Beating Logic

Dependent types with time for synchronous circuits

Master's Thesis

Sybren van Elderen

August 19, 2014

University of Twente
Faculty of EEMCS
Computer Architecture for Embedded Systems (CAES)

Committee:
Dr. Ir. Jan Kuper
Ir. Christiaan Baaij
Prof. Dr. Ir. Arend Rensink

Abstract

Existing functional structural hardware description languages are based on single-cycle descriptions, which makes it difficult to understand the computational essence of the circuit. Multi-cycle descriptions may provide more intuitive descriptions. In this thesis, we present a system which allows such multi-cycle descriptions through the use of timed types. This means that the temporal spread of a computation is encoded in its type. The innovation of this thesis is that we use dependent types as a basis for the timed types. This allows us to describe an instance of a computation, and abstract it over time to form a circuit description. As a side-effect, we gain the potential to describe time varying circuits. We show that by adding a simplistic theorem prover, we gain a rudimentary ability to check the synchronisation of circuits without the need for explicit delay mechanisms.

As a proof of concept, we have implemented the type system. Aside from validating the use of dependent types as a basis for timed types, this revealed that the definition of timed sequences affects the folding behaviour over those sequences, and the kinds of circuits that can be described with those folds.

Contents

1 · Introduction

Earlier research has shown that functional languages and hardware can go, to some extent, hand in hand. The nature of hardware, easily thought of as a composition of computational blocks by means of wires, lies close to the nature of functional progamming, where the composition of functions is a fundamental operation. This has lead to the birth of variety of functional hardware description languages (HDLs), such as Lava [BCSS98], ForSyDe [SJ04], and CλaSH [Baa09, Koo09].

It is a property of all existing functional structural HDLs that they model hardware on a cycle-per-cycle basis: circuit blocks are defined by the computations they perform during a single clock cycle. For example, CλaSH models circuits as (compositions of) Mealy machines, with functions that act on an input and a state, and that return an output and a new state, on a per-cycle basis. However, the intended computations performed by circuits as a whole may span multiple cycles. It is not always obvious from a single-cycle description what the intended behaviour of the corresponding circuit is: the single-cycle descriptions obfuscate the multi-cycle computational essence of the circuits. If we want to understand what the circuit does conceptually, we may have to (mentally) simulate the circuit over multiple cycles. Given the complexity of present day hardware architectures, it will be clear that that is a cumbersome task and will give rise to errors. A definition that describes the multi-cycle behaviour of a circuit would be easier to understand, because it shows the intended computation directly.

This thesis is an initial attempt to provide an abstraction away from the single-cycle descriptions. With this abstraction, circuits can be defined on a higher level by their intended behaviour. Ideally, a translation mechanism might then take these definitions back to single-cycle descriptions, from which the hardware realisations can easily be extracted, for example, by the compiler CλaSH. A first task, however, is to check whether a composition of components, of which the overall behaviour is specified, can be synchronized correctly.

A similar attempt has been made by [Ott13], which introduces *timed types*. This notion separates a computation from its distribution in time: a circuit definition describes *what* needs to be done, while its type specifies *when* it needs to be done.

Because moments in time (clock cycles) are naturally described by values (as opposed to types), it seems also natural to regard timed types as types depending on values. Such types are called *dependent types*. The incorporation of dependent types into programming languages is a growing research topic, and may extend itself to hardware description languages.

The apparent resemblance of timed types to dependent types gives rise to our main research question:

Can we regard timed types as a specific form of dependent types, and if

> so, how can we construct a type system, based on dependent types, that allows type checking of functional multi-cycle hardware specifications?

The perspective of this research question is different from the one taken in [Ott13], which is based on a constraint solving approach.

In this report we first give an informal introduction to dependent types (Chapter 2), followed by a chapter with a description of their type-theoretical background (Chapter 3). Chapter 4 presents a formal definition of timed types, a type system and its implementation, followed by some small examples. The final chapter of this report contains some discussion and conclusions, and we mention some possibilities for future work (Chapter 5). Finally, the appendices contain a description of the dependently typed languages Idris and Agda (Appendix A), the full type system (Appendix B), and its implementation (Appendix C).

2 · A beginning with dependent types

This chapter will give a short informal introduction into dependent types. We start with Haskell-style types, and expand towards dependent types. This chapter is programming oriented, and is only meant to give a little taste of dependent types. For a more detailed exposition of dependently typed programming, we refer to appendix A, which discusses two dependently typed programming languages.

2.1 Types in Haskell

Anyone who has programmed in Haskell is probably familiar with its type system. Every term has a type, which can be written as $term : type$.[1] There are base types, like *Bool* and *Char*, and compound types which are built with other types, like the function type $Char \to Bool$ and the tuple type $(Char, Bool)$.

Since Haskell is based on the Hindley-Milner type system, it also has so-called parametric polymorphic types[Jon03, DM82]. These types are parametrised by other types: they contain type variables, which can act as a type when multiple types are possible. For example, the identity function *id* has type $a \to a$, where a is such a type variable. It means that the function can be used for values of any type. Haskell implicitly binds type variables with forall-quantifiers at the top level, such that the type of *id* actually is $\forall a.a \to a$. Whenever *id* is used, the type inference algorithm will try to find the actual type and substitute it for a. If, for example, *id* is applied to a boolean value, a will be substituted with *Bool* and the type of *id* becomes $Bool \to Bool$.

