the type will not change.

- ($E^{\rightarrow}$) In the case of an application we distribute substitution according to its recursive definition on both subterms. We obtain the types for the sub-terms through inversion on the type derivation. We can then use the inductive hypothesis on each sub-term, and apply ImpElim to obtain a term of the original type.

- Other rules are similar to the ImpElim case

□

Theorem 4.3.4 (Preservation)**.** *Given a derivation* $\Delta\ ;\Gamma \vdash r\ :\ A$*, then if* $r \rightsquigarrow s$ *we have that* $\Delta\,;\Gamma \vdash s\ :\ A$*.*

Sketch. Again the proof proceeds by structural induction on r. Each case proceeds by inversion on the typing derviation and application of the inductive hypothesis and then applications of the original proof rule. □

The idea of *progress* is that we can always perform some further reduction unless we have a *value* which is given in Definition 2.4.1.

Theorem 4.3.5 (Progress)**.** *Given a derivation* $\cdot\ ;\cdot \vdash t\ :\ A$ *we have that* $t \rightsquigarrow s$ *or* $t \in \mathcal{V}$*.*

Sketch. We have two cases depending on whether the term is a value or not. If the term is a value, then we are done, if it is not a value then we perform inversion on the type derivation to obtain one case for each of the various proof rules which is not a value. In this case, we apply the inductive hypothesis on the sub-term. Each sub-term will either exhibit a step of evaluation, which is progressive, or will return a value of the appropriate type (by type preservation). If it is a value of the appropriate type then we can then perform one step of evaluation to obtain a new term ($t \rightsquigarrow s$). □

4.4 Normalisation

Some changes in the form of the proof which go beyond a simple application of the reduction relation $\rightsquigarrow_n$ will turn out to be useful. Specifically these will include distributing reducing contexts such that they are closer to the producer. This can help in both finding similar terms and eliminating intermediate constructors. We would like to find forms for our expressions prior to checking for recurrences and unfolding since this will eliminate intermediate computations and simplify the structure of the program. The term-level algorithm for normalisation is given in Figure 4.7 and Figure 4.8. This function associates a new term together with its derivation such that it is bisimilar to the original term and has the same type, meeting our notion of proof-equivalence. The function is deterministic and total. This presentation follows on work presented in [53].

Definition 4.4.1 (Normal Form). A term t has normal form t', written $[\![t]\!]_N = t'$ which is given by the rules in Figure 4.7 and Figure 4.8.

Normal Forms

$$\begin{array}{lcl}
\mathcal{I} \ni i & := & x \mid \mathtt{f} \mid i\ n \mid i[A] \mid \text{out}(i, A) \\
\mathcal{V}_n \ni v & := & \lambda x{:}\,A.\ n \mid \Lambda A.\ n \mid \text{in}_\alpha(n, U) \mid (n, m) \\
 & & \mid \text{left}(n, A + B) \mid \text{right}(n, A + B) \\
\mathcal{N} \ni n, m & := & v \mid i \mid \text{case } i \text{ of } \{x \Rightarrow n \mid y \Rightarrow m\} \mid \text{split } i \text{ as } (x, y) \text{ in } n
\end{array}$$

Figure 4.6: Normalisation Grammar

We can characterise the terms after normalisation with the grammar given in Figure 4.6. A term after normalisation will be a member of this set, that is, $[\![t]\!]_N \in \mathcal{N}$. This characterisation gives some insight as to the reason for shuffling around the various elimination rules. The final form of our term is either a *value* in which subterms are normal, in which case it is in the set $\mathcal{V}_n$, or it will have no more than *one* elimination at the top level after normalisation and the sub-term is *irreducible*. The irreducible terms are in a set $\mathcal{I}$ which is a set of terms which are *stuck*. Later, when we perform supercompilation, we will use the fact that the scrutinee of an elimination is in $\mathcal{I}$.

$$
\begin{array}{lcl}
\llbracket x \rrbracket_N & := & x \\
\llbracket \texttt{f} \rrbracket_N & := & \texttt{f} \\
\llbracket () \rrbracket_N & := & () \\
\llbracket \text{left}(t, A) \rrbracket_N & := & \text{left}(\llbracket t \rrbracket_N, A) \\
\llbracket \text{right}(t, A) \rrbracket_N & := & \text{right}(\llbracket t \rrbracket_N, A) \\
\llbracket (r, s) \rrbracket_N & := & (\llbracket r \rrbracket_N, \llbracket s \rrbracket_N) \\
\llbracket \text{in}_\alpha(t, A) \rrbracket_N & := & \text{in}_\alpha(\llbracket t \rrbracket_N, A) \\
\llbracket \lambda x\!:\!A.\ r \rrbracket_N & := & \lambda x\!:\!A.\ \llbracket r \rrbracket_N \\
\llbracket \Lambda X.\ r \rrbracket_N & := & \Lambda X.\ \llbracket r \rrbracket_N \\
\llbracket r\ s \rrbracket_N & := & \text{if } \llbracket r \rrbracket_N = \lambda x\!:\!A.\ t \\
 & & \text{then } \llbracket t[x := s] \rrbracket_N \\
 & & \text{else if } \llbracket r \rrbracket_N = \text{case } t \text{ of } \{x \Rightarrow w \mid y \Rightarrow u\} \\
 & & \text{then } \llbracket \text{case } t \text{ of } \{x \Rightarrow w\ s \mid y \Rightarrow u\ s\} \rrbracket_N \\
 & & \text{else if } \llbracket r \rrbracket_N = \text{split } t \text{ as } (x, y) \text{ in } u \\
 & & \text{then } \llbracket \text{split } t \text{ as } (x, y) \text{ in } u\ s \rrbracket_N \\
 & & \text{else } \llbracket r \rrbracket_N\ \llbracket s \rrbracket_N \\
\llbracket r[A] \rrbracket_N & := & \text{if } \llbracket r \rrbracket_N = \Lambda X.\ t \\
 & & \text{else if } \llbracket r \rrbracket_N = \text{case } t \text{ of } \{x \Rightarrow w \mid y \Rightarrow u\} \\
 & & \text{then } \llbracket \text{case } t \text{ of } \{x \Rightarrow w[A] \mid y \Rightarrow u[A]\} \rrbracket_N \\
 & & \text{else if } \llbracket r \rrbracket_N = \text{split } t \text{ as } (x, y) \text{ in } u \\
 & & \text{then } \llbracket \text{split } t \text{ as } (x, y) \text{ in } u[A] \rrbracket_N \\
 & & \text{else } \llbracket r \rrbracket_N[A] \\
\llbracket \text{out}(r, A) \rrbracket_N & := & \text{if } \llbracket r \rrbracket_N = \text{in}_\alpha(t, A) \\
 & & \text{then } \llbracket t \rrbracket_N \\
 & & \text{else if } \llbracket r \rrbracket_N = \text{case } t \text{ of } \{x \Rightarrow s \mid y \Rightarrow u\} \\
 & & \text{then } \llbracket \text{case } t \text{ of } \{x \Rightarrow \text{out}_\alpha(s, A) \mid y \Rightarrow \text{out}_\alpha(u, A)\} \rrbracket_N \\
 & & \text{else if } \llbracket r \rrbracket_N = \text{split } t \text{ as } (x, y) \text{ in } s \\
 & & \text{then } \llbracket \text{split } t \text{ as } (x, y) \text{ in } \text{out}_\alpha(s, A) \rrbracket_N \\
 & & \text{else } \text{out}_\alpha(\llbracket r \rrbracket_N, A)
\end{array}
$$

Figure 4.7: Normalisation

Theorem 4.4.2 (Type Preservation for Normalisation)**.** *For a term with pre-proof derivation* $\Gamma \vdash t : A$ *we can construct a pre-proof derivation* $\Gamma \vdash \llbracket t \rrbracket_N : A$

Proof. The *partial* correctness for preservation of types is based on an inductive argument on t. Total correctness relies on a further argument about normalisation in System F. The proof follows immediately for all introduction rules. For the elimination rules we use substitution, but substitution preserves types. The remaining cases for elimination rules involve the following term distribution laws.

- *App-Split:* Given $\Delta\ ; \Gamma \vdash \text{split } t \text{ as } (x, y) \text{ in } r\ s\ :\ C$, by inversion on the derivation for this sequent we have $\Delta\ ; \Gamma \vdash \text{split } t \text{ as } (x, y) \text{ in } r\ :\ A \to C$ together with $\Delta\ ; \Gamma \vdash s\ :\ A$ for

$$
\left[\!\!\left[\begin{array}{l}\text{case } t \text{ of}\\ \quad\{\ x \Rightarrow r\\ \quad\mid y \Rightarrow s\ \}\end{array}\right]\!\!\right]_N \quad := \quad
\begin{array}{l}
\text{if } [\![t]\!]_N = \text{left}(u, A)\\
\text{then } [\![r[x := u]]\!]_N\\
\text{if } [\![t]\!]_N = \text{right}(u, A)\\
\text{then } [\![s[y := u]]\!]_N\\
\text{else if } [\![t]\!]_N = \text{case } t' \text{ of } \{w \Rightarrow r' \mid z \Rightarrow s'\}\\
\text{then } \left[\!\!\left[\begin{array}{l}\text{case } t' \text{ of}\\ \quad\{\ w \Rightarrow \text{case } r' \text{ of } \{x \Rightarrow r \mid y \Rightarrow s\}\\ \quad\mid z \Rightarrow \text{case } s' \text{ of } \{x \Rightarrow r \mid y \Rightarrow s\}\ \}\end{array}\right]\!\!\right]_N\\
\text{else if } [\![r]\!]_N = \text{split } t' \text{ as } (w, z) \text{ in } s'\\
\text{then } [\![\text{split } t' \text{ as } (x, y) \text{ in case } s' \text{ of } \{x \Rightarrow r \mid y \Rightarrow s\}]\!]_N\\
\text{else case } [\![t]\!]_N \text{ of } \{x \Rightarrow [\![r]\!]_N \mid y \Rightarrow [\![s]\!]_N\}
\end{array}
$$

$$
\left[\!\!\left[\begin{array}{l}\text{split } t \text{ as } (x, y)\\ \text{in } r\end{array}\right]\!\!\right]_N \quad := \quad
\begin{array}{l}
\text{if } [\![t]\!]_N = (s, u)\\
\text{then } [\![r[x := s]]\!]_N[y := u]]\\
\text{else if } [\![t]\!]_N = \text{case } t' \text{ of } \{x' \Rightarrow r' \mid y' \Rightarrow s'\}\\
\text{then } \left[\!\!\left[\begin{array}{l}\text{case } t' \text{ of}\\ \quad\{\ x' \Rightarrow \text{split } r' \text{ as } (x, y) \text{ in } r\\ \quad\mid y' \Rightarrow \text{split } s' \text{ as } (x, y) \text{ in } s\ \}\end{array}\right]\!\!\right]_N\\
\text{else if } [\![t]\!]_N = \text{split } t' \text{ as } (x', y') \text{ in } r'\\
\text{then } [\![\text{split } t' \text{ as } (x', y') \text{ in split } r' \text{ as } (x, y) \text{ in } r]\!]_N\\
\text{else split } [\![t]\!]_N \text{ as } (x', y') \text{ in } [\![r]\!]_N
\end{array}
$$

Figure 4.8: Normalisation (Cont.)

some A as can be seen from the following pre-proof fragment:

$$
\frac{\Delta\ ;\Gamma \vdash \text{split } t \text{ as } (x, y) \text{ in } r\ :\ A \to C \qquad \Delta\ ;\Gamma \vdash s\ :\ A}{\Delta\ ;\Gamma \vdash (\text{split } t \text{ as } (x, y) \text{ in } r)\ s\ :\ C}
$$

We can therefore construct a proof of $\Delta\ ;\Gamma\ \vdash$ split t as (x, y) in $r\ s\ :\ C$ provided we weaken the sequent $\Delta\ ;\Gamma \vdash s\ :\ A$ to $\Delta\ ;\Gamma \cup \{x\!: A, y\!: B\} \vdash s\ :\ A$ in order to obtain the proof:

$$
\frac{\Delta\ ;\Gamma \vdash t\ :\ D \times E \qquad \dfrac{\Delta\ ;\Gamma \cup \{x\!: A, y\!: B\} \vdash r\ :\ A \to C \qquad \Delta\ ;\Gamma \cup \{x\!: A, y\!: B\} \vdash s\ :\ A}{\Delta\ ;\Gamma \cup \{x\!: A, y\!: B\} \vdash r\ s\ :\ C}}{\Delta\ ;\Gamma \vdash \text{split } t \text{ as } (x, y) \text{ in } r\ s\ :\ C}
$$

- Other *App* cases are similar.
- *Split-Split:* Given $\Delta\ ;\Gamma\ \vdash$ split (split t as (x, y) in r) as (x', y') in $r'\ :\ C$, by inversion on the derivation for this sequent we have $\Delta\ ;\Gamma \vdash t\ :\ A \times B$ together with $\Delta\ ;\Gamma \cup \{x\!: A, y\!: B\} \vdash$

$r \ : \ D \times E$ and $\Delta \ ; \Gamma\{x'\!:\!D, y'\!:\!E\} \vdash r' \ : \ C$ as can be seen from the following pre-proof fragment:

$$\mathcal{E} := \Delta \ ; \Gamma \cup \{x\!:\!A, y\!:\!B\} \vdash r \ : \ D \times E$$

$$\dfrac{\dfrac{\Delta \ ; \Gamma \vdash t \ : \ A \times B \qquad \mathcal{E}}{\Delta \ ; \Gamma \vdash \text{split } t \text{ as } (x,y) \text{ in } r \ : \ D \times E} \qquad \Delta \ ; \Gamma \cup \{x'\!:\!D, y'\!:\!E\} \vdash r' \ : \ C}{\Delta \ ; \Gamma \vdash \text{split (split } t \text{ as } (x,y) \text{ in } r) \text{ as } (x',y') \text{ in } r' \ : \ C}$$

We can therefore construct a proof of $\Delta \ ; \Gamma \vdash$ split t as (x,y) in split r as (x',y') in $r' \ : \ C$ provided we weaken the sequent $\Delta \ ; \Gamma \cup \{x'\!:\!D, y'\!:\!D\} \vdash r' \ : \ C$ to $\Delta \ ; \Gamma \cup \{x\!:\!A, y\!:\!B, x'\!:\!D, y'\!:\!D\} \vdash r' \ : \ C$ in order to obtain the proof:

$$\mathcal{E} := \dfrac{\Delta \ ; \Gamma \cup \{x\!:\!A, y\!:\!B\} \vdash r \ : \ D \times E \qquad \Delta \ ; \Gamma \cup \{x\!:\!A, y\!:\!B, x'\!:\!D, y'\!:\!D\} \vdash r' \ : \ C}{\Delta \ ; \Gamma \cup \{x : A, y : B\} \vdash \text{split } r \text{ as } (x',y') \text{ in } r' \ : \ C}$$

$$\dfrac{\Delta \ ; \Gamma \vdash t \ : \ A \times B \qquad \mathcal{E}}{\Delta \ ; \Gamma \vdash \text{split } t \text{ as } (x,y) \text{ in split } r \text{ as } (x',y') \text{ in } r' \ : \ C}$$

- *Split-Case:* Given $\Delta \ ; \Gamma \vdash$ split (case t of $\{x \Rightarrow r \mid y \Rightarrow s\}$) as (x',y') in $r' \ : \ C$, by inversion on the derivation for this sequent we obtain the sequents $\Delta \ ; \Gamma \vdash t \ : \ E + F$ and $\Delta \ ; \Gamma \cup \{x\!:\!E\} \vdash r \ : \ A \times B$ and $\Delta \ ; \Gamma \cup \{y\!:\!F\} \vdash s \ : \ A \times B$ and finally $\Delta \ ; \Gamma \cup \{x'\!:\!A, y'\!:\!B\} \vdash r' \ : \ C$ which we can see from the following pre-proof fragment:

$$\mathcal{E} := \dfrac{\Delta \ ; \Gamma \vdash t \ : \ E + F \qquad \Delta \ ; \Gamma \cup \{x\!:\!E\} \vdash r \ : \ A \times B \qquad \Delta \ ; \Gamma \cup \{y\!:\!F\} \vdash s \ : \ A \times B}{\Delta \ ; \Gamma \vdash \text{case } t \text{ of } \{x \Rightarrow r \mid y \Rightarrow s\} \ : \ A \times B}$$

$$\dfrac{\mathcal{E} \qquad \Delta \ ; \Gamma \cup \{x'\!:\!A, y'\!:\!B\} \vdash r' \ : \ C}{\Delta \ ; \Gamma \vdash \text{split (case } t \text{ of } \{x \Rightarrow r \mid y \Rightarrow s\}) \text{ as } (x',y') \text{ in } r' \ : \ C}$$

Using these sequents we can weaken the sequent $\Delta \ ; \Gamma \cup \{x'\!:\!A, y'\!:\!B\} \vdash r' \ : \ C$ to $\Delta \ ; \Gamma \cup \{x\!:\!E, x'\!:\!A, y'\!:\!B\} \vdash r' \ : \ C$ and to $\Delta \ ; \Gamma \cup \{y\!:\!F, x'\!:\!A, y'\!:\!B\} \vdash r' \ : \ C$ to obtain the sequent
$\Delta \ ; \Gamma \vdash$ case t of $\{x \Rightarrow$ split r as (x',y') in $r' \mid y \Rightarrow$ split s as (x',y') in $r'\} \ : \ C$ seen in the pre-proof fragment:

$\mathcal{E} :=$

$$\frac{\Delta\,;\Gamma \cup \{x\!:\!E\} \vdash r \;:\; A \times B \qquad \Delta\,;\Gamma \cup \{x\!:\!E, x'\!:\!A, y'\!:\!B\} \vdash r' \;:\; C}{\Delta\,;\Gamma \cup \{x\!:\!E\} \vdash \text{split } r \text{ as } (x', y') \text{ in } r' \;:\; C}$$

$\mathcal{F} :=$

$$\frac{\Delta\,;\Gamma \cup \{y\!:\!F\} \vdash s \;:\; A \times B \qquad \Delta\,;\Gamma \cup \{y\!:\!F, x'\!:\!A, y'\!:\!B\} \vdash r' \;:\; C}{\Delta\,;\Gamma \cup \{y\!:\!F\} \vdash \text{split } s \text{ as } (x', y') \text{ in } r' \;:\; C}$$

$$\frac{\Delta\,;\Gamma \vdash t \;:\; E + F \qquad \mathcal{E} \qquad \mathcal{F}}{\Delta\,;\Gamma \vdash \text{case } t \text{ of } \{x \Rightarrow \text{split } r \text{ as } (x', y') \text{ in } r' \mid y \Rightarrow \text{split } s \text{ as } (x', y') \text{ in } r'\} \;:\; C}$$

- *Case-Split:* Given $\Delta\,;\Gamma \vdash \text{case split } t \text{ as } (x, y) \text{ in } r \text{ of } \{x' \Rightarrow r' \mid y' \Rightarrow s'\} \;:\; C$, by inversion we obtain the sequents $\Delta\,;\Gamma \vdash t \;:\; D \times E$ and $\Delta\,;\Gamma \cup \{x\!:\!D, y\!:\!E\} \vdash r \;:\; A + B$ and $\Delta\,;\Gamma \cup \{x'\!:\!A\} \vdash r' \;:\; C$ and $\Delta\,;\Gamma \cup \{y'\!:\!B\} \vdash s' \;:\; C$ from the following proof fragment.

  $\mathcal{E} :=$

  $$\frac{\Delta\,;\Gamma \vdash t \;:\; D \times E \qquad \Delta\,;\Gamma \cup \{x\!:\!D, y\!:\!E\} \vdash r \;:\; A + B}{\Delta\,;\Gamma \vdash \text{split } t \text{ as } (x, y) \text{ in } r \;:\; A + B}$$

  $$\frac{\mathcal{E} \qquad \Delta\,;\Gamma \cup \{x'\!:\!A\} \vdash r' \;:\; C \qquad \Delta\,;\Gamma \cup \{y'\!:\!B\} \vdash s' \;:\; C}{\Delta\,;\Gamma \vdash \text{case split } t \text{ as } (x, y) \text{ in } r \text{ of } \{x' \Rightarrow r' \mid y' \Rightarrow s'\} \;:\; C}$$

  From these we can weaken two sequents to obtain $\Delta\,;\Gamma \cup \{x\!:\!D, y\!:\!E, x'\!:\!A\} \vdash r' \;:\; C$ and $\Delta\,;\Gamma \cup \{x\!:\!D, y\!:\!E, y'\!:\!B\} \vdash s' \;:\; C$ which we can obtain the following proof fragment:

  $\Gamma' := \Gamma \cup \{x\!:\!D, y\!:\!E\}$

  $\mathcal{E} :=$

  $$\frac{\Delta\,;\Gamma' \vdash r \;:\; A + B \qquad \Delta\,;\Gamma', x'\!:\!A \vdash r' \;:\; C \qquad \Delta\,;\Gamma', y'\!:\!A \vdash s' \;:\; C}{\Delta\,;\Gamma \cup \{x\!:\!D, y\!:\!E\} \vdash \text{case } r \text{ of } \{x' \Rightarrow r' \mid y' \Rightarrow s\} \;:\; C}$$

  $$\frac{\Delta\,;\Gamma \vdash t \;:\; D \times E \qquad \mathcal{E}}{\Delta\,;\Gamma \vdash \text{split } t \text{ as } (x, y) \text{ in case } r \text{ of } \{x' \Rightarrow r' \mid y' \Rightarrow s'\} \;:\; C}$$

- *Case-Case:* Given $\Delta\,;\Gamma \vdash \text{case case } t \text{ of } \{x \Rightarrow r \mid y \Rightarrow s\} \text{ of } \{x' \Rightarrow r' \mid y' \Rightarrow s'\} \;:\; C$, by inversion we obtain the sequents $\Delta\,;\Gamma \vdash t \;:\; D + E$ and $\Delta\,;\Gamma \cup \{x\!:\!D\} \vdash r \;:\; A + B$ and $\Delta\,;\Gamma \cup \{y\!:\!E\} \vdash s \;:\; A + B$ and $\Delta\,;\Gamma \cup \{x'\!:\!A\} \vdash r' \;:\; C$ and $\Delta\,;\Gamma \cup \{y\!:\!B\} \vdash s' \;:\; C$

from the following proof fragment:

$$\mathcal{E} :=$$
$$\dfrac{\Delta\,;\Gamma \vdash t \,:\, D+E \qquad \Delta\,;\Gamma \cup \{x\!:\!D\} \vdash r \,:\, A+B \qquad \Delta\,;\Gamma \cup \{y\!:\!E\} \vdash s \,:\, A+B}{\Delta\,;\Gamma \vdash \text{case } t \text{ of } \{x \Rightarrow r \mid y \Rightarrow s\} \,:\, A+B}$$

$$\dfrac{\mathcal{E} \qquad \Delta\,;\Gamma \cup \{x'\!:\!A\} \vdash r' \,:\, C \qquad \Delta\,;\Gamma \cup \{y'\!:\!B\} \vdash s' \,:\, C}{\Delta\,;\Gamma \vdash \text{case case } t \text{ of } \{x \Rightarrow r \mid y \Rightarrow s\} \text{ of } \{x' \Rightarrow r' \mid y' \Rightarrow s'\} \,:\, C}$$

We weaken these sequents appropriately to obtain the following proof fragment:

$$\Gamma' := \quad \Gamma \cup \{x\!:\!D\}$$
$$\mathcal{E} := \quad \dfrac{\Delta\,;\Gamma' \vdash r \,:\, A+B \qquad \Delta\,;\Gamma', x'\!:\!A \vdash r' \,:\, C \qquad \Delta\,;\Gamma', y'\!:\!B \vdash s' \,:\, C}{\Delta\,;\Gamma' \vdash \text{case } r \text{ of } \{x' \Rightarrow r' \mid y' \Rightarrow s'\} \,:\, C}$$

$$\Gamma'' := \quad \Gamma \cup \{y\!:\!E\}$$
$$\mathcal{F} := \quad \dfrac{\Delta\,;\Gamma'' \vdash s \,:\, A+B \qquad \Delta\,;\Gamma'', x'\!:\!A, \vdash r' \,:\, C \qquad \Delta\,;\Gamma'', y'\!:\!B \vdash s' \,:\, C}{\Delta\,;\Gamma'' \vdash \text{case } s \text{ of } \{x' \Rightarrow r' \mid y' \Rightarrow s'\} \,:\, C}$$

$$\dfrac{\Delta\,;\Gamma \vdash t \,:\, D+E \qquad \mathcal{E} \qquad \mathcal{F}}{\begin{array}{ll}\Delta\,;\Gamma \vdash \text{case } t \text{ of} & \qquad\qquad\qquad : C \\ \qquad \{\; x \Rightarrow \text{case } r \text{ of } \{x' \Rightarrow r' \mid y' \Rightarrow s'\} & \\ \qquad \mid y \Rightarrow \text{case } s \text{ of } \{x' \Rightarrow r' \mid y' \Rightarrow s'\} & \end{array}}$$

- *Unfold-Case:* Given the sequent $\Delta\,;\Gamma \vdash \text{out}(\text{case } t \text{ of } \{x \Rightarrow r \mid y \Rightarrow s\}, \alpha\hat{X}.\,A) \,:\, A[X := \alpha\hat{X}.\,A]$, by inversion we obtain $\Delta\,;\Gamma \vdash t \,:\, D+E$ and $\Delta\,;\Gamma \cup \{x\!:\!D\} \vdash r \,:\, \alpha\hat{X}.\,A$ and $\Delta\,;\Gamma \cup \{y\!:\!E\} \vdash s \,:\, \alpha\hat{X}.\,A$ from the proof fragment:

$$\dfrac{\dfrac{\Delta\,;\Gamma \vdash t \,:\, D+E \qquad \Delta\,;\Gamma \cup \{x\!:\!D\} \vdash r \,:\, \alpha\hat{X}.\,A \qquad \Delta\,;\Gamma \cup \{y\!:\!E\} \vdash s \,:\, \alpha\hat{X}.\,A}{\Delta\,;\Gamma \vdash \text{case } t \text{ of } \{x \Rightarrow r \mid y \Rightarrow s\} \,:\, \alpha\hat{X}.\,A}}{\Delta\,;\Gamma \vdash \text{out}(\text{case } t \text{ of } \{x \Rightarrow r \mid y \Rightarrow s\}, \alpha\hat{X}.\,A) \,:\, A[X := \alpha\hat{X}.\,A]}$$

From these sequents we can construct the proof fragment:

$\mathcal{E} :=$

$$\frac{\Delta\,;\Gamma\cup\{x\!:\!D\}\vdash r\,:\,\alpha\hat{X}.\,A}{\Delta\,;\Gamma\cup\{x\!:\!D\}\vdash \text{out}(r,\alpha\hat{X}.\,A)\,:\,A[X:=\alpha\hat{X}.\,A]}$$

$\mathcal{F} :=$

$$\frac{\Delta\,;\Gamma\cup\{y\!:\!E\}\vdash s\,:\,\alpha\hat{X}.\,A}{\Delta\,;\Gamma\cup\{y\!:\!E\}\vdash \text{out}(s,\alpha\hat{X}.\,A)\,:\,A[X:=\alpha\hat{X}.\,A]}$$

$$\frac{\Delta\,;\Gamma\vdash t\,:\,D+E \qquad \mathcal{E} \qquad \mathcal{F}}{\Delta\,;\Gamma\vdash \text{case } t \text{ of } \{x\Rightarrow \text{out}(r,\alpha\hat{X}.\,A)\mid y\Rightarrow \text{out}(s,\alpha\hat{X}.\,A)\}\,:\,A[X:=\alpha\hat{X}.\,A]}$$

- *Unfold-Split:* Given the sequent $\Delta\,;\Gamma\vdash \text{out}(\text{split } t \text{ as } (x,y) \text{ in } r,\alpha\hat{X}.\,A)\,:\,A[X:=\alpha\hat{X}.\,A]$, by inversion we obtain $\Delta\,;\Gamma\vdash t\,:\,D\times E$ and $\Delta\,;\Gamma\cup\{x\!:\!D,y\!:\!E\}\vdash r\,:\,\alpha\hat{X}.\,A$ from the proof fragment:

$$\cfrac{\cfrac{\Delta\,;\Gamma\vdash t\,:\,D\times E \qquad \Delta\,;\Gamma\cup\{x\!:\!D,y\!:\!E\}\vdash r\,:\,\alpha\hat{X}.\,A}{\Delta\,;\Gamma\vdash \text{split } t \text{ as } (x,y) \text{ in } r\,:\,\alpha\hat{X}.\,A}}{\Delta\,;\Gamma\vdash \text{out}(\text{split } t \text{ as } (x,y) \text{ in } r,\alpha\hat{X}.\,A)\,:\,A[X:=\alpha\hat{X}.\,A]}$$

From these sequents we can construct the proof fragment:

$$\cfrac{\Delta\,;\Gamma\vdash t\,:\,D\times E \qquad \cfrac{\Delta\,;\Gamma\cup\{x\!:\!D,y\!:\!E\}\vdash r\,:\,\alpha\hat{X}.\,A}{\Delta\,;\Gamma\cup\{x\!:\!D\}\vdash \text{out}(r,\alpha\hat{X}.\,A)\,:\,A[X:=\alpha\hat{X}.\,A]}}{\Delta\,;\Gamma\vdash \text{split } t \text{ as } x \text{ in out}(r,\alpha\hat{X}.\,A)\,:\,A[X:=\alpha\hat{X}.\,A]}$$

□

Theorem 4.4.3 (Characterisation of Normalisation)**.** *For any term t, with derivation* $\Delta\,;\Gamma\vdash t\,:\,A$, $[\![t]\!]_N\in\mathcal{V}_n$.

Proof. First we show the partial correctness of the theorem based on the structure of the definition of the normalisation function, and the result follows from its termination.

- $[\![x]\!]_N = x$: $x\in\mathcal{I}$, therefore $x\in\mathcal{N}$.
- $[\![\mathtt{f}]\!]_N = \mathtt{f}$: $\mathtt{f}\in\mathcal{I}$ therefore $f\in\mathcal{N}$.
- $[\![()]\!]_N = ()$: $()\in\mathcal{V}_n$ therefore $()\in\mathcal{N}$.

- $[\![\text{right}(t, A)]\!]_N = ()$: $[\![t]\!]_N \in \mathcal{N}$ therefore $\text{right}([\![t]\!]_N, A) \in \mathcal{V}_n$.
- $[\![\text{left}(t, A)]\!]_N = ()$: $[\![t]\!]_N \in \mathcal{N}$ therefore $\text{left}([\![t]\!]_N, A) \in \mathcal{V}_n$.
- $[\![(r, s)]\!]_N = ([\![r]\!]_N, [\![s]\!]_N)$: $[\![r]\!]_N, [\![s]\!]_N \in \mathcal{N}$ therefore $([\![r]\!]_N, [\![s]\!]_N) \in \mathcal{V}_n$.
- $[\![\text{in}_\alpha(t, A)]\!]_N = \text{in}_\alpha([\![t]\!]_N, A)$: $[\![t]\!]_N \in \mathcal{N}$ therefore $\text{in}_\alpha(t, A) \in \mathcal{V}_n$.
- $[\![\lambda x\!:\!A.\ r]\!]_N = \lambda x\!:\!A.\ [\![r]\!]_N$: $[\![t]\!]_N \in \mathcal{N}$ therefore $\lambda x\!:\!A.\ [\![r]\!]_N \in \mathcal{V}_n$.
- $[\![\Lambda X.\ r]\!]_N = \Lambda X.\ [\![r]\!]_N$: $[\![t]\!]_N \in \mathcal{N}$ therefore $\Lambda X.\ [\![r]\!]_N \in \mathcal{V}_n$.
- $[\![r\ s]\!]_N$: There are two cases:
  - $[\![t[x := s]]\!]_N \in \mathcal{N}$.
  - else $[\![r]\!]_N \neq \lambda x\!:\!A.\ t$ and hence it must be an elimination by inversion on the typing relation, and by type preservation of normalisation. The elimination forms possible are:
    * $[\![r]\!]_N = \text{case } t \text{ of } \{x \Rightarrow w \mid y \Rightarrow u\}$: $[\![\text{case } t \text{ of } \{x \Rightarrow w\ s \mid y \Rightarrow u\ s\}]\!]_N \in \mathcal{N}$
    * $[\![r]\!]_N = \text{split } t \text{ as } (x, y) \text{ in } u$: $[\![\text{case } t \text{ of } \{x \Rightarrow w\ s \mid y \Rightarrow u\ s\}]\!]_N \in \mathcal{N}$
    * $[\![r]\!]_N = t\ u$: $[\![r]\!]_N \in \mathcal{I}$ and $[\![s]\!]_N \in \mathcal{N}$ therefore $[\![r]\!]_N\ [\![s]\!]_N \in \mathcal{I}$.
    * $[\![r]\!]_N = t[A]$: $[\![r]\!]_N \in \mathcal{I}$ and $[\![s]\!]_N \in \mathcal{N}$ therefore $[\![r]\!]_N\ [\![s]\!]_N \in \mathcal{I}$.
    * $[\![r]\!]_N = \text{out}(t, A)$: $[\![r]\!]_N \in \mathcal{I}$ and $[\![s]\!]_N \in \mathcal{N}$ therefore $[\![r]\!]_N\ [\![s]\!]_N \in \mathcal{I}$.
- The remaining cases are similar to the case for application.

□

Here we have firmly established that the types are invariant under normalisation by suitable manipulation of proof fragments obtained through inversion and occasional weakening.

However, our concept of equivalence requires that our proofs are not irrelevant up to the type but must also maintain the behavioural characteristics up to bisimilarity.

Theorem 4.4.4 (Bisimilarity of Normalisation)**.** *For any term t with derivation $\Delta\ ; \Gamma \vdash t\ :\ A$ we have that $t \sim [\![t]\!]_N$*

Proof. The proof of bisimilarity for normalisation proceeds by induction on terms.

- $v \sim [\![v]\!]_N$: Since we are working with the Γ-closure, v will be a meta-variable for a term and given that $[\![v]\!]_N := v$ they are identical.

- $f \sim [\![f]\!]_N, () \sim [\![()]\!]_N$: These cases are trivial using the definition of $[\![t]\!]_N$.