Polymorphic types allow more abstraction and thus more flexibility. One can define a single list type without needing to be specific as to what type its elements should have. Similarly, one can write functions that act on every such list, regardless of the type of the elements — think of the *head* function, which returns the first element of the list: it doesn't really need to know the type of the elements. At the same time, polymorphic types are more specific: instead of having one generic list type that disregards the type of its elements, it is possible to create lists of booleans, lists of naturals, lists of strings, et cetera. With more specific types, more specific properties can be expressed, and with more specific properties, there is more confidence in the correctness of the program. Parametric polymorphism is therefore an important concept.

[1] Note on notation: This report will use the type-theoretic ':' to indicate typing, instead of Haskell's '::'.

Parametric polymorphism allows types to depend on other types via type variables, but they cannot depend on *values*: it is for example impossible to create a list type where the length of the list, expressed as natural number, acts as parameter. In fact, in systems like Hindley-Milner, there is a strict separation between types and terms. Types can't be used as values in functions, so is impossible to define operations on types like regular functions. In Haskell, it is possible to copy the behaviour of term values to the type level to some extent, to get natural numbers as types, for example, but this is quite tedious and seemingly redundant.

Dependent types do not have such a separation between terms and types. Types become terms and can use the same language constructs like lambda abstraction and application, so that it is possible to make functions that act on types. The relation between terms and types represented by ':' becomes more general in a hierarchy of values, types, types of types, and types of types of types, etcetera. Every level of this hierarchy may depend on the same and all lower levels, and a type may therefore depend not only on other types, but also on values. Functions can now be generalised as dependent functions: because lambda abstractions are available for types, parameters can become actual arguments of functions. Dependent function types are of the form $\forall x{:}A.B(x)$, which says that the function takes any x of type A (which may be a type of types) to some type B depending on x. If B does not depend on x, the type is equivalent to $A \to B$. To illustrate, the type of the identity function can be written as

$$\forall a{:}\mathit{Type}.\forall x{:}a.a \qquad \text{or} \qquad \forall a{:}\mathit{Type}.a \to a$$

If the identity function is applied to some type A, the result is

$$\mathit{id}\ A : A \to A$$

such that the function $\mathit{id}\ A$ takes any element of type A to itself. Type signatures tend to become quite large if all quantifications are kept explicit. To keep the signatures concise, any quantification that can be inferred and that is not a function parameter can be made implicit, such that a valid type for the identity function is still $a \to a$. As a consequence, unbound variables occuring in types should be understood as bound by a forall-quantifier at top-level.

With dependent types, new possibilites arise to describe more precise properties of programs. The 'list of certain length', or vector, is a common example. This type depends on the number of elements in the vector, such that a vector with three boolean values for example has type $\mathit{Vect}\ \mathit{Bool}\ 3$. If it were a list, it would have type *List Bool*, which is the same type for all lists of booleans regardless of their length.

Similar to lists, there are two constructors for vectors. One is the *Nil* constant, which is defined to be the vector of length zero. The other constructor is *Cons*, which puts some element in front of another vector, and therefore has an element and a vector as arguments. Both constructors are parametrised by types, so they need a type as argument. Because the result type of *Cons* depends on the length of the vector argument (is is one element longer), it also needs that length as argument. The types of the constructors are therefore as follows:

$$\begin{aligned} \mathit{Nil} &: \forall a : \mathit{Type}.\mathit{Vect}\ a\ O \\ \mathit{Cons} &: \forall a : \mathit{Type}.\forall n : \mathbb{N}.a \to \mathit{Vect}\ a\ n \to \mathit{Vect}\ a\ (S\ n) \end{aligned}$$

A bit of the power of dependent types becomes clear when two vectors are concatenated. The vectors can have different lengths, and if they are concatenated, these lengths should be added together for the resulting type. Because types can depend on values, they can depend on terms that evaluate to values, like $n+m$ (with n and m both natural numbers). This makes it possible to guarantee that the length of the resulting vector is the sum of lengths of the argument vectors:

$$\begin{array}{llcl} \mathit{concat} & : \mathit{Vect}\ a\ n \to \mathit{Vect}\ a\ m \to \mathit{Vect}\ a\ (n+m) & & \\ \mathit{concat} & \mathit{Nil} \qquad\quad\ \ y & = & y \\ \mathit{concat} & (\mathit{Cons}\ x\ xs) \quad y & = & \mathit{Cons}\ x\ (\mathit{concat}\ xs\ y) \end{array}$$

The $+$-operator in the type is just the ordinary operator acting on natural numbers.

The body of *concat* defines the concatenation recursively and with pattern matching, like it is commonly done for lists in Haskell: When the first vector is the empty vector *Nil*, the other vector is returned, and when the first vector is not empty (it is a *Cons*) then it takes the first element and puts it in front of the concatenation of the tail and the other vector. Notice that this definition uses the implicit notation. The explicit type of *concat* would be

$$\forall a{:}\mathit{Type}\,.\,\forall n{:}\mathbb{N}\,.\,\forall m{:}\mathbb{N}\,.\,\mathit{Vect}\ a\ n \to \mathit{Vect}\ a\ m \to \mathit{Vect}\ a\ (n+m)$$

The variables a, n and m are quantified, and are therefore function arguments. They would have to be pattern-matched and explicitly passed around, to for example *Cons*. However, because they can be inferred from the vector arguments, they can be made implicit, which removes a lot of clutter.

Another illustrative function with vectors is *replicate*:

$$\begin{array}{llcl} \mathit{replicate} & : \forall n{:}\mathbb{N}\,.\,a \to \mathit{Vect}\ a\ n & & \\ \mathit{replicate} & O \qquad\quad\ x & = & \mathit{Nil} \\ \mathit{replicate} & (S\ k) \quad x & = & \mathit{Cons}\ x\ (\mathit{replicate}\ k\ x) \end{array}$$

This function takes some natural number n and some element of type a, and returns a vector containing n times that element. Like *concat*, it is defined recursively, but now the induction is over n: when n is O, then the empty vector is returned, and when n is the successor of some k, x is pasted in front of the vector with k x's. Notice that because we are now pattern matching on n, it has to be an explicit argument to the function.

3 · Type Theory

In the previous chapter we showed some examples with dependent types, and tried to bring some of its usefulness to light. In this chapter we will explore the theory. Through its history and foundations, functional programming is tightly bound to formal logic, and one of the goals of this chapter is to unveil this relationship. We will start with typeless first-order logic and introduce a version called constructive logic, which already has some computational elements. We then switch to the untyped lambda calculus, which is the basis for many programming languages. By adding types to the lambda calculus, the set of allowable terms will be restricted, and we will show that the result corresponds to constructive first-order logic: lambda terms (programs) will be proofs, and propositions will be types. We will present this language first as the simply typed lambda calculus, which, with some extentions, matches to propositional logic, and then expand it to a dependently typed language, corresponding to first-order predicate logic.

We used several main sources for this chapter: [Tho91] gives introductory expositions of logic and type theory; [End01] is an introduction to mathematical logic; [Bar93] gives a nice overview of lambda calculi with types, and seems to encompass most topics relevant to this chapter from the canoncical work [Bar84]. Finally, [ML84] has some nice informal explanations regarding type theory.

3.1 Classical and constructive first-order logic

3.1.1 *Propositional logic*

In this section we will describe a couple of formal logical systems. Such logical systems describe mathematical reasoning, in a very strict way. Mathematical arguments are considered as patterns of symbols, and any meaning we may give to those symbols is ignored. The correctness of the argument is embodied in the *form* of those patterns, such that it can be judged mechanically, without knowledge of their meaning. Manipulation of the argument can also be done entirely on just the patterns, because the formal system reflects the mathematical reasoning. Of course, in the end the result should be given an interpretation, so a formal system should include a way to give back the meaning to those symbols. In the following discussion, we do not separate form and meaning so rigorously. We do not need it for this thesis, and the forms are more easily understood with informal explanations.

We start with the simplest logic: classical propositional logic. It is the logic of statements that are either true or false, like “the window is open” or “if the window is open, then there’s a draft”. The latter statement is a compound: it consists of the two statements “the window is open” and “there’s a draft”, connected to each other with

"if ... then ..." to form an implication. If a statement cannot be split like this, we call it an atomic proposition.

To build compound statements from arbitrary statements, propositional logic introduces five connectives. Informally, these are "and", "or", "not", "if ... then ..." (implication), and "if and only if" (bi-implication). In the realm of logical formulas we write them as $\wedge$, $\vee$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$ respectively. If we use A and B to denote arbitrary formulas, we can then for example write $A \wedge B$ and say that it is a formula.

Later on in this discussion, we will want to be able to refer within our proofs to a proposition that is always false. For this, we use $\bot$ (falsum).

In short, we can now build formulas according to the following definition:

Definition 3.1.1. A formula is either:

- an atomic proposition denoted by X_0, X_1, X_2, ..., or
- a compound formula of the form $(A \wedge B)$, $(A \vee B)$, $(A \Rightarrow B)$, $(A \Leftrightarrow B)$ or $(\neg A)$, where A and B are formulas, or
- the falsum $\bot$.

If we are to use this definition exactly, we will have to write down a lot of parentheses. When no ambiguity can arise, we will leave them out in favor of looks and clarity.

Formulas give us a way of writing down propositions. Some propositions, the axioms, we will accept without proof. Other propositions we will want to prove. If we succeed the proof, the proposition becomes a theorem.

A logical system lays down deduction rules for constructing proofs. These rules dictate how certain propositions may be inferred from others. Each rule therefore states what are the premisses, and what is the conclusion. A proof is then built by using these rules to infer new propositions from (possibly zero) previously inferred propositions.

In the following, we will use the natural deduction system, which allows to write down proofs in a very natural way. In this system, proofs are constructed in a tree-like structure, like this:

$$\dfrac{\dfrac{A \quad B}{C} \qquad D}{E}$$

The root of the tree (E) is the proposition we want to prove. Each bar indicates the application of a rule: the premisses are above the bar, the conclusion is underneath. In the example above A, B, and D are premisses, and E is a conclusion. C is both a premiss and a conclusion.

Above we said that the rules are used to infer new propositions from possibly zero previously inferred propositions. Most rules prescribe one or more premisses. One rule allows us to conclude propositions from nothing. These propositions are the assumptions we need for the proof, and the rule is therefore called the assumption rule. It says that to assume a proposition, we simply write it down:

$$A$$

Because this is the only rule without premisses, the assumptions will form the leaves of the tree. It is important to keep in mind that the theorem will only hold if the

assumptions hold. If the theorem is to be used in another proof, the assumptions will have to be accounted for in the new proof. Some rules allow certain assumptions to be discharged. This means that they do occur somewhere in the proof, but they do not have to be accounted for when using the theorem (they are discharged from their role as assumption). This happens when a rule has as premiss the fact that we can derive some proposition, and not the proposition itself. The use of discharged assumptions will become more clear when we discuss the individual rules.

To distinguish discharged assumptions from the other assumptions we will surround them with square brackets, and sometimes add labels to show at which steps they are discharged:

$$1\ \frac{[A]^1 \qquad B}{C}$$

The following rules are almost all related to the connectives. Either they introduce a connective into the argument (introduction rules), or they eliminate a connective (elimination rules) from it, allowing us to build up compound formulas and to extract sub-formulas.