- $\text{left}(t, A) \sim [\![\text{left}(t, A)]\!]_N$: Using the definition of normalisation we have $[\![\text{left}(t, A)]\!]_N :=$ $\text{left}([\![t]\!]_N, A)$. We have the transitions $\text{left}(t, A) \xmapsto{left} t$ and $\text{left}([\![t]\!]_N, A) \xmapsto{left} [\![t]\!]_N$ hence we must show that $t \sim [\![t]\!]_N$ but this is the inductive hypothesis.

- $\text{right}(t, A) \sim [\![\text{right}(t, A)]\!]_N$: This is just as with $\text{left}(t, A)$.

- $(r, s) \sim [\![(r, s)]\!]_N$: Since $[\![(r, s)]\!]_N := ([\![r]\!]_N, [\![s]\!]_N)$ We have the transitions $(r, s) \xmapsto{fst} r$ and $([\![r]\!]_N, [\![s]\!]_N) \xmapsto{fst} [\![r]\!]_N$ which means we must show that $r \sim [\![r]\!]_N$, but this follows from the inductive hypothesis. Similarly, $(r, s) \xmapsto{snd} s$ and $([\![r]\!]_N, [\![s]\!]_N) \xmapsto{snd} [\![s]\!]_N$: which leads to the goal $s \sim [\![s]\!]_N$ again proved using the inductive hypothesis.

- $\text{in}_\alpha(t, A) \sim [\![\text{in}_\alpha(t, A)]\!]_N$: Using the definition of normalisation we obtain $\text{in}_\alpha(t, A) \sim$ $\text{in}_\alpha([\![t]\!]_N, A)$. We have the transition $\text{in}_\alpha(t, A) \xmapsto{fold} t$ and $\text{in}_\alpha([\![t]\!]_N, A) \xmapsto{fold} [\![t]\!]_N$. This leads to the inductive hypothesis $t \sim [\![t]\!]_N$.

- $\lambda x : A.r \sim [\![\lambda x : A.r]\!]_N$: Again we obtain $\lambda x : A.r \sim \lambda x : A.[\![r]\!]_N$ from the definition of normalisation. Using the transitions $\lambda x : A.r \xmapsto{@a} r[x := a]$ and $\lambda x : .[\![r]\!]_N \xmapsto{@a} [\![r]\!]_N[x :=$ $a]$ we have the goal $r[x := a] \sim [\![r]\!]_N[x := a]$ carefully choosing a from the Gamma closure, which is in fact the goal.

- $\Lambda X.r \sim [\![\Lambda X.r]\!]_N$: By the definition of normalisation we have $\Lambda X.r \sim \Lambda X.n[r]$. Using the transitions $\Lambda X.r \xmapsto{@A} r[X := A]$ and $\Lambda X.[\![r]\!]_N \xmapsto{@A} [\![r]\!]_N[X := A]$ leads to the goal $r[X := A] \sim [\![r]\!]_N[X := A]$ which is also the inductive hypothesis.

- $r\ s \sim [\![r\ s]\!]_N$: Here, we have several cases depending on the normalisation $[\![r]\!]_N$.
  - $[\![r]\!]_N = \lambda x : A.t$: Here we have that $[\![r\ s]\!]_N = [\![t[x := s]]\!]_N$ which is β-equivalent to $[\![[\![r]\!]_N\ s]\!]_N$, hence we must show that $r\ s \sim [\![t[x := s]]\!]_N$, By the inductive hypothesis we have that $r \sim [\![r]\!]_N$ hence $r \sim \lambda x : A.t$. Since $-\ s$ is an experiment $r\ s \sim$ $(\lambda x : A.t)\ s$. This gives us $r\ s \sim t[x := s]$ by β-equivalence. Using the fact that $t[x := s] \sim [\![t[x := s]]\!]_N$ we have finally that $r\ s \sim [\![t[x := s]]\!]_N$.
  - $[\![r]\!]_N = \text{case } t \text{ of } \{x \Rightarrow w \mid y \Rightarrow u\}$: If $t \xmapsto{left} t'$ then we have $w[x := t']\ s \sim$ $w[x := t']\ s[x := t']$ but since x is not free in s this is true by reflexivity. Similarly for $t \xmapsto{right} t'$. If $t \rightsquigarrow t'$ then we step once in both proofs and use the coinductive hypothesis.

- $[\![r]\!]_N = \text{split } t \text{ as } (x, y) \text{ in } u$: If $t \overset{fst}{\longmapsto} t'$ and $t \overset{snd}{\longmapsto} s'$ then we have $(u[x := t'][y := s'])\ s \sim (u[x := t'][y := s'])\ (s[x := t'][y := s'])$ but since x and y are not free in s, this is true by reflexivity.
- $[\![r]\!]_N\ [\![s]\!]_N$: Here we can simply use the composition directly.

- $r\ A \sim [\![r\ A]\!]_N$: Again we have two cases depending on the normalisation of $[\![r]\!]_N$.
  - $[\![r]\!]_N = \Lambda X.\ t$: We have that $r \sim [\![r]\!]_N$ hence $r \sim \Lambda X.\ t$. since $-\ A$ is an experiment, this gives us that $r\ A \sim t[X := A]$. Since $t[X := A] \sim [\![t[X := A]]\!]_N$ we have finally that $r\ A \sim [\![t[X := A]]\!]_N$.
  - As with the eliminations in the application case.
  - $[\![r]\!]_N\ A$: Since $r \sim [\![r]\!]_N$, and $-\ A$ is an experiment $r\ A \sim [\![r]\!]_N\ A$.

- $\text{out}(r, A) \sim [\![\text{out}(r, A)]\!]_N$: Here we have several cases depending on the normalisation $[\![r]\!]_N$.
  - $[\![r]\!]_N = \text{in}_\alpha(t, A)$: Here we have the goal $\text{out}(r, A) \sim [\![t]\!]_N$. Since $r \sim [\![r]\!]_N$ and $[\![r]\!]_N = \text{in}_\alpha(t, A)$ we have $r \sim \text{in}_\alpha(t, A)$. But then by composition $E[r] \sim E[\text{in}_\alpha(t, A)$ for some experiment E, hence we choose $E = \text{out}(-, A)$ and arive at $\text{out}(r, A) \sim \text{out}_\alpha(\text{in}_\alpha([\![t]\!]_N, A), A)$ which by the evaluation lemma is $\text{out}(r, A) \sim [\![t]\!]_N$.
  - $[\![r]\!]_N = \text{case } t \text{ of } \{x \Rightarrow s \mid y \Rightarrow u\}$: Here the goal is $\text{out}(r, A) \sim [\![\text{case } t \text{ of } \{x \Rightarrow \text{out}_\alpha(s, A) \mid y \Rightarrow \text{out}_\alpha(u, A)\}]\!]_N$. We proceed as with the elimination forms in the application case.
  - $[\![r]\!]_N = \text{split } t \text{ as } (x, y) \text{ in } s$:
  - $\text{out}_\alpha([\![r]\!]_N, A)$: Since $r \sim [\![r]\!]_N$, and $\text{out}(-, A)$ is an experiment $\text{out}_\alpha(r, A) \sim \text{out}_\alpha([\![r]\!]_N, A)$.

□

The use of the inductive hypothesis in the above proof assumes that $[\![t]\!]_N$ terminates for arbitrary terms. This means that the above proof is only partially correct. To establish total correctness we need to show termination.

Theorem 4.4.5 (Termination of Normalisation)**.** *For all terms t which have a pre-proof $\Delta\ ; \Gamma \vdash t : A$, $[\![t]\!]_N$ terminates.*

Proof. Each step in the normalisation proof applies to one of two cases. It applies to a reduction step using the evaluation relation, or it applies to subterms. The normalisation steps involving subterms are terminating since we do not unfold function constants and terms are therefore finite. The normalisation steps which perform a reduction however require that the term is smaller according to the union of the subterm relation and another relation used to show that reduction is strongly normalising for System-F given a suitable restriction on types (for instance some positivity condition). We assume that such a relation is given by our particular choice of restriction on types and we use it here to show that subsequent reduction steps are decreasing monotonically in normalisation. □

The normalisation algorithm, as described here, coupled with the proof of type preservation can easily be generalised to deal with normalisation of derivations. In the actual implementation we merely use these facts to transform the term and produce the type derivation from this transformed term. We sometimes overload the meaning of $[\![t]\!]_N$ such that it can also work on sequents and its corresponding derivation $[\![\Delta\ ; \Gamma \vdash t\ :\ A]\!]_N$.

Finally, we need a concept of information propagation. This captures the idea that the result of a computation after decomposition yields information about the variable which is decomposed.

Definition 4.4.6 (Information Propagation)**.** The following rewrites are used to propagate information for terms.

- For (case c of $\{x \Rightarrow r \mid y \Rightarrow s\}$) we have the rewrite $c :=$ left$(a, A + B)$ in r
- For (case c of $\{x \Rightarrow r \mid y \Rightarrow s\}$) we have the rewrite $c :=$ right$(b, A + B)$ in s
- For (split c as (x, y) in r) we have the rewrite $c := (a, b)$ in r
- For (out$_\alpha(c, \alpha\hat{X}.\ T)$) we have the rewrite $c :=$ in$_\alpha(a, \alpha\hat{X}.\ T)$ in the reduction context.

These rules are justified by bisimilarity. We can see that each of these cases arises directly from the transition edge corresponding to the operational behaviour of the term *or* there is some further reduction which will take place. As long as we don't skip the reductions which must take place, information propagation will preserve bisimilarity. We demonstrate this formally in Section 5.3. The full process of recursively rewriting a term t using information propagation is given by the function $\mathbb{I}[t]$.

4.5 Function Unfolding

When transforming proofs using function constants to cyclic proofs it is critical that we be able to *unfold* any function constant f and replace it with the associated body $\Omega(f)$. Once a function constant is unfolded, there are possibilities for evaluation that were not there previously. We can capture the idea of unfolding a function constant in a context of experiments E^* by performing normalisation on the term after replacing a function constant with its body associated under the Ω function. This provides us with a new derived proof rule I^δ. This rule preserves not only the types but also preserves our notion of behavioural equivalence as has was proved in Section 4.4. We show this rule in Figure 4.9.

$$\frac{\Delta\,;\Gamma \vdash \mathbb{I}[[\![E^*[\Omega(\mathtt{f})]]\!]_N] \,:\, A}{\Delta\,;\Gamma \vdash E^*[\mathtt{f}] \,:\, A}\,I^\delta$$

Figure 4.9: Function Unfolding Rule

4.6 Proofs

In order to move from weak soundness to a notion of totality, or *soundness* we need to deal with circularities in the proofs which can lead to non-terminating behaviour. To do this requires some concept of semantics. We use a notion of soundness based on the transition systems induced by the structural operational semantics of our language. Since we have not yet defined this we are unable to show that any of our pre-proofs are in fact proofs.

However, for pragmatic reasons, instead of demonstrating soundness directly we give two syntactic criteria, one for inductive and one for co-inductive types. These form sufficient conditions to ensure soundness. The criteria are similar to those widely used for correctness of total languages such as Coq and Agda. The justification for these rules is given later in Section 6.2 where we show that these syntactic criteria on pre-proofs imply soundness and can be treated as proofs.

Definition 4.6.1 (Structural Ordering)**.** A term t is said to be *less in the structural ordering* than a term s, or $t <^s s$ using the relation $<^s$ given by the inductive definition in Figure 4.10.

Definition 4.6.2 (Structural Recursion)**.** A derivation is said to be structurally recursive if for every sequent used in a Cyc rule, there exists a privileged variable x such that for all Cyc rules, with substitution σ_i, using that sequent we have that $x \in dom(\sigma_i)$ and $\sigma(x) <^s x$.

$$\frac{\text{case } r \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\}}{x \ \mathcal{S} \ r} \qquad \frac{\text{case } r \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\}}{y \ \mathcal{S} \ r}$$

$$\frac{\text{split } r \text{ as } (x, y) \text{ in } t}{x \ \mathcal{S} \ r} \qquad \frac{\text{split } r \text{ as } (x, y) \text{ in } t}{y \ \mathcal{S} \ r}$$

$$\frac{\text{out}_\alpha(t, \alpha\hat{X}.\ A)}{\text{out}_\alpha(t, \alpha\hat{X}.\ A) \ \mathcal{S} \ t}$$

$$<^s := S^* \quad \text{Transitive closure of } S$$

Figure 4.10: Structural Ordering

It should be mentioned that there is nothing in particular needed for this definition aside from some guarantee that the cycles lead to a well founded recursion scheme. As such this represents a particular implementation *strategy* and we could very well have used a more liberal approach. One such approach is *size-change termination* as described by Neil Jones et al. in [41]. This was adapted to dependent type theory by David Wahlstedt [93]. Andreas Abel and Thorsten Altenkirch have also described a similar termination checking algorithm which forms the basis of Agda's termination and productivity checker [4]. Abel has introduced a notion of a *sized type* which allows definitions which are not possible with a strictly syntactic criteria on terms[1]. It would be interesting to see if there is some relation to transformation combined with syntactic criteria. A visualisation of the definition adopted here is depicted in Figure 4.11.

$$\cfrac{\cfrac{\Delta \,; \Gamma \vdash \sigma \ \circ \ (x,t) \ \circ \ \sigma' : «\Delta' \,; \Gamma'» \ \textbf{sub}}{\zeta \cup \{\star\} \cup \zeta' \,; \Delta' \,; \Gamma' \vdash (s \ : \ A) \Leftarrow \sigma \ \circ \ (x,t) \ \circ \ \sigma'} \ \text{Cyc}(\star)}{\cfrac{\vdots}{\zeta \,; \Delta \,; \Gamma \vdash s \ : \ A \ \textbf{label} \ \star}}$$

Where $t <^s x$

Figure 4.11: Structural Recursion

Similarly, we must produce a rule for coinductive types which ensures that all terms of coinductive type are *productive*. We here develop a *guardedness* condition specific to our type theory of cyclic proofs. Essentially this condition ensures we encounter an introduction of a constructor which cannot be eliminated on all coinductive cyclic paths. The only intermediate terms must reduce finitely through eliminations of finite or inductively defined terms, ensuring that we will not compute indefinitely prior to producing a constructor.

While structural recursion is focused on determining whether the arguments of a recursive term are subterms of some previously destructured term, the dual problem is of determining if a recursive term's context ensures that the term grows. This means we need ways of describing the surrounding context of a term. However, the contexts we have developed thus far are structured in terms of *experiments*. With coinductive terms we need exactly the opposite variety of contexts, those surrounding terms which are *not* experiments.

The key important features of the contexts we are interested in turns out to be whether or not they introduce constructors, and whether they are guaranteed not to remove them. These properties are necessary in the construction of our proof that guardedness leads to *productivity*.

We can describe the relevant features of the context by describing a *path*. This *path* is a series of constructors that allows us to demonstrate which directions to take down a proof tree to arive at a recurrence.

Definition 4.6.3 (Path)**.** A *path* is a finite sequence of pairs of a proof rule from Figure 2.10 and an index denoting which antecedent it decends from. This pair is described as a *rule-index-pair*.

An example of such a path would be the following:

$$I_L^{+1}, I^{\times 2}, I^{\to 1}$$

This denotes the context:

$$\text{left}((\lambda x : B.-, s), A)$$

With some unknown (and for the purpose of proving productivity, inconsequential) variable x, term s and types A and B.

With this in hand we can establish conditions for guardedness with recursive definitions based on constraints on paths.

Definition 4.6.4 (Admissible)**.** A path is called admissible if the first element c of the path $p = c, p'$ is one of the rule-index-pairs I_L^{+1}, I_R^{+1}, $I^{\times 1}$, $I^{\times 2}$, $I^{\forall 1}$, $I^{\alpha 1}$, $I^{\to 1}$, E^{+2}, E^{+3}, $E^{\times 2}$,$E^{\forall 1}$, $I^{\Omega 1}$ and p' is an admissible path.

Definition 4.6.5 (Guardedness)**.** A path is called guarded if it terminates at a Cyc Rule, with the sequent having a coinductive type and the path can be partitioned such that $p = p', [I^{\nu 1}], p''$ and p' and p'' are admissible paths. We call $I^{\nu 1}$ the *guard*.

The idea behind the guardedness condition is that we have to be assured that as we take a cyclic path we produce an Intro rule which will never be removed by the reduction relation. The left hand-side of an elimination rule will never cause the elimination of such an introduction and so is *safe*. However, the right hand side of an elimination rule may in fact cause the removal of the introduction rule when we use the evaluation relation. Again, we give a visualisation of the definition in Figure 4.12.

$$\begin{array}{ccccc}
\dfrac{\Delta ; \Gamma \vdash \sigma_1 : \text{«}\Delta_1 ; \Gamma_1\text{»} \ \texttt{sub}}{\zeta \cup \{\mathcal{S}\} \cup \zeta_1 ; \Delta_1 ; \Gamma_1 \vdash (s : \nu\hat{X}.\, A) \Leftarrow \sigma_1} & & & & \dfrac{\Delta ; \Gamma \vdash \sigma_n : \text{«}\Delta_n ; \Gamma_n\text{»} \ \texttt{sub}}{\zeta \cup \{\mathcal{S}\} \cup \zeta_n ; \Delta_n ; \Gamma_n \vdash (s : \nu\hat{X}.\, A) \Leftarrow \sigma_n} \\
\vdots & & & & \vdots \\
\mathcal{P}_1 & \cdots & \mathcal{P}_i & \cdots & \mathcal{P}_n \\
\ddots & & \vdots & & \cdot^{\cdot^{\cdot}} \\
 & & \zeta ; \Delta ; \Gamma \vdash s : \nu\hat{X}.\, A \ \textbf{label}\ \mathcal{S} & &
\end{array}$$

Where $\mathcal{P}_i$ is guarded for i from 1 to n.

Figure 4.12: Guardedness

Using these two conditions of structural induction and guardedness we can define a proof.

Definition 4.6.6 (Proof)**.** A *proof* is a *pre-proof* of a sequent $(\zeta ; \Delta ; \Gamma \vdash t : A)$ in which every Cyc rule is either structurally recursive or productive. We will write the sequent for the proof associated with the pre-proof $(\zeta ; \Delta ; \Gamma \vdash t : A)$ as $(\zeta ; \Delta ; \Gamma \Vdash t : A)$.

4.7 Example

In order to get a better handle on how cyclic proof works practically, we present some examples of programs and their transformations. We carry out the process quite manually such that none of the considerations discussed in detail in Chapter 3 are needed.

We use over-bars to represent types which use greatest fixed points as a visual cue to the reader. The type $\overline{\mathbb{N}}$ is defined as $\nu\hat{X}.\ 1 + \hat{X}$ and represents the type of natural numbers including the point at infinity. The type $\overline{[A]}$ is defined as $\nu\hat{X}.\ 1 + (A \times \hat{X})$ and represents potentially infinite lists with elements of type A. In Figure 4.15 we give a program which calculates the length plus the sum of elements of a potentially infinite list. The example is somewhat contrived, but the addition of the length is necessary to ensure that the program is productive. An infinite stream of zeros for instance would not yield a productive sum.

We ascribe a type to each constant at the top level using the function Ξ. This serves two purposes: it allows us to avoid a bi-directional type-inferencing and type-checking algorithm, simplifying our presentation, but also helps us ensure that each constant corresponds to a unique type. The program $\Omega(\omega) := \omega$ is an example which can inhabit arbitrary types and whose type would be ambiguous if one was not ascribed.

$$
\begin{array}{lcl}
\Omega(\texttt{zero}) & := & \mathrm{in}_\nu(\mathrm{left}(, 1 + \overline{\mathbb{N}}), \overline{\mathbb{N}}) \\
\Omega(\texttt{succ}) & := & \lambda x : \overline{\mathbb{N}}.\ \mathrm{in}_\nu(\mathrm{right}(x, 1 + \overline{\mathbb{N}}), \overline{\mathbb{N}}) \\
\Omega(\texttt{plus}) & := & \lambda x\ y : \overline{\mathbb{N}}. \\
 & & \mathrm{case}\ (\mathrm{out}_\nu(x, \overline{\mathbb{N}}))\ \mathrm{of} \\
 & & \quad \{\, z\ \Rightarrow\ y \\
 & & \quad \mid n\ \Rightarrow \texttt{succ}\ (\texttt{plus}\ n\ y)\} \\
\Xi(\texttt{zero}) & := & \overline{\mathbb{N}} \\
\Xi(\texttt{succ}) & := & \overline{\mathbb{N}} \to \overline{\mathbb{N}} \\
\Xi(\texttt{plus}) & := & \overline{\mathbb{N}} \to \overline{\mathbb{N}} \to \overline{\mathbb{N}}
\end{array}
$$

Figure 4.13: Program for `plus`

The program for the term `plus` is given in Figure 4.13. The cyclic pre-proof associated with the term `plus`, is given in Figure 4.14. Since we have a cyclic pre-proof we can now attempt to show that this term is productive. We must demonstrate that all cycles meet the conditions imposed by the guardedness and structural recursion criteria given above. Since we only have one cycle, and this cycle is coinductive, we need only write down the associated path and check that it meets the condition of a guarded path. We will take the liberty of eliding elements of the context from the sequent when they are irrelevant. For instance, in the sequent $(\zeta\ ; \Delta\ ; \{x : A\} \vdash x\ :\ \mathbb{N})$, ζ and Δ are not relevant, as there is no Cyc rule and no free type variables in A.

$$p = I^{\Omega 1}, I^{\to 1}, E^{+3}, I^{\Omega 1}, I^{\nu 1}, I_R^{+1}, I^{\Omega 1}$$

We can see that the prefix $p' = I^{\Omega 1}$, $I^{\to 1}$, E^{+3}, $I^{\Omega 1}$ and the suffix $p'' = I_R^{+1}$, $I^{\Omega 1}$ are *admissible* and the $I^{\nu 1}$ is the guard. This meets the condition of guardedness and hence our pre-

$$
\dfrac{\dfrac{\dfrac{\dfrac{\cdot\ ; \{x : \overline{\mathbb{N}}, y : \overline{\mathbb{N}}\} \vdash (x, n) : «\cdot\ ; \{y : \overline{\mathbb{N}}, n : \overline{\mathbb{N}}\}»\ \texttt{\textbf{sub}}}{\{\star\}\ ; \cdot\ ; \{y : \overline{\mathbb{N}}, n : \overline{\mathbb{N}}\} \vdash (\mathrm{case}\ \mathrm{out}_\nu(x, \overline{\mathbb{N}})\ \mathrm{of}\ \{z \Rightarrow y \mid n \Rightarrow \texttt{succ}\ (\texttt{plus}\ n\ y)\}\ :\ \overline{\mathbb{N}}) \Leftarrow \{(x, n)\}}\ \mathrm{Cyc}(\star)}{\{\star\}\ ; \cdot\ ; \{y : \overline{\mathbb{N}}, n : \overline{\mathbb{N}}\} \vdash \texttt{plus}\ n\ y\ :\ \overline{\mathbb{N}}}\ I^\delta}{\{\star\}\ ; \cdot\ ; \{y : \overline{\mathbb{N}}, n : \overline{\mathbb{N}}\} \vdash \mathrm{right}(\texttt{plus}\ n\ y, 1 + \overline{\mathbb{N}})\ :\ 1 + \overline{\mathbb{N}}}\ I_R^+}{\{\star\}\ ; \cdot\ ; \{y : \overline{\mathbb{N}}, n : \overline{\mathbb{N}}\} \vdash \mathrm{in}_\nu(\mathrm{right}(\texttt{plus}\ n\ y, 1 + \overline{\mathbb{N}}), \overline{\mathbb{N}})\ :\ \overline{\mathbb{N}}}\ I^\nu \qquad (\mathcal{A})
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\{\star\}\ ; \cdot\ ; \{x : \overline{\mathbb{N}}\} \vdash x\ :\ \overline{\mathbb{N}}}{\{\star\}\ ; \cdot\ ; \{x : \overline{\mathbb{N}}\} \vdash \mathrm{out}_\nu(x, \overline{\mathbb{N}})\ :\ 1 + \overline{\mathbb{N}}}\ E^\nu \quad \{\star\}\ ; \cdot\ ; \{y : \overline{\mathbb{N}}\} \vdash y\ :\ \overline{\mathbb{N}} \quad \dfrac{\mathcal{A}}{\{\star\}\ ; \cdot\ ; \{y : \overline{\mathbb{N}}, n : \overline{\mathbb{N}}\} \vdash \texttt{succ}\ (\texttt{plus}\ n\ y)\ :\ \overline{\mathbb{N}}}\ I^\delta}{\cdot\ ; \cdot\ ; \{x : \overline{\mathbb{N}}, y : \overline{\mathbb{N}}\} \vdash \mathrm{case}\ \mathrm{out}_\nu(x, \overline{\mathbb{N}})\ \mathrm{of}\ \{z \Rightarrow y \mid n \Rightarrow \texttt{succ}\ (\texttt{plus}\ n\ y)\}\ :\ \overline{\mathbb{N}}\ \texttt{\textbf{label}}\ \star}\ E^+}{\cdot\ ; \cdot\ ; \cdot \vdash \lambda x\ y : \overline{\mathbb{N}}.\ \mathrm{case}\ \mathrm{out}_\nu(x, \overline{\mathbb{N}})\ \mathrm{of}\ \{z \Rightarrow y \mid n \Rightarrow \texttt{succ}\ (\texttt{plus}\ n\ y)\}\ :\ \overline{\mathbb{N}}}\ I^\to}{\cdot\ ; \cdot\ ; \cdot \vdash \texttt{plus}\ :\ \overline{\mathbb{N}} \to \overline{\mathbb{N}} \to \overline{\mathbb{N}}}\ I^\delta \qquad (\mathcal{B})
$$

Figure 4.14: Plus

proof is in fact a proof.

Similarly, we can use the exact same proof tree with the slight modification of the type from using the co-natural numbers $\overline{\mathbb{N}}$ to the natural numbers $\mathbb{N}$. In this case we need to show that $(n <^s x)$. This can be done simply by walking through the proof rules. From the transitivity of $<^s$ we have that $n <^s x$ as $(\mathrm{out}_\mu(x, \mathbb{N}) <^s x)$ together with $n <^s \mathrm{out}_\mu(x, \mathbb{N})$, since we have case $(\mathrm{out}_\mu(x, \mathbb{N}))$ of $\{z \Rightarrow y \mid n \Rightarrow \mathtt{succ}\ (\mathtt{plus}\ n\ y)\}$.

$$
\begin{array}{lll}
\Omega(\mathtt{sumlen}) & := & \lambda xs : \overline{[\overline{\mathbb{N}}]}. \\
 & & \text{case } (\mathrm{out}_\nu(xs, \overline{[\overline{\mathbb{N}}]})) \text{ of} \\
 & & \quad \{\ nil \Rightarrow \mathtt{zero} \\
 & & \quad \mid p \Rightarrow \\
 & & \qquad \text{split } p \text{ as } (n, xs') \text{ in } \mathtt{succ}\ (\mathtt{plus}\ n\ (\mathtt{sumlen}\ xs'))\} \\
\Xi(\mathtt{zero}) & := & \overline{\mathbb{N}} \\
\Xi(\mathtt{succ}) & := & \overline{\mathbb{N}} \to \overline{\mathbb{N}} \\
\Xi(\mathtt{plus}) & := & \overline{\mathbb{N}} \to \overline{\mathbb{N}} \to \overline{\mathbb{N}} \\
\Xi(\mathtt{sumlen}) & := & \overline{[\overline{\mathbb{N}}]} \to \overline{\mathbb{N}}
\end{array}
$$

Figure 4.15: Sumlen Program

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdot\,;\cdot \vdash \cdot : «\cdot\,;\cdot» \ \ \mathbf{sub}}{\{\dagger,\star\}\,;\cdot\,;\cdot \vdash (\mathtt{plus} : \overline{\mathbb{N}} \to \overline{\mathbb{N}} \to \overline{\mathbb{N}}) \Leftarrow \cdot}\,\mathrm{Cyc}(\star) \qquad \cdot\,;\cdot\,;\{n : \overline{\mathbb{N}}\} \vdash n : \overline{\mathbb{N}} \qquad \cdot\,;\cdot\,;\{y : \overline{\mathbb{N}}\} \vdash y : \overline{\mathbb{N}}}{\{\dagger,\star\}\,;\cdot\,;\{y : \overline{\mathbb{N}}, n : \overline{\mathbb{N}}\} \vdash \mathtt{plus}\ n\ y : \overline{\mathbb{N}}}}{\{\dagger,\star\}\,;\cdot\,;\{y : \overline{\mathbb{N}}, n : \overline{\mathbb{N}}\} \vdash \mathrm{right}(\mathtt{plus}\ n\ y, 1 + \overline{\mathbb{N}}) : 1 + \overline{\mathbb{N}}}}{\{\dagger,\star\}\,;\cdot\,;\{y : \overline{\mathbb{N}}, n : \overline{\mathbb{N}}\} \vdash \mathrm{in}_\nu(\mathrm{right}(\mathtt{plus}\ n\ y, 1 + \overline{\mathbb{N}}), \overline{\mathbb{N}}) : \overline{\mathbb{N}}}}{\{\dagger,\star\}\,;\cdot\,;\{y : \overline{\mathbb{N}}, n : \overline{\mathbb{N}}\} \vdash \mathtt{succ}\ (\mathtt{plus}\ n\ y) : \overline{\mathbb{N}}} \qquad (\mathcal{G})
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\cdot\,;\cdot\,;\{x : \overline{\mathbb{N}}\} \vdash x : \overline{\mathbb{N}}}{\cdot\,;\cdot\,;\{x : \overline{\mathbb{N}}\} \vdash \mathrm{out}_\nu(x, \overline{\mathbb{N}}) : 1 + \overline{\mathbb{N}}} \qquad \cdot\,;\cdot\,;\{y : \overline{\mathbb{N}}\} \vdash y : \overline{\mathbb{N}} \qquad \dfrac{\mathcal{G}}{\{\dagger,\star\}\,;\cdot\,;\{y : \overline{\mathbb{N}}, n : \overline{\mathbb{N}}\} \vdash \mathtt{succ}\ (\mathtt{plus}\ n\ y) : \overline{\mathbb{N}}}}{\{\dagger,\star\}\,;\cdot\,;\{x : \overline{\mathbb{N}}, y : \overline{\mathbb{N}}\} \vdash \text{case } \mathrm{out}_\nu(x, \overline{\mathbb{N}}) \text{ of } \{z \Rightarrow y \mid n \Rightarrow \mathtt{succ}\ (\mathtt{plus}\ n\ y)\} : \overline{\mathbb{N}}}}{\{\dagger,\star\}\,;\cdot\,;\cdot \vdash \lambda x\ y : \overline{\mathbb{N}}.\ \text{case } \mathrm{out}_\nu(x, \overline{\mathbb{N}}) \text{ of } \{z \Rightarrow y \mid n \Rightarrow \mathtt{succ}\ (\mathtt{plus}\ n\ y)\} : \overline{\mathbb{N}}}}{\{\dagger\}\,;\cdot\,;\cdot \vdash plus : \overline{\mathbb{N}} \to \overline{\mathbb{N}} \to \overline{\mathbb{N}}\ \mathbf{label}\ \star} \qquad (\mathcal{F})
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{F}}{\{\dagger\}\,;\cdot\,;\cdot \vdash \mathtt{plus} : \overline{\mathbb{N}} \to \overline{\mathbb{N}} \to \overline{\mathbb{N}}} \qquad \cdot\,;\cdot\,;\{n : \overline{\mathbb{N}}\} \vdash n : \overline{\mathbb{N}} \qquad \dfrac{\dfrac{\cdot\,;\cdot \vdash \cdot : «\cdot\,;\cdot» \ \ \mathbf{sub}}{\{\dagger\}\,;\cdot\,;\cdot \vdash (\cdots : \overline{[\overline{\mathbb{N}}]} \to \overline{\mathbb{N}}) \Leftarrow \cdot}\,\mathrm{Cyc}(\dagger) \qquad \cdot\,;\cdot\,;\{xs' : \overline{[\overline{\mathbb{N}}]}\} \vdash xs' : \overline{[\overline{\mathbb{N}}]}}{\{\dagger\}\,;\cdot\,;\{xs' : \overline{[\overline{\mathbb{N}}]}\} \vdash \mathtt{sumlen}\ xs' : \overline{\mathbb{N}}}}{\{\dagger\}\,;\cdot\,;\{n : \overline{\mathbb{N}}, xs' : \overline{[\overline{\mathbb{N}}]}\} \vdash \mathtt{plus}\ n\ (\mathtt{sumlen}\ xs') : \overline{\mathbb{N}}}}{\{\dagger\}\,;\cdot\,;\{n : \overline{\mathbb{N}}, xs' : \overline{[\overline{\mathbb{N}}]}\} \vdash \mathrm{right}(\mathtt{plus}\ n\ (\mathtt{sumlen}\ xs'), 1 + \overline{\mathbb{N}}) : 1 + \overline{\mathbb{N}}}}{\{\dagger\}\,;\cdot\,;\{n : \overline{\mathbb{N}}, xs' : \overline{[\overline{\mathbb{N}}]}\} \vdash \mathrm{in}_\nu(\mathrm{right}(\mathtt{plus}\ n\ (\mathtt{sumlen}\ xs'), 1 + \overline{\mathbb{N}}), \overline{\mathbb{N}}) : \overline{\mathbb{N}}}
$$

$$
\{\dagger\}\,;\cdot\,;\{n : \overline{\mathbb{N}}, xs' : \overline{[\overline{\mathbb{N}}]}\} \vdash \mathtt{succ}\ (\mathtt{plus}\ n\ (\mathtt{sumlen}\ xs')) : \overline{\mathbb{N}} \qquad (\mathcal{E})
$$

$$
\dfrac{\{\dagger\}\,;\cdot\,;\{p : (\overline{\mathbb{N}} \times \overline{[\overline{\mathbb{N}}]})\} \vdash p : (\overline{\mathbb{N}} \times \overline{[\overline{\mathbb{N}}]}) \qquad \mathcal{E}}{\{\dagger\}\,;\cdot\,;\{p : (\overline{\mathbb{N}} \times \overline{[\overline{\mathbb{N}}]})\} \vdash \text{split } p \text{ as } (n, xs') \text{ in } \mathtt{succ}\ (\mathtt{plus}\ n\ (\mathtt{sumlen}\ xs')) : \overline{\mathbb{N}}} \qquad (\mathcal{D})
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\{\dagger\}\,;\cdot\,;\{xs : \overline{[\overline{\mathbb{N}}]}\} \vdash xs : \overline{[\overline{\mathbb{N}}]}}{\{\dagger\}\,;\cdot\,;\{xs : \overline{[\overline{\mathbb{N}}]}\} \vdash \mathrm{out}_\mu(xs, \overline{[\overline{\mathbb{N}}]}) : 1 + (\overline{\mathbb{N}} \times \overline{[\overline{\mathbb{N}}]})} \qquad \{\dagger\}\,;\cdot\,;\cdot \vdash \mathtt{zero} : \overline{\mathbb{N}} \qquad \mathcal{D}}{\{\dagger\}\,;\cdot\,;\{xs : \overline{[\overline{\mathbb{N}}]}\} \vdash \text{case } (\mathrm{out}_\mu(xs, \overline{[\overline{\mathbb{N}}]})) \text{ of } \{nil \Rightarrow \mathtt{zero} \mid p \Rightarrow \text{split } p \text{ as } (n, xs') \text{ in } \mathtt{succ}\ (\mathtt{plus}\ n\ (\mathtt{sumlen}\ xs'))\} : \overline{[\overline{\mathbb{N}}]} \to \overline{\mathbb{N}}}}{\{\dagger\}\,;\cdot\,;\cdot \vdash \lambda xs : \overline{[\overline{\mathbb{N}}]}.\ \text{case } (\mathrm{out}_\mu(xs, \overline{[\overline{\mathbb{N}}]})) \text{ of } \{nil \Rightarrow \mathtt{zero} \mid p \Rightarrow \text{split } p \text{ as } (n, xs') \text{ in } \mathtt{succ}\ (\mathtt{plus}\ n\ (\mathtt{sumlen}\ xs'))\} : \overline{[\overline{\mathbb{N}}]} \to \overline{\mathbb{N}}}}{\cdot\,;\cdot\,;\cdot \vdash \mathtt{sumlen} : \overline{[\overline{\mathbb{N}}]} \to \overline{\mathbb{N}}\ \mathbf{label}\ \dagger}
$$

Figure 4.16: Cyclic Pre-Proof for Sumlen

The cyclic derivation of the type of the $\mathtt{sumlen}$ term is given in Figure 4.16. We cut the pre-proof into several parts for the sake of presentation. It turns out that this pre-proof can be transformed into a proof, however this is not immediately evident from the structure of the proof as presented, as the original cyclic pre-proof does not meet the guardedness condition.