$\wedge$-rules

Let's start with the rules for $\wedge$-introduction and elimination:

$$\frac{A \qquad B}{A \wedge B}\ (\wedge\mathrm{I}) \qquad \frac{A \wedge B}{A}\ (\wedge\mathrm{E}_1) \qquad \frac{A \wedge B}{B}\ (\wedge\mathrm{E}_2)$$

The first rule, ($\wedge$I), is the introduction rule, and says that when we have two propositions A and B, we may also conclude their conjunction $A \wedge B$. The other two rules are the elimination rules: they say that when we have a conjunction, we may infer either of its parts.

As an example, we can prove a part of the commutativity of the conjunction: if we assume $A \wedge B$, we can derive $B \wedge A$. We can do this by eliminating the conjunction twice: once for A and once for B. We can then combine them again with the introduction rule:

$$\frac{\dfrac{A \wedge B}{B} \qquad \dfrac{A \wedge B}{A}}{B \wedge A}$$

In the natural deduction system every rule has only a single proposition as conclusion. Therefore, to conclude A and B, we have to use two assumptions. However, because they are the same, we may regard them as a single assumption.

Another example regards associativity. When we assume $A \wedge (B \wedge C)$, we can infer $(A \wedge B) \wedge C$. We first prove A, B and C with the elimination rules:

$$\frac{A \wedge (B \wedge C)}{A} \qquad \frac{\dfrac{A \wedge (B \wedge C)}{B \wedge C}}{B} \qquad \frac{\dfrac{A \wedge (B \wedge C)}{B \wedge C}}{C}$$

And then we can prove the goal with the introduction rule:

$$\frac{\dfrac{A \qquad B}{A \wedge B} \qquad C}{(A \wedge B) \wedge C}$$

∨-rules

The disjunction rules are as follows:

$$\frac{A}{A \vee B}\,(\vee \mathrm{I}_1) \qquad \frac{B}{A \vee B}\,(\vee \mathrm{I}_2) \qquad \frac{A \vee B \qquad \begin{array}{c}[A]\\ \vdots \\ C\end{array} \qquad \begin{array}{c}[B]\\ \vdots \\ C\end{array}}{C}\,(\vee \mathrm{E})$$

We now have two introduction rules. The first says that when we have A, we may conclude 'A or something else'. The second says the same, but with the disjuncts swapped. The elimination rule is a bit more complicated. Here we see the first use of discharged assumptions. The rule says that when we know $A \vee B$, and we can prove C once by assuming A and once by assuming B, we can conclude C. In other words, as long as we know that C can be proved for both assumptions independently, we can conclude C directly from $A \vee B$. We do not need to prove which of A and B is true: according to the premisses, at least one of them is true, and C will hold either way. That means that the conclusion only depends on the assumption $A \vee B$, and not on the assumptions A and B. Therefore, A and B can be discharged. Should we want to use C in another proof, then we only need to take $A \vee B$ into account as assumption.

As an example, we can use again the commutativity property:

$$\frac{A \vee B \qquad \dfrac{[A]^1}{B \vee A} \qquad \dfrac{[B]^1}{B \vee A}}{B \vee A}\,1$$

Using the introduction rules we can prove $B \vee A$ from both A and B, and therefore we can use the elimination rule to conclude $B \vee A$ from $A \vee B$. A and B are discharged by the elimination rule, so we have proved $B \vee A$ from the single assumption $A \vee B$.

⇒-rules

The implication rules are as follows:

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \Rightarrow B}\,(\Rightarrow \mathrm{I}) \qquad \frac{A \qquad A \Rightarrow B}{B}\,(\Rightarrow \mathrm{E})$$

The introduction rule states that if we can derive B from A, we can conclude that A implies B. The elimination rule says that if we know that A implies B, and if we know A holds, we can conclude that B holds.

The conclusion $A \Rightarrow B$ does not depend on whether or not A holds, the only premiss is that B is *derivable* from A. Therefore, A is discharged in the introduction rule. This does not mean that we can't use other assumptions in the derivation of B, but the conclusion will depend on them as well if we do. In the elimination rule A does become a proper assumption, because now we use it together with $A \Rightarrow B$ to derive B.

As an example, we use the transitivity property of implication. If we assume $A \Rightarrow B$, and $B \Rightarrow C$, then we can conclude $A \Rightarrow C$:

$$\frac{\dfrac{\dfrac{[A]^1 \qquad A \Rightarrow B}{B} \qquad B \Rightarrow C}{C}}{A \Rightarrow C}\,1$$

Here we used the elimination rule twice to derive first B and then C, under the assumptions A, $A \Rightarrow B$ and $B \Rightarrow C$. Since we derived C from A, we can conclude $A \Rightarrow C$ with the introduction rule. The conclusion does not depend anymore on whether A holds or not: we have proved that *if* it holds, then C holds as well, which is what we understand as implication. The assumption A can therefore be discharged. Of course, the other assumptions are still necessary.

Note that we could remove the remaining assumptions from the proof by using the implication rule. This would leave us with a tautology:

$$\dfrac{\dfrac{\dfrac{\dfrac{[A]^1 \qquad [A \Rightarrow B]^3}{B} \qquad [B \Rightarrow C]^2}{C}\,1}{A \Rightarrow C}\,2}{(B \Rightarrow C) \Rightarrow (A \Rightarrow C)}\,3$$
$$(A \Rightarrow B) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \Rightarrow C))$$

$\Leftrightarrow$-rules

Bi-implication is very similar to implication:

$$\dfrac{\begin{array}{cc}[A] & [B] \\ \vdots & \vdots \\ B & A\end{array}}{A \Leftrightarrow B}\,(\Leftrightarrow\text{I}) \qquad \dfrac{A \qquad A \Leftrightarrow B}{B}\,(\Leftrightarrow\text{E}_1) \qquad \dfrac{B \qquad A \Leftrightarrow B}{A}\,(\Leftrightarrow\text{E}_2)$$

To introduce bi-implication, A needs to be derivable from B and B needs to be derivable from A. We can eliminate bi-implication using either A or B as premiss, and concluding B or A respectively.

$\neg$-rules and $\bot$

Before discussing negation, we will take a look at $\bot$. This is the symbol to denote absurdity or contradiction. We can regard it as a proposition that is always false, and it should therefore be impossible to prove. If we can prove $\bot$, we can prove anything. This notion is embodied by the following elimination rule:

$$\dfrac{\bot}{A}\,(\bot\text{E})$$

We cannot introduce $\bot$, because it is impossible to prove. However, we can use it to define negation, because if A does not hold, any proof of A leads to a contradiction:

$$\neg A \equiv A \Rightarrow \bot$$

With this definition and the $\bot$-elimination rule, we can derive the rules for $\neg$-introduction and elimination. Elimination can be done by using a contradiction: if we have both A and $\neg A$, we can conclude $\bot$, and the $\bot$-elimination rule subsequently allows us to prove anything. This becomes clear if we use the implicative form of the negation:

$$\dfrac{\dfrac{A \qquad A \Rightarrow \bot}{\bot}\,(\Rightarrow\text{E})}{B}\,(\bot\text{E})$$

By extracting the assumptions and the conclusion, we can form the elimination rule:

$$\frac{A \qquad \neg A}{B}\ (\neg\text{E})$$

To introduce a negation, we want to show that the assumption A leads to $\bot$ (since that is the definition). If we assume A and derive both B and $\neg B$, we conclude $\bot$, like we did earlier. The implication rule then allows us to conclude $A \Rightarrow \bot$, or $\neg A$:

$$\dfrac{\dfrac{\begin{array}{c}[A]^1\\ \vdots\\ B\end{array} \qquad \begin{array}{c}[A]^1\\ \vdots\\ B \Rightarrow \bot\end{array}}{\bot}\ (\Rightarrow\text{E})}{A \Rightarrow \bot}\ 1\ (\Rightarrow\text{I})$$

Again we can turn this into a concise rule:

$$\frac{\begin{array}{c}[A]\\ \vdots\\ B\end{array} \qquad \begin{array}{c}[A]\\ \vdots\\ \neg B\end{array}}{\neg A}\ (\neg\text{I})$$

At the beginning of this section we said that classical propositional logic concerns statements that are either true or false. It implies that there is no grey area between true and false. This principle is called 'the law of the excluded middle'. A consequence of this law is that if we can prove that a statement is not false, we may conclude it is true and vice versa. By definition, we can conlude $\neg A$ when A brings us to a contradiction. If we replace A with $\neg A$, we can conclude $\neg\neg A$ when $\neg A$ does not hold. However, we have no rules yet to conclude A from not $\neg A$. In other words: the law of excluded middle does not hold. To remedy this, we can add one or more of the following equivalent rules.

First, we can introduce the law as an explicit rule:

$$\frac{}{A \vee \neg A}\ (\text{LEM})$$

We can also choose a rule in which we conclude A from $\neg\neg A$, called the rule of double negation:

$$\frac{\neg\neg A}{A}\ (\text{DN})$$

Finally, we can also introduce a rule similar to the introduction rule ($\neg$I). Instead of deriving a contradiction from A and concluding $\neg A$, we assume $\neg A$ and conclude A. This is the classical proof by contradiction:

$$\frac{\begin{array}{c}[\neg A]\\ \vdots\\ B\end{array} \qquad \begin{array}{c}[\neg A]\\ \vdots\\ \neg B\end{array}}{A}\ (\text{CC})$$

3.1.2 *Predicate logic*

In the propositional logic of the previous section, the only objects of interest were propositions. In a mathematical discourse we generally also recognise other kinds of objects, and we make statements *about* these objects. For instance, the statement "42 is even" asserts that the number 42 has the property that it is even, via the predicate "is even". If we extend propositional logic with the means to reason about objects and their properties, we get predicate logic.

In the language of predicate logic objects are referred to with terms:

Definition 3.1.2. A term is either:

- a constant (we will use a, b and c for arbitrary constants),
- a variable (we will use x, y and z for arbitrary variables), or
- a function of arity n, $n \geqslant 0$, applied to n terms (we will use f, g and h to denote arbitrary functions, and write applications as for example $f(x, y)$).

If we refer directly to an object, we call the term a constant. For example, we associate the term 42 directly with the number 42, much like how we associate the name Bertrand Russell with a particular polymath. We can also refer to arbitrary objects, with variables. In the sentence "x is prime", the term x can point to any object. However, a variable will (within boundaries we will discuss later) always point to the same object: In "x is prime and x is even", the x's denote the same object, while in "x is prime and y is even", the objects of x and y may be different. The last kind of term is the function, which allows us to use objects to refer to other objects. For example, the term $x+4$ points to the object we get from adding 4 to x.

We have now defined how we represent objects within predicate logic. If we combine them with predicates, we get atomic propositions: Let P be an arbitrary n-ary predicate (n>0) and let $t_1, t_2, \ldots, t_n$ be terms. $P(t_1, t_2, \ldots, t_n)$ is then an atomic proposition. For example, if P is the binary predicate "is greater than", then $P(7,4)$ means "7 is greater than 4". When we encounter common predicates like this, we will use the regular notation, e.g. $7 > 4$.