The pre-proof of `plus`, given by $\mathcal{F}$, does meet the guardedness condition as was shown above. However, the same is not true of `sumlen`. The pre-proof as given does not meet the guardedness condition as there is an implication elimination that takes place in part $\mathcal{E}$. In principle, this application could strip some number of constructors from the subsequent call of `sumlen`.

In order to show that the *semantic* condition of productivity is indeed met we can make use of proof transformation using the evaluation relation, and a restructuring of the cyclicity of the proof to derive an equivalent pre-proof which meets the *syntactic* condition of guardedness.

We do this as a series of steps the first of which is presented in Figure 4.17. This pre-proof simply unfolds more definitions and uses the evaluation relation to simplify proof steps rather than using ImpElim steps in the proof. The resulting proof then uses the Cyc rule to produce cycles in order to give a finite presentation.

$$\dfrac{\cdot\,;\{xs:\overline{[\overline{\mathbb{N}}]}\}\vdash(xs',xs):«\cdot\,;\{xs':\overline{[\overline{\mathbb{N}}]}\}»\ \textbf{sub}}{\{\dagger,\star\}\,;\cdot\,;\{xs:\overline{[\overline{\mathbb{N}}]}\}\vdash(\begin{array}{l}\text{case }(\text{out}_\mu(xs',\overline{[\overline{\mathbb{N}}]}))\text{ of}\\ \{\,nil\Rightarrow\texttt{zero}\\ \mid p\Rightarrow\text{split }p\text{ as }(n,xs')\\ \qquad\text{in }\texttt{succ (plus }n\texttt{ (sumlen }xs'\texttt{))}\end{array}\}:\overline{\mathbb{N}})\Leftarrow(xs',xs)}\,\text{Cyc}(\dagger) \tag{$\mathcal{H}$}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot\,;\{xs':\overline{[\overline{\mathbb{N}}]},n':\overline{\mathbb{N}}\}\vdash(n',n):«\cdot\,;\{xs':\overline{[\overline{\mathbb{N}}]},n:\overline{\mathbb{N}}\}»\ \textbf{sub}}{\{\dagger,\star\}\,;\cdot\,;\{xs':\overline{[\overline{\mathbb{N}}]},n:\overline{\mathbb{N}}\}\vdash(\texttt{plus }n'\texttt{ (sumlen }xs'\texttt{)}:\overline{\mathbb{N}})\Leftarrow(n',n)}\,\text{Cyc}(\star)}{\{\dagger,\star\}\,;\cdot\,;\{xs':\overline{[\overline{\mathbb{N}}]},n:\overline{\mathbb{N}}\}\vdash\text{right}(\texttt{plus }n\texttt{ (sumlen }xs'\texttt{)},1+\overline{\mathbb{N}}):1+\overline{\mathbb{N}}}}{\{\dagger,\star\}\,;\cdot\,;\{xs':\overline{[\overline{\mathbb{N}}]},n:\overline{\mathbb{N}}\}\vdash\text{in}_\nu(\text{right}(\texttt{plus }n\texttt{ (sumlen }xs'\texttt{)},1+\overline{\mathbb{N}}),\overline{\mathbb{N}}):\overline{\mathbb{N}}}}{\{\dagger,\star\}\,;\cdot\,;\{xs':\overline{[\overline{\mathbb{N}}]},n:\overline{\mathbb{N}}\}\vdash\texttt{succ (plus }n\texttt{ (sumlen }xs'\texttt{))}:\overline{\mathbb{N}}} \tag{$\mathcal{G}$}$$

$$\dfrac{\dfrac{\dfrac{\cdot\,;\cdot\,;\{n:\overline{\mathbb{N}}\}\vdash n:\overline{\mathbb{N}}}{\cdot\,;\cdot\,;\{n:\overline{\mathbb{N}}\}\vdash\text{out}_\nu(n,\overline{\mathbb{N}}):1+\overline{\mathbb{N}}}\quad\mathcal{H}\quad\dfrac{\mathcal{G}}{\{\dagger,\star\}\,;\cdot\,;\{xs':\overline{[\overline{\mathbb{N}}]},n:\overline{\mathbb{N}}\}\vdash\texttt{succ (plus }n\texttt{ (sumlen }xs'\texttt{))}:\overline{\mathbb{N}}}}{\{\dagger,\star\}\,;\cdot\,;\{n:\overline{\mathbb{N}},xs':\overline{[\overline{\mathbb{N}}]}\}\vdash\text{case out}_\nu(n,\overline{\mathbb{N}})\text{ of }\{z\Rightarrow\texttt{sumlen }xs'\mid n\Rightarrow\texttt{succ (plus }n\texttt{ (sumlen }xs'\texttt{))}\}:\overline{\mathbb{N}}}}{\{\dagger\}\,;\cdot\,;\{n:\overline{\mathbb{N}},xs':\overline{[\overline{\mathbb{N}}]}\}\vdash\texttt{plus }n\texttt{ (sumlen }xs'\texttt{)}:\overline{\mathbb{N}}\ \textbf{label}\ \star} \tag{$\mathcal{F}$}$$

$$\dfrac{\dfrac{\dfrac{\mathcal{F}}{\{\dagger\}\,;\cdot\,;\{n:\overline{\mathbb{N}},xs':\overline{[\overline{\mathbb{N}}]}\}\vdash\text{right}(\texttt{plus }n\texttt{ (sumlen }xs'\texttt{)},1+\overline{\mathbb{N}}):1+\overline{\mathbb{N}}}}{\{\dagger\}\,;\cdot\,;\{n:\overline{\mathbb{N}},xs':\overline{[\overline{\mathbb{N}}]}\}\vdash\text{in}_\nu(\text{right}(\texttt{plus }n\texttt{ (sumlen }xs'\texttt{)},1+\overline{\mathbb{N}}),\overline{\mathbb{N}}):\overline{\mathbb{N}}}}{\{\dagger\}\,;\cdot\,;\{n:\overline{\mathbb{N}},xs':\overline{[\overline{\mathbb{N}}]}\}\vdash\texttt{succ (plus }n\texttt{ (sumlen }xs'\texttt{))}:\overline{\mathbb{N}}} \tag{$\mathcal{E}$}$$

$$\dfrac{\cdot\,;\cdot\,;\{p:(\overline{\mathbb{N}}\times\overline{[\overline{\mathbb{N}}]})\}\vdash p:(\overline{\mathbb{N}}\times\overline{[\overline{\mathbb{N}}]})\qquad\mathcal{E}}{\{\dagger\}\,;\cdot\,;\{p:(\overline{\mathbb{N}}\times\overline{[\overline{\mathbb{N}}]})\}\vdash\text{split }p\text{ as }(n,xs')\text{ in }\texttt{succ (plus }n\texttt{ (sumlen }xs'\texttt{))}:\overline{\mathbb{N}}} \tag{$\mathcal{D}$}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot\,;\cdot\,;\{xs:\overline{[\overline{\mathbb{N}}]}\}\vdash xs:\overline{[\overline{\mathbb{N}}]}}{\cdot\,;\cdot\,;\{xs:\overline{[\overline{\mathbb{N}}]}\}\vdash\text{out}_\mu(xs,\overline{[\overline{\mathbb{N}}]}):1+(\overline{\mathbb{N}}\times\overline{[\overline{\mathbb{N}}]})}\qquad\cdot\,;\cdot\,;\cdot\vdash\texttt{zero}:\overline{\mathbb{N}}\qquad\mathcal{D}}{\cdot\,;\cdot\,;\{xs:\overline{[\overline{\mathbb{N}}]}\}\vdash\text{case }(\text{out}_\mu(xs,\overline{[\overline{\mathbb{N}}]}))\text{ of }\{nil\Rightarrow\texttt{zero}\mid p\Rightarrow\text{split }p\text{ as }(n,xs')\text{ in }\texttt{succ (plus }n\texttt{ (sumlen }xs'\texttt{))}\}:\overline{\mathbb{N}}\ \textbf{label}\ \dagger}}{\cdot\,;\cdot\,;\cdot\vdash\lambda xs:\overline{[\overline{\mathbb{N}}]}.\ \text{case }(\text{out}_\mu(xs,\overline{[\overline{\mathbb{N}}]}))\text{ of }\{nil\Rightarrow\texttt{zero}\mid p\Rightarrow\text{split }p\text{ as }(n,xs')\text{ in }\texttt{succ (plus }n\texttt{ (sumlen }xs'\texttt{))}\}:\overline{[\overline{\mathbb{N}}]}\to\overline{\mathbb{N}}}}{\cdot\,;\cdot\,;\cdot\vdash\texttt{sumlen}:\overline{[\overline{\mathbb{N}}]}\to\overline{\mathbb{N}}}$$

Figure 4.17: Cyclic Proof for Sumlen

In this final proof in Figure 4.17 we can demonstrate the productivity by checking the guardedness condition. For the path given by the Cyc rule associated with the †, we have $p = E^{+3}$, $I^{\times 2}$, $I^{\nu 1}$, I_R^{+1}, E^{+2}, $I^{\Omega 1}$ and $p' = I_R^{+1}$. This is a prefix and suffix which is admissible, together with the guarded rule $I^{\nu 1}$, hence this path is guarded.

For the path given by the Cyc rule associated with the $\star$, we have $p = I^{\Omega 1}$, E^{+3}, $I^{\Omega 1}$, $I^{\nu 1}$, I_R^{+1} and $p' = I_R^{+1}$. This path has an admissible prefix and suffix together with the guard $I^{\nu 1}$. Since there are no other cycles in the pre-proof, and the cycles are all guarded, this pre-proof is a proof.

$$\begin{array}{lll}
\Omega(\texttt{f}) & := & \lambda s : \overline{[\overline{\mathbb{N}}]}. \\
 & & \text{case out}_\nu(s, [\![\overline{\mathbb{N}}]\!]) \text{ of} \\
 & & \quad \{\ \mathit{nil} \Rightarrow \texttt{zero} \\
 & & \quad \mid p \Rightarrow \text{split } p \text{ as } (n, s') \text{ in } \texttt{g}\ s'\ n\} \\
\Omega(\texttt{g}) & := & \lambda s' : \overline{[\overline{\mathbb{N}}]},\ n : \overline{\mathbb{N}}. \\
 & & \text{in}(\text{right}(\text{case } (\text{out}(n, \overline{\mathbb{N}})) \text{ of} \\
 & & \qquad\qquad \{\ z \Rightarrow \texttt{f}\ s' \\
 & & \qquad\qquad \mid n' \Rightarrow \texttt{g}\ s'\ n'\}, \mathbb{1} + \overline{\mathbb{N}}), \overline{\mathbb{N}})
\end{array}$$

Figure 4.18: Residual Sumlen Program

Associated with this cyclic proof is a program which has the same structure as the proof and is given in Figure 4.18. The technical procedure for generating this residual program is given in Section 5.4. We can clearly see that the program contains constructors in `g` prior to calling `f` or calling `g` and does not attempt to destructure the results of these calls but merely returns them it is manifestly productive. We have removed intermediate computations such as the function `plus` which could have been destroying the productivity of our computation. After transformation the fact that it was not in fact doing anything untoward which might damage productivity is made obvious in the syntax.

By contrast to the productive examples we give an example of an addition function, `badplus` given in Figure 4.19, which would be perfectly suitable on an inductive type, however the prograrm is rightly rejected as it is not guarded.

$$\begin{array}{lll}
\Omega(\texttt{succ}) & := & \lambda x : \overline{\mathbb{N}}.\ \text{in}_\nu(\text{right}(x, \mathbb{1} + \overline{\mathbb{N}}), \overline{\mathbb{N}}) \\
\Omega(\texttt{badplus}) & := & \lambda x, y : \overline{\mathbb{N}}. \\
 & & \text{case } (\text{out}_\nu(x, \overline{\mathbb{N}})) \text{ of} \\
 & & \quad \{\ z \ \Rightarrow\ y \\
 & & \quad \mid x' \ \Rightarrow\ \texttt{badplus}\ x'\ (\texttt{succ}\ y)\} \\
\Xi(\texttt{succ}) & := & \overline{\mathbb{N}} \to \overline{\mathbb{N}} \\
\Xi(\texttt{badplus}) & := & \overline{\mathbb{N}} \to \overline{\mathbb{N}} \to \overline{\mathbb{N}}
\end{array}$$

Figure 4.19: Unproductive Plus

$$
\dfrac{
\dfrac{\cdot\,;\cdot\,;\{x:\overline{\mathbb{N}},y:\overline{\mathbb{N}}\}\vdash x : \overline{\mathbb{N}}}{\cdot\,;\cdot\,;\{x:\overline{\mathbb{N}},y:\overline{\mathbb{N}}\}\vdash \mathrm{out}(x,\overline{\mathbb{N}}) : \mathbb{1}+\overline{\mathbb{N}}}\,E^{\nu}
\qquad
\cdot\,;\cdot\,;\{x:\overline{\mathbb{N}},y:\overline{\mathbb{N}}\}\vdash y : \overline{\mathbb{N}}
\qquad
\dfrac{\dfrac{\cdot\,;\{x':\overline{[\overline{\mathbb{N}}]}\}\vdash (x',x) : «\cdot\,;\{x:\overline{\mathbb{N}}\}»\ \mathbf{sub}}{\begin{array}{l}\cdot\,;\cdot\,;\{x':\overline{\mathbb{N}},y:\overline{\mathbb{N}}\}\vdash \text{case out}(x,\overline{\mathbb{N}})\text{ of} \qquad : \overline{\mathbb{N}}\\ \quad\{\ z\Rightarrow \mathtt{succ}\ y\\ \quad\mid x'\Rightarrow \mathtt{badplus}\ x'\\ \qquad(\mathtt{succ}\ (\mathtt{succ}\ y))\}\end{array}}\,\mathrm{Cyc}(\dagger)}{\cdot\,;\cdot\,;\{x:\overline{\mathbb{N}},y:\overline{\mathbb{N}}\}\vdash \mathtt{badplus}\ x'(\mathtt{succ}\ y) : \overline{\mathbb{N}}}\,I^{\Omega}
}{
\dfrac{\begin{array}{l}\cdot\,;\cdot\,;\{x:\overline{\mathbb{N}},y:\overline{\mathbb{N}}\}\vdash \text{case out}(x,\overline{\mathbb{N}})\text{ of} \qquad : \overline{\mathbb{N}}\ \mathbf{label}\ \dagger\\ \quad\{\ z\Rightarrow y\\ \quad\mid x'\Rightarrow \mathtt{badplus}\ x'(\mathtt{succ}\ y)\}\end{array}}{\dfrac{\begin{array}{l}\cdot\,;\cdot\,;\cdot\vdash \lambda x,y:\overline{\mathbb{N}}.\ \text{case out}(x,\overline{\mathbb{N}})\text{ of} \qquad : \overline{\mathbb{N}}\to\overline{\mathbb{N}}\to\overline{\mathbb{N}}\\ \quad\{\ z\Rightarrow y\\ \quad\mid x'\Rightarrow \mathtt{badplus}\ x'(\mathtt{succ}\ y)\}\end{array}}{\cdot\,;\cdot\,;\cdot\vdash \mathtt{badplus} : \overline{\mathbb{N}}\to\overline{\mathbb{N}}\to\overline{\mathbb{N}}}\,I^{\Omega}}\,I^{\to},I^{\to}
}\,E^{+}
$$

Figure 4.20: Unproductive Plus Cyclic Pre-Proof

We see in Figure 4.20 that the path for † consists of E^{+^3},I^{Ω^1} and therefore does not have a guard.

It is also possible to have programs which are productive, but for which productivity will not be discovered, even *after* the transformation rules which are given in Chapter 5. In the conclusion we give a simple example of a clearly productive program (Figure 8.1which we are not able to transform into a cyclic proof. We additionally demonstrate how such programmes might be included. Of course, in the final analysis it must be impossible to transform all productive programs into a syntactic form demonstrating productivity. We can make a pathological case that requires us to determine if a given program halts prior to emiting a guard. This reduces the problem to the halting problem.

4.8 Related Work

The importance of cyclic proof was probably first recognised by the model checking community in providing proofs that transition systems with a potentially infinite number of states, satisfied some temporal formula. The modal μ-calculus in particular is interesting because it has alternating greatest and least fixed points. Tableau methods for demonstrating satisfaction of μ-calculus formulae were investigated by Bradfield and Stirling in [12]. These will be looked at in more detail in Chapter 6.

Bradfield and Stirling's work is what initially inspired the approach taken in the present work. It differs in dealing with a Tableau system rather than a type theory and being concerned with transition systems, not with a term calculus which is itself directly a programming language.

The work on cyclic proofs was initiated by Brotherston [13]. The formulation that was given by Brotherston is in terms of a logic with induction rules. He first gives a formulation quite similar to the one given for cyclic proofs without explicit reference to terms.

In Brotherston's formulation inductive definition sets are used which introduce inductive predicates by way of *productions*. These productions are introduction rules for the predicates.

The monotone operator which ensures that this is meaningful (and does not lead to unsoundness) is constructed explicitly. In the present work we rely on the positivity of types to ensure that such a monotone operator exists rather than constructing it. In addition, we define recursive types using an explicit least and greatest fixed-point notation rather than simply giving a set of productions.

The method given in this chapter differs since it makes reference to a term calculus. We use a notion of equivalence based on the operational behaviour of the evaluation relation such that we can use similar methods in the setting of a functional programming language. This opens the door to the use of the evaluation relation to give us more freedom in the transformation proofs. We extend beyond simply representing inductive relationships as proof cycles, to include the generation of new cyclic configurations of the same proof and include both induction and coinduction.

Our work shares important similarities with Cockett's work on deforestation [18]. He describes cut-elimination in cyclic proofs with both inductive and coinductive types.

The present work differs from the one given by Cockett in the use of pre-proofs. Cockett uses the Charity [19] programming language which is a total functional programming language. The work here differs in that it deals with a general programming language (which is Turing complete) with the aim of proving particular terms to be total. Cockett's motivation for the use of similar techniques was to explore equivalence of terms. Within Cockett's framework soundness does not need to be determined in each case. Only the rules and the transformations need to be justified.

We give a notion of bisimiliarity in Chapter 3 in order to define equivalence of proofs which allows us to work with programs without regard to their termination. We use this result to demonstrate in Chapter 5 how the more general supercompilation family of algorithms fit into the framework of program transformation as cut-elimination (and cut-introduction) in cyclic proof extending Cockett's work. In this case we seek to use automatic methods for establishing soundness for programs which do not meet the restrictions required for deforestation. This method can cope with programs which are not known to be terminating in contrast to the total programs given by Cockett.

Santocanale has also investigated cyclic proof [76]. Santocanale's approach is to start with categorical notions of initial algebras and final coalgebras and to use this to uncover a term language with a cyclic proof system which is suitable as a programming language. Rather than using function constants, as is done here, Santocanale looks at the more general notion of systems of directed equations.

The investigation given here starts with the aim of bridging the gap between presently existing programming languages, such as Haskell and Coq, with techniques from cyclic proof. For this reason, the cyclic proof system given here differs significantly from the one given by Santocanale, and more closely resembles a typical type system with a coinductive typing derivation relation rather than an inductive one.

Chapter 5

Program Transformation

5.1 Introduction

The study of program transformation is the broad study of methods of altering programs such that they preserve program behaviours up to some notion of equivalence. For this reason, notions of inclusion or equivalence of behaviours are critical to the study of program transformation.

Historically, program transformation is most often used to increase the efficiency of programs, either in space or time or both. However, Turchin noted quite early that it can also be used as a method for determining properties of software [86]. Similar approaches, using program transformation for verification have been taken up in several other works [48][35].

The present work uses program transformation for the purpose of establishing program (co)termination behaviour, using new methods. We are interested in dealing with program termination behaviour because we want to be able to view types as establishing properties of the programs for which they have proofs. Using cyclic pre-proofs all types are inhabited and types which describe some intended behaviour can be given proofs which have no behaviour at all. The generation of a proof from a pre-proof will give us a guarantee that the type of the final program describes the actual behaviours of the original program prior to transformation.

5.2 General Framework

Program transformation can be seen as a family of systems for the manipulation of terms. A very general system was given by Burstall and Darlington [14]. Following on that work we present some of their results in the context of proof transformations. We use those results to describe the *supercompilation algorithm* in Section 5.3. Supercompilation is a particular instance of this more general framework which shares the twin properties of ensuring bisimilarity and termination of the transformation.

The techniques can be broadly described as falling into the following basic steps: *unfolding*, which is the replacement of function symbols with the terms that they represent, *elimination*, which makes use of equivalence under the evaluation relation, *generalisation* which will introduce new forms which are behaviourally equivalent and *information propagation*, which keeps track of which behaviours we can assume to be true of a term. Folding involves creating a finite representation of the infinite proof tree by recognising a recurrence. We define all of these formally, and then motivate their use with a particular example.

Definition 5.2.1 (Unfolding)**.** A term r with derivation ($\Delta\ ; \Gamma \vdash r\ :\ A$) is said to be an unfolding of a term f with derivation ($\Delta\ ; \Gamma \vdash \Omega(f)\ :\ \Xi(f)$) such that $\Omega(f) = r$ and $\Xi(f) = A$.

Since we have built the proof rules to include unfolding as a derivation rule, we can use this rule in an unrestricted way. It has no effect on the termination properties of programs as it simply replaces terms with their definitions. It is what allows us to represent terms with infinite derivations in a finite way.

Definition 5.2.2 (Folding)**.** If a term r, with derivation ($\Delta'\ ; \Gamma' \vdash r\ :\ A$) is encountered while transforming a proof to a cyclic proof, such that t is a substitution instance of some term s with derivation ($\Delta\ ; \Gamma \vdash s\ :\ B$) and substitution ($\Delta\ ; \Gamma \vdash \sigma\ :\ «\Delta'\ ; \Gamma'»\quad \mathbf{sub}$), that is, ($\Delta\ ; \Gamma \vdash s\ :\ B$) $\Leftarrow \sigma =$ ($\Delta'\ ; \Gamma' \vdash r\ :\ A$) and further we have that ($\Delta\ ; \Gamma \vdash s\ :\ B$) $\in \zeta$, where ζ is the current sequent context, we can simply rewrite the pre-proof as a Cyc rule, and say that we have made a *fold*.

If we think of the pre-proof as being a coinductive representation of the proof-rules, this folding rule is simply a co-fixpoint representation of the infinite proof tree utilising the recurrence to give a finite presentation.

We also need to make use of *generalisation*. The *least general generalisation* (or LGG) is in a strict sense the dual to unification as was first demonstrated by Plotkin [71]. Generalisation simply allows us to introduce a number of implications and for-all proof rules using the assumed equality modulo β-reduction. The need for generalisation comes from the need to produce finite representations of our pre-proofs. We give a definition of generalisation in Definition 5.3.8. We see an example of the use of generalisation in Section 5.5.

The choices of generalisation are somewhat arbitrary as the calculus presented here admits no single least general generalisation[49]. The choice of ordering which gives rise to the notion of *least* will impact what results we obtain. This means that we must choose the generalisation algorithm that we use in practice, based on heuristics, or leave the choice open to a user.

5.3 The Supercompilation Algorithm

Supercompilation [87] is a particular family of algorithms which make use of folding, unfolding, generalisation and information propagation. It is a superclass of the *partial evaluation* and *deforestation* program transformations, as it can perform these in addition to more sophisticated transformations.

Supercompilation is characterised by composing the features: unfolding, folding, generalisation, reduction and information propagation in such a way as to ensure termination of the transformation algorithm. It can be thought of as a method of producing bisimilar programs by design. Just as bisimilarity is coinductively defined and makes use of self-reference, we will find that construction of bisimilar programs using supercompilation follows a similar approach to the structure of a bisimulation argument. We construct a term which has the syntax for the appropriate behaviours and makes use of self-reference.

The fact that some elimination proof steps constrain the form of terms which share proof structure with the term under elimination is known as *information propagation*. The full algorithm requires that we find a finite representation of the cyclic pre-proof produced by driving, and therefore we must use *generalisation* to avoid unbounded pre-proof sizes, by replacing some terms with variables and *folding* to substitution instances.

Information propagation is quite straightforward in our framework. The definition of the rewrites performed by information propagation are given in 4.4.6. It is essentially book keeping of meta-variables with inversion on the typing derivations of these metavariables to arrive at equations. These equations can then be used as rewrites on further meta-variables in subsequent proof steps. We can be assured that this is acceptable because the reduction relation is deterministic and will always attempt to reduce the term under consideration for elimination first. Non-termination of this term will not be affected by replacement of terms further up the proof tree as they will never be reached in the event of non-termination.

Lemma 5.3.1 (Information Propagation)**.** *Information Propagation for a term t leads to a term t'*

such that $t \sim t'$.

Proof. The proof of the bisimulation of the term after rewrites follows directly from inversion on the typing derivation and the fact that simulation is a compatible refinement. The metavariables, a,b and c are introduced through the definition of the transition relation.

- For case terms, case c of $\{x \Rightarrow r \mid y \Rightarrow s\}$, we have that the term c is of type $\cdot\ ; \cdot \vdash c\ :\ A + B$. If $c \Uparrow$ then (case c of $\{x \Rightarrow r \mid y \Rightarrow s\}) \Uparrow$. If $c \Downarrow$ then by inversion of the type we arrive at two cases. In the first case we have that $c = \text{left}(a, A + B)$. We can then rewrite c with $\text{left}(a, A + B)$ in the term resulting in the reduction (case c of $\{x \Rightarrow r \mid y \Rightarrow s\}) \rightsquigarrow^* r[x := a]$. In the second case, similarly we have (case c of $\{x \Rightarrow r \mid y \Rightarrow s\}) \rightsquigarrow^* s[y := a]$, in which we can rewrite c with $\text{right}(a, A + B)$. Since simulation includes the reduction relation we can have our goal.

- For split terms, split c as (x, y) in r, we have that the term c is of type $\cdot\ ; \cdot \vdash c\ :\ A \times B$. If $c \Uparrow$ then (split c as (x, y) in $r) \Uparrow$. If $c \Downarrow$ We can then rewrite c with (a, b) in the term resulting from the reduction (split c as (x, y) in $r) \rightsquigarrow^* r[x := a][y := b]$. Since simulation includes the reduction relation we can have our goal.

- For unfolds we have that $\text{out}_\alpha(a, \alpha\hat{X}.\,T)$ requires $\cdot\ ; \cdot\ \vdash\ a\ :\ \alpha\hat{X}.\,T$. If $a \Uparrow$ then $(\text{out}_\alpha(a, \alpha\hat{X}.\,T)) \Uparrow$. If $a \Downarrow$ then by inversion on this type derivation we have one case, that a is formed by $\text{in}_\alpha(b, \alpha\hat{X}.\,T)$ for some b. We then have the rewrite $a = \text{in}_\alpha(b, \alpha\hat{X}.\,T)$ for the term. Since simulation includes the reduction relation we can have our goal.

□

The subject of the supercompilation algorithm can be thought of as the potentially infinite pre-proof tree in which intermediate elimination steps are removed by evaluation and information propagation.

Definition 5.3.2 (Driving)**.** Driving is the production of a pre-proof tree which has removed all Elimination Introduction rule pairs by use of the evaluation relation $\rightsquigarrow$ and rewritten by information propagation. This driven pre-proof tree is potentially infinite.

Folding however is a more complex issue. In general program transformation we find that folding can sometimes lead to unsound proofs and indeed transformed terms may lose behaviour that they once had. Various methods and mechanisms have been designed to cope with this problem. Notably, David Sands developed a condition of *improvement* and a syntactic calculus which ensures improvement [73].

It is the case, however, that if a pre-proof is a proof, then we have no such concerns. The folding cannot lead to non-termination as the proof tree shows it to be guarded and therefore we cannot lose behaviours [24]. We can therefore dispense with complicated proofs of preserving correctness and simply show that we have a proof.

We also find it necessary to generalise terms in order both to find cycles and to find a decomposition into pre-proofs which meet our syntactic restrictions. The notion of a generalisation assumes a *poset*, in our case $(\mathbf{Terms}, \leqslant)$ for a suitably defined $\leqslant$.

Definition 5.3.3 (Poset)**.** A *poset* is a pair of a set P with a relation $\leqslant \subseteq P \times P$ such that for some pairs of elements in the set $(a, b) \in P \times P$, one of the elements *precedes* the other, that is, $(a, b) \in \leqslant$ or $a \leqslant b$. The relation $\leqslant$ must be reflexive, anti-symmetric and transitive.

The *application ordering* is one choice of ordering for generalisations in System F, though others are possible [66]. We base our relation on the one given in [49]. The idea is similar to various types of substitution based generalisations but the substitutions are left implicit by leaving them to the reduction relation. This helps us to ensure that the substitutions are correct if the applications are type correct, and simplifies reasoning about the relation.

Definition 5.3.4 (Application Ordering)**.** The application ordering states that a term $s \leqslant t$ (t is more general than s) for any terms s and t where there exists a vector of terms $\overrightarrow{u}$, which represents some number of term and type applications, such that $t\ \overrightarrow{u} =_\beta s$ with $\overrightarrow{u}$ possibly empty.

Here we have $=$ denoting syntactic equality modulo the $\rightsquigarrow^*$ reduction relation. No use is made of function-constant unfolding, which is important as we need to restrict to a normalisable fragment. It may be noticed that the particular definition here does not deal with permutations of abstractions. This however simply means that fewer terms are considered in the ordering. To remedy this, one can simply permute the order of abstractions in the generalisation.