From atomic propositions, we can again build formulas:

Definition 3.1.3. A formula is either:

- an atomic proposition
- a compound formula of the form $(A \wedge B)$, $(A \vee B)$, $(A \Rightarrow B)$, $(A \Leftrightarrow B)$ or $(\neg A)$, where A and B are formulas, or
- a quantified formula of the form $\forall x.A$ or $\exists x.A$, where A is a formula and x is a variable.

The formulas are constructed similarly to propositional logic. The new elements are formulas with the universal quantifier $\forall$ (for all) and the existential quantifier $\exists$ (exists). These quantifiers allow us say something about the extent to which predicates are valid: Say we have the non-quantified proposition $x > 3$. This proposition says that whatever object x refers to (we limit ourselves to the natural numbers), this object at least satisfies the predicate > 3. When we add a quantifier, we don't say something about the object x itself, but about the range of objects for which the proposition holds. If we write $\forall x.A$, we say that proposition A holds no matter what object we choose for x: "for all x, A holds". Therefore, if we write $\forall x.(x+1) > 0$, we say that whatever we choose for x, it will be true that $(x+1) > 0$. Similarly, if we write $\exists x.A$, we say that there is at least one object for which the proposition A holds: "there exists an x, such that A holds". For instance, $\exists x.x > 3$ means that there is at least one object x such that $x > 3$.

Quantifiers affect how we regard variables in formulas. They bind to a variable (the one immediatly following the quantifier), to indicate which part of the following proposition varies over the respective range. For example, in $\exists x.x > y$, the $\exists$-quantifier binds to x, such that x may range over one or more objects. The variable

y is not bound: we call it a free variable. When a quantifier binds a variable that has already been used, the newly bound variable holds no relation to the old one. So in $(x > 2) \wedge (\forall x.\exists y.y > x) \wedge (x < 4)$, the first and fourth x are free and refer to the same object. The third one is bound, and is independent of the other x's. The same goes for nested quantifiers: in $\exists x.(x > 3 \wedge \exists x.x < 10)$, the second x is bound to the first quantifier, while the fourth x is bound to the second quantifier.

The rules from propositional logic also apply to predicate logic. We only need to add rules for the quantified formulas. We again have an introduction rule and an elemination rule for each quantifier.

$\forall$-rules

These are the rules for $\forall$-introduction and elimination:

$$\frac{A}{\forall x.A}\ (\forall \text{I}) \qquad \frac{\forall x.A}{A[x := t]}\ (\forall \text{E})$$

The elimination rule ($\forall$E) introduces a new notation: We use $A[x := t]$ to say that in the formula A, all free occurences of x must be substituted with the term t. This definition alone invites a problem: t may contain variables that become bound in A. Take for example the proposition $\exists x.x > y$. We could substitute y with a free variable x, different from the bound x:

$$(\exists x.x > y)[y := x]$$

This results in the proposition $\exists x.x > x$, where suddenly the free x we substituted becomes bound as well. This phenomenon is called variable capture. Since bound variables are dummies used for the quantification, we can replace them with other variables: $\exists x.x > y$ means the same as $\exists z.z > y$. We can therefore avoid variable capture by changing the quantification variables to "fresh" variables if they also occur in t. We can make the above definition more rigorous by accounting for variable capture, but we postpone the effort and assume we always use unique variables, keeping in mind that we can always rename bound variables if necessary.

Now that we know substitution, we can explain the elimination formula: it says that when a proposition A holds for all x, we can replace x with any other term t. For example (we limit ourselves again to the natural numbers):

$$\frac{\forall z.z^2 \geqslant 0}{(x + y)^2 \geqslant 0}$$

If z^2 holds for all z, then we can choose $x + y$ for z, where x and y are fresh variables. Whatever numbers x and y are, the square of their sum is greater than or equal to 0, because the square of every natural number is greater than or equal to 0.

The introduction rule ($\forall$I) states that if we have proved A, we have proved A for all x. Application of this rule calls for some caution, because there is a restriction on the use of x in the derivation of A: x may not appear free in any of the assumptions (that includes A itself, if it is an assumption). This is called the side condition of the rule. Consider what happens if x appears free in one of the assumptions: In that case, the assumption states some property that must hold of x. Although we may not know x, it can only be one of those objects that satisfy the assumption. Therefore we cannot conclude that A holds for all objects. For example, if A would be the assumption $x > 10$, we clearly can't conclude $\forall x.x > 10$.

To illustrate the introduction rule, we can build on the previous example:

$$\dfrac{\dfrac{\dfrac{\forall z.z^2 \geqslant 0}{(x+y)^2 \geqslant 0}}{\forall y.(x+y)^2 \geqslant 0}}{\forall x.\forall y.(x+y)^2 \geqslant 0}$$

This proof shows how we can use the quantified variables of the conclusion free in the derivation. In this case, x and y are names for unknown objects, introduced via the $\forall$-elimination rule. Because we don't make assumptions about x and y, we may conclude that they may range over all objects, such that $\forall x.\forall y.(x+y)^2 \geqslant 0$.

$\exists$-rules

These are the rules for the existential quantifier:

$$\dfrac{A[x := t]}{\exists x.A}\ (\exists\text{I}) \qquad \dfrac{\exists x.A \qquad \begin{array}{c}[A[x := a]] \\ \vdots \\ B\end{array}}{B}\ (\exists\text{E})$$

The introduction rule again mentions substitution. It says that if the proposition A holds whith t substituted for x, then there exists an x such that A holds. A small example:

$$\dfrac{2 < 3}{\exists x.x < 3}$$

We know that $2 < 3$, so we know that there exists an x such that $x < 3$.