Lemma 5.3.5 ($\leqslant$ is reflexive)**.** *For any term t, $t \leqslant t$.*

Proof. $t\ \overrightarrow{u} = t$ when $\overrightarrow{u}$ is empty. □

Lemma 5.3.6 ($\leqslant$ is antisymmetric)**.** *For any terms t and s such that $s \leqslant t$, yet $s \neq t$, then it is not the case that $t \leqslant s$.*

Proof. Since we have that $s \leqslant t$ for some t and s, we know that there are some types and terms $\overrightarrow{u}$ such that $t\ \overrightarrow{u} = s$ by the definition of $\leqslant$. Since the application has a type having $|\overrightarrow{u}|$ fewer $\forall$ and $\rightarrow$ steps, the relation cannot be symmetric, as this would imply $n - |\overrightarrow{u}| = m$ and $m - |\overrightarrow{u}| = n$. This is only true if $|\overrightarrow{u}| = 0$ which means that $t = s$, a contradiction with the hypothesis. □

Lemma 5.3.7 ($\leqslant$ is Transitive)**.** *For all terms t, s and v, if $t \leqslant s$ and $s \leqslant v$ then $t \leqslant v$.*

Proof. This proof is straightforward though we have to be careful about the ordering of terms. Since $s\ \overrightarrow{u_1} = t$ and $v\ \overrightarrow{u_2} = s$. This means that $t = v\ \overrightarrow{u_2}\ \overrightarrow{u_1}$. We can then take $\overrightarrow{u} = \overrightarrow{u_2}\ \overrightarrow{u_1}$ and have that $t = v\ \overrightarrow{u}$ and hence $t \leqslant v$. □

We have previously given the definition of an instance in Definition 2.6.1. The particular instantiation algorithm used to provide instances is a partial function which provides us with a term which is driven by the structure of the two terms under consideration. It will not find instances in all cases in which it is possible, but necessarily returns an instance when it succeeds. We define a type-instance partial-function *tyinst* : $\mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \mathbf{Var} \rightarrow (\mathbf{TypeVar} \rightarrow \mathbf{Type})$ as given in Figure 5.1. We use the function *fresh* which chooses a variable or type variable (depending on context) such that it is new with respect to a set.

```
$tyinst[X, Y, \mathcal{T}, \sigma] :=$                        if $X = Y$ then $\sigma$ else $fail$

$tyinst[1, 1, \mathcal{T}, \sigma] :=$                        $\sigma$

$tyinst[A \times B, C \times D, \mathcal{T}, \sigma] :=$      let $\sigma_1 = tyinst[A, C, \mathcal{T}, \sigma]$
                                                  in $tyinst[B, D, \mathcal{T}, \sigma_1]$

$tyinst[A + B, C + D, \mathcal{T}, \sigma] :=$                let $\sigma_1 = tyinst[A, C, \mathcal{T}, \sigma]$
                                                  in $tyinst[B, D, \mathcal{T}, \sigma_1]$

$tyinst[A \rightarrow B, C \rightarrow D, \mathcal{T}, \sigma] :=$  let $\sigma_1 = tyinst[A, C, \mathcal{T}, \sigma]$
                                                  in $tyinst[B, D, \mathcal{T}, \sigma_1]$

$tyinst[\alpha\hat{X}.\ A, \alpha\hat{Y}.\ B, \mathcal{T}, \sigma] :=$  let $\hat{Z} = fresh(FV(A) \cup FV(B) \cup \mathcal{V})$
                                                  in $tyinst[A[\hat{X} := \hat{Z}], B[\hat{Y} := \hat{Z}], \mathcal{T} \cup \{\hat{Z}\}, \sigma]$

$tyinst[\forall X.A, \forall Y.B, \mathcal{T}, \sigma] :=$    let $Z = fresh(FV(A) \cup FV(B) \cup \mathcal{V})$
                                                  in $tyinst[A[X := Z], B[Y := Z], \mathcal{T} \cup \{Z\}, \sigma]$

$tyinst[X, A, \mathcal{T}, \sigma] :=$                        if $X \in \mathcal{T}$ then $fail$
                                                  else if $\exists (X, B) \in \sigma$
                                                      then $tyinst[B, A, \mathcal{T}, \sigma]$
                                                      else $\sigma \cup (X, A)$
```

Figure 5.1: Type Instance

The term instance algorithm makes no direct reference to $tyinst$ but the compatibility of types is assumed by the algorithm, hence it must follow application of $tyinst$. We give the partial function $inst$ in Figure 5.2.

$$
\begin{array}{lcl}
inst[x, y, \mathcal{V}, \sigma] & := & \text{if } x \notin \mathcal{V} \wedge x = y \text{ then } \sigma \text{ else } fail \\[1ex]
inst[f, g, \mathcal{V}, \sigma] & := & \text{if } f = g \text{ then } \sigma \text{ else } fail \\[1ex]
inst[\text{left}(t, A), \text{left}(s, B), \mathcal{V}, \sigma] & := & inst[t, s, \mathcal{V}, \sigma] \\[1ex]
inst[\text{right}(t, A), \text{right}(s, B), \mathcal{V}, \sigma] & := & inst[t, s, \mathcal{V}, \sigma] \\[1ex]
inst[\text{in}(t, A), \text{in}(s, A), \mathcal{V}, \sigma] & := & inst[t, s, \mathcal{V}, \sigma] \\[1ex]
inst[(t, s), (r, u), \mathcal{V}, \sigma] & := & \text{let } \sigma_1 = inst[t, r, \mathcal{V}, \sigma] \\
 & & \text{in } inst[s, u, \mathcal{V}, \sigma_1] \\[1ex]
inst[\Lambda X.\ t, \Lambda Y.\ s, \mathcal{V}, \sigma] & := & inst[t, s, \mathcal{V}, \sigma] \\[1ex]
inst[\lambda x\!:\!A.\ t, \lambda y\!:\!B.\ s, \mathcal{V}, \sigma] & := & \text{let } z = fresh(FV(t) \cup FV(s) \cup \mathcal{V}) \\
 & & \text{in } inst[t[x := z], s[y := z], \mathcal{V} \cup \{z\}, \sigma] \\[1ex]
inst[\text{out}(t, A), \text{out}(s, B), \mathcal{V}, \sigma] & := & inst[t, s, \mathcal{V}, \sigma] \\[1ex]
inst[t\ s, r\ u, \mathcal{V}, \sigma] & := & \text{let } \sigma_1 = inst[t, r, \mathcal{V}, \sigma] \\
 & & \text{in } inst[s, u, \mathcal{V}, \sigma_1] \\[1ex]
inst[t[A], s[B], \mathcal{V}, \sigma] & := & inst[t, s, \mathcal{V}, \sigma] \\[1ex]
inst\begin{bmatrix} \begin{array}{l}\text{case } t \text{ of} \\ \ \{\, x \Rightarrow r \\ \ \ \mid y \Rightarrow s \,\}\end{array}, & \begin{array}{l}\text{case } u \text{ of} \\ \ \{\, v \Rightarrow m \\ \ \ \mid w \Rightarrow n \,\}\end{array}, \mathcal{V}, \sigma \end{bmatrix} & := & \\
 & & \text{let } z_1 = fresh(FV(r) \cup FV(m) \cup \mathcal{V}) \\
 & & \quad z_2 = fresh(FV(s) \cup FV(w) \cup \mathcal{V}) \\
 & & \quad \sigma_1 = inst[t, u, \mathcal{V}, \sigma] \\
 & & \quad \sigma_2 = inst[r[x := z_1], u[v := z_1], \mathcal{V} \cup \{z_1\}, \sigma_1] \\
 & & \text{in } inst[s[y := z_2], n[w := z_2], \mathcal{V} \cup \{z_2\}, \sigma_2] \\[1ex]
inst\begin{bmatrix} \begin{array}{l}\text{split } t \text{ as } (x, y) \\ \text{in } r\end{array}, & \begin{array}{l}\text{split } u \text{ as } (v, w) \\ \text{in } s\end{array}, \mathcal{V}, \sigma \end{bmatrix} & := & \\
 & & \text{let } \sigma_1 = inst[t, u, \mathcal{V}, \sigma] \\
 & & \text{in } inst[r, s, \mathcal{V} \cup \{x, y\}, \sigma_1] \\[1ex]
inst[x, t, \mathcal{V}, \sigma] & := & \text{if } x \in \mathcal{V} \text{ then } fail \\
 & & \text{else if } \exists (x, s) \in \sigma \\
 & & \quad \text{then } inst[s, t, \mathcal{V}, \sigma] \\
 & & \quad \text{else } \sigma \cup (x, s)
\end{array}
$$

Figure 5.2: Instance

Now that we have terms as a poset, we can continue to define generalisation in terms of that ordering.

Definition 5.3.8 (Generalisation)**.** A generalisation of two terms, t and s is a triple of a term g and a vector of terms $\overrightarrow{u}$ and $\overrightarrow{v}$ denoted $(g, \overrightarrow{u}, \overrightarrow{v}) = t \sqcap s$. The term g is constructed such that $g\ \overrightarrow{u} =_\beta t$ and $g\ \overrightarrow{v} =_\beta s$.

Generalisation can consist of any such term which is less than in the application ordering. $\leqslant$. This means that there may be very many generalisations.

The generalisation algorithm actually implemented is relatively naive though it is more complicated than most algorithms presented in the supercompilation literature because of the need to represent types and the generalisation of types. Because we have type application and a reduction relation that includes type substitution, we will start with describing the generalisation of types. The algorithm for type generalisation is given by a partial function $\mathcal{G}_{ty} : \mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \mathbf{TyCtx} \rightarrow \mathcal{P}(\mathbf{TyVar}) \rightarrow \mathbf{TyCtx} \times \Theta \times \mathbf{Type}$ shown in Figure 5.3.

In the algorithm we make use of a map that associates a variable with a pair of types $\theta : \Theta$ where $\Theta \equiv \mathbf{TyVar} \rightarrow \mathbf{Type} \times \mathbf{Type}$. This takes the place of a pair of substitutions which make reference only to the same variables and simplifies the implementation.

Type generalisation is required before pursuing term generalisation. This ensures the constraint that the types are compatible is maintained during the term generalisation algorithm. We will again need a map that associates variables with pairs of terms, constrained to have the same type which we denote with the variable $\pi : \Pi$ where $\Pi \equiv \mathbf{Term} \rightarrow \mathbf{Term} \times \mathbf{Term} \times \mathbf{Type}$.

Once we have generalised types to obtain a suitably general type, we can proceed with term generalisation using the partial function $\mathcal{G} : \mathbf{Term} \rightarrow \mathbf{Term} \rightarrow \mathbf{Type} \rightarrow \mathbf{TyCtx} \rightarrow \mathbf{TyCtx} \rightarrow \mathbf{Ctx} \rightarrow \mathbf{Ctx} \rightarrow \mathbf{TyCtx} \rightarrow \mathbf{Ctx} \rightarrow \mathcal{P}(\mathbf{Var}) \rightarrow \mathcal{P}(\mathbf{TyVar}) \rightarrow \Theta \rightarrow \Pi \rightarrow \mathbf{Ctx} \times \Theta \times \Pi \times \mathbf{Term}$.

The algorithm for term generalisation is described in Figure 5.5 and Figure 5.6. In this description we see that there are three pairs of contexts: one context pair Δ^a and Γ^a which describes the context in which the first term has a derivation, Δ^b and Γ^b in which the second term has a derivation and Δ^c and Γ^c which is an extension which will be required for the resultant term to have a successful derivation. The set $\mathcal{V}$ contains information about bound variables, and $\mathcal{T}$ about bound type-variables. The algorithm is designed to return a generalised term together with an extended context which can be used to find the derivation of the resultant term, along with a type and term double-substitution allowing us to reconstruct either of the two original terms by substitution, or to fail in an attempt to do so. Since our supercompilation algorithm is non-deterministic, failure is an acceptable outcome.

$$\mathcal{G}_{ty}[X, Y, \Delta, \mathcal{V}, \theta]\ s.t.\ X = Y \quad := \quad (\Delta, \theta, X)$$

$$\mathcal{G}_{ty}[1, 1, \Delta, \mathcal{V}, \theta] \quad := \quad (\Delta, \theta, 1)$$

$$\begin{array}{lll}
\mathcal{G}_{ty}[A \times B, C \times D, \Delta, \mathcal{V}, \theta] & := & \text{let } (\Delta_1, \theta_1, E) = \mathcal{G}_{ty}[A, C, \Delta, \mathcal{V}, \theta] \\
& & \quad (\Delta_2, \theta_2, F) = \mathcal{G}_{ty}[B, D, \Delta_1, \mathcal{V}, \theta_1] \\
& & \text{in } (\Delta_2, \theta_2, E \times F)
\end{array}$$

$$\begin{array}{lll}
\mathcal{G}_{ty}[A + B, C + D, \Delta, \mathcal{V}, \theta] & := & \text{let } (\Delta_1, \theta_1, E) = \mathcal{G}_{ty}[A, C, \Delta, \mathcal{V}, \theta] \\
& & \quad (\Delta_2, \theta_2, F) = \mathcal{G}_{ty}[B, D, \Delta_1, \mathcal{V}, \theta_1] \\
& & \text{in } (\Delta_2, \theta_2, E + F)
\end{array}$$

$$\begin{array}{lll}
\mathcal{G}_{ty}[A \to B, C \to D, \Delta, \mathcal{V}, \theta] & := & \text{let } (\Delta_1, \theta_1, E) = \mathcal{G}_{ty}[A, C, \Delta, \mathcal{V}, \theta] \\
& & \quad (\Delta_2, \theta_2, F) = \mathcal{G}_{ty}[B, D, \Delta_1, \mathcal{V}, \theta_1] \\
& & \text{in } (\Delta_2, \theta_2, E \to F)
\end{array}$$

$$\begin{array}{lll}
\mathcal{G}_{ty}[\forall X.\ A, \forall Y.\ B, \Delta, \mathcal{V}, \theta] & := & \text{let } Z = \mathit{fresh}(FV(A) \cup FV(B) \cup \mathcal{V} \cup \mathit{dom}(\Delta)) \\
& & \quad (\Delta_1, \theta_1, C) = \\
& & \qquad \mathcal{G}_{ty}[A[X := Z], B[Y := Z], \Delta, \mathcal{V} \cup \{Z\}, \theta] \\
& & \text{in } (\Delta_1, \theta_1, \forall X.C)
\end{array}$$

$$\begin{array}{lll}
\mathcal{G}_{ty}[\alpha\hat{X}.\ A, \alpha\hat{Y}.\ B, \Delta, \mathcal{V}, \theta] & := & \text{let } \hat{Z} = \mathit{fresh}(FV(A) \cup FV(B) \cup \mathcal{V} \cup \mathit{dom}(\Delta)) \\
& & \quad (\Delta_1, \theta_1, C) = \\
& & \qquad \mathcal{G}_{ty}[A[\hat{X} := \hat{Z}], B[\hat{Y} := \hat{Z}], \Delta, \mathcal{V} \cup \{\hat{Z}\}, \theta] \\
& & \text{in } (\Delta_1, \theta_1, \alpha\hat{Z}.\ C)
\end{array}$$

$$\begin{array}{lll}
\mathcal{G}_{ty}[A, B, \Delta, \mathcal{V}, \theta] & := & \text{if } \exists (X \mapsto (A, B)) \in \theta \\
& & \text{then } (\Delta, \theta, X) \\
& & \text{else let } Y = \mathit{fresh}(\mathit{dom}(\Delta) \cup \mathcal{V}) \\
& & \qquad \text{in } (\Delta \cup Y, \theta \cup (Y \mapsto (A, B)), Y)
\end{array}$$

Figure 5.3: Type Generalisation

We also need to be able to create *patterns* that is, terms which are suitably abstracted to capture only those bound variables in their context, but which might then be supplied. A definition of a pattern as we use it in the algorithm is as follows.

Definition 5.3.9 (Pattern)**.** A *pattern*, p, of a given term t, with derivation $\Delta \cup \Delta' \,; \Gamma \cup \Gamma' \vdash t : C$ is a generalisation of this term to a lambda (type and term) abstraction of the form $p = \Lambda\Delta'.\ \lambda\Gamma'.\ t$ which only generalises variables from t which are bound in some context Γ' such that $\Delta \,; \Gamma \vdash p : \forall\Delta'.\ \Gamma' \rightarrow C$ and $(p[\Delta'])\ \Gamma' \rightsquigarrow t$.

The utility of such a definition of a pattern stems from the ability to create functions whose instantiations reduce to terms which make different use of the currently bound context. This use can be seen clearly in the following pair of terms:

$$\text{case } t \text{ of } \{x \Rightarrow r\ x \mid y \Rightarrow r\ y\}$$

and

$$\text{case } t \text{ of } \{x \Rightarrow r \text{ in}((), A) \mid y \Rightarrow r \text{ in}((), A)\}$$

We can derive from these a pair of patterns for r in generalisation, namely: $(\lambda x\!:\!A.\ r\ x)$ and $(\lambda x\!:\!A.\ r \text{ in}((), A))$ which allows us to represent a generalised term for the pair as $(\lambda f\!:\!A \rightarrow B.\ \text{case } t \text{ of } \{x \Rightarrow$ $f\ x \mid y \Rightarrow f\ y\})$. The first term uses a variable in the bound context, the second does not. In the general case we can imagine any number of bound variables used or not used in either of the two terms, but the union of which is expressed in the abstraction given in the pattern. It is convenient to be able to abstract functions of this form. I provide a definition of the partial function $pattern$ which implements our algorithm for creating patterns in Figure 5.4. In the actual implementation the result term is *strengthened* to remove any unnecessary type or term variables from the context, leading to a compact representation.

```
$pattern[z, s, t, \Delta, \Gamma, \mathcal{V}, \mathcal{T}, \theta, \pi] :=$

let $\mathcal{B} = (FV(s) \cup FV(t)) - \mathcal{V}$
    $\mathcal{B}_{ty} = (FV_{ty}(s) \cup FV_{ty}(t)) - \mathcal{T}$
    $a = (z[\mathcal{B}_{ty}])\ \mathcal{B}$
    $u = \Lambda\mathcal{B}_{ty}.\ \lambda\mathcal{B}.\ s$
    $w = \Lambda\mathcal{B}_{ty}.\ \lambda\mathcal{B}.\ t$
in $(a, u, w)$
```

Figure 5.4: Creating Patterns

$\mathcal{G}[x, y, A, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi]$ s.t. $x = y \wedge x \in \mathcal{V} := (\Delta^c, \Gamma^c, \theta, \pi, x)$

$\mathcal{G}[\mathtt{f}, \mathtt{g}, A, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi]$ s.t. $\mathtt{f} = \mathtt{g} := (\Delta^c, \Gamma^c, \theta, \pi, \mathtt{f})$

$\mathcal{G}[(), (), \mathbb{1}, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi] := (\Delta^c, \Gamma^c, \theta, \pi, ())$

$\mathcal{G}[s\ t, r\ u, E, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi] :=$

```
  if $\exists A, B.(\Delta^a ; \Gamma^a \vdash s : A) \wedge (\Delta^b ; \Gamma^b \vdash r : B)$
  then let $(\Delta^c_1, \theta_1, F \to E) = \mathcal{G}_{ty}[A, B, \Delta^c, \mathcal{T}, \theta]$
           $(\Delta^c_2, \Gamma^c_2, \theta_2, \pi_2, n) = \mathcal{G}[s, r, F \to E, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c_1, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta_1, \pi]$
           $(\Delta^c_3, \Gamma^c_3, \theta_3, \pi_3, m) = \mathcal{G}[t, u, F, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c_2, \Gamma^c_2, \mathcal{V}, \mathcal{T}, \theta_1, \pi_1]$
       in $(\Delta^c_3, \Gamma^c_3, \theta_3, \pi_3, n\ m)$
```

$\mathcal{G}[(\lambda x{:}A.\ t), (\lambda y{:}B.\ s), C \to D, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi] :=$

```
  let $z = \mathit{fresh}(\mathit{dom}(\Gamma^a) \cup \mathit{dom}(\Gamma^b) \cup \mathcal{V})$
    $(\Delta^c_1, \Gamma^c_1, \theta_1, \pi_1, r) =$
      $\mathcal{G}[t[x := z], s[x := z], \Delta^a, \Delta^b, \Gamma^a \cup \{(z, C)\}, \Gamma^b \cup \{(z, C)\}, \Delta^c, \Gamma^c, \mathcal{V} \cup \{z\}, \mathcal{T}, \theta, \pi]$
  in $(\Delta^c_1 \Gamma^c_1, \theta_1, \pi_1, \lambda x{:}C.\ r)$
```

$\mathcal{G}[(\Lambda X.\ t), (\Lambda Y.\ s), \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi] :=$

```
  let $Z = \mathit{fresh}(\mathit{dom}(\Delta^a) \cup \mathit{dom}(\Delta^b) \cup \mathcal{T})$
    $(\Delta^c_1, Gamma^c_1, \theta_1, \pi_1, r) =$
      $\mathcal{G}[t[X := Z], s[Y := Z], \Delta^a \cup \{Z\}, \Delta^b \cup \{Z\}, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T} \cup \{Z\}, \theta, \pi]$
  in $(\Gamma^c_1, \theta_1, \pi_1, \Lambda Z.r)$
```

$\mathcal{G}[(s, t), (r, u), A \times B, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi] :=$

```
  let $(\Delta^c_1, \Gamma^c_1, \theta_1, \pi_1, m) = \mathcal{G}[s, r, A, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi]$
      $(\Delta^c_2, \Gamma^c_2, \theta_2, \pi_2, n) = \mathcal{G}[s, r, B, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c_1, \mathcal{V}, \mathcal{T}, \theta, \pi]$
  in $(\Delta^c_2, \Gamma^c_2, \theta_2, \pi_2, (m, n))$
```

$\mathcal{G}[\text{left}(t, A), \text{left}(s, B), C + D, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi] :=$

```
  let $(\Delta^c_1, \Gamma^c_1, \theta_1, \pi_1, r) = \mathcal{G}[t, s, C, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi]$
  in $(\Delta^c_1, \Gamma^c_1, \theta_1, \pi_1, \text{left}(r, C + D))$
```

$\mathcal{G}[\text{right}(t, A), \text{right}(s, B), C + D, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi] :=$

```
  let $(\Delta^c_1, \Gamma^c_1, \theta_1, \pi_1, r) = \mathcal{G}[t, s, D, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi]$
  in $(\Delta^c_1, \Gamma^c_1, \theta_1, \pi_1, \text{right}(r, C + D))$
```

Figure 5.5: Term Generalisation

$$\mathcal{G}[\text{out}_\alpha(t, \alpha\hat{X}.\ A), \text{out}_\alpha(s, \alpha\hat{Y}.\ B), C, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi] :=$$

$$\begin{array}{l}
\text{let } (\Delta^c_1, \theta_1, D) = \mathcal{G}_{ty}[A, B, \Delta^c, \mathcal{T} \cup \{X\}, \theta] \\
\quad (\Delta^c_2, \Gamma^c_2, \theta_2, \pi_2, r) = \mathcal{G}[t, s, \alpha\hat{X}.\ D, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c_1, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta_1, \pi] \\
\text{in } (\Delta^c_2, \Gamma^c_2, \theta_2, \pi_2, \text{out}(r, \alpha\hat{X}.\ D))
\end{array}$$

$$\mathcal{G}[\text{in}_\alpha(t, A), \text{in}_\alpha(s, B), \alpha\hat{X}.\ C, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi] :=$$

$$\begin{array}{l}
\text{let } (\Delta^c_1, \Gamma^c_1, \theta_1, \pi_1, r) = \mathcal{G}[t, s, C[X := \alpha\hat{X}.\ C], \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi] \\
\text{in } (\Delta^c_1, \Gamma^c_1, \theta_1, \pi_1, \text{in}_\alpha(r, \alpha\hat{X}.\ C))
\end{array}$$

$$\mathcal{G}\left[\begin{array}{llll}
\text{case } t \text{ of} & & \text{case } t' \text{ of} & \\
\ \{\ x \Rightarrow r & , & \ \{\ x' \Rightarrow r' & , C, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi \\
\ \mid y \Rightarrow s\ \} & & \ \mid y' \Rightarrow s'\ \} &
\end{array}\right] :=$$

$$\begin{array}{l}
\text{let } z = \mathit{fresh}(\mathit{dom}(\Gamma^a) \cup \mathit{dom}(\Gamma^b) \cup \mathit{dom}(\Gamma^c)) \\
\quad z' = \mathit{fresh}(\mathit{dom}(\Gamma^a) \cup \mathit{dom}(\Gamma^b) \cup \mathit{dom}(\Gamma^c)) \\
\quad D = \mathit{typeof}[\Delta^a, \Gamma^a, t] \\
\quad E = \mathit{typeof}[\Delta^a, \Gamma^b, t'] \\
\quad A + B = \mathcal{G}_{ty}[D, E, \Delta^c, \mathcal{T}, \theta] \text{ or } \mathit{fail} \\
\quad (\Delta^c_1, \Gamma^c_1, \theta_1, \pi_1, t'') = \mathcal{G}[t, t', A + B, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi] \\
\quad (\Delta^c_2, \Gamma^c_2, \theta_2 \pi_2, r'') = \\
\qquad \mathcal{G}[r[x := z], r'[x' := z], C, \Delta^a, \Delta^b, \Gamma^a \cup \{(z, A)\}, \Gamma^b \cup \{(z, A)\}, \Delta^c_2, \Gamma^c_2, \mathcal{V} \cup \{x\}, \mathcal{T}, \theta_1, \pi_1] \\
\quad (\Delta^c_3, \Gamma^c_3, \theta_3, \pi_3, s'') = \\
\qquad \mathcal{G}[s[y := z'], s'[y' := z'], C, \Delta^a, \Delta^b, \Gamma^a \cup \{(y, B)\}, \Gamma^b \cup \{(y', B)\}, \Delta^c_3, \Gamma^c_3, \mathcal{V} \cup \{y\}, \mathcal{T}, \theta_2, \pi_2] \\
\text{in } (\Gamma^c_3, \theta_3, \pi_3, \text{case } t'' \text{ of } \{x \Rightarrow r'' \mid y \Rightarrow s''\})
\end{array}$$

$$\mathcal{G}\left[\begin{array}{lll}
\text{split } t \text{ as } (x, y) & \text{split } t' \text{ as } (x', y') & \\
\text{in } s & ,\text{in } s' & , C, \Gamma^a, \Gamma^b, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi
\end{array}\right] :=$$

$$\begin{array}{l}
\text{let } z = \mathit{fresh}(\mathit{dom}(\Gamma^a) \cup \mathit{dom}(\Gamma^b) \cup \mathit{dom}(\Gamma^c)) \\
\quad z' = \mathit{fresh}(\mathit{dom}(\Gamma^a) \cup \mathit{dom}(\Gamma^b) \cup \mathit{dom}(\Gamma^c)) \\
\quad D = \mathit{typeof}[\Delta^a, \Gamma^a, t] \\
\quad E = \mathit{typeof}[\Delta^b, \Gamma^b, t'] \\
\quad A \times B = \mathcal{G}_{ty}[D, E, \Delta^c, \mathcal{T}, \theta] \text{ or } \mathit{fail} \\
\quad (\Delta^c_1, \Gamma^c_1, \theta_1, \pi_1, t'') = \mathcal{G}[t, t', A + B, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi] \\
\quad (\Delta^c_2, \Gamma^c_2, \theta_2, \pi_2, s'') = \\
\qquad \mathcal{G}[s[x := z][y := z'], s'[x' := z][y' := z'], C, \Delta^a, \Delta^b, \Gamma^a \cup \{(z, A), (z', B)\}, \\
\qquad\quad \Gamma^b \cup \{(z, A), (z', B)\}, \Gamma^c, \mathcal{V} \cup \{z, z'\}, \mathcal{T}, \theta_1, \pi_1] \\
\text{in } (\Delta^c_2, \Gamma^c_2, \theta_2, \pi_2, \text{split } t'' \text{ as } (x, y) \text{ in } s'')
\end{array}$$

$$\mathcal{G}[s, t, C, \Delta^a, \Delta^b, \Gamma^a, \Gamma^b, \Delta^c, \Gamma^c, \theta, \pi] :=$$

$$\begin{array}{l}
\text{if } \exists x.(x \mapsto (s, t)) \in \pi \\
\text{then } (\Delta^c, \Gamma^c, \theta, \pi, x) \\
\text{else let } z = \mathit{fresh}(\mathit{dom}(\Gamma^a) \cup \mathit{dom}(\Gamma^b) \cup \mathit{dom}(\Gamma^c)) \\
\qquad\quad (r, s', t', A) = \mathit{pattern}[z, s, t, \Delta^a \cup \Delta^c, \Gamma^a \cup \Gamma^c, \mathcal{V}, \mathcal{T}, \theta, \pi] \\
\qquad \text{in } (\Delta^c, \Gamma^c \cup (z, A), \theta, \pi \cup \{v \mapsto (s', t')\}, r)
\end{array}$$

Figure 5.6: Term Generalisation (Cont.)

With the definition of generalisation, all of the necessary pieces are in place for a supercompilation algorithm which generates cyclic proofs. To ensure that we indeed have an *algorithm* we must still choose some constraints to ensure that the process of program transformation will, in fact, terminate. In much of the literature on program transformation, a relation known as the *homeomorphic embedding* relation has become fashionable as a means of ensuring termination [46]. However, any termination condition is acceptable.

For the purpose of describing supercompilation we make reference to a *whistle* predicate that has access to the path and current sequent under scrutiny. This is written as $whistle[h, p]$ with h being the sequent and p the path. This *whistle* tells us on when to give up, so that we can be sure that the algorithm terminates.

In the actual implementation, a simple depth bound is used on the length of paths. The reasoning behind this is that implementations of the homeomorphic embedding are expensive and its basis is fairly arbitrary. Using the homeomorphic embedding to control unfolding leads to premature termination in some cases, and in some cases may lead to unnecessary search depth.

5.3.1 The Parts Assembled

Using these pieces we can now demonstrate a supercompilation algorithm over pre-proofs. The pieces, and how they fit together can now be explained in a pseudocode algorithm given in Figure 5.7.

The supercompilation algorithm keeps track of the formerly encountered sequents in order to find folds. This is kept as the current *path*, p which is simply a list of sequents. The supercompilation algorithm itself is defined non-deterministically. We return a stream of pre-proofs. This stream we can use to either obtain pre-proofs which are bisimulation equivalent, or we can filter these for correct proofs meeting the syntactic conditions required of terms with total proofs.

We use a non-deterministic algorithm, rather than the more traditional deterministic approach. This choice is driven by the need to find a term which meets the global condition of totality. In our implementation this global condition is expressed as a predicate which can be used to filter a potentially infinite stream monad M^{ω}. The stream, however, is generated lazily, so we need not actually develop the full space in order to find the term of interest. Further, we join these infinite streams with an (Or) operator which makes a fair choice selection from the streams and generates a new stream. The term $Fail$ we use to denote non-deterministic failure, which is the zero of the (Or) operator.

We write the initial application of supercompilation, with an empty path, to a sequent $\cdot\ ; \cdot \vdash t\ :\ A$ as $\mathcal{S}[t]$.

```
$super\ history\ sequent$ =
  $h$ := normalise and perform information propagation on $sequent$
  if our history exceeds the depth bound
  then $Fail$
  else
    for every sequent $h'$, an instance of a sequent in the history
    $history$ with substitution $\sigma$
    apply $super$ to each term of the substitution $\sigma$ to generate a $\sigma'$
    return a Cyc rule with substitution $\sigma'$
   Or
    for every sequent $h'$ in $history$, with root proof rule identical to $h$
    generalise $h$ $h'$
   Or
    unfold a function constant and return $I^\delta$
   Or
    Attempt supercompilation on subterms and introduce a type level proof rule
```

Figure 5.7: Pseudocode of Supercompilation

The basic idea of the algorithm is as follows:

- Try folds whenever there is an instance.
- Attempt generalisations whenever there is some structural similarity.
- Unfold function symbols when they are in a redex position.
- Otherwise descend structurally into the term.

We attempt all of these approaches for each sequent and join them together with a disjunction.

Because we have described bisimilarity over terms, and the supercompilation algorithm given here results in cyclic pre-proof it is necessary to describe Reification given in Section 5.4 before giving the correctness theorem for supercompliation, described in Theorem 5.4.1.

5.4 Reification

Reification is the process of producing a residual term from a cyclic proof. It is essentially an application of the Curry Howard relationship between proof steps and terms. Each proof step is reified as a term excepting for cycles, which are represented by function constants whose bodies are the remaining term. We give rules for reifying a cyclic proof in Figure 5.8 and Figure 5.9. The function $\mathcal{R} : (\mathbb{N} \to \mathbf{F}) \to D \to (\mathbf{F} \to \mathbf{Term}) \times \mathbf{Term}$ returns both a term and a function Ω between a countable number of function constants and terms. The function $\mathcal{R}$ is total since we insist that the proof or pre-proof to which it is applied is well-formed with respect to the Cyc rule. This ensures that $\mathcal{F}$ will never be applied to a natural number for which there is no function constant.

The function Ω is built up by turning cycles in the proof (which have already been marked) into functions. The process may require abstraction if the context is not empty, in which case all type variables free in the context must be turned into Λ-forms and variables free in the context must be turned into λ-forms. These can then be applied to the respective variables and the term which is stored is consequently a closed term which is a requirement on the form of Ω.

The correctness of reification is established by bisimilarity with the cyclic type derivation produced by supercompilation. It may be possible to show equivalence with a greater class of proofs but this is unnecessary.

Theorem 5.4.1 (Supercompilation is Correct)**.** *For any term t we have the bisimilarity $\forall d \in \mathcal{S}[t].t \sim \mathcal{R}[\emptyset, d]$.*

$$\mathcal{R}\left[\mathcal{F}, \frac{D}{S}\right] ::=$$

if S is a labeled recurrence with label $\dagger$
then $(\{\mathtt{f} \mapsto \mathcal{R}[\mathcal{F} \cup \{\dagger \mapsto \mathtt{f}\}, D]\}, f\ |\sigma|^{tyvar}\ |\sigma|^{tvar})$ with a fresh function constant f
otherwise

$$\mathcal{R}\left[\mathcal{F}, \frac{}{S}Unit\right] ::= (\emptyset, ())$$

$$\mathcal{R}\left[\mathcal{F}, \frac{}{\zeta\ ; \Delta\ ; \Gamma \vdash x\ :\ A}Var\right] ::= (\emptyset, x)$$

$$\mathcal{R}\left[\mathcal{F}, \frac{\Delta\ ; \Gamma \vdash \sigma : «\Delta'\ ; \Gamma'»\ \ \mathtt{\textbf{sub}}}{S}\text{Cyc}(\dagger)\right] ::=$$

$$\begin{aligned} &let\ (\Omega_1, r_1) = \mathcal{R}[\mathcal{F}, D_1] \\ &\qquad \vdots \\ &\qquad (\Omega_n, r_n) = \mathcal{R}[\mathcal{F}, D_n] \\ &in\ (\Omega_1 \cup \cdots \cup \Omega_n, \mathcal{F}(\dagger)\ |\sigma|^{type}\ |\sigma|^{term}) \end{aligned}$$

$$\mathcal{R}\left[\mathcal{F}, \frac{D}{\zeta\ ; \Delta\ ; \Gamma \vdash \lambda x\!:\!A.\ t\ :\ A \to B}I^{\to}\right] ::=$$

$$let\ (\Omega, r) = \mathcal{R}[\mathcal{F}, D]\ in\ (\Omega, \lambda x\!:\!A.\ r)$$

$$\mathcal{R}\left[\mathcal{F}, \frac{D}{\zeta\ ; \Delta\ ; \Gamma \vdash \Lambda X.\ t\ :\ \forall X.\ A}I^{\forall}\right] ::=$$

$$let\ (\Omega, r) = \mathcal{R}[\mathcal{F}, D]\ in\ (\Omega, \Lambda X.\ r)$$

$$\mathcal{R}\left[\mathcal{F}, \frac{D}{\zeta\ ; \Delta\ ; \Gamma \vdash \text{left}(t, A + B)\ :\ A + B}I^{+}_{L}\right] ::=$$

$$let\ (\Omega, r) = \mathcal{R}[\mathcal{F}, D]\ in\ (\Omega, \text{left}(r, A + B))$$

$$\mathcal{R}\left[\mathcal{F}, \frac{D}{\zeta\ ; \Delta\ ; \Gamma \vdash \text{right}(t, A + B)\ :\ A + B}I^{+}_{R}\right] ::=$$

$$let\ (\Omega, r) = \mathcal{R}[\mathcal{F}, D]\ in\ (\Omega, \text{right}(r, A + B))$$

Figure 5.8: Reification Rules

$$\mathcal{R}\left[\mathcal{F}, \frac{D1\ D2}{S} I^{\times}\right] ::=$$

$$\mathit{let}\ (\Omega, t) = \mathcal{R}[\mathcal{F}, D1]\ (\Omega', s) = \mathcal{R}[\mathcal{F}, D2]\ \mathit{in}\ (\Omega \cup \Omega', (t, s))$$

$$\mathcal{R}\left[\mathcal{F}, \frac{D}{\zeta\ ; \Delta\ ; \Gamma \vdash \mathrm{in}_{\alpha}(t, \alpha\hat{X}.\ A)\ :\ \alpha\hat{X}.\ A} I^{\alpha}\right] ::=$$

$$\mathit{let}\ (\Omega, r) = \mathcal{R}[\mathcal{F}, D]\ \mathit{in}\ (\Omega, \mathrm{in}_{\alpha}(r, \alpha\hat{X}.\ A))$$

$$\mathcal{R}\left[\mathcal{F}, \frac{D1\ D2}{S} E^{\rightarrow}\right] ::=$$

$$\mathit{let}\ (\Omega, s) = \mathcal{R}[\mathcal{F}, D1]\ (\Omega', t) = \mathcal{R}[\mathcal{F}, D2]\ \mathit{in}\ (\Omega \cup \Omega', s\ t)$$

$$\mathcal{R}\left[\mathcal{F}, \frac{D}{\zeta\ ; \Delta\ ; \Gamma \vdash t[A]\ :\ B} E^{\forall}\right] ::= \mathit{let}\ (\Omega, r) = \mathcal{R}[\mathcal{F}, D]\ \mathit{in}\ (\Omega, r[A])$$

$$\mathcal{R}\left[\mathcal{F}, \frac{D1\quad D1\quad D3}{\zeta\ ; \Delta\ ; \Gamma \vdash \text{case}\ r\ \text{of}\ \{x \Rightarrow s \mid y \Rightarrow t\}\ :\ C} E^{+}\right] ::=$$

$$\begin{array}{l} \mathit{let}\ (\Omega, r) = \mathcal{R}[\mathcal{F}, D1] \\ \quad\ (\Omega', s) = \mathcal{R}[\mathcal{F}, D2] \\ \quad\ (\Omega'', t) = \mathcal{R}[\mathcal{F}, D3] \\ \mathit{in}\ (\Omega \cup \Omega' \cup \Omega'', \text{case}\ r\ \text{of}\ \{x \Rightarrow s \mid y \Rightarrow t\}) \end{array}$$

$$\mathcal{R}\left[\mathcal{F}, \frac{D1\quad D2}{\zeta\ ; \Delta\ ; \Gamma \vdash \text{split}\ t\ \text{as}\ (x, y)\ \text{in}\ s\ :\ C} E^{\times}\right] ::=$$

$$\begin{array}{l} \mathit{let}\ (\Omega, r) = \mathcal{R}[\mathcal{F}, D1] \\ \quad\ (\Omega', u) = \mathcal{R}[\mathcal{F}, D2] \\ \mathit{in}\ (\Omega \cup \Omega', \text{split}\ r\ \text{as}\ (x, y)\ \text{in}\ u) \end{array}$$

$$\mathcal{R}\left[\mathcal{F}, \frac{D}{\zeta\ ; \Delta\ ; \Gamma \vdash t\ :\ \alpha\hat{X}.\ A} E^{\alpha}\right] ::= \mathit{let}\ (\Omega, r) = \mathcal{R}[\mathcal{F}, D]\ \mathit{in}\ (\Omega, \mathrm{out}_{\alpha}(r, \alpha\hat{X}.\ A))$$

$$\mathcal{R}\left[\mathcal{F}, \frac{D}{S} \mathit{Delta}\right] ::= \mathcal{R}[\mathcal{F}, D]$$

Figure 5.9: Reification Rules (Cont.)

Proof. The bisimilarity of terms here is established by looking at the proofs which can be produced by supercompilation for a given term t and then showing that reification faithfully preserves the important behavioural aspects of proof structure.

The proof proceeds by coinduction.

We ensure termination with a whistle hence we can assume the inductive hypothesis holds for any computed output.

The supercompilation algorithm will result in one of four cases. We will either create a cycle, unfold, generalise or supercompile subterms.

- Subterms: Since normalisation preserves bisimulation equivalence we can assume that the original term is bisimulation equivalent if the subterms are bisimulation equivalent.
- Generalisation: Generalisations are bisimilar by β-equivalence.
- Unfolding: Function constants are unfolded in the term. Function constant unfolding occurs in the redex position and therefore is consistent with the evaluation relation and consequently bisimulation.
- Cycles: In the production of cycles we look only at the list of previously encountered terms in terms of outer-proof steps. This means that any cycle created must necessarily include a proof step which is either an introduction or elimination rule, or a Delta rule. The Delta rule is the only one that can lead to arbitrary non-termination because otherwise terms are finite. This leads to two cases, either we have only Delta steps or we have Delta steps with intermediate proof steps.
  - Delta: If there are only Delta steps before a cycle this means that we are respecting the non-termination of the original program since a cycle with only Deltas in the original program was a program Ω such that two or more constants refer to each other. We can therefore use the fact that application of bisimilar constants is bisimilar together with the coinductive hypothesis.
  - Intermediate: If there is an intermediate proof-step prior to creating a cycle, we produce the action associated with that proof-step for our bisimulation proof on both terms (since $\mathcal{R}$ will produce a term for all but Delta steps). This leads to an action which is duplicated by the original term since the behaviour arose from normalisation which is

bisimilar to the original. Again we can use the fact that application of bisimilar terms is bisimilar together with the coinductive hypothesis.

$\square$

5.5 Example

The example given here comes from work done by Bertot and Komendantskaya in [9]. It was noted in that paper that some programs do not meet the guardedness criteria but are plainly productive. The particular example is notable since it makes use of both greatest and least fixed points. The framework presented in the paper made use of relations which demonstrate *always, eventually* type behaviour, which we also make use of in Section 6.2.

The natural numbers are defined identically to the co-natural numbers that were presented earlier, aside from the use of the least fixed point, which excludes the point at infinity and is given by the type $\mathbb{N} = \mu X.1 + X$. We define streams of natural numbers co-inductively as the greatest fixed point of a pair of a natural number and stream of natural numbers. The stream type over an arbitrary type A is given by $[\![A]\!] = \nu X.(A \times X)$.

```
$\Omega(\texttt{true})$   :=  $\mathrm{left}((), \mathbb{1} + \mathbb{1})$
$\Omega(\texttt{false})$  :=  $\mathrm{right}((), \mathbb{1} + \mathbb{1})$
$\Omega(\texttt{zero})$   :=  $\mathrm{in}_\mu(\mathrm{left}((), \mathbb{1} + \mathbb{N}), \mathbb{N})$
$\Omega(\texttt{succ})$   :=  $\lambda x : \mathbb{N}.\ \mathrm{in}_\nu(\mathrm{right}(x, \mathbb{1} + \mathbb{N}), \mathbb{N})$
$\Omega(\texttt{cons})$   :=  $\lambda x : \mathbb{N}, s : [\![\mathbb{N}]\!].\ \mathrm{in}_\nu((x, s), [\![\mathbb{N}]\!])$
$\Omega(\texttt{le})$     :=  $\lambda x\ y : \mathbb{N}.$
                case $(\mathrm{out}_\mu(x, \mathbb{N}))$ of
                  $\{\ z \Rightarrow$ true
                  $|\ x' \Rightarrow$
                    case $(\mathrm{out}_\mu(y, \mathbb{N}))$ of
                      $\{\ z \Rightarrow$ false
                      $|\ y' \Rightarrow$ le $x'\ y'\}\}$
$\Omega(\texttt{pred})$   :=  $\lambda x\ y : \mathbb{N}.$
                case $(\mathrm{out}_\mu(x, \mathbb{N}))$ of
                  $\{\ z \Rightarrow$ zero
                  $|\ n \Rightarrow n\}$
$\Omega(\texttt{f})$      :=  $\lambda s : [\![\mathbb{N}]\!].$
                split $(\mathrm{out}_\nu(s, \mathbb{N}))$ as as $(x, s')$ in
                  split $(\mathrm{out}_\nu(s', \mathbb{N}))$ as $(y, s'')$ in
                    case (le $x\ y$) of
                      $\{\ yes \Rightarrow$ cons $x$ (f $s'$)
                      $|\ no \Rightarrow$ f (cons (pred $x$) $s'$)$\}\}$
```

Figure 5.10: Stream Program

From the program in Figure 5.10 we can inspect the term for the function constant `f`. This term is productive in the sense that in all cases, it will eventually produce some element of an output stream. However, it does so by recursion on its first argument until it has reduced it below the subsequent element of the stream. This means that the program, as written does not pass the guardedness condition. The failure to pass the condition is apparent in the cyclic pre-proof given in Figure 5.11.

$$\dfrac{\cdot\,;\cdot \vdash \texttt{f} : \llbracket\mathbb{N}\rrbracket \to \llbracket\mathbb{N}\rrbracket \qquad \dfrac{\cdot\,;\cdot \vdash \texttt{cons} : \mathbb{N} \to \llbracket\mathbb{N}\rrbracket \to \llbracket\mathbb{N}\rrbracket \qquad \dfrac{\cdot\,;\cdot \vdash \texttt{pred} : \mathbb{N} \to \mathbb{N} \quad \cdot\,;\{x:\mathbb{N}\} \vdash x : \mathbb{N}}{\cdot\,;\{x:\mathbb{N}\} \vdash \texttt{pred}\ x : \mathbb{N}} \qquad \cdot\,;\{s':\llbracket\mathbb{N}\rrbracket\} \vdash s' : \llbracket\mathbb{N}\rrbracket}{\cdot\,;\{x:\mathbb{N}, s':\llbracket\mathbb{N}\rrbracket\} \vdash \texttt{cons (pred}\ x\texttt{)}\ s' : \llbracket\mathbb{N}\rrbracket}}{\cdot\,;\{x:\mathbb{N}, s':\llbracket\mathbb{N}\rrbracket\} \vdash \texttt{f (cons (pred}\ x\texttt{)}\ s'\texttt{)} : \llbracket\mathbb{N}\rrbracket} \qquad (\mathcal{C})$$

$$\dfrac{\cdot\,;\cdot \vdash \texttt{cons} : \mathbb{N} \to \llbracket\mathbb{N}\rrbracket \to \llbracket\mathbb{N}\rrbracket \qquad \cdot\,;\{x:\mathbb{N}\} \vdash x : \mathbb{N} \qquad \dfrac{\cdot\,;\cdot \vdash \texttt{f} : \llbracket\mathbb{N}\rrbracket \to \llbracket\mathbb{N}\rrbracket \qquad \cdot\,;\{s':\llbracket\mathbb{N}\rrbracket\} \vdash s' : \llbracket\mathbb{N}\rrbracket}{\cdot\,;\{s':\llbracket\mathbb{N}\rrbracket\} \vdash \texttt{f}\ s' : \llbracket\mathbb{N}\rrbracket}}{\cdot\,;\{x:\mathbb{N}, s':\llbracket\mathbb{N}\rrbracket\} \vdash \texttt{cons}\ x\ \texttt{(f}\ s'\texttt{)} : \llbracket\mathbb{N}\rrbracket} \qquad (\mathcal{D})$$

$$\dfrac{\dfrac{\cdot\,;\{y:\mathbb{N}\} \vdash y : \mathbb{N}}{\cdot\,;\{y:\mathbb{N}\} \vdash \mathrm{out}_\mu(y,\mathbb{N}) : \mathbb{1}+\mathbb{N}} \qquad \cdot\,;\cdot \vdash \texttt{false} : \mathbb{1}+\mathbb{1} \qquad \cdot\,;\{x':\mathbb{N}, y':\mathbb{N}\} \vdash \texttt{le}\ x'\ y' : \mathbb{1}+\mathbb{1}}{\cdot\,;\{x':\mathbb{N}, y:\mathbb{N}\} \vdash \mathrm{case\ out}_\mu(y,\mathbb{N})\ \mathrm{of}\ \{z \Rightarrow \texttt{false} \mid y' \Rightarrow \texttt{le}\ x'\ y'\} : \mathbb{1}+\mathbb{1}} \qquad (\mathcal{E})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot\,;\{x:\mathbb{N}\} \vdash x : \mathbb{N}}{\cdot\,;\{x:\mathbb{N}\} \vdash \mathrm{out}_\mu(x,\mathbb{N}) : \mathbb{1}+\mathbb{N}} \qquad \cdot\,;\cdot \vdash \texttt{true} : \mathbb{1}+\mathbb{1} \qquad \mathcal{E}}{\cdot\,;\{x:\mathbb{N}, y:\mathbb{N}\} \vdash \mathrm{case\ out}_\mu(x,\mathbb{N})\ \mathrm{of}\ \{z \Rightarrow \texttt{true} \mid x' \Rightarrow \mathrm{case\ out}_\mu(y,\mathbb{N})\ \mathrm{of}\ \{z \Rightarrow \texttt{false} \mid y' \Rightarrow \texttt{le}\ x'\ y'\}\} : \mathbb{1}+\mathbb{1}}}{\begin{array}{ll}\cdot\,;\cdot \vdash \lambda x\ y:\mathbb{N}.\ \mathrm{case}\ (\mathrm{out}_\mu(x,\mathbb{N}))\ \mathrm{of} & : \mathbb{N} \to \mathbb{N} \to \mathbb{1}+\mathbb{1} \\ \quad \{\ z \Rightarrow \texttt{true} & \\ \quad \mid x' \Rightarrow \mathrm{case\ out}_\mu(y,\mathbb{N})\ \mathrm{of} & \\ \qquad \{\ z \Rightarrow \texttt{false} & \\ \qquad \mid y' \Rightarrow \texttt{le}\ x'\ y'\}\} & \end{array}}}{} \qquad (\mathcal{F})$$

$$\dfrac{\dfrac{\mathcal{F} \qquad \cdot\,;\{x:\mathbb{N}\} \vdash x : \mathbb{N} \qquad \cdot\,;\{y:\mathbb{N}\} \vdash y : \mathbb{N}}{\cdot\,;\{x:\mathbb{N}, y:\mathbb{N}\} \vdash \texttt{le}\ x\ y : \mathbb{1}+\mathbb{1}} \qquad \mathcal{D} \qquad \mathcal{C}}{\cdot\,;\{x:\mathbb{N}, y:\mathbb{N}, s':\llbracket\mathbb{N}\rrbracket\} \vdash \mathrm{case}\ (\texttt{le}\ x\ y)\ \mathrm{of}\ \{\mathit{yes} \Rightarrow \texttt{cons}\ x\ \texttt{(f}\ s'\texttt{)} \mid \mathit{no} \Rightarrow \texttt{f (cons (pred}\ x\texttt{)}\ s'\texttt{)}\} : \llbracket\mathbb{N}\rrbracket} \qquad (\mathcal{G})$$

$$\dfrac{\dfrac{\dfrac{\cdot\,;\{s:\llbracket\mathbb{N}\rrbracket\} \vdash s : \llbracket\mathbb{N}\rrbracket}{\cdot\,;\{s:\llbracket\mathbb{N}\rrbracket\} \vdash \mathrm{out}(s,\llbracket\mathbb{N}\rrbracket) : \mathbb{N} \times \llbracket\mathbb{N}\rrbracket} \qquad \dfrac{\dfrac{\cdot\,;\{s':\llbracket\mathbb{N}\rrbracket\} \vdash s' : \llbracket\mathbb{N}\rrbracket}{\cdot\,;\{s':\llbracket\mathbb{N}\rrbracket\} \vdash \mathrm{out}(s',\llbracket\mathbb{N}\rrbracket) : \mathbb{N} \times \llbracket\mathbb{N}\rrbracket} \qquad \mathcal{G}}{\begin{array}{ll}\cdot\,;\{x:\mathbb{N}, s':\llbracket\mathbb{N}\rrbracket\} \vdash \mathrm{split\ out}(s',\llbracket\mathbb{N}\rrbracket)\ \mathrm{as}\ (y,s'') & : \llbracket\mathbb{N}\rrbracket \\ \quad \mathrm{in\ case}\ (\texttt{le}\ x\ y)\ \mathrm{of} & \\ \qquad \{\ \mathit{yes} \Rightarrow \texttt{cons}\ x\ \texttt{(f}\ s'\texttt{)} & \\ \qquad \mid \mathit{no} \Rightarrow \texttt{f (cons (pred}\ x\texttt{)}\ s'\texttt{)}\} & \end{array}}}{\begin{array}{ll}\cdot\,;\{s:\llbracket\mathbb{N}\rrbracket\} \vdash \mathrm{split\ out}(s,\llbracket\mathbb{N}\rrbracket)\ \mathrm{as}\ (x,s') & : \llbracket\mathbb{N}\rrbracket \\ \quad \mathrm{in\ split\ out}(s',\llbracket\mathbb{N}\rrbracket)\ \mathrm{as}\ (y,s'') & \\ \quad \mathrm{in\ case}\ (\texttt{le}\ x\ y)\ \mathrm{of} & \\ \qquad \{\ \mathit{yes} \Rightarrow \texttt{cons}\ x\ \texttt{(f}\ s'\texttt{)} & \\ \qquad \mid \mathit{no} \Rightarrow \texttt{f (cons (pred}\ x\texttt{)}\ s'\texttt{)}\} & \end{array}}}{\dfrac{\begin{array}{ll}\cdot\,;\cdot \vdash \lambda s:\llbracket\mathbb{N}\rrbracket.\ \mathrm{split\ out}(s,\llbracket\mathbb{N}\rrbracket)\ \mathrm{as}\ (x,s') & : \llbracket\mathbb{N}\rrbracket \to \llbracket\mathbb{N}\rrbracket \\ \quad \mathrm{in\ split\ out}(s',\llbracket\mathbb{N}\rrbracket)\ \mathrm{as}\ (y,s'') & \\ \quad \mathrm{in\ case}\ (\texttt{le}\ x\ y)\ \mathrm{of} & \\ \qquad \{\ \mathit{yes} \Rightarrow \texttt{cons}\ x\ \texttt{(f}\ s'\texttt{)} & \\ \qquad \mid \mathit{no} \Rightarrow \texttt{f (cons (pred}\ x\texttt{)}\ s'\texttt{)}\} & \end{array}}{\cdot\,;\cdot \vdash \texttt{f} : \llbracket\mathbb{N}\rrbracket \to \llbracket\mathbb{N}\rrbracket}} \qquad (\mathcal{H})$$

Figure 5.11: Pre-proof for f

We can transform this pre-proof into a valid proof using supercompilation. The supercompiled version of this proof, which meets the guardedness criterion is given in Figure 5.12 and Figure 5.13.

This alternative proof form is produced simply by removing intermediate elimination rules and folding. We should pay special attention to the recurrences which are labelled by †,‡ and □.

Every fold to † yields cycles which meet the guardedness condition. This can easily be seen as the prior proof rules are all admissible and the final rule prior to the Cyc rule is guarded by a `cons` (which unfolds to an (right) and (in_μ)). For reasons of space we use binary derived rule (I^{Cons}) for `cons` which the reader can easily see is simply (right) combined with (in_ν) and is therefore a guard.

$$\dfrac{\zeta\,;\Delta\,;\Gamma \vdash x : \mathbb{N} \qquad \zeta\,;\Delta\,;\Gamma \vdash s : [\![\mathbb{N}]\!]}{\zeta\,;\Delta\,;\Gamma \vdash \mathsf{cons}\ x\ s : [\![\mathbb{N}]\!]}\ I^{Cons}$$

$$\Downarrow$$

$$\dfrac{\dfrac{\dfrac{\zeta\,;\Delta\,;\Gamma \vdash x : \mathbb{N} \qquad \zeta\,;\Delta\,;\Gamma \vdash s : [\![\mathbb{N}]\!]}{\zeta\,;\Delta\,;\Gamma \vdash (x,s) : \mathbb{N} \times [\![\mathbb{N}]\!]}\ I^{\times}}{\zeta\,;\Delta\,;\Gamma \vdash \mathrm{in}_\nu((x,s),[\![\mathbb{N}]\!]) : [\![\mathbb{N}]\!]}\ I^{\nu}}{\zeta\,;\Delta\,;\Gamma \vdash \mathsf{cons}\ x\ s : [\![\mathbb{N}]\!]}\ I^{\delta}$$

Every other symbol is used to mark an inductive cycle having a structural reduction in some parameter. Since all cycles are inductive or coinductive, this final form yields a total proof which meets the syntactic conditions on proofs simply by means of supercompilation.

It is important for the reader to note the use of I^δ in the proof. This particular proof relies critically on information propagation and not simply normalisation. Without the propagation of information there will be redundant impossible branches that are not eliminated which leads to a failure to meet the guardedness condition. Specifically this fact is required in the ‡ branch.

From this proof we can use reification to obtain a bisimilar program which is given in Figure 5.14. The resulting program demonstrates the productivity much more directly. Each function is either descending recursively on an inductive argument and then terminating with a guard, as in the case of (`g`) and (`h`) or is directly productive, as with (`f`).

5.6 Related Work

Program transformation has an extensive literature and various techniques have been described for a range of practical programming languages. Program transformation was a central point behind Backus's idea of liberation from the Von Neumann machine [6] and gave one motivation for the development of functional languages.

The most well known technique from program transformation is partial evaluation, however there are many techniques including unfold/fold[14], deforestation[92], supercompilation [87] and others. Many compiler optimisations can be seen as special cases of program transformational techniques.

$$\dfrac{\cdot\,;\cdot\,;\{x:\mathbb{N}\}\vdash x : \mathbb{N} \qquad \dfrac{\cdot\,;\{s:\llbracket\mathbb{N}\rrbracket\}\vdash (s,(\texttt{cons}\ y\ s'')) : \text{«}\cdot\,;\{y:\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\text{»}\ \textbf{sub}}{\{\dagger\}\,;\cdot\,;\{x':\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash \texttt{f}\ (\texttt{cons}\ y\ s'') : \llbracket\mathbb{N}\rrbracket}\,\mathrm{Cyc}(\dagger)}{\{\dagger\}\,;\cdot\,;\{x:\mathbb{N},y:\mathbb{N},s:\llbracket\mathbb{N}\rrbracket\}\vdash \texttt{cons}\ x\ (\texttt{f}\ (\texttt{cons}\ y\ s'')) : \llbracket\mathbb{N}\rrbracket}\,I^{Cons} \tag{$\mathcal{A}$}$$

$$\dfrac{\cdot\,;\cdot\,;\cdot\vdash \texttt{zero} : \mathbb{N} \qquad \dfrac{\cdot\,;\{s:\llbracket\mathbb{N}\rrbracket\}\vdash (s,(\texttt{cons zero}\ s'')) : \text{«}\cdot\,;\{s'':\llbracket\mathbb{N}\rrbracket\}\text{»}\ \textbf{sub}}{\{\square,\dagger\}\,;\cdot\,;\{s'':\llbracket\mathbb{N}\rrbracket\}\vdash \texttt{f}\ (\texttt{cons zero}\ s'') : \llbracket\mathbb{N}\rrbracket}\,\mathrm{Cyc}(\dagger)}{\{\square,\dagger\}\,;\cdot\,;\{s'':\llbracket\mathbb{N}\rrbracket\}\vdash \texttt{cons zero}\ (\texttt{f}\ (\texttt{cons zero}\ s''))) : \llbracket\mathbb{N}\rrbracket}\,I^{Cons} \tag{$\mathcal{B}$}$$

$$\dfrac{\cdot\,;\{x:\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash (x,x') : \text{«}\cdot\,;\{x':\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\text{»}\ \textbf{sub}}{\{\square,\dagger\}\,;\cdot\,;\{x':\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash \texttt{f}\ (\texttt{cons}\ (\texttt{pred}\ x')\ (\texttt{cons zero}\ s'')) : \llbracket\mathbb{N}\rrbracket}\,\mathrm{Cyc}(\square) \tag{$\mathcal{C}$}$$

$$\dfrac{\{\dagger\}\,;\cdot\,;\{x:\mathbb{N},y:\mathbb{N}\}\vdash \texttt{le}\ x'\ y' : \mathbb{B} \qquad \mathcal{A} \qquad \mathcal{Z}}{\begin{array}{l}\{\dagger,\ddagger\}\,;\cdot\,;\{x':\mathbb{N},y':\mathbb{N},x:\mathbb{N},y:\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash \text{case}\ (\texttt{le}\ x'\ y')\ \text{of} \qquad : \llbracket\mathbb{N}\rrbracket\\ \qquad\qquad \{\ t \Rightarrow \texttt{cons}\ x\ (\texttt{f}\ (\texttt{cons}\ y\ s''))\\ \qquad\qquad \mid f \Rightarrow \texttt{f}\ (\texttt{cons}\ (\texttt{pred}\ x)\ s'')\}\end{array}}\,E^{+} \tag{$\mathcal{D}$}$$

$$\dfrac{\dfrac{\dfrac{\cdot\,;\cdot\,;\{x':\mathbb{N}\}\vdash x' : \mathbb{N}}{\cdot\,;\cdot\,;\{x':\mathbb{N}\}\vdash \text{out}(x',\mathbb{N}) : \mathbb{1}+\mathbb{N}} \qquad \mathcal{B} \qquad \mathcal{C}}{\begin{array}{l}\{\square,\dagger,\ddagger\}\,;\cdot\,;\{x':\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash \text{case out}(x',\mathbb{N})\ \text{of} \qquad : \llbracket\mathbb{N}\rrbracket\\ \qquad\qquad \{\ z \Rightarrow \texttt{cons zero}\ (\texttt{f}\ (\texttt{cons zero}\ s'')))\\ \qquad\qquad \mid x'' \Rightarrow \texttt{f}\ (\texttt{cons}\ (\texttt{pred}\ x')\ (\texttt{cons zero}\ s''))\}\end{array}}\,E^{+}}{\{\dagger,\ddagger\}\,;\cdot\,;\{x':\mathbb{N},y':\mathbb{N},x:\mathbb{N},y:\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash \texttt{f}\ (\texttt{cons}\ (\texttt{pred}\ x)\ (\texttt{cons zero}\ s'')) : \llbracket\mathbb{N}\rrbracket\ \textbf{label}\ \square}\,I^{\delta} \tag{$\mathcal{E}$}$$

$$\dfrac{\dfrac{\{\dagger\}\,;\cdot\,;\{y:\mathbb{N}\}\vdash y : \mathbb{N}}{\{\dagger\}\,;\cdot\,;\{y:\mathbb{N}\}\vdash \text{out}(y,\mathbb{N}) : \mathbb{1}+\mathbb{N}}\,E^{\mu} \qquad \mathcal{E} \qquad \mathcal{D}}{\begin{array}{l}\{\dagger\}\,;\cdot\,;\{x':\mathbb{N},x:\mathbb{N},y:\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash \text{case}\ (\text{out}(y,\mathbb{N}))\ \text{of} \qquad : \llbracket\mathbb{N}\rrbracket\\ \qquad\qquad \{\ z \Rightarrow \texttt{f}\ (\texttt{cons}\ (\texttt{pred}\ x)\ (\texttt{cons zero}\ s''))\\ \qquad\qquad \mid y' \Rightarrow \cdots\}\end{array}}\,E^{+} \tag{$\mathcal{F}$}$$

$$\dfrac{\{\dagger\}\,;\cdot\,;\cdot\vdash \texttt{zero} : \mathbb{N} \qquad \dfrac{\cdot\,;\{s:\llbracket\mathbb{N}\rrbracket\}\vdash (s,\texttt{cons}\ y\ s'') : \text{«}\cdot\,;\{y:\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\text{»}\ \textbf{sub}}{\{\dagger\}\,;\cdot\,;\{x:\mathbb{N},y:\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash \texttt{f}\ (\texttt{cons}\ y\ s'') : \llbracket\mathbb{N}\rrbracket}\,\mathrm{Cyc}(\dagger)}{\{\dagger\}\,;\cdot\,;\{x:\mathbb{N},y:\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash \texttt{cons zero}\ (\texttt{f}\ (\texttt{cons}\ y\ s'')) : \llbracket\mathbb{N}\rrbracket}\,I^{Cons} \tag{$\mathcal{G}$}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\dagger\}\,;\cdot\,;\{x:\mathbb{N}\}\vdash x : \mathbb{N}}{\{\dagger\}\,;\cdot\,;\{x:\mathbb{N}\}\vdash \text{out}(x,\mathbb{N}) : \mathbb{1}+\mathbb{N}}\,E^{\mu} \qquad \mathcal{G} \qquad \mathcal{F}}{\begin{array}{l}\{\dagger\}\,;\cdot\,;\{x:\mathbb{N},y:\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash \text{case}\ (\text{out}(x,\mathbb{N}))\ \text{of} \qquad : \llbracket\mathbb{N}\rrbracket\\ \qquad\qquad \{\ z \Rightarrow \texttt{cons zero}\ (\texttt{f}\ (\texttt{cons}\ y\ s''))\\ \qquad\qquad \mid x' \Rightarrow \cdots\}\end{array}}\,E^{+}}{}}{\begin{array}{l}\{\dagger\}\,;\cdot\,;\{x:\mathbb{N},y:\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash \text{case}\ (\texttt{le}\ x\ y)\ \text{of} \qquad : \llbracket\mathbb{N}\rrbracket\\ \qquad\qquad \{\ \mathit{yes} \Rightarrow \texttt{cons}\ x\ (\texttt{f}\ (\texttt{cons}\ y\ s''))\\ \qquad\qquad \mid \mathit{no} \Rightarrow \texttt{f}\ (\texttt{cons}\ (\texttt{pred}\ x)\ (\texttt{cons}\ y\ s''))\}\end{array}}\,I^{\delta} \tag{$\mathcal{H}$}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot\,;\cdot\,;\{s:\llbracket\mathbb{N}\rrbracket\}\vdash s : \llbracket\mathbb{N}\rrbracket}{\cdot\,;\cdot\,;\{s:\llbracket\mathbb{N}\rrbracket\}\vdash \text{out}(s,\llbracket\mathbb{N}\rrbracket) : \mathbb{N}\times\llbracket\mathbb{N}\rrbracket}\,E^{\mu} \qquad \dfrac{\dfrac{\cdot\,;\cdot\,;\{s':\llbracket\mathbb{N}\rrbracket\}\vdash s' : \llbracket\mathbb{N}\rrbracket}{\cdot\,;\cdot\,;\{s':\llbracket\mathbb{N}\rrbracket\}\vdash \text{out}(s',\llbracket\mathbb{N}\rrbracket) : \mathbb{N}\times\llbracket\mathbb{N}\rrbracket} \qquad \mathcal{H}}{\begin{array}{l}\{\dagger\}\,;\cdot\,;\{x:\mathbb{N},s':\llbracket\mathbb{N}\rrbracket\}\vdash \text{split out}(s',\llbracket\mathbb{N}\rrbracket)\ \text{as}\ (y,s'') \qquad : \llbracket\mathbb{N}\rrbracket\\ \qquad\qquad \text{in case}\ (\texttt{le}\ x\ y)\ \text{of}\\ \qquad\qquad\quad \{\ \mathit{yes} \Rightarrow \texttt{cons}\ x\ (\texttt{f}\ s')\\ \qquad\qquad\quad \mid \mathit{no} \Rightarrow \texttt{f}\ (\texttt{cons}\ (\texttt{pred}\ x)\ s')\}\end{array}}\,E^{\times}}{\begin{array}{l}\cdot\,;\cdot\,;\{s:\llbracket\mathbb{N}\rrbracket\}\vdash \text{split out}(s,\llbracket\mathbb{N}\rrbracket)\ \text{as}\ (x,s') \qquad : \llbracket\mathbb{N}\rrbracket\ \textbf{label}\ \dagger\\ \qquad\qquad \text{in split out}(s',\llbracket\mathbb{N}\rrbracket)\ \text{as}\ (y,s'')\\ \qquad\qquad\quad \text{in case}\ (\texttt{le}\ x\ y)\ \text{of}\\ \qquad\qquad\qquad \{\ \mathit{yes} \Rightarrow \texttt{cons}\ x\ (\texttt{f}\ s')\\ \qquad\qquad\qquad \mid \mathit{no} \Rightarrow \texttt{f}\ (\texttt{cons}\ (\texttt{pred}\ x)\ s')\}\end{array}}\,E^{\times}}{\begin{array}{l}\cdot\,;\cdot\,;\cdot\vdash \lambda s:\llbracket\mathbb{N}\rrbracket.\ \text{split out}(s,\llbracket\mathbb{N}\rrbracket)\ \text{as}\ (x,s') \qquad : \llbracket\mathbb{N}\rrbracket\to\llbracket\mathbb{N}\rrbracket\\ \qquad\qquad \text{in split out}(s',\llbracket\mathbb{N}\rrbracket)\ \text{as}\ (y,s'')\\ \qquad\qquad\quad \text{in case}\ (\texttt{le}\ x\ y)\ \text{of}\\ \qquad\qquad\qquad \{\ \mathit{yes} \Rightarrow \texttt{cons}\ x\ (\texttt{f}\ s')\\ \qquad\qquad\qquad \mid \mathit{no} \Rightarrow \texttt{f}\ (\texttt{cons}\ (\texttt{pred}\ x)\ s')\}\end{array}}\,I^{\to}}{\cdot\,;\cdot\,;\cdot\vdash \texttt{f} : \llbracket\mathbb{N}\rrbracket\to\llbracket\mathbb{N}\rrbracket}\,I^{\delta} \tag{$\mathcal{I}$}$$

Figure 5.12: Proof for f, part I

$$\dfrac{\cdot\,;\{x:\mathbb{N}, y':\mathbb{N}:\llbracket\mathbb{N}\rrbracket\} \vdash (x,x') \circ (y',y') \circ (s'',s'') : \text{«}\cdot\,;\{x':\mathbb{N},y':\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\text{»} \ \mathbf{sub}}{\{\ddagger\}\,;\cdot\,;\{x':\mathbb{N},y:\mathbb{N},s'':\mathbb{N}\}\vdash \texttt{f (pred } x'\texttt{) (cons (succ } y'\texttt{) } s''\texttt{)} : \llbracket\mathbb{N}\rrbracket}\,\mathrm{Cyc}(\ddagger) \tag{$\mathcal{W}$}$$

$$\dfrac{\cdot\,;\cdot\,;\{x':\mathbb{N}\}\vdash x' : \mathbb{N} \qquad \dfrac{\cdot\,;\{s:\llbracket\mathbb{N}\rrbracket\}\vdash (s,(\texttt{cons } y\ s'')) : \text{«}\cdot\,;\{y:\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\text{»}\ \mathbf{sub}}{\{\dagger\}\,;\cdot\,;\{y':\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash \texttt{f (cons (succ } y'\texttt{) } s''\texttt{)} : \llbracket\mathbb{N}\rrbracket}\,\mathrm{Cyc}(\dagger)}{\{\dagger\}\,;\cdot\,;\{x':\mathbb{N},y:\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash \texttt{cons } x'\ \texttt{(f (cons (succ } y'\texttt{) } s''\texttt{))} : \llbracket\mathbb{N}\rrbracket}\,I^{Cons} \tag{$\mathcal{V}$}$$

$$\dfrac{\cdot\,;\cdot\,;\cdot\vdash \texttt{zero} : \mathbb{N} \qquad \dfrac{\cdot\,;\{s:\llbracket\mathbb{N}\rrbracket\}\vdash (s,(\texttt{cons (succ } y'\texttt{) } s'')) : \text{«}\cdot\,;\{y':\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\text{»}\ \mathbf{sub}}{\{\dagger\}\,;\cdot\,;\{y':\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash \texttt{f (cons (succ } y'\texttt{) } s''\texttt{)} : \llbracket\mathbb{N}\rrbracket}\,\mathrm{Cyc}(\dagger)}{\{\dagger\}\,;\cdot\,;\{y':\mathbb{N},s'':\llbracket\mathbb{N}\rrbracket\}\vdash \texttt{cons zero (f (cons (succ } y'\texttt{) } s''\texttt{))} : \llbracket\mathbb{N}\rrbracket}\,I^{Cons} \tag{$\mathcal{X}$}$$

$$\dfrac{\{\dagger,\ddagger\}\,;\cdot\,;\{x'':\mathbb{N},y':\mathbb{N}\}\vdash \texttt{le } x''\ y' : \mathbb{B} \qquad \mathcal{V} \qquad \mathcal{W}}{\begin{array}{l}\{\dagger,\ddagger\}\,;\cdot\,;\{x:\mathbb{N},x':\mathbb{N},x'':\mathbb{N},y':\mathbb{N}\}\vdash \text{case } \texttt{le } x''\ y' \text{ of} \qquad : \llbracket\mathbb{N}\rrbracket \\ \qquad\qquad\qquad \{\ \mathit{yes} \Rightarrow \texttt{cons } x'\ \texttt{(f (cons (succ } y'\texttt{) } s''\texttt{))} \\ \qquad\qquad\qquad \mid \mathit{no} \Rightarrow \texttt{f (pred } x'\texttt{) (cons (succ } y'\texttt{) } s''\texttt{)}\}\end{array}}\,E^{+} \tag{$\mathcal{Y}$}$$

$$\dfrac{\dfrac{\dfrac{\cdot\,;\cdot\,;\{x:\mathbb{N}\}\vdash x : \mathbb{N}}{\cdot\,;\cdot\,;\{x:\mathbb{N}\}\vdash \mathrm{out}_{\mu}(x,\mathbb{N}) : \mathbb{1}+\mathbb{N}}\,E^{\mu} \qquad \mathcal{X} \qquad \mathcal{Y}}{\begin{array}{l}\{\ddagger,\dagger\}\,;\cdot\,;\{x:\mathbb{N},x':\mathbb{N},y':\mathbb{N}\}\vdash \text{case out}(x',\mathbb{N}) \text{ of} \qquad : \llbracket\mathbb{N}\rrbracket \\ \qquad\qquad \{\ z \Rightarrow \texttt{cons zero (f (cons (succ } y'\texttt{) } s''\texttt{))} \\ \qquad\qquad \mid x'' \Rightarrow \cdots\}\end{array}}\,E^{+}}{\{\dagger\}\,;\cdot\,;\{x:\mathbb{N},y:\mathbb{N},x':\mathbb{N},y':\mathbb{N}\}\vdash \texttt{f (pred } x\texttt{) (cons (succ } y'\texttt{) } s''\texttt{)} : \llbracket\mathbb{N}\rrbracket\ \texttt{\textbf{label}}\ \ddagger}\,I^{\delta} \tag{$\mathcal{Z}$}$$

Figure 5.13: Proof for f, part II

Most techniques of program transformation are simpler in the context of functional or logic programming since program transformation in the presence of side effects is much more complex.