The elimination rule says the following: if we know that there exist one or more x such that A holds, and from $A[x := a]$ we can derive B, we may conclude B. The premiss $\exists x.A$ guarantees the existence of an object such that A holds. Because we don't know for which object(s) A holds, we cannot derive anything directly from it. To be able to use A, we can pretend to know the object and give it a fresh name, for example a (for the purposes of this chapter, we'll call a name fresh if it hasn't been used earlier in the proof). We may then assume $A[x := a]$ holds, and use it to prove B. The premiss $\exists x.A$ justifies $A[x := a]$, so we may discharge this assumption, and conclude B just from $\exists x.A$ and its derivability from $A[x := a]$.

However, like the $\forall$-introduction rule ($\forall$I), this rule needs some caution. Besides the requirement of a being fresh, there are two other side-conditions that concern the use of a. The first condition is that a should not appear free in B, which leads to trouble when we discharge the assumption. Consider the following derivation:

$$\dfrac{\exists x.x < 3 \qquad \begin{array}{c}[a < 3] \\ \vdots \\ a < 4\end{array}}{a < 4}\ \text{(wrong!!)}$$

This derivation says that we can prove $a < 4$ merely from the fact that $\exists x.x < 3$. In the subderivation a is not arbitrary: we assumed $a < 3$. However, this assumption is discharged when we apply the rule. a is then an arbitrary object, and it is clearly incorrect to say that an arbitrary object is less than four because there exists some object less than three.

The second side-condition prevents similar problems. This condition states that in deriving B, we may not use a free in any other assumption other than $A[x := a]$.

If we ignore this condition, we are assuming more about a than is justified by $\exists x.A$. We cannot discharge these assumptions, and as a consequence B still depends on assumptions about a. When these assumptions are more strict than $A[x := a]$ (so that the objects that make these assumptions together hold are a subset of those that make $A[x := a]$ hold), we don't rely on $\exists x.A$ at all, because the existence of those objects x is implied by our stricter assumptions. If the assumptions are less strict, we run into similar problems as when we allow a to be free in B:

$$\dfrac{\exists x.x < 3 \qquad \dfrac{\begin{array}{c}[a < 3]\\ a > 1\\ \vdots\\ a = 2\end{array}}{\exists y.y = 2}}{\exists y.y = 2}\ \text{(wrong!!)}$$

Here we have "proved" that we can conclude $\exists y.y = 2$ from the assumptions $\exists x.x < 3$ and $a > 1$, which is clearly wrong: we also need $a < 3$ to conlude $\exists y.y = 2$.

3.1.3 *Constructive logic*

Up to this point, we regarded propositions as statements that can only be either true or false. We embedded this principle in our propositional logic at the end of section 3.1.1 as the rule of the excluded middle: $A \vee \neg A$. To use this rule, we can apply $\vee$-elimination: we prove some other proposition B once by assuming A and once by assuming $\neg A$. If we succeed both proofs, we have proved B. A classical example is a proof that there exist two irrational numbers a and b such that a^b is rational. We start with the LEM,

$$\sqrt{2}^{\sqrt{2}} \text{ is rational or } \sqrt{2}^{\sqrt{2}} \text{ is not rational,}$$

and proceed to prove the hypothesis for both cases:

- Assume $\sqrt{2}^{\sqrt{2}}$ is rational. Let $a = \sqrt{2}$ and $b = \sqrt{2}$. Then $a^b = \sqrt{2}^{\sqrt{2}}$, which we assumed rational, so in this case there are an irrational a and b such that a^b is rational.
- Assume $\sqrt{2}^{\sqrt{2}}$ is irrational. Let $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$. Then $a^b = \sqrt{2}^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2$, which is rational, so in this case too there are an irrational a and b such that a^b is rational.

Since in both cases we have found an a and b such that a^b is rational, we have proved the hypothesis.

This proof does not depend on knowing whether or not $\sqrt{2}^{\sqrt{2}}$ is rational. While the law of the excluded middle says it's one or the other, our interpretation of $A \vee B$ allows us to leave undecided which one it is exactly. We can therefore prove the existence of a, without knowing a.

The ability to prove the existence of some object without being able to pinpoint the exact object is not accepted by all mathematicians. An underlying principle of constructive mathematics is that objects should be constructable, such that a proof of the existence of some object can only be given by presenting a construction of the object itself. This also means that an assertion of existence is accompanied by such

a construction, and that we can use this construction in a further line of proof. In a similar vein, proofs themselves form constructions, as shown by what is now called the BHK-interpretation of Brouwer, Heyting and Kolmogorov:

- A proof of $A \wedge B$ consists of a proof of A and a proof of B.
- A proof of $A \vee B$ consists of a proof of A or a proof of B, together with an indication of which proof it is.
- A proof of $A \Rightarrow B$ is a method to turn a proof of A into a proof of B.
- A proof of $\exists x.A$ consists of the construction of a a and a proof of $A[x := a]$.
- A proof of $\forall x.A$ is a method to turn a construction of a into a proof of $A[x := a]$.
- A proof of $\neg A$ is a proof of $A \Rightarrow \bot$.
- A proof of $\bot$ does not exist.

When we assume $A \wedge B$, we may conclude either A or B. In a constructive setting this means we need proof of both A and B to conclude $A \wedge B$. In classical logic a proof of $\neg(\neg A \vee \neg B)$ can be used to prove $A \wedge B$, but since this provides no proofs of A and B, it is not a generally valid method in constructive logic. Similarly, $A \vee B$ requires a direct proof of either A or B, which is not implied by the classically equivalent $\neg(\neg A \wedge \neg B)$.