For program transformation in a setting which includes side-effects it is possible to either carefully deal with side effects, or first translate into a declarative intermediate language. The former approach is often used in partial evaluators and other compiler optimisations. An example of the later approach is described in [60].

A survey of partial evaluation techniques is given in [20]. In terms of the unfold/fold framework it can be thought of most simply as a special case which makes use only of unfolding and instantiation.

Turchin described supercompilation in the early 70s. An overview of supercompilation is provided in [87]. As we have seen in this chapter supercompilation is an automated technique of program tranformation that generalises partial evaluation by making use of folding from the unfold/fold framework and the introduction of generalisations.

One of the most clear expositions of supercompilation was given for *positive supercompilation* by Sørensen, Glück and Jones[78]. This restricted the algorithm to the propagation of positive information while not propagating negative information which results when some predicate fails to be satisfied in the course of computation.

A Coq mechanised verification of a supercompiler for a very simple language has been presented by Krustev in [45]. The language is much simpler than the term language used here, but

```
Ω(true)    :=  left((), 𝟙 + 𝟙)
Ω(false)   :=  right((), 𝟙 + 𝟙)
Ω(zero)    :=  $\text{in}_\mu(\text{left}((), \mathbb{1} + \mathbb{N}), \mathbb{N})$
Ω(succ)    :=  $\lambda x : \mathbb{N}.\ \text{in}_\nu(\text{right}(x, \mathbb{1} + \mathbb{N}), \mathbb{N})$
Ω(cons)    :=  $\lambda x : \mathbb{N}\ s : [\![\mathbb{N}]\!].\text{in}_\nu((x, s), [\![\mathbb{N}]\!])$
Ω(le)      :=  $\lambda x\ y : \mathbb{N}.$
               case $(\text{out}_\mu(x, \mathbb{N}))$ of
                 { $z$ ⇒ true
                 | $x'$ ⇒
                   case $(\text{out}_\mu(y, \mathbb{N}))$ of
                     { $z$ ⇒ false
                     | $y'$ ⇒ le $x'$ $y'$}}
Ω(f)       :=  $\lambda s : [\![\mathbb{N}]\!].$
               split $(\text{out}_\nu(s, \mathbb{N}))$ as $(x, s')$ in
                 split $(\text{out}(s', \mathbb{N}))$ as $(y, s'')$ in
                   case $x$ of
                     { $z$ ⇒ cons zero (f (cons $y$ $s''$))
                     | $x'$ ⇒
                       case $y$ of
                         { $z$ ⇒ g $x'$ $s''$(cons $y$ $s''$))
                         | $y'$ ⇒
                           case (le $x$ $y$) of
                             { $yes$ ⇒ cons (succ $x$) (f ((cons $x$) $s''$))
                             | $no$ ⇒ h $x'$ $y'$ $s''$}}}
Ω(g)       :=  $\lambda x : \mathbb{N}\ s' : [\![\mathbb{N}]\!].$
               case $(\text{out}_\mu(x, \mathbb{N}))$; of
                 { $z$ ⇒ cons zero (f (cons (succ $y$) $s$))
                 | $x'$ ⇒ g $x'$ $s$}
Ω(h)       :=  $\lambda x\ y : \mathbb{N}\ s : [\![\mathbb{N}]\!].$
               case $(\text{out}_\mu(x, \mathbb{N}))$ of
                 { $z$ ⇒ cons zero ($f$ (cons (succ $y$) $s$))
                 | $x'$ ⇒
                 case (le $x'$ $y$) of
                   { $yes$ ⇒ cons $x$ (f (cons (succ $y$) $s$)
                   | $no$ ⇒ h $x'$ $y$ $s$}}
```

Figure 5.14: Supercompiled Stream Program

the mechanisation is also more complete.

The approach of supercompilation for the production of equivalent cyclic proofs is very close to Cockett's work on deforestation [18]. We extend the approach to supercompilation, and additionally manipulate terms which may or may not be total in contrast to the term language given by Cockett.

The *distillation algorithm* is a program transformation algorithm which was described by Hamilton[36] and which is capable of more sophisticated transformations than supercompilation. It may be possible to use such more advanced program transformation techniques to expand further the programs which we can show to be correct.

The present work uses a variant supercompilation algorithm which is non-deterministic. Non-deterministic variants of supercompilation have been described before by Klyuchnikov and Romanenko[43]. The particular variant which we describe differs in that it is used to lazily provide a stream of supercompiled programs to a totality checker. This allows us to short-circuit when proofs fail to demonstrate totality by violating the syntactic restrictions.

Chapter 6

Soundness

6.1 Introduction

The use of bisimulation in a functional programming setting makes use of the fact that transition edges can be associated with terms to create a transition system. This technique suggests that we might also look at soundness as the satisfaction of a relation contingent on the structure of the type. Such an approach of showing that a formula is satisfied by a transition system is known as *model checking*. We construct a soundness proof in analogy with techniques from *model checking*.

The technique that we employ is quite close to infinite state local model checking using the concept of an analytic tableau as in Bradfield and Stirling [12]. Instead of describing a tableau system, we use a coinductive relation in the style of Milner and Tofte [56]. We however must carefully require that the relation consumes only edges from the transition system, ensuring that our concept of soundness corresponds with program behaviour and non-termination is excluded.

6.2 (Co)-Inductive Constructive Types

The supercompilation algorithm given in Section 5.3 results in pre-proofs rather than proofs. In order to obtain proofs, we need to use the syntactic restrictions given in Section 4.6.

In demonstrating bisimulation in supercompilation, the question of termination is implicit since reductions require input that will always behave identically *if the input converges*. The convergence need only be to a *value*, that is, a term with some observable behaviour. If reductions lead us to divergence, we need only worry that it does so in both cases in order to show they are bisimilar. For the inhabitation of types we need to be more careful.

To give an example we can look at the question of termination of the program which implements an evenness predicate given in Figure 6.1.

$$
\begin{array}{lll}
\Omega(\texttt{even}) & := & \lambda x : \mathbb{N}. \\
& & \text{case } x \text{ of} \\
& & \quad \{ \, z \Rightarrow \texttt{true} \\
& & \quad \; \mid x' \Rightarrow \\
& & \qquad \text{case } x' \text{ of} \\
& & \qquad \{ \, z \Rightarrow \texttt{false} \\
& & \qquad \; \mid x'' \Rightarrow \text{case } \texttt{even } x'' \text{ of } \{t \Rightarrow \texttt{false} \mid f \Rightarrow \texttt{true}\}\}\} \\
\\
\Xi(\texttt{even}) & := & \mathbb{N} \rightarrow \mathbb{B}
\end{array}
$$

Figure 6.1: Even

This program makes use of the function constant `even` within an experiment. The destructuring of `even` x'' requires a chain of Delta unfoldings, one of which is necessarily as long as the x'' variable persists in producing an inr transition.

We have however, by assumption, taken x to be an inductive variable, and this limits the process of eliminations to a finite one. The fact that it is finite is dictated by the meaning of the variable belonging to the type. Namely that there is some finite sequence of transitions and that it successfully satisfies the least-fixed point meaning of its associated formula.

Our problem then is to show that totality implies that given suitable total input, we always provide the transitions required of the type when interpreted as a formula.

The use of transition systems to describe the behaviour of terms is suggestive that we might be able to use techniques from model checking in order to demonstrate type correctness of proofs. We can do this by producing a relation corresponding to the type of interest which, if satisfied will demonstrate the type soundness of a term. The theorem which we need to prove is of the following form.

Theorem 6.2.1 (Soundness)**.** $\cdot \Vdash t : A \rightarrow \vDash t : A$

The relation on the left, is a cyclic proof, a pre-proof meeting the necessary syntactic criteria. The relation on the right is one which demonstrates soundness by way of a coinductive relation. It essentially acts to test that the correct (as dictated by the type) transitions are available to be accepted. The relation is itself coinductive as is the soundness proof. We show the relation in Figure 6.2.

The relation demonstrates soundness by requiring that our transition system provide us with an edge suitable for the type of the term. In this way it mirrors model checking since we ask for

$$
\begin{array}{ll}
\forall a. \vDash a : 1 & \mathit{SUnit} \\
\forall aAB.(\forall ct. \cdot \Vdash c : A \to a \overset{c:A}{\longmapsto} t \to \vDash t : B) \to \vDash a : A \to B & \mathit{SImp} \\
\forall aA.(\forall Ct. a \overset{C}{\longmapsto} t \to \vDash t : A[X := C]) \to \vDash a : \forall X.A & \mathit{SAll} \\
(\forall atsAB. a \overset{\mathit{fst}}{\longmapsto} t \wedge \vDash t : A) \wedge & \\
\quad (\forall s. a \overset{\mathit{snd}}{\longmapsto} s \wedge \vDash s : B) \to \vDash a : A \wedge B & \mathit{SAnd} \\
\forall atAB. a \overset{\mathit{left}}{\longmapsto} t \wedge \vDash t : A \to \vDash a : A + B & \mathit{SOrL} \\
\forall atAB. a \overset{\mathit{right}}{\longmapsto} t \wedge \vDash t : B \to \vDash a : A + B & \mathit{SOrR} \\
\forall atA. a \overset{\mathit{in}}{\longmapsto} t \wedge \vDash t : A[X := \mu X.A] \to \vDash a : \mu X.A & \mathit{SFoldMu} \\
\forall atA. a \overset{\mathit{in}}{\longmapsto} t \wedge \vDash t : A[X := \nu X.A] \to \vDash a : \nu X.A & \mathit{SFoldNu} \\
\forall aA.(\exists tB.(t, B, a, A) \in R^{+} \to \vDash t : B) \to \vDash a : A & \mathit{SRecNu} \\
\forall aA.(\exists tB.(t, B, a, A) \in R \to \vDash t : B) \to \vDash a : A & \mathit{SRecMu}
\end{array}
$$

Figure 6.2: Soundness Relation

the formula of interest to be satisfied by appropriate behaviours. It does not admit terms which fail to provide edges, and so $\perp$ is excluded by design.

We note that we can give any term as having a type of $\mathbb{1}$ and this does not present a problem for us as the static semantics do not allow us to form terms which might look for behaviour from terms of type $\mathbb{1}$. Since there are no experiments for programs of unit type, we will never seek behaviour from them and our transition system will have no edge.

The relations R and R^{+} are free in the statement of the soundness relation and must be known beforehand in order to demonstrate soundness in this manner. The relations must additionally be monotonic and well-founded for R and anti-well-founded for R^{+}. Because of the restrictions on cyclic proofs, we already have these relations in hand. They are the relations induced by the guardedness and structural recursion. Note, that $<^{s}$ by itself would not be sufficient, the requirement of monotonicity requires that the relation be a relation which is monotonic globally. Without this restriction it would be possible to have recursive terms which alternate increasing and decreasing arguments leading to divergence.

However there is one feature to which notice should be drawn. The *SImp* rule makes use of the relation $\cdot \Vdash c : A$ for the parameter c of type A. It might seem more natural to make this the relation $\vDash c : A$, however this would lead to the relation failing to respect the positivity condition. Instead, we will find that this definition will be sufficient provided that we demonstrate substitution preservation for the relation $\cdot \Vdash c : A$. The lemma required is as follows:

Lemma 6.2.2 (Type Preservation)**.** $\cdot \Vdash c : A \wedge x : A \Vdash t : B \to \cdot \Vdash t[x := c] : A$

This type preservation requires only that we can re-establish the syntactic conditions for guardedness and structural recursion.

Type Preservation. The proof proceeds by induction on the term and inversion on the derivation. □

We will also need a decomposition Lemma which allows us to reduce each term to a value, or a function symbol blocking evaluation. This makes use of the reduction relation without Ω.

Lemma 6.2.3 (Decomposition)**.** $\forall t. \cdot \Vdash t : A \to t \Downarrow v \vee \exists f. t \Downarrow C[f]$ *where* $C[f]$ *is composed strictly of atomic experiments.*

Proof. The proof of Decomposition is straightforward since System-F$^+$ is strongly normalising. The only possibilities for the innermost redex are either a value, which is not reducible, a variable, which is eliminated by inversion since the context is empty or a function symbol. □

We now return to the proof that our cyclic proofs do in fact demonstrate soundness, that is $\cdot \Vdash t : A \to \vDash t : A$.

Soundness. The proof proceeds coinductively, with inversion on the typing relation. We will use the syntactic criteria and the restriction on the form of types to positive types to ensure that we can always produce the necessary edges.

For all applications of induction on the transition edge (after finitely many evaluation steps we must obtain some labeled transition since the relation is finitely formed), we have two cases that will arise based on the inductive structure of the transition relation: a concrete syntactic value or an evaluation and an edge.

Type preservation for induction on the transition edge, when the edge is given by the evaluation case, is proved by use of the type preservation of the transitive closure of the evaluation relation.

We have the following cases for application of the decomposition lemma on $\cdot \Vdash s : B$.

- $\cdot \Vdash \lambda x : A.t : A \to C$: From this we can apply the SImp rule. By induction on the transition edge $s \overset{x:c}{\longmapsto} s'$ we obtain for s' a substitution $t[x := c]$. By substitution preservation we obtain $\cdot \Vdash t[x := c] : C$. We can now apply the coinductive hypothesis. The necessary type preservation for the induction on the transition edge for the case that the edge is given by a reduction which leads to an edge arises from the type preservation of the evaluation relation.

- $\cdot \Vdash \Lambda X.\ t : \forall X.\ A$: Here we apply the SAll rule. By induction on the transition edge $s \overset{C}{\longmapsto} s'$ we obtain for s' a substitution $t[X := C]$. This gives us the $\cdot \Vdash t[X := C] : A[X := C]$ using the type substitution lemma. We apply the coinductive hypothesis.

- $\cdot \Vdash v : B$: eliminated as there is no context from which to form a derivation.
- $\cdot \Vdash (r, s) : A \times C$: Here we apply the SAnd rule, followed with the relation $(r, s) \xmapsto{fst/snd} r/s$ respectively and finally the coinductive hypothesis.
- $\cdot \Vdash \text{left}(r, A + C) : A + C$: Here we apply the SOrL rule, followed with the $s \xmapsto{left} r$ transition rule and finally the coinductive hypothesis.
- $\cdot \Vdash \text{right}(r, A + C) : A + C$: Here we apply the SOrR rule, followed with the $s \xmapsto{right} r$ transition rule and finally the coinductive hypothesis.
- $\cdot \Vdash \text{in}_\mu(x, \mu X.A) : \mu X.A$: We apply the SFoldMu rule, followed with the $s \xmapsto{in} x$ transition rule and finally the coinductive hypothesis.
- $\cdot \Vdash \text{in}_\nu(x, \mu X.A) : \nu X.A$: We apply the SFoldNu rule, followed with the $s \xmapsto{in} x$ transition rule and finally the coinductive hypothesis.
- $C[f^n]$: Here we proceed on the sub argument using well founded induction. We have two cases:
  - $f^n : \forall \overrightarrow{X}.\overrightarrow{A} \to \mu\hat{X}.\ .C \to \overrightarrow{D}$: In this case we unfold to reach a term $C[\Omega(f^n)] \Downarrow s$. We reapply the decomposition lemma leading to one of the above cases or we obtain $C'[f^m]$ for some m. We use the product order of $<^s$, one for each constant m below r the bound on the number of function symbols. We proceed using the inductive hypothesis. This means that eventually we must have that $s \Downarrow v$ for $v \in \mathcal{V}$ or we have $C''[f^o]$ for the coinductive case.
  - $f^n : \forall \overrightarrow{X}.\overrightarrow{A} \to \nu X.C$: In this case we reach a term $C[\Omega(f^n)] \Downarrow s$. We reapply the decomposition lemma leading to one of the above value cases or we obtain $C'[f^m]$ for some m. If it is the case that f^m is a function with an inductive parameter we apply the inductive hypothesis. We can only encounter it a finite number of times hence reduction to a $C[v]$ is inevitable. We can repeat this argument for every unfolding of f^i which is by necessity bounded by some constant r since our program is finite. Because of our guardedness condition which ensures each unfolding of a coinductive argument introduces only a finite number of contexts together with the above well founded relation $<^s$ and the finite size of terms, we will eventually reach a value that will reduce a finite number of times one for each context (by the progress lemma) until

we reach a constructor. We now have $C[\Omega(f^n)] \rightsquigarrow^* v$ for a $v \in Value$ and can use one of the above value cases to construct the soundness relation.

□

6.3 Examples

$$\begin{array}{lcl}
\Omega(\texttt{zero}) & := & \text{in}_\mu(\text{left}(,1+\mathbb{N}),\mathbb{N}) \\
\Omega(\texttt{succ}) & := & \lambda x : \mathbb{N}.\text{in}_\mu(\text{right}(x,1+\mathbb{N}),\mathbb{N}) \\
\Omega(\texttt{plus}) & := & \lambda x\ y : \mathbb{N}. \\
 & & \text{case } (\text{out}_\mu(x,\mathbb{N})) \text{ of} \\
 & & \quad \{\, z \;\Rightarrow\; y \\
 & & \quad \mid x' \;\Rightarrow \\
 & & \quad\quad \texttt{succ}\ (\texttt{plus}\ x'\ y)\} \\
\Xi(\texttt{zero}) & := & \mathbb{N} \\
\Xi(\texttt{succ}) & := & \mathbb{N} \to \mathbb{N} \\
\Xi(\texttt{plus}) & := & \mathbb{N} \to \mathbb{N} \to \mathbb{N}
\end{array}$$

Figure 6.3: Program for `plus`

We can see how this soundness relation can be constructed in the particular case by looking at a few examples. We start with Example 6.3.1 which demonstrates satisfaction of the soundness relation using inductive types with the program from Figure 6.3.

Example 6.3.1. Since we have that $\Vdash \texttt{plus} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ we can write the transitions: $\texttt{plus} \xmapsto{c:\mathbb{N}} \cdot \xmapsto{d:\mathbb{N}} \text{case out}(c,\mathbb{N}) \text{ of } \{z \Rightarrow d \mid x' \Rightarrow \texttt{succ}\ (\texttt{plus}\ x'\ y)\}$

We can make use of the constructors for *SImp* twice. Now we perform inversion on the proof of c to obtain two cases.

- $c = \text{in}(\text{left}(x', 1+\mathbb{N}),\mathbb{N})$:

  $\text{case out}(c,\mathbb{N}) \text{ of } \{z \Rightarrow d \mid x' \Rightarrow \texttt{succ}\ (\texttt{plus}\ x'\ y)\} \rightsquigarrow d$

  Since $\Vdash d : \mathbb{N}$ holds by assumption we are done with this case.

- $c = \text{in}(\text{right}(x', 1+\mathbb{N}),\mathbb{N})$:

  $\text{case } c \text{ of } \{z \Rightarrow d \mid x' \Rightarrow \texttt{succ}\ (\texttt{plus}\ x'\ y)\} \rightsquigarrow \texttt{succ}\ (\texttt{plus}\ x'\ y)$

  Now we have:

  $\texttt{succ}\ (\texttt{plus}\ x'\ y) \xmapsto{in} \cdot \xmapsto{right} \texttt{plus}\ x'\ y$

  We can apply *SFold* and then *SOrR* and finally we have:

plus x' y

With this we can apply the $SRecMu$ rule using the fact that $x' <^s c$, and then apply the coinductive hypothesis.

$$\begin{array}{lcl}
\Omega(\texttt{zero}) & := & \text{in}_\mu(\text{left}(,1+\mathbb{N}),\mathbb{N}) \\
\Omega(\texttt{succ}) & := & \lambda x:\mathbb{N}.\text{in}_\mu(\text{right}(x,1+\mathbb{N}),\mathbb{N}) \\
\Omega(\texttt{alt}) & := & \lambda x\ y:\mathbb{N}. \\
 & & \text{case } (\text{out}_\mu(x,\mathbb{N})) \text{ of} \\
 & & \quad \{\ z \ \Rightarrow \\
 & & \qquad \text{case } (\text{out}_\mu(y,\mathbb{N})) \text{ of} \\
 & & \qquad\quad \{\ z \ \Rightarrow \ \texttt{zero} \\
 & & \qquad\quad \mid y' \ \Rightarrow \ \texttt{alt}\ (\texttt{succ}\ x)\ y'\} \\
 & & \quad \mid x' \ \Rightarrow \ \texttt{alt}\ x'\ (\texttt{succ}\ y)\} \\
\Xi(\texttt{zero}) & := & \mathbb{N} \\
\Xi(\texttt{succ}) & := & \mathbb{N}\to\mathbb{N} \\
\Xi(\texttt{alt}) & := & \mathbb{N}\to\mathbb{N}\to\mathbb{N}
\end{array}$$

Figure 6.4: Program for alt

We can turn to another example program given in Figure 6.4 which demonstrates why we need to give the restrictions on the relation R.

Example 6.3.2. Since we have that $\Vdash \texttt{alt} : \mathbb{N}\to\mathbb{N}\to\mathbb{N}$ we can write the transitions:

$\texttt{alt} \xmapsto{c:\mathbb{N}} \cdot \xmapsto{d:\mathbb{N}} t$

with $t =$ case out$_\mu(c,\mathbb{N})$ of .

$\{\ x' \Rightarrow$ case out$_\mu(d,\mathbb{N})$ of

$\{\ z \Rightarrow \texttt{zero}$

$\mid y' \Rightarrow \texttt{alt}\ (\texttt{succ}\ c)\ y'\}$

$\mid x' \Rightarrow \texttt{alt}\ x'\ d\}$

We can make use of the constructors for *SImp* twice. Now we perform inversion on the proof of c to obtain two cases.

- $c = \text{in}_\mu(\text{left}(x', 1+\mathbb{N}),\mathbb{N})$:

  We have that $t \rightsquigarrow$ case out$_\mu(d,\mathbb{N})$ of $\{z \Rightarrow zero \mid y' \Rightarrow \texttt{alt}\ (\texttt{succ}\ c)\ y'\}$.

  We perform inversion on the proof of d to obtain two further cases.

  – $d = \text{in}_\mu(\text{left}(y', 1+\mathbb{N}),\mathbb{N})$:

    In this case $t \rightsquigarrow^* zero$. Since $zero \xmapsto{in} \cdot \xmapsto{left} U$ we can apply *SFoldMu*, *SOrL* and $SUnit$ and we are done.

  - $d = \mathrm{in}_\mu(\mathrm{right}(y', 1+\mathbb{N}), \mathbb{N})$:

    Here we have $t \rightsquigarrow^* \texttt{alt}\ (\texttt{succ}\ c)\ y'$. We can attempted to use the *SRecMu* rule plus the coinductive hypothesis given that $y' <^s d$.

- $c = \mathrm{in}_\mu(\mathrm{right}(x', 1+\mathbb{N}), \mathbb{N})$:

  We have here that $t \rightsquigarrow^* \texttt{alt}\ x'\ (\texttt{succ}\ d)$. We can now use the *SRecMu* rule plus the coinductive hypothesis given that $x' <^s c$.

However, now the relation R which we have built is of the form

$$(\forall cd.\texttt{alt}\ c\ (\texttt{succ}\ d) < \texttt{alt}\ (\texttt{succ}\ c)\ d) \wedge (\forall cd.\texttt{alt}\ (\texttt{succ}\ c)\ d < \texttt{alt}\ c\ (\texttt{succ}\ d)).$$

This relation can be proved not to be anti-symmetric, so it is not possible to have supplied it to the soundness relation.

This shows the importance of the indexing of the soundness relation according to the particular relations that we have constructed from our cyclic proofs. The program term here is not sound and an attempt to run this program in Haskell or a similar programming language will demonstrate that it does not terminate unless both arguments are `zero`.

In order to see how our example works on a coinductively defined program we can deal again with `plus` except this time using the program in Figure 4.13 where we deal with $\overline{\mathbb{N}}$ rather than $\mathbb{N}$.

Example 6.3.3. Since we have that $\Vdash \texttt{plus} : \overline{\mathbb{N}} \to \overline{\mathbb{N}} \to \overline{\mathbb{N}}$ we can write the transitions:

$\mathit{plus} \xmapsto{c:\overline{\mathbb{N}}} \cdot \xmapsto{d:\overline{\mathbb{N}}} \text{case out}_\nu(c, \overline{\mathbb{N}}) \text{ of } \{z \Rightarrow d \mid x' \Rightarrow \texttt{succ}\ (\texttt{plus}\ x'\ y)\}$

We can make use of the constructors for *SImp* twice. Now we perform inversion on the proof of c to obtain two cases.

- $c = \mathrm{in}_\nu(\mathrm{left}(x', 1+\overline{\mathbb{N}}), \overline{\mathbb{N}})$:

  Since $\Vdash d : \overline{\mathbb{N}}$ holds by assumption we are done with this case.

- $c = \mathrm{in}_\nu(\mathrm{right}(x', 1+\overline{\mathbb{N}}), \overline{\mathbb{N}})$:

  Here we have:

  $\text{case out}_\nu(c, \overline{\mathbb{N}}) \text{ of } \{z \Rightarrow d \mid x' \Rightarrow \texttt{succ}\ (\texttt{plus}\ x'\ y)\} \rightsquigarrow^* \texttt{succ}\ (\texttt{plus}\ x'\ y)$.

  From here we have $\texttt{succ}\ (\texttt{plus}\ x'\ y) \xmapsto{in} \cdot \xmapsto{inr} \texttt{plus}\ x'\ y$ and so we can apply the *SFoldNu*, *SOrR* rules followed by *SRecNu* with the fact that for the relation R^+ we have $\texttt{succ}\ (\texttt{plus}\ x'\ y) > \texttt{plus}\ (\texttt{succ}\ x')\ y$.

6.4 Related Work

Model checking is a very well studied field with a large amount of literature. There are numerous modal logics of interest including LTL, CTL, CTL*[91] and the modal μ-calculus [10] among others. Moller describes the various relationships between some of these calculi in [58].

Milner and Tofte give a co-inductive relational description of type inhabitation in [56]. They use a method of representing recursive functions that is quite close to the method used here excepting that their map between function constants includes closures over an environment. The type system in their presentation however is quite simple, being essentially quite close to the simply typed λ-calculus. Additionally they *only* include a description for what we refer to as co-inductive types. Namely, they use the greatest fixed-point to define inhabitation.

Stirling and Bradfield [12] give a tableau method for showing that a given model meets a formula. In contrast, this work applies the same techniques to a transition system model generated from a term. The novel contribution of the present work is to synthesise the approach taken by Milner and Tofte and the already well developed approach of Stirling and Bradfield in order to obtain a method of showing type inhabitation for functional programs.

There are a number of ways that model-checking can be related to proof and proof-search. A unification of model checking and proof-search is described in [54][84][85]. A type system equivalent to model-checking is described in [61] which helps to shed light on the connections and differences between model-checking and type-theory. Our work differs in that it consists in the reverse problem of starting with a term calculus and looking for suitable fragments of temporal formulae which can be used to show type correctness.

There has been work on the use of program transformation techniques as applied to model-checking, such as [90] and [47]. However, these techniques do not make use of program transformation as a means to find terms which meet a syntactic restriction for soundness in a type theory.

Chapter 7

Implementation

The implementation of the work presented here is divided into two main pieces. The first is a mechanised implementation of System-F with an extension including general recursion which allows the production of cyclic proofs (using coinductively defined formation rules) in the Coq proof assistant.

The second piece is an implementation of a supercompiler for the enriched System-F^+ combined with a totality checker implemented in the Haskell programming language.

7.1 Mechanisation in Coq

The mechanisation of the theory in Coq is done with a modified De Bruijin indexing style for variables. We take the natural numbers as the index for type-variables, variables and function constants.

The data type which describes type formation is given in Figure 7.1. It consists of type-variables *TV*, implication *Imp*, universal quantification *All*, pairs for conjunction given by *And*, disjunction by injection into a sum type given by *Or*, greatest fixed-points given by *Nu*, least fixed-points given by *Mu* and a type with one constructor, *One*.

The data type describing terms is given in Figure 7.2. It consists of function constants *F*, variables *V*, application *App*, type-application *TApp*, term abstraction *Abs* and type abstraction *Lam*, pair introduction *Pair*, two constructors for injection into sum types, *Inl* and *Inr*, *Unit* which has no elimination rule, *Fold* which is used to explicitly introduce *Nu* or *Mu* types, *Case* for elimination of sums, *Split* for elimination of pairs and *Unfold* for elimination of *Folds*.

```
Inductive Ty : Set :=
| TV : nat → Ty
| One : Ty
| Imp : Ty → Ty → Ty
| All : Ty → Ty
| And : Ty → Ty → Ty
| Or : Ty → Ty → Ty
| Nu : Ty → Ty
| Mu : Ty → Ty.
```

Figure 7.1: Types

```
Inductive Term : Set :=
| F : nat → Term
| V : nat → Term
| App : Term → Term → Term
| TApp : Term → Ty → Term
| Abs : Ty → Term → Term
| Lam : Term → Term
| Fold : Term → Ty → Term
| Unfold : Term → Ty → Term
| Inl : Term → Ty → Term
| Inr : Term → Ty → Term
| Case : Term → Term → Term → Term
| Pair : Term → Term → Term
| Split : Term → Term → Term
| Unit : Term.
Lemma term_eq_dec : ∀ (t1 t2 : Term), {t1 = t2} + {t1 ≠ t2}.
```

Figure 7.2: Terms

7.1.1 Derivations

To this term and type syntax we add a data-type representing both type and term variable contexts. These will hold all of the free variables in an open term and will be further constrained by our formation rules. We add a notation:

```
[n ; l |= t @ ty]
```

```
Inductive Ctx : Set :=
| ctx : nat → list Ty → Ctx.
Inductive Holds : Set :=
| H : Ctx → Term → Ty → Holds.
Notation "[ n ; l |= t @ ty ]" := (H (ctx n l) t ty) (at level 0).
```

Figure 7.3: Contexts

which gives the number of free type variables, n, a list of the types of the free variables, l, the term, t, and the type of that term, ty. The data-type is given in Figure 7.3. The directive *at level 0* simply refers to the precedent table maintained by Coq to determine how it should interpret new notations.

The de Bruijn notation allows us to describe variables simply by the use of natural numbers and abstraction without explicit reference to the variable being captured. This is done by interpreting the variable as a count of the number of λ-binders that must be traversed in order to reach the binding lambda, or in the case of free-variables, as the index past the last λ-binder into the free variable context.

	Named	**de Bruijn**
1.	$A, B; x\!:\!A \to B \vdash (\lambda y\!:\!A.\ x\ y)\ :\ A \to B$	$2\ ;\ [\mathbf{0} \to \mathbf{1}] \vdash (\lambda\mathbf{0}.\ 1\ 0)\ :\ \mathbf{0} \to \mathbf{1}$
2.	$A, B; \cdot \vdash (\lambda y\!:\!A.\ (\lambda x\!:\!A \to B.\ x\ y))$ $: A \to (A \to B) \to B$	$2\ ; \cdot \vdash (\lambda\mathbf{0}.\ (\lambda\mathbf{0} \to \mathbf{1}.\ 0\ 1))$ $:\ \mathbf{0} \to (\mathbf{0} \to \mathbf{1}) \to \mathbf{1}$
3.	$\cdot; \cdot \vdash (\Lambda A.(\lambda x : A.x)) : \forall A.A \to A$	$0\ ; \cdot \vdash (\Lambda.(\lambda\mathbf{0}.0)) : \forall\mathbf{0} \to \mathbf{0}$
4.	$C; \cdot \vdash (\lambda x : C.x) : C \to C$	$1; \cdot \vdash (\lambda\mathbf{0}.0) : \mathbf{0} \to \mathbf{0}$

Figure 7.4: Named versus de Bruijn

To get an idea of how this notation works we demonstrate in Figure 7.4 the named and nameless representation of some terms and their sequents, assuming two types themselves containing no free variables, A and B.

In the first entry in the table, the variable binder for y is replaced with a 0 which represents the fact that it is bound to the first λ form above it. The type is given for that variable, but no variable name needs to be used. The variable x which is a free variable is represented by 1. We subtract one for each λ form we encounter, in this case, 1, and then use the resulting natural number as an index to our variable context, in this case 0. Since there are no free type variables, the free-type variable count is zero.

In the second case we see similar accounting with two bound variables.

In the third entry we see the use of type-variable accounting. We represent type variable indexes in bold to help distinguish them from term variables. Finally in entry 4 we see the use of free variables in the type context.

The use of a natural number to represent free variables in the type-context can be seen in analogy with the term variables. Since in our nameless representation, term variables are simply indices into our variable context from which we can recover their type, the types, which are not themselves typed in System-F can be represented as nothing more than a bound on the size of the context with the variable index demonstrating that the type is constrained to be below that maximum.

The use of nameless representation requires functions which manipulate indices. For the description of our formation rules, we will require the function *tyshift* and *tyshiftn*, given in Figure 7.5 which manipulates type variables by *shifting* them by increasing integers above a certain threshold. This ensures that we can bring terms under binders during substitution while keeping their references correct.

The function *tyshiftn* takes a number by which to increase the variable, and a cut-off which represents which variables are currently free. Anything under the cut-off will not be shifted.

```
Fixpoint tyshiftn (n : nat) (d : nat) (ty : Ty) {struct ty} : Ty :=
  match ty with
    | TV m ⇒ if le_lt_dec d m then TV (n+m) else TV m
    | Imp t s ⇒ Imp (tyshiftn n d t) (tyshiftn n d s)
    | All t ⇒ All (tyshiftn n (S d) t)
    | And t s ⇒ And (tyshiftn n d t) (tyshiftn n d s)
    | Or t s ⇒ Or (tyshiftn n d t) (tyshiftn n d s)
    | Mu t ⇒ Mu (tyshiftn n (S d) t)
    | Nu t ⇒ Nu (tyshiftn n (S d) t)
    | One ⇒ One
  end.
```

Figure 7.5: Type Shifting

With type-shifting in hand we can turn to type substitution. This implements the substitution described in Figure 7.6 in Section 2.4. The substitution function here shows how we have to raise the indexes of types which we take under an All constructor as the index to the free variables increases by one.

Additionally when we encounter a variable, we need to test for three cases. The first case is whether the variable is the variable to be substituted. If so we simply return the substituting term, which has now been appropriately shifted by the recursion. If we encounter a bound variable, we simply return it. If we encounter a variable which is free, we need to shift it down by one, since one lambda binder has been removed. Coq conveniently allows us to prove that we can always find a predecessor and this means we need not worry about using subtraction, something which can often complicate the implementation of de Bruijn indices. This proof is achieved by using $False\_rec$, which essentially means that we can eliminate cases with hypotheses that lead to contradiction, which makes use of the principle of *ex falso quodlibet*.

We introduce a concept of a valid type in Figure 7.7. A *valid type* is a type which has free variables which are never larger than some cut-off. This will be needed for our formation rules, to ensure that we do not use types which refer to variables which are not in context.

There are also some necessary invariants on the form of programs which must be maintained. The first is that there is a total function *Xi* or Ξ which associates types with all function constants. The second is that none of these types has free variables. Ξ is needed because System-F type inference is undecidable. It is relatively easy to provide Ξ as a total function when writing functional programs by giving a type of 0 to anything which represents a function constant not in our program.

With these pieces in hand we can describe the basic structure of our type derivations which are given as a data-type in Figure 7.8. We notice here that we have a co-inductive description of our derivations. This means that cyclicity is allowable in the structure of the derivation. Despite that fact, we will find that it is relatively straightforward to produce the standard progress and preservation laws used for functional programs.

The Derivation data-type uses the familiar introduction and elimination rules altered slightly to use the nameless representation. *ImpIntro* for example simply allows the use of a type from the variable context to introduce an abstraction. The *AllIntro* step is perhaps more interesting. Here we require that we step the number of free-type variables down by one, hence a requirement that our antecedent is $1 + n$ or *S n* for some n and that every type variable is one smaller in terms of

```
Definition tysub : ∀ (ty : Ty) (n : nat) (s : Ty), Ty.
Proof.
  refine
    (fix tysub (ty : Ty) (n : nat) (s : Ty) {struct ty} : Ty :=
      match ty with
        | TV m ⇒ match le_lt_dec n m with
                    | left p ⇒ match eq_nat_dec n m with
                                 | left _ ⇒ s
                                 | right p' ⇒
                                   (match m as m' return (m = m' → Ty)
                                     with
                                     | 0 ⇒ (fun p'' ⇒ False_rec _ _)
                                     | S m' ⇒ (fun _ ⇒ TV m')
                                    end) (refl_equal m)
                               end
                    | right _ ⇒ TV m
                  end
        | Imp ty1 ty2 ⇒ Imp (tysub ty1 n s) (tysub ty2 n s)
        | All t ⇒ All (tysub t (S n) (tyshift s))
        | Mu t ⇒ Mu (tysub t (S n) (tyshift s))
        | Nu t ⇒ Nu (tysub t (S n) (tyshift s))
        | One ⇒ One
        | And ty1 ty2 ⇒ And (tysub ty1 n s) (tysub ty2 n s)
        | Or ty1 ty2 ⇒ Or (tysub ty1 n s) (tysub ty2 n s)
      end).
  destruct m. apply le_n_O_eq in p. apply p'. auto. inversion p''.
Defined.

Fixpoint tysubt (t : Term) (n : nat) (s : Ty) {struct t} : Term :=
  match t with
    | F m ⇒ F m
    | V m ⇒ V m
    | Abs ty t ⇒ Abs (tysub ty n s) (tysubt t n s)
    | Lam t ⇒ Lam (tysubt t (S n) (tyshift s))
    | App f g ⇒ App (tysubt f n s) (tysubt g n s)
    | TApp f ty ⇒ TApp (tysubt f n s) (tysub ty n s)
    | Inl t ty ⇒ Inl (tysubt t n s) (tysub ty n s)
    | Inr t ty ⇒ Inr (tysubt t n s) (tysub ty n s)
    | Case t u v ⇒ Case (tysubt t n s) (tysubt u n s) (tysubt v n s)
    | Pair t u ⇒ Pair (tysubt t n s) (tysubt u n s)
    | Split t u ⇒ Split (tysubt t n s) (tysubt u n s)
    | Fold t ty ⇒ Fold (tysubt t n s) (tysub ty n s)
    | Unfold t ty ⇒ Unfold (tysubt t n s) (tysub ty n s)
    | Unit ⇒ Unit
  end.
```

Figure 7.6: Type Substitution

```
Fixpoint valid (ty : Ty) (n : nat) {struct ty} : Prop :=
  match ty with
    | TV m ⇒
      if le_lt_dec n m
        then False
        else True
    | Imp s t ⇒ valid s n ∧ valid t n
    | Or s t ⇒ valid s n ∧ valid t n
    | And s t ⇒ valid s n ∧ valid t n
    | One ⇒ True
    | All t ⇒ valid t (S n)
    | Nu t ⇒ valid t (S n)
    | Mu t ⇒ valid t (S n)
  end.
Variable Xi : nat → Ty.
Variable ProgTy : ∀ m, valid (Xi m) 0.
```

Figure 7.7: Valid

the number of free variables, a fact ensured by the *tyshift* applied to every type in the term-context for our antecedent.

AllElim is essentially identical to the derivations given earlier, aside from the need to check that our substituted type is in our type variable context.

Var requires that we introduce a variable with a type by ensuring that the natural number used for formation is a valid index into our term-variable context, and formed with a type which is valid at our current number of free type variables.

We will also need two additional parameters which must be provided with any program that we want to use with our derivations to ensure that our theorems hold. These are given in Figure 7.9. Essentially they state that we have a *program* which combines a total function $Delta$ or Ω and associates every function constant with a term and a corresponding proof that if we assume that each function-constant is typed by Ξ then every term associated with that type has a derivation. This is in fact the well known typing law from functional programming that allows us to introduce recursive terms and leads to weak-soundness results, but also the possibility of general recursion.

The formulation we give here is straightforward to pass as a parameter with our programs in order to form our derivations. We need only type-check each term under the assumption that function constants are well typed, a process which has been mechanised in the implementation.

```
CoInductive Derivation : Holds → Set :=
| FunIntro : ∀ n m l, Derivation [n ; l |= F m @ Xi m]
| ImpIntro : ∀ n l t ty xty, valid xty n → Derivation [n ; xty::l |= t @ ty] →
    Derivation [n ; l |= (Abs xty t) @ (Imp xty ty) ]
| ImpElim : ∀ n l t f ty xty, Derivation [n ; l |= t @ xty] →
    Derivation [n ; l |= f @ (Imp xty ty) ] → Derivation [n ; l |= (App f t) @ ty]
| AllIntro : ∀ n l t ty, Derivation [S n ; map tyshift l |= t @ ty] →
    Derivation [n ; l |= (Lam t) @ All ty]
| AllElim : ∀ n l t ty xty, valid xty n → Derivation [n ; l |= t @ All ty] →
    Derivation [n ; l |= TApp t xty @ (tysub ty 0 xty) ]
| VarIntro : ∀ n l ty i, valid ty n → i < length l → nth i l One = ty →
    Derivation [n ; l |= V i @ ty]
| AndIntro : ∀ n l t s A B, Derivation [n ; l |= t @ A] →
    Derivation [n ; l |= s @ B] → Derivation [n ; l |= Pair t s @ And A B]
| AndElim : ∀ n l t s A B C, Derivation [n ; l |= t @ And A B] →
    Derivation [n ; A::B::l |= s @ C] → Derivation [n ; l |= Split t s @ C]
| OrIntroL : ∀ n l t A B, valid B n → Derivation [n ; l |= t @ A] →
    Derivation [n ; l |= Inl t B @ Or A B]
| OrIntroR : ∀ n l t A B, valid A n → Derivation [n ; l |= t @ B] →
    Derivation [n ; l |= Inr t A @ Or A B]
| OrElim : ∀ n l t u v A B C, Derivation [n ; l |= t @ Or A B] →
    Derivation [n ; A::l |= u @ C] → Derivation [n ; B::l |= v @ C] →
    Derivation [n ; l |= Case t u v @ C]
| MuIntro : ∀ n l t A, valid A (S n) →
    Derivation [n ; l |= t @ tysub A 0 (Mu A) ] →
    Derivation [n ; l |= Fold t (Mu A) @ Mu A ]
| MuElim : ∀ n l t A, valid A (S n) →
    Derivation [n ; l |= t @ Mu A] →
    Derivation [n ; l |= Unfold t (Mu A) @ tysub A 0 (Mu A)]
| NuIntro : ∀ n l t A, valid A (S n) →
    Derivation [n ; l |= t @ tysub A 0 (Nu A) ] →
    Derivation [n ; l |= Fold t (Nu A) @ Nu A ]
| NuElim : ∀ n l t A, valid A (S n) →
    Derivation [n ; l |= t @ Nu A] →
    Derivation [n ; l |= Unfold t (Nu A) @ tysub A 0 (Nu A)]
| OneIntro : ∀ n l, Derivation [n ; l |= Unit @ One].
```

Figure 7.8: Type Derivations

```
Variable Delta : nat → Term.
Variable Prog : ∀ n l m, Derivation [n ; l |= F m @ Xi m] →
    Derivation [n ; l |= Delta m @ Xi m].
```

Figure 7.9: The Program Axioms

7.1.2 Progress and Preservation

The term substitution function is very similar to the type substitution function. It also requires the use of a shifting function to help with the manipulation of indices. We also need a way to shift type indices in terms for performing substitution of types in terms which is required by the evaluation relation for System-F. These functions are given in Figure 7.11. Again we see the use of a short proof to eliminate the possibility that no predecessor can be found to allow us to shift downward variables which were above the cutoff.

```
Fixpoint shiftn (n : nat) (d : nat) (t : Term) {struct t} : Term :=
  match t with
    | F m ⇒ F m
    | V m ⇒ if le_lt_dec d m then V (n+m) else V m
    | App r s ⇒ App (shiftn n d r) (shiftn n d s)
    | Lam r ⇒ Lam (shiftn n d r)
    | Abs ty r ⇒ Abs ty (shiftn n (1+d) r)
    | TApp r ty ⇒ TApp (shiftn n d r) ty
    | Fold r ty ⇒ Fold (shiftn n d r) ty
    | Unfold r ty ⇒ Unfold (shiftn n d r) ty
    | Pair r s ⇒ Pair (shiftn n d r) (shiftn n d s)
    | Split r s ⇒ Split (shiftn n d r) (shiftn n (2+d) s)
    | Inl r ty ⇒ Inl (shiftn n d r) ty
    | Inr r ty ⇒ Inr (shiftn n d r) ty
    | Case r u v ⇒ Case (shiftn n d r) (shiftn n (1+d) u) (shiftn n (1+d) v)
    | Unit ⇒ Unit
  end.
Definition shift := shiftn 1.

Fixpoint tyshift_term (d : nat) (t : Term) {struct t} : Term :=
  match t with
    | F m ⇒ F m
    | V m ⇒ V m
    | App r s ⇒ App (tyshift_term d r) (tyshift_term d s)
    | Lam r ⇒ Lam (tyshift_term (S d) r)
    | Abs ty r ⇒ Abs (tyshiftn 1 d ty) (tyshift_term d r)
    | TApp r ty ⇒ TApp (tyshift_term d r) (tyshiftn 1 d ty)
    | Fold t ty ⇒ Fold (tyshift_term d t) (tyshiftn 1 d ty)
    | Unfold t ty ⇒ Unfold (tyshift_term d t) (tyshiftn 1 d ty)
    | Inr t ty ⇒ Inr (tyshift_term d t) (tyshiftn 1 d ty)
    | Inl t ty ⇒ Inl (tyshift_term d t) (tyshiftn 1 d ty)
    | Case t u v ⇒ Case (tyshift_term d t) (tyshift_term d u) (tyshift_term d v)
    | Pair t s ⇒ Pair (tyshift_term d t) (tyshift_term d s)
    | Split t s ⇒ Split (tyshift_term d t) (tyshift_term d s)
    | Unit ⇒ Unit
  end.
```

Figure 7.10: Shifting

```
Definition sub : ∀ (t : Term) (n : nat) (s : Term), Term.
Proof.
  refine
    (fix sub (t : Term) (n : nat) (s : Term) {struct t} : Term :=
      match t with
        | F m ⇒ F m
        | V m ⇒ match le_lt_dec n m with
                    | left p ⇒ match eq_nat_dec n m with
                                  | left p' ⇒ s
                                  | right p' ⇒
                                    (match m as m' return (m = m' → Term)
                                       with
                                       | 0 ⇒ (fun p'' ⇒ False_rec _ _)
                                       | S m' ⇒ (fun _ ⇒ V m')
                                     end) (refl_equal m)
                                end
                    | right p ⇒ V m
                  end
        | Abs ty r ⇒ Abs ty (sub r (S n) (shift 0 s))
        | Lam r ⇒ Lam (sub r n (tyshift_term 0 s))
        | App f g ⇒ App (sub f n s) (sub g n s)
        | TApp r ty ⇒ TApp (sub r n s) ty
        | Fold r ty ⇒ Fold (sub r n s) ty
        | Unfold r ty ⇒ Unfold (sub r n s) ty
        | Pair r u ⇒ Pair (sub r n s) (sub u n s)
        | Split r u ⇒ Split (sub r n s) (sub u (S (S n)) (shiftn 2 0 s))
        | Inl r ty ⇒ Inl (sub r n s) ty
        | Inr r ty ⇒ Inr (sub r n s) ty
        | Case r u v ⇒ Case (sub r n s) (sub u (S n) (shift 0 s)) (sub v (S n) (shift 0 s))
        | Unit ⇒ Unit
      end). destruct m. apply le_n_O_eq in p. apply p'. auto. inversion p''.
Defined.
```

Figure 7.11: Substitution

The implementation provides a number of important lemmas and theorems which will help us to prove progress and preservation and to ensure that our implementation is indeed correct.

First we need a lemma which demonstrates that the shifting implementation is correct. This theorem states that we can insert an arbitrary type into a context at a cutoff by shifting our term at the index of the insertion.

```
Lemma shift_correct : ∀ n s xty ty G L,
  Derivation [n; G ++ L |= s @ ty] →
  Derivation [n; G ++ (xty :: L) |= shift (length G) s @ ty].
```

Figure 7.12: Shift Correctness

The substitution preservation theorem given in Figure 7.13 demonstrates that substitution preserves types given that we substitute a variable with a term of the same type as that variable in our context.

The proof of this result is done by induction on the term t and requires a *strengthening lemma*, which states that we can remove irrelevant variables from our contexts. In addition it uses the shift correctness lemma.

```
Theorem sub_preservation : ∀ t s n xty ty G L,
  Derivation [n ; G++xty::L |= t @ ty] →
  Derivation [n ; G++L |= s @ xty] →
  Derivation [n ; G++L |= sub t (length G) s @ ty].
```

Figure 7.13: Substitution Type Preservation

Finally, we will need a similar result for type substitution. This is given in Figure 7.14. Essentially this result states that for any valid type and a type derivation for a term t we can derive a valid derivation for that term with one free type substituted. It turns out practically that this is by far the hardest result to prove. It requires the lemma which is given in Figure 7.15.

```
Lemma tysub_derivation : ∀ t n m l ty tyx,
  valid tyx (n+m) →
  Derivation [S (n+m); map (tyshiftn 1 m) l |=t @ ty] →
  Derivation [ (n+m) ; l |=tysubt t m tyx @ tysub ty m tyx].
```

Figure 7.14: Type Substitution Type Preservation

This lemma states that given an arbitrary term, we can substitute in both the context and the term and the type to determine another valid type derivation. It is critical in proving the inductive case for λ-abstraction. The rest of the proof is relatively straightforward.

From here we can describe our deterministic, normal-order evaluation relation, which we give in Figure 7.16.

```
Lemma tysub_all : ∀ t ty tyx n m l,
  valid tyx (n+m) →
  Derivation [S (n+m); l |= t @ ty] →
  Derivation [(n+m); map (fun ty ⇒ tysub ty m tyx) l |= tysubt t m tyx @ tysub ty m tyx].
```

Figure 7.15: Lemma for Type Substitution

```
Inductive Ev : Term → Term → Set :=
  | ev_f : ∀ n, Ev (F n) (Delta n)
  | ev_app : ∀ t t' s, Ev t t' → Ev (App t s) (App t' s)
  | ev_abs : ∀ t s ty, Ev (App (Abs ty t) s) (sub t 0 s)
  | ev_tapp : ∀ t t' ty, Ev t t' → Ev (TApp t ty) (TApp t' ty)
  | ev_lam : ∀ t ty, Ev (TApp (Lam t) ty) (tysubt t 0 ty)
  | ev_fold : ∀ t ty ty', Ev (Unfold (Fold t ty') ty) t
  | ev_unfold : ∀ t t' ty, Ev t t' → Ev (Unfold t ty) (Unfold t' ty)
  | ev_inl : ∀ t r s ty, Ev (Case (Inl t ty) r s) (sub r 0 t)
  | ev_inr : ∀ t r s ty, Ev (Case (Inr t ty) r s) (sub s 0 t)
  | ev_case : ∀ t t' r s, Ev t t' → Ev (Case t r s) (Case t' r s)
  | ev_pair : ∀ t s u, Ev (Split (Pair t s) u) (sub (sub u 0 (shift 0 t)) 0 s)
  | ev_split: ∀ t t' u, Ev t t' → Ev (Split t u) (Split t' u).
```

Figure 7.16: Evaluation Relation

With this relation we can describe our theorem of type preservation which is given in Figure 7.17. This simply shows that any term which is the result of a single step evaluation in any type and term variable context will have a derivation at the same type.

```
Theorem ev_preservation : ∀ t t' n l ty, Derivation Xi [n ; l |= t @ ty] → Ev t t' → Derivation
Xi [n ; l |= t' @ ty].
```

Figure 7.17: Type Preservation

Similar results hold for the transitive and transitive-reflexive closures of evaluation as presented in Figure 7.18.

In addition to the evaluation relation, a program which can calculate an evaluated term is also provided with a strong type specification. Because of the fact that System-F with general recursion is not strongly normalising, it is not possible to represent this function directly unless some additional constraint is given. Here we add an upper-bound on the number of computation steps taken.

The strong specification states that given any natural number bound and a term t we can calculate a term t' which is in the reflexive transitive closure.

We also have a progress result which states that anything in the empty context which is not a value will reduce. Values are defined inductively as being either a λ-abstraction or a Λ-abstraction. These are given in Figure 7.20. Proof of these results for the transitive and reflexive closures is

```
Inductive Evplus : Term → Term → Set :=
| Evplus_base : ∀ t t', Ev t t' → Evplus t t'
| Evplus_next : ∀ t t' t'', Evplus t t' → Ev t' t'' → Evplus t t''.

Inductive Evstar : Term → Term → Set :=
| Evstar_refl : ∀ t t', t = t' → Evstar t t'
| Evstar_plus : ∀ t t', Evplus t t' → Evstar t t'.

Notation "t ~> t'" := (Ev t t') (at level 60).
Notation "t ~>+ t'" := (Evplus t t') (at level 60).
Notation "t ~>* t'" := (Evstar t t') (at level 60).

Theorem evplus_preservation : ∀ t t' n l ty, Derivation Xi [n ; l |= t @ ty] → t ~>+ t' →
Derivation Xi [n ; l |= t' @ ty].

Theorem evstar_preservation : ∀ t t' n l ty, Derivation Xi [n ; l |= t @ ty] → t ~>* t' →
Derivation Xi [n ; l |= t' @ ty].
```

Figure 7.18: Type Preservation of Transitive and Reflexive Closures

```
Definition eval : ∀ (bound : nat) (t : Term), { t' : Term & t ~>* t'}.
```

Figure 7.19: Strong Eval

straightforward.

```
Inductive Value : Term → Set :=
| Value_Lam : ∀ t, Value (Lam t)
| Value_Abs : ∀ t ty, Value (Abs ty t)
| Value_Fold : ∀ t ty, Value (Fold t ty)
| Value_Pair : ∀ t s, Value (Pair t s)
| Value_Inl : ∀ t ty, Value (Inl t ty)
| Value_Inr : ∀ t ty, Value (Inr t ty)
| Value_Unit : Value Unit.

Theorem ev_progress : ∀ t A, Derivation Xi [0; nil |= t @ A] → { s : Term & t ~> s} + (Value
t).
```

Figure 7.20: Progress

7.1.3 Transition Systems

Using these important results we can move to the description of transition systems and simulation. We do this with two inductively defined data-types, describing labels in Figure 7.21 and our transition system relation in Figure 7.22.

We can form labels by introducing an arbitrary term with a correct type derivation and appropriate type, as a test to look at the behaviour of some term which can be reduced to a λ-abstraction and similarly for Λ-abstractions and types.

With labelled transition systems defined, we can move forward to the definition of simulation.

```
Inductive label : Set :=
| lt : Term → label
| lty : Ty → label
| lft : label
| rgt : label
| fst : label
| snd : label
| fld : label.
```

Figure 7.21: Transition System Labels

```
Inductive trans : Term → label → Term → Type :=
| trans_app : ∀ t1 t2 A B,
   Derivation Xi [0; nil |= t1 @ Imp A B] →
   Derivation Xi [0; nil |= t2 @ A] →
   trans t1 (lt t2) (App t1 t2)
| trans_tapp : ∀ t ty A,
   Derivation Xi [0; nil |= t @ All A] →
   valid ty 0 →
   trans t (lty ty) (TApp t ty)
| trans_inl : ∀ t A B,
   Derivation Xi [0; nil |= (Inl t B) @ Or A B] →
   trans (Inl t B) lft t
| trans_inr : ∀ t A B,
   Derivation Xi [0; nil |= (Inr t A) @ Or A B] →
   trans (Inr t A) rgt t
| trans_fst : ∀ s t A B,
   Derivation Xi [0; nil |= (Pair s t) @ And A B] →
   trans (Pair s t) fst s
| trans_snd : ∀ s t A B,
   Derivation Xi [0; nil |= (Pair s t) @ And A B] →
   trans (Pair s t) snd t
| trans_fold : ∀ t ty,
   Derivation Xi [0; nil |= (Fold t ty) @ ty] →
   trans (Fold t ty) fld t
| trans_next : ∀ l t1 t2 t3, t1 ~>* t2 → trans t2 l t3 → trans t1 l t3.
```

Figure 7.22: Transition System

This essentially follows directly from Gordon's presentation [32]. If we want to show that a term b simulates a we need to show that given any possible state a' that a can arrive at by any given label l, that b similarly can arrive at a new state b' by that same label l and we can additionally prove that b' simulates a'. This is essentially a form of Parks' principle for transition systems.

```
CoInductive simulates : Term → Term → Type :=
| simulates_base : ∀ a b,
  (∀ a' l,
     trans a l a' → {b' : Term & trans b l b' & simulates a' b'}) →
  simulates a b.
```

Figure 7.23: Simulation

We have not yet produced a mechanisation of contextual equivalence for System-F, though we expect that this result is possible using the given formulation. The theorem that one would most likely attempt to prove is given in Figure 7.24. While different than the usual formulation for contextual equivalence, this formulation should be more straight forward to prove. The one-hole contexts would act as arbitrary semi-decidability predicates over terms. Essentially here we would describe contextual equivalence as the inability to distinguish between two terms by the means of any arbitrary constructible (*continuous* in the language of Synthetic Topology) semi-decidability predicate[27] in System-F with general recursion.

```
Theorem contextual_equivalence : ∀ C a b A,
  Derivation Xi [0 ; nil |= a @ A] →
  Derivation Xi [0 ; nil |= b @ A] →
  a :<: b →
  (∀ t : Term, Derivation Xi [0 ; nil |= t @ A] →
    Derivation Xi [0 ; nil |= insert C t @ One]) →
  evaluates (insert C a) Unit →
  evaluates (insert C b) Unit.
```

Figure 7.24: Contextual Equivalence

7.2 Haskell Implementation

The Haskell implementation builds on the mechanisation in Coq and provides the expanded syntax and formation rules for System-F$^+$ given in the introduction, along with a type-checker, an interpreter, a supercompiler, a natural-deduction-style proof printer and a totality checker. We use the name *Cocktail* to denote the supercompiler.

The representation of sequents, derivations and terms using the modified de Bruijn scheme follows the same pattern as in the Coq implementation. This turns out to simplify the process of abstraction, strengthening and generalisation as compared to a locally nameless representation, which was used in an earlier design.

The supercompiler is non-deterministic, and we have implemented this using streams of results. The stream itself is formed using Luke Palmer's ω-monad [64]. The ω-monad (which we write M^{ω} in the pseudocode) is specifically designed to allow non-deterministic interleaving between various streams in such a manner as to ensure that it enumerates over all possibilities. It is so named because it can deal with the enumeration of cross-products of denumerable sets, those of order type ω. It is similar to the list monad, though it technically violates the monad laws unless we view lists modulo permutation of elements.

The ability to combine two streams in the ω-monad is given with the $mplus$ operator. We can think of these combinations as allowing us to specify alternative non-deterministic paths to take in our supercompilation.

The syntax for source files is structured as in Figure 7.25 and the representation of the various term and type level constructors are given in Figure 7.26 and Figure 7.27 respectively.

```
 term
where
 function_constant :  type =  term
    ⋮
 function_constant :  type =  term
```

Figure 7.25: File Syntax

`\ x : (A) . r`	:=	$\lambda x : A.r$
`inr(r,A)`	:=	right(r, A)
`inl(r,A)`	:=	left(r, A)
`(r,s)`	:=	(r, s)
`/\ X . r`	:=	$\Lambda X.r$
`case t of { inl(x) => r | inr(y) => s }`	:=	case t of $\{x \Rightarrow r \mid y \Rightarrow s\}$
`split t as (x,y) in {r}`	:=	split t as (x, y) in r
`fold(r,A)`	:=	in$_{\alpha}(r, A)$
`unfold(r,A)`	:=	out$_{\alpha}(r, A)$

Figure 7.26: Textual Term Representation

`A * B`	:=	$A \times B$
`A + B`	:=	$A + B$
`A -> B`	:=	$A \to B$
`\-/ X . A`	:=	$\forall X.A$
`nu X .(A)`	:=	$\nu X.A$
`mu X .(A)`	:=	$\mu X.A$

Figure 7.27: Textual Type Representation

In order to run our supercompiler we can invoke Main. Since no readline facility is used, it is best to use the emacs shell interaction mode.

The online help can be invoked by typing:

```
:help
```

This leads to the options given in Figure 7.28. The program provides a number of program manipulation tools and allows the user to manually produce bisimulation equivalent terms.

By loading a source file using the command:

```
:load ourfile.sup
```

followed by

```
:total 1
```

We can perform a search through lazily produced supercompiled programs and the first such total program will be returned to the user.

```
Cocktail> :help

:load filename To Load a file
:quit To quit Cocktail
:out [filename] Output program to file
:proof [filename] Output derivation to PDF file
:super Supercompile the current program
:reify Reify term as program
:check Check totality of the program
:total [n] Supercompile the current program searching for a
provably total representative over n proofs
:help Show this message
:program To print the current program
:down [n] Descend further into a term taking the n'th branch
:up Ascend once step to the containing context term
:top Ascend to the top level closed term
:norm Normalise the present term
:display Show pdf of derivation
:unfold Unfold the blocking term
:fold Fold a term with a prior term
:term Show the present term
Cocktail>
```

Figure 7.28: On-line Help

In order to view the proof which the supercompiler has determined is total, we can output the current proof using:

```
:display
```

This will export a pdf containing the proof, with some attempts made to strengthen rules implicitly to avoid overly large contexts. The output looks as in Figure 7.29.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\dfrac{}{\cdot,[x:\nu N.1\vee N]\vdash x:\nu N.1\vee N}\;{\scriptstyle\mathrm{Var}}}{\cdot,[x:\nu N.1\vee N]\vdash \mathrm{unfold}(x,\nu N.1\vee N):1\vee(\nu N.1\vee N)}\;{\scriptstyle\mathrm{AlphaE}}
\quad
\dfrac{}{\cdot,[y:\nu N.1\vee N]\vdash y:\nu N.1\vee N}\;{\scriptstyle\mathrm{Var}}
\quad
\dfrac{
\dfrac{}{\cdot,\cdot\vdash \mathit{succ}:(\nu N.1\vee N)\rightarrow\nu N.1\vee N}\;{\scriptstyle\mathrm{FunI}}
\quad
\dfrac{
\dfrac{
\dfrac{}{\cdot,[y:\nu N.1\vee N]\vdash y:\nu N.1\vee N}\;{\scriptstyle\mathrm{Var}}
\quad
\dfrac{}{\cdot,[x':\nu N.1\vee N]\vdash x':\nu N.1\vee N}\;{\scriptstyle\mathrm{Var}}
}{
\cdot,[x':\nu N.1\vee N,y:\nu N.1\vee N]\vdash \begin{array}{l}\mathrm{case}\ \mathrm{unfold}(x',\nu N.1\vee N)\ \mathrm{of}\\ \quad\mathrm{inl}(z)\Rightarrow y\\ \quad\mid\mathrm{inr}(x')\Rightarrow \mathit{succ}\ (\mathit{plus}\ x'\ y)\end{array} :\nu N.1\vee N
}\;{\scriptstyle\mathrm{Rec}^3}
}{
\cdot,[x':\nu N.1\vee N,y:\nu N.1\vee N]\vdash \mathit{plus}\ x'\ y:\nu N.1\vee N
}\;{\scriptstyle\Delta}
}{
\cdot,[x':\nu N.1\vee N,y:\nu N.1\vee N]\vdash \mathit{succ}\ (\mathit{plus}\ x'\ y):\nu N.1\vee N
}\;{\scriptstyle\mathrm{ImpE}}
}{
\cdot,[y:\nu N.1\vee N,x:\nu N.1\vee N]\vdash \begin{array}{l}\mathrm{case}\ \mathrm{unfold}(x,\nu N.1\vee N)\ \mathrm{of}\\ \quad\mathrm{inl}(z)\Rightarrow y\\ \quad\mid\mathrm{inr}(x')\Rightarrow \mathit{succ}\ (\mathit{plus}\ x'\ y)\end{array} :\nu N.1\vee N
}\;{\scriptstyle\mathrm{OrElim}}
}{
\cdot,[x:\nu N.1\vee N]\vdash \lambda(y:\nu N.1\vee N).\ \begin{array}{l}\mathrm{case}\ \mathrm{unfold}(x,\nu N.1\vee N)\ \mathrm{of}\\ \quad\mathrm{inl}(z)\Rightarrow y\\ \quad\mid\mathrm{inr}(x')\Rightarrow \mathit{succ}\ (\mathit{plus}\ x'\ y)\end{array} :(\nu N.1\vee N)\rightarrow\nu N.1\vee N
}\;{\scriptstyle\mathrm{ImpI}}
}{
\cdot,\cdot\vdash \lambda(x:\nu N.1\vee N),(y:\nu N.1\vee N).\ \begin{array}{l}\mathrm{case}\ \mathrm{unfold}(x,\nu N.1\vee N)\ \mathrm{of}\\ \quad\mathrm{inl}(z)\Rightarrow y\\ \quad\mid\mathrm{inr}(x')\Rightarrow \mathit{succ}\ (\mathit{plus}\ x'\ y)\end{array} :(\nu N.1\vee N)\rightarrow(\nu N.1\vee N)\rightarrow\nu N.1\vee N
}\;{\scriptstyle\mathrm{ImpI}}
}{
\cdot,\cdot\vdash \mathit{plus}:(\nu N.1\vee N)\rightarrow(\nu N.1\vee N)\rightarrow\nu N.1\vee N
}\;{\scriptstyle\Delta}
$$

Figure 7.29: Total Proof of Co-Natural Addition

Chapter 8

Conclusion and Further Work

We have presented a framework which utilises transformations on cyclic proof to perform program manipulation with the intention of finding equivalent programs which can be demonstrated to be total by a syntactic check.

In Chapter 2 we introduced a language based on System-F^{+} which enriches the base language with sums, products and (co)inductive types and general recursion. This produces a language rich enough to be comparable to the pure fragments of more standard functional programming languages such as Haskell.

In Chapter 4 we introduced a novel framework for presenting a functional programming language in a natural deduction style where we make use of recurrences to represent recursive structure. This provides a framework for program transformation in a strongly typed setting. The totality of proofs is given by a restriction resembling structural recursion and a guardedness condition which is novel.

The formal footing provided in this chapter should be helpful in understanding the process of supercompilation and other cyclic program transformations such as the worker/consumer and loop-rolling type.

Future work would extend the notion of cyclic proofs of the form we have presented to include a language such as the calculus of constructions. This would allow much more sophisticated specifications of program terms to be described.

In addition it would be useful to include a bisimulation substitution law. This would give an extensional type which would allow the use of cyclic proofs which were bisimulation equivalent to be substituted in the proof tree. Several example proofs appear to require such an extension to demonstrate totality by way of program transformation.

Perhaps most importantly this would provide a formal foundation for some approaches currently known as higher-order, or higher-level supercompilation [43]. The strength of type-theory is in the ability to check the correctness of proofs quickly. Theorem proving in the manner of proof assistants such as ACL2 [42] which use transformation of terms do not provide a checkable proof of their chain of reasoning. If we insist that transformation techniques produce the proof of the bisimulation then the problem of checking type correctness is possible given a suitable extensional type theory with a bisimulation substituion rule. This will allow us to make better use of program transformation techniques for theorem proving. Observational Type Theory[5] is probably the appropriate tool for providing an evidential approach to program transformations of this nature. The ability to provide explicit substitutions of bisimilar terms is required to type check a number of interesting examples.

```
Ω(t)       :=  $\mathrm{in}_\nu(\mathrm{left}(1, 1+\mathbb{T}), \mathbb{T})$
Ω(delay)   :=  $\lambda x : \mathbb{T}.\mathrm{in}_\nu(\mathrm{right}(x, 1+\mathbb{T}), \mathbb{T})$
Ω(f)       :=  $delay\ f$
Ω(join)    :=  $\lambda x : \mathbb{T}\ y : \mathbb{T}.$
               case $\mathrm{out}_\nu(x, \mathbb{T})$ of
                   $z \Rightarrow t$
                 $\mid x' \Rightarrow$
                   case $\mathrm{out}_\nu(y, \mathbb{T})$ of
                       $z \Rightarrow t$
                     $\mid y' \Rightarrow delay\ (join\ x'\ y')$

Ω(ex)      :=  $\lambda l : \overline{[A]}\ p : (\overline{[A]} \to \mathbb{T}).$
               case $l$ of
                   $z \Rightarrow f$
                 $\mid pair \Rightarrow$
                   split $pair$ as $(x, xs)$
                   in $delay\ (join\ (p\ x)\ (ex\ p\ xs))$

Ξ(t)       :=  $\mathbb{T}$
Ξ(f)       :=  $\mathbb{T}$
Ξ(delay)   :=  $\mathbb{T} \to \mathbb{T}$
Ξ(join)    :=  $\mathbb{T} \to \mathbb{T} \to \mathbb{T}$
Ξ(ex)      :=  $\overline{[A]} \to (A \to \mathbb{T}) \to \mathbb{T}$
```

Figure 8.1: Semi-Decidable Existential Functional

One example can be given for the type of an existential quantifier over the domain of semi-decidable truth values given by the delay-monad: $\nu X.A + X$, which either returns an A or delays a step, over the Sierpinski type 1 which is $\nu X.1 + X$, and which we will write as $\mathbb{T}$. As we can see from this type, it is isomorphic with the co-natural numbers. The program representing the existential functional is given in Figure 8.1.

If we attempt to perform program transformation on this term, we can in fact find a bisimilar term for ex which meets the guardedness condition, however it relies on the associativity of $join$. Supercompilation can easily show that $join$ is associative as the terms $join\ (join\ x\ y)\ z$ and $join\ (join\ x\ y)\ z$ both supercompile to identical terms. This means we can establish the type correctness of the term ex provided we are allowed to substitute bisimilar terms. Both higher level supercompilation [43] and distillation [36] are capable of automatically transforming this example.

In Chapter 3 we gave a description of a transition system semantics for System-F^{+}. The presentation here was developed based on Gordon's published works, which we extended to include universal quantification over types. Our mechanisation of this theory is novel.

It would be useful to extend this work to include a transition system for a richer calculus such as the calculus of constructions. The inclusion of existential quantification may present difficulties, but it is possible that this could be overcome using techniques such as used in environmental bisimulations [44].

The extension could also potentially allow libraries mixing total and non-total terms. This can make programming in total languages more flexible since some partial terms are in fact total in a context, and this can sometimes be decided by program manipulation of cyclic proofs. Essentially the restriction to particular terms as arguments of a partial function can be total provided it is restricted in some way to ensure that these arguments are in the domain of the image. We can imagine how totality would hold for a filter function over streams, provided that the filter predicate was searching for a finite number in a list of natural numbers. Program transformation and synthesis techniques could transform this into a single recursive term with a structurally reducing argument, which would be total.

In Chapter 5 we provided a non-deterministic program transformation algorithm based on supercompilation. The transformation methodology used is novel in several ways. The non-deterministic search is breadth first but limited by a predicate which rejects paths which will not meet conditions on cyclic proofs required to ensure totality. The use of program transformation to obtain manifestly type correct terms is a novel development.

It would be useful to export a demonstration of the equivalence between the original and transformed term. This could be done by producing a Coq term which demonstrates the bisimulation relation with respect to the transition system semantics of the programming language. This would dovetail nicely with a type theory which allowed substitutions of bisimilar terms. In addition this appears to be a fruitful way of ensuring that various optimising transformations in programs such as compilers can demonstrate the correctness of their transforms and limit the scope for the introduction of changes in termination behaviour.

In Chapter 6 we demonstrated a coinductive relational approach to looking at totality based on ideas given by Bradfield and Stirling [12] and Milner and Tofte [56]. Our application of the approach is unique and differentiates itself from that of Bradfield and Stirling by applying the technique to a term language. We differentiate from Milner and Tofte in presenting relations which do not include non-termination.

Much work is yet to be done on understanding the connections between models and transition system approaches to the structural operational semantics of terms. The extension to term calculi such as the calculus of constructions is one obvious step. In addition it would be useful to explore other possible restrictions on cyclic proofs which might lead to satisfaction of relations of this type. It would also be desirable to have a complete mechanisation of the approach given in this work.

In Chapter 7 we gave two implementations. One of the implementations is the mechanisation of sections of the theory provided in the proof assistant Coq. The mechanisation of System-F enriched with general recursion and the transition system relation and simulation are novel.

It is hoped that the work done here can provide a basis for a full mechanisation of a supercompiler in Coq. This would be a major step forward for program correctness. In order to do this there are a number of steps which must be taken. A proof of contextual equivalence would need to be completed for terms satisfying a bisimulation. The generalisation algorithm which has been implemented in Haskell would need to be provided in Coq, and the supercompilation core algorithm would also need to be implemented.

In conclusion we have produced novel contributions to the theory of cyclic type systems, discovered new uses for supercompilation and provided a novel mechanisation of much of this theory in Coq.

It seems likely that future programming languages in areas that require high availability or extremely low failure rates will be forced to adopt formal methodologies for connecting specifications to their software. The need for automation in the process of checking specifications against their programs will only become more important. We have made a contribution to this end.

Bibliography

[1] Andreas Abel. Semi-continuous sized types and termination. In Zoltán Ésik, editor, *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 21-24, 2006, Proceedings*, volume 4207 of *Lecture Notes in Computer Science*, pages 72–88. Springer-Verlag, 2006.

[2] Samson Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[3] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, New York, NY, USA, 2007.

[4] Thorsten Altenkirch and Nils Anders Danielsson. Termination checking in the presence of nested inductive and coinductive types. In Ekaterina Komendantskaya, Ana Bove, and Milad Niqui, editors, *PAR-10*, volume 5 of *EPiC Series*, pages 100–105. EasyChair, 2012.

[5] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In Aaron Stump and Hongwei Xi, editors, *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, pages 57–68. ACM, 2007.

[6] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[7] H. P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[8] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[9] Yves Bertot and Ekaterina Komendantskaya. Inductive and Coinductive Components of Corecursive Functions in Coq. *Electron. Notes Theor. Comput. Sci.*, 203(5):25–47, 2008.

[10] Dietmar Berwanger, Erich Grädel, and Giacomo Lenzi. On the variable hierarchy of the modal mu-calculus. In *Computer Science Logic, CSL 2002*, volume 2471 of *LNCS*, pages 352–366. Springer-Verlag, 2002.

[11] J. Bradfield and C. Stirling. *Modal logics and mu-calculi: an introduction*. Elsevier, 2001.

[12] Julian C. Bradfield and Colin Stirling. Local model checking for infinite state spaces. *Theor. Comput. Sci.*, 96(1):157–174, 1992.

[13] James Brotherston. Cyclic Proofs for First-Order Logic with Inductive Definitions. In B. Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods: Proceedings of TABLEAUX 2005*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005.

[14] Rod M. Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[15] J. Chapman. *Type checking and normalisation*. PhD thesis, University of Nottingham, 2008.

[16] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 1940.

[17] Alonzo Church and J. Barkley Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[18] J. Robin B. Cockett. Deforestation, program transformation, and cut-elimination. *Electr. Notes Theor. Comput. Sci.*, 44(1), 2001.

[19] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[20] Charles Consel and Olivier Danvy. Partial Evaluation: Principles and Perspectives. *Journees Francophones des Langages Applicatifs*, February 1993.

[21] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[22] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, jan 1977.

[23] Haskell Curry. Functionality in Combinatory Logic. *Proceedings of the National Academy of Sciences*, 20:584–590, 1934.

[24] Nils A. Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 41, pages 206–217, New York, NY, USA, January 2006. ACM.

[25] Nils Anders Danielsson. Beating the productivity checker using embedded languages. In *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers (PAR2010)*, 2010.

[26] Nicolaas Govert De Bruijn. Lambda Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[27] M. H. Escardo. Synthetic topology of data types and classical spaces. *ENTCS, Elsevier*, 87:21–156, 2004.

[28] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen*. Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam, 1969.

[29] Jean-Yves Girard. Une Extension de l'Interpretation de Gödel à l'Analyse, et son Application à l'Élimination des Coupures dans l'Analyse et la Théorie des Types. *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, 1971.

[30] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types (Cambridge Tracts in Theoretical Computer Science)*. Cambridge University Press, April 1989.

[31] Andrew D. Gordon. A tutorial on co-induction and functional programming. In *IN GLASGOW FUNCTIONAL PROGRAMMING WORKSHOP*, pages 78–95. Springer, 1994.

[32] Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theor. Comput. Sci.*, 228(1-2):5–47, 1999.

[33] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for $f_{<}$:. 1995.

[34] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *In Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 386–395. ACM Press, 1996.

[35] G. W. Hamilton. Poitín: Distilling Theorems From Conjectures. *Electronic Notes in Theoretical Computer Science*, 151(1):143–160, 2006.

[36] G. W. Hamilton. Distillation: extracting the essence of programs. In *PEPM âĂŹ07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 61–70, New York, NY, USA, 2007. ACM.

[37] G. W. Hamilton and G. Mendel-Gleason. A graph-based definition of distillation. In *SECOND INTERNATIONAL WORKSHOP ON METACOMPUTATION IN RUSSIA (META 2010)*, pages 47–63. Ailamazyan University of Pereslavl, 2010.

[38] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[39] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, pages 103–112, 1996.

[40] P. Hudak and P. Wadler. Report on the Programming Language Haskell. Technical report, 1990.

[41] Neil D. Jones. Program termination analysis by size-change graphs (abstract). In *IJCAR*, pages 1–4, 2001.

[42] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[43] Ilya Klyuchnikov and Sergei Romanenko. Towards higher-level supercompilation. In *SECOND INTERNATIONAL WORKSHOP ON METACOMPUTATION IN RUSSIA*, pages 82–101. Ailamazyan University of Pereslavl, 2010.

[44] V. Koutavas, P. B. Levy, and E. Sumii. From applicative to environmental bisimulation. In *Proceedings of the 27th Conference of the Mathematical Foundations of Programming Semantics*, 2011.

[45] Dimitur Krustev. A simple supercompiler formally verified in coq. In *SECOND INTERNATIONAL WORKSHOP ON METACOMPUTATION IN RUSSIA (META 2010)*, pages 102–127. Ailamazyan University of Pereslavl, 2010.

[46] M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In *Proceedings of the International Static Analysis Symposium*, pages 230–245, 1998.

[47] Michael Leuschel and Thierry Massart. Infinite state model checking by abstract interpretation and program specialisation. In *LOPSTR*, pages 62–81, 1999.

[48] Alexei Lisitsa and Andrei Nemytykh. Towards verification via supercompilation. *Computer Software and Applications Conference, Annual International*, 2:9–10, 2005.

[49] Jianguo Lu, Masateru Harao, and Masami Hagiya. Higher Order Generalization. In *JELIA '98: Proceedings of the European Workshop on Logics in Artificial Intelligence*, pages 368–381, London, UK, 1998. Springer-Verlag.

[50] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[51] Conor McBride. Let's See How Things Unfold: Reconciling the Infinite with the Intensional (Extended Abstract). In *CALCO*, pages 113–126, 2009.

[52] Gavin Mendel-Gleason and Geoff Hamilton. Cyclic proofs and coinductive principles. In *PAR-2010: Workshop on Partiality and Recursion in Interactive Theorem Provers*, 2010.

[53] Gavin E. Mendel-Gleason and Geoff Hamilton. Supercompilation and normalisation by evaluation. In *SECOND INTERNATIONAL WORKSHOP ON METACOMPUTATION IN RUSSIA (META 2010)*, pages 128–145. Ailamazyan University of Pereslavl, 2010.

[54] Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, October 2005.

[55] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[56] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, September 1991.

[57] Robin Milner. *A Calculus of Communicating Systems*. Springer Verlag, 1980.

[58] Faron Moller and Alexander Rabinovich. On the Expressive Power of CTL*. In *LICS '99: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, page 360, Washington, DC, USA, 1999. IEEE Computer Society.

[59] James H. Morris. Lambda Calculus Models of Programming Languages. *LCS Technical Report*, 1968.

[60] M. Mottl. Automating functional program transformation, 2000. MSc Thesis. Division of Informatics, University of Edinburgh.

[61] Mayur Naik and Jens Palsberg. A type system equivalent to a model checker . In Sagiv Mooly, editor, *ESOP : European symposium on programming No14* , pages 374–388. Springer-Verlag, Berlin, April 2005.

[62] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[63] Ulf Norell. Agda 2.

[64] Luke Palmer. Omega Monad, 2012.

[65] Simon L. Peyton Jones and David R. Lester. *Implementing functional languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[66] Frank Pfenning. Unification and anti-unification in the calculus of constructions . In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS 1991)*, pages 74–85. IEEE Computer Society Press, July 1991.

[67] Frank Pfenning. On a logical foundation for explicit substitutions. In *RTA*, page 19, 2007.

[68] Frank Pfenning and Christine Paulin-mohring. Inductively defined types in the calculus of constructions. pages 209–228. Springer-Verlag, 1990.

[69] B.C. Pierce. *Basic category theory for computer scientists*. Foundations of computing. MIT Press, 1991.

[70] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[71] G. D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[72] John Reynolds. Towards a Theory of Type Structure. *Colloque sur la Programmation*, pages 408–425, 1974.

[73] D. Sands. Total Correctness by Local Improvement in the Transformation of Functional Programs. *ACM Transactions on Programming Languages and Systems*, 18(2), mar 1996.

[74] Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31:15:1–15:41, May 2009.

[75] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 33:5:1–5:69, January 2011.

[76] Luigi Santocanale. A Calculus of Circular Proofs and its Categorical Semantics. pages 357–371. Springer, 2002.

[77] Dana Scott. Data Types as Lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976.

[78] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[79] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. Elsevier Science Inc., New York, NY, USA, 2006.

[80] Thomas Studer. On the Proof Theory of the Modal mu-Calculus. *Studia Logica*, 89(3):343–363, 2008.

[81] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM.

[82] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. *SIGPLAN Not.*, 40:63–74, January 2005.

[83] W. W. Tait. Intensional Interpretations of Functionals of Finite Type I. *Journal of Symbolic Logic*, 32:198–212, 1967.

[84] A. Tiu, G. Nadathur, and D. Miller. Mixing Finite Success and Finite Failure in an Automated Prover.

[85] Alwen Tiu. Model checking for pi-calculus using proof search. In *Proceedings of CONCUR 2005*, volume 3653, pages 36–50. Springer-Verlag, 2005.

[86] V. F. Turchin. The Language Refal–The Theory of Compilation and Metasystem Analysis. Technical Report 18, Curant Institute of Mathematics, 1980.

[87] Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.

[88] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–65, 1936.

[89] D. A. Turner. Total Functional Programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.

[90] Shobha Vasudevan and Jacob A. Abraham. Static program transformations for efficient software model checking. In *IFIP Congress Topical Sessions*, pages 257–282, 2004.

[91] Y. Venema. *Temporal Logic*, pages 203–223. Blackwell Publishers, Malden, USA, 2001.

[92] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *European Symposium on Programming*, pages 344–358, 1988.

[93] David Wahlstedt. Type theory with first-order data types and size-change termination. Technical report, 2004. Licentiate thesis 2004, No. 36L.

[94] Pierre Wolper. Constructing Automata from Temporal Logic Formulas: A Tutorial. *Lecture Notes in Computer Science*, 2090:261, 2001.

Appendix A

Supercompilation Implementation

```
super :: Path -> Holds -> Omega PreProof
super p h =
    let h' = sequentNorm h
    in if whistle h' p
       then mzero
       else do
         -- * Instances *
         -- find all potential instances in the history
         -- and try them each, in turn.
         (i,ts) <-
             each . concat $
               map (\ (i,h'') ->
                      maybeToList $
                      fmap (\ s -> (i, nub \$ map snd s)) (h'' >- h'))
                   (zip [0..] p)
         -- Make all possible supercompiled derivations of
         -- subterms not in the parent
         ds <- sequence $
               map (\textbackslash t -> do
                       let (Holds fctx vctx tctx _ _) =  h'
                       ty <- each . eitherToList \$ typeof fctx vctx tctx t
                       let h'' = (Holds fctx vctx tctx t ty)
                       super (h':p) h'') ts
         return $ Pointer h' (i+1) ds
        `mplus` do
         -- * Coupling / Generalisation *
         (_,h'') <- each . maybeToList $ couples h' p
```

```
  let oldpath = p
  (gf@@(Holds fctx vctx tctx _ _),typargs,termargs) <-
      each . eitherToList $ generalised_function h' h''
  -- Make sure pointers are
  -- offset properly for upcoming applications.
  let stubpath = replicate (length typargs + length termargs)
                           placeholder
  gd <- super (stubpath++oldpath) gf
  let f = proofTerm gd
      tyapp = foldr (\ ty f -> TApp f ty) f typargs
      typath = replicate (length typargs) placeholder
  (t,_) <- foldrM (\ ha (f,p) -> do
                     d <- super p ha
                     --let a = sequentTerm ha --
                     let a = proofTerm d
                         t = App f a
                     return (t,placeholder:p))
           (tyapp,typath++oldpath)
           (each termargs)
  d' <- each . eitherToList $  makeProof fctx vctx tctx t
  return d'
`mplus` do
  -- * Function constant unfolding *
  h'' <- each . maybeToList $ unfold h' -- unfold
  r <- super (h':p) h''
  return $ DeltaRule h' r
`mplus`
  -- * Subterm supercompilation *
  super' p h'
```

Appendix B

System-F Embedding in Coq

```
Definition Zero := ∀ (a : Prop), a.
Notation "'0" := Zero (at level 1).
Lemma Zero_uninhabited : ∀ t : '0, False.
  intros.
  unfold Zero in t. apply t.
Defined.
Definition One := ∀ (a : Prop), a → a.
Notation "'1" := One (at level 1).
Definition unit : '1 := (fun (a : Prop) (x : a) ⇒ x).
Implicit Arguments unit [a].
Notation "'()" := unit (at level 1).
Definition And (a : Prop) (b : Prop) := ∀ (z : Prop), (a → b → z) → z.
Notation "a |*| b" := (And a b) (at level 90, b at next level).
Definition pair : ∀ (a b : Prop), a → b → And a b :=
    fun (a b : Prop) ⇒
      fun (x : a) (y : b) ⇒
        fun (z : Prop) (f : a → b → z) ⇒ f x y.
Implicit Arguments pair [a b].
Notation "[ x , y ]" := (pair x y) (at level 1, y at next level).
Definition fst : ∀ (a b : Prop), a |*| b → a :=
  fun (a b : Prop) ⇒
```

```
    fun (p : ∀ (z : Prop), (a → b → z) → z) ⇒
      p a (fun (x : a) (y : b) ⇒ x).
Implicit Arguments fst [a b].

Definition snd : ∀ (a b : Prop), a |*| b → b :=
  fun (a b : Prop) ⇒
    fun (p : ∀ (z : Prop), (a → b → z) → z) ⇒
      p b (fun (x : a) (y : b) ⇒ y).
Implicit Arguments snd [a b].

Definition Or (a : Prop) (b : Prop) := ∀ (z : Prop), (a → z) → (b → z) → z.
Notation "a |+| b" := (Or a b) (at level 90, b at next level).

Definition inl : ∀ (a b : Prop), a → a |+| b :=
    fun (a b : Prop) ⇒
      fun (x : a) ⇒
        fun (z : Prop) (left : a → z) (right : b → z) ⇒ left x.
Implicit Arguments inl [a].
Definition inr : ∀ (a b : Prop), b → a |+| b :=
    fun (a b : Prop) ⇒
      fun (y : b) ⇒
        fun (z : Prop) (left : a → z) (right : b → z) ⇒ right y.
Implicit Arguments inr [b].

Definition case : ∀ (a b c: Prop), a |+| b → (a → c) → (b → c) → c :=
  fun (a b c : Prop) ⇒
    fun (x : a |+| b) (f : a → c) (g : b → c) ⇒
      x c f g.

Lemma or_inl : ∀ (a b c : Prop) (f : a → c) (g : b → c) (x : a),
  case a b c (inl b x) f g = f x.
Proof.
  unfold case.
  unfold inl. auto.
Defined.

Lemma or_inr : ∀ (a b c : Prop) (f : a → c) (g : b → c) (x : a),
  case a b c (inl b x) f g = f x.
Proof.
  unfold case.
  unfold inl. auto.
Defined.

Definition bool := '1 |+| '1.
Definition true := inl '1 '().
Definition false := inr '1 '().

Definition natF := (fun x : Prop ⇒ x |+| '1).
Definition mu (F: Prop → Prop) := all (fun (x : Prop) ⇒ (F x → x) → x).

Definition fold (F : Prop → Prop) : ∀ (x : Prop), (F x → x) → mu F → x.
Proof.
  intros x k t.
  apply (t x k).
Defined.

Definition inmu (F : Prop → Prop) (FM : ∀ (a b : Prop), (a → b) → (F a → F b))
  : F (mu F) → mu F.
Proof.
  intros s.
  unfold mu. unfold all.
  intro x. intro k.
  refine (k (FM (mu F) x (fold F x k) s)).
Defined.

Definition inmu' (F : Prop → Prop) (FM : ∀ (a b : Prop), (a → b) → (F a → F b))
  : F (mu F) → mu F.
Proof.
  intros s.
  unfold mu. unfold all.
  intro x. intro k.
  refine (k (FM (mu F) x (fun t : mu F ⇒ t x k) s)).
Defined.
```

```
Definition outmu (F : Prop → Prop) (FM : ∀ (a b : Prop), (a → b) → (F a → F b))
  : mu F → F (mu F).
Proof.
  intros s.
  apply (fold F).
  apply FM. apply inmu. auto. auto.
Defined.

Definition outmu' (F : Prop → Prop) (FM : ∀ (a b : Prop), (a → b) → (F a → F b))
  : mu F → F (mu F).
Proof.
  intros s.
  refine (s (F (mu F)) (FM (F (mu F)) (mu F) (inmu F FM))).
Defined.

Definition ex (F : Prop → Prop) := ∀ y: Prop, (∀ x : Prop, F x → y) → y.

Definition pack (F : Prop → Prop) : ∀ (x : Prop), F x → ex F :=
  fun (x : Prop) ⇒
    fun (e : F x) ⇒
      fun (y: Prop) ⇒
        fun (f : ∀ (z : Prop), F z → y) ⇒ f x e.

Definition unpack (F : Prop → Prop)
  : ex F → ∀ (y : Prop), (∀ (x : Prop), F x → y) → y :=
    fun (u : ex F) (y : Prop) (f : ∀ (x : Prop), F x → y) ⇒
      u y f.

Definition nu (F : Prop → Prop) := ex (fun (x : Prop) ⇒ (x → F x) |*| x).
Definition unfold (F : Prop → Prop)
  : ∀ (x : Prop), (x → F x) → (x → nu F) :=
    fun (x : Prop) ⇒
      fun (f : x → F x) ⇒
        fun (e : x) ⇒
          pack (fun x ⇒ (x → F x) |*| x) x [f, e].

Definition outnu (F : Prop → Prop) (FM : ∀ (a b : Prop), (a → b) → (F a → F b))
```

```
  : nu F → F (nu F).
Proof.
  refine
    (fun (u : nu F) ⇒
      unpack _ u (F (nu F))
      (fun (x : Prop) (w : (x → F x) |*| x) ⇒
        FM _ _ (unfold _ x (fst w)) ((fst w) (snd w)))).
Defined.

Definition innu (F : Prop → Prop) (FM : ∀ (a b : Prop), (a → b) → (F a → F b))
  : F (nu (fun z ⇒ F z)) → nu (fun z ⇒ F z).
Proof.
  refine (unfold _ (F (nu (fun z ⇒ F z))) (FM _ _ (outnu F FM))).
Defined.
```

Examples

```
Definition NatF := fun n : Prop ⇒ '1 |+| n.
Definition NatFM : ∀ a b : Prop, (a → b) → NatF a → NatF b.
Proof.
  unfold NatF. intros a b f n. unfold Or in n. apply n.
  intros. apply inl. auto.
  intros. apply inr. apply f. exact H.
Defined.
Definition Nat := mu NatF.
Definition z : Nat := inmu NatF NatFM (inl Nat '()).
Definition s : Nat → Nat := fun n : Nat ⇒ inmu NatF NatFM (inr '1 n).

Definition FL := (fun N ⇒ '1 |+| (Nat |*| N)).
Definition CoList := nu FL.

Definition FML : ∀ (a b : Prop), (a → b) → (FL a → FL b).
Proof.
  unfold FL.
  intros a b f c.
  refine (c ('1 |+| (Nat |*| b)) (fun x : '1 ⇒ inl (Nat |*| b) '()) (fun x : (Nat |*|
a) ⇒ inr _ _)).
  refine [_,_].
  exact (fst x).
  exact (f (snd x)).
Defined.

Definition FN := (fun N ⇒ '1 |+| N).
Definition Conat := nu FN.

Definition FMN : ∀ (a b : Prop), (a → b) → (FN a → FN b).
Proof.
  unfold FN.
  intros a b f c.
  apply (c ('1 |+| b) (fun x : '1 ⇒ inl b '()) (fun x : a ⇒ inr '1 (f x))).
Defined.

Definition out_FN := outnu FN FMN.
Definition inn_FN := innu FN FMN.

Definition cz : Conat.
Proof.
  unfold Conat. unfold FN.
  cut '1. apply unfold. intros. apply inl. auto.
  exact '().
Defined.

Definition cs : Conat → Conat.
Proof.
  unfold Conat.
  intros.
  apply (fun (c : nu FN) ⇒ inr '1 c) in H.
  change (FN (nu FN)) in H.
  apply inn_FN.
  unfold FN at 2.
  change (FN (nu FN)). auto.
Defined.
```

```
Definition inf : Conat := unfold FN Conat (fun x : Conat ⇒ inr '1 x) cz.

Definition clt (n : nat) (c : Conat) : bool.
Proof.
  refine
    ((fix clt (n : nat) (c : Conat) : bool :=
      match n with
        | 0 ⇒ true
        | S n' ⇒ (out_FN c) bool (fun x : '1 ⇒ false) (fun y ⇒ clt n' y)
      end) n c).
Defined.

Lemma inf_infinity : ∀ (n : nat), clt n inf = true.
Proof.
  induction n.
  simpl. auto.
  simpl.
  unfold out_FN.
  unfold outnu. unfold unpack.
  unfold inf at 1.
  unfold unfold at 1. unfold pack at 1.
  unfold FMN. unfold fst at 1.
  unfold pair at 1. unfold inr at 1.
  unfold inr at 1.
  unfold fst at 1. unfold pair at 1.
  unfold snd at 1. unfold pair at 1. unfold inf in IHn.
  auto.
Defined.

Lemma inf_infinity2 : ∀ (n : nat), clt n inf = true.
Proof.
  induction n ; compute in × ; auto.
Defined.

Definition List (A : Prop) := ∀ (X : Prop), X → (A → X → X) → X.
Definition nil := fun (A : Prop) (X : Prop) (n : X) (c : A → X → X) ⇒ n.
Definition cons := fun (A : Prop) (a : A) (l : List A) ⇒
  fun (X : Prop) (n : X) (c : A → X → X) ⇒ c a (l X n c).
Implicit Arguments cons [A].
Definition mapl : ∀ (a b : Prop), (a → b) → List a → List b.
Proof.
  refine
    (fun (a b : Prop) ⇒
      (fun (f : a → b) (foldr : List a) ⇒
        foldr (List b) (nil b) (fun (x : a) (l : List b) ⇒ cons (f x) l))).
Defined.
Definition foldr : ∀ (a b : Prop), (a → b → b) → b → List a → b.
Proof.
  refine
    (fun (a b : Prop) ⇒
      fun (f : a → b → b) (n : b) (l : List a) ⇒
        l b n (fun (x : a) (y : b) ⇒ f x y)).
Defined.
```

Appendix C

Non-positive Types in System-F

```
Definition Nat := ∀ (N : Prop), N → (N → N) → N.

Definition zero : Nat := fun (N : Prop) (z : N) (s : N → N) ⇒ z.

Definition succ : Nat → Nat := fun (n : Nat) ⇒
  fun (N : Prop) (z : N) (s : N → N) ⇒ s (n N z s).

Definition List (A : Prop) := ∀ (X : Prop), X → (A → X → X) → X.

Definition nil := fun (A : Prop) (X : Prop) (n : X) (c : A → X → X) ⇒ n.

Definition cons := fun (A : Prop) (a : A) (l : List A) ⇒
  fun (X : Prop) (n : X) (c : A → X → X) ⇒ c a (l X n c).

Implicit Arguments cons [A].

Definition LamMu := ∀ (X : Prop),
  (Nat → X) →
  (Nat → List X → X) →
  ((∀ (Y : Prop), ((X → Y) → Y)) → X) → X.

Definition var : Nat → LamMu :=
  fun (n : Nat) ⇒
    fun (X : Prop)
      (v : Nat → X)
      (f : Nat → List X → X)
      (m : (∀ (Y : Prop), ((X → Y) → Y)) → X) ⇒
      v n.

Definition func : Nat → List LamMu → LamMu :=
  fun (n : Nat) ⇒
    fun (t : List LamMu) ⇒
      fun (X : Prop)
        (v : Nat → X)
        (f : Nat → List X → X)
        (m : (∀ (Y : Prop), ((X → Y) → Y)) → X) ⇒
        f n (t (List X) (nil X) (fun (x : LamMu) (y : List X) ⇒ cons (x X v f m) y)).
Definition mu : (∀ (Y : Prop), (LamMu → Y) → Y) → LamMu.
Proof.
  unfold LamMu.
  refine
    (fun (zi : (∀ (Y : Prop), (LamMu → Y) → Y)) ⇒
      fun (X : Prop)
        (v : Nat → X)
        (f : Nat → List X → X)
        (m : (∀ (Y : Prop), ((X → Y) → Y)) → X) ⇒
        m (fun (Y : Prop) (g : X → Y) ⇒
          g (zi X (fun e : LamMu ⇒ e X v f m)))).
Defined.
```