A constructive proof of implication should show that any proof of the first proposition in some way also justifies the second: it is a method to transform the proof of one into a proof of the other. The proof of a universally quantified proposition is likewise a method, but now from objects to proofs. A proof of a proposition is built from facts implied by the construction of an object; a universal proof is a method to provide such a proof for any object.

As the constructive interpretation rejects indirect proofs, it also rejects the law of excluded middle and its siblings. The assertion $A \vee \neg A$ requires either a proof of A or a proof of $\neg A$. As long as there is no proof of either (i.e. a problem is unsolved), $A \vee \neg A$ cannot be asserted. Similarly $\neg\neg A$ only asserts that $\neg A$ leads to a contradiction. It does not provide a direct proof of A, so the law of double negation also cannot hold in general.[1]

3.2 LAMBDA CALCULUS

In the previous section we looked at logic, and arrived at a version with a computational nature. In this section we will start with a computational formalism, and work towards a union of logic and computation. The computational formalism is the untyped lambda calculus, which was actually part of a formal logic proposed by Alonso Church in the 1930s. This logic was proved inconsistent by two of his students, after which Church excised the calculus and combined it with ideas from the theory of types, resulting in the simply typed lambda calculus. Starting in the late 1950s, the lambda calculus started influencing the development of programming languages, and eventually lead to languages like ML, and later Haskell. However, the relation to logic was not entirely lost. In the 1960s, De Bruijn and Howard independently posited interpretations of terms as proofs and types as propositions. This resulted in proof checking systems like for example AUTOMATH. The mathematical and programming

[1] There are cases where the law of excluded middle can be used, but it goes too far to discuss that here.

communities influenced each other, and many different calculi and type systems evolved. [CH09, Bar93]

In this chapter, we will start with the untyped lambda calculus, and expand it first to the simply typed lambda calculus, and then to a system with dependent types, while also highlighting the logical interpretation.

3.2.1 *Untyped lambda calculus*

The lambda calculus is a function oriented calculus. Its only objects are functions, and at its heart it is concerned only with the construction and use of these objects. A function can be constructed with abstraction: in a common mathematical context we may see $f(x) = x+2$ as a generalisation of for instance the operation $5+2$, where we abstracted from the particular number we added to 2. When we have such a function, we can use it by applying it to some argument: f applied to 3, written $f(3)$, is to mean the particular operation $3+2$. In the pure lambda calculus, application and abstraction are the key ingredients. In fact, together with variables, they are the sole ingredients, as we can see in its syntax:

Definition 3.2.1. The syntax of the pure lambda calculus is:

$$t \coloneqq v \mid \lambda v\,.\,t \mid t\,t$$

The category v represents the variables; we will use letters near the end of the alphabet for them (x, y, z an the like). The abstraction of a term e over variable x is written as $\lambda x\,.\,e$. The application of a term f to a term e is written as $f\,e$.

To keep the notation concise yet unambiguous, we will adhere to the following rules: applications associate to the left, and parentheses are used when necessary to delimit lambda-abstractions and applications. Thus $x\,y\,c$ will mean $(x\,y)\,z$ (the application of $x\,y$ to z), which is different from $x\,(y\,z)$ (the application of x to $y\,z$), and $\lambda x\,.\,x\,y$ is an abstraction of $x\,y$, while $(\lambda x\,.\,y)z$ is the term $\lambda x\,.\,y$ applied to z.

A lambda-abstraction indicates a parameter of a term. In $\lambda x\,.\,t$, x is a parameter of the term t, and we say that it is *bound*. If a variable occurs in t not bound, then it occurs *free*. If a term contains free variables we call it *open*, and if it contains no free variables we call it *closed*. So $\lambda y\,.\,x\,y$ is an open term, because x is free, whereas $\lambda x\,.\,\lambda y\,.\,x\,y$ is closed because both x and y are bound. Because a bound variable is a formal parameter, we can change its name without altering the meaning of the term: the function $\lambda x\,.\,x$ will do the exact same thing as $\lambda y\,.\,y$. We will therefore regard such terms as equal.

As we mentioned above, when we write $f(3)$, the application of f to 3, we actually intend to substitute 3 for the parameter in the definition of f. When we have defined $f(x) = x+2$, $f(3)$ simply means $3+2$. We can say that $f(3)$ *reduces* to $3+2$. Of course, $+$ itself is also a function, and we could substitute 3 and 2 for the respective parameters of whatever the definition of $+$ is. In the end, we would expect $3+2$ to reduce to 5. This process of reduction is reflected in the lambda calculus by the so-called β-reduction rule:

Definition 3.2.2. β-reduction:

$$(\lambda y\,.\,t_1)\;t_2 \to_\beta t_1[y \coloneqq t_2]$$

It says that when we encounter a lambda-abstraction $\lambda y\,.\,t_1$ applied to t_2, we may substitute t_2 for the parameter y in t_1, such that $(\lambda y\,.\,t_1)\,t_2$ becomes $t_1[y \coloneqq t_2]$. The

symbol $\rightarrow_\beta$ can be read as "reduces to". For example,

$$(\lambda x\,.\,x)\,y \rightarrow_\beta y$$

says that the identity function applied to y reduces to y. We will call a term of the form $(\lambda y\,.\,t_1)\,t_2$ a reducible expression (redex), and the reduced form $t_1[y := t_2]$ the reduct.

The lambda calculus does not allow functions with multiple parameters. However, it is possible to represent such functions by means of currying: we can abstract a term repeatedly to add the parameters, such that it becomes a function of one parameter which returns another function of the next parameter, which returns another function et cetera. For example, we may view the function $\lambda x\,.\,\lambda y\,.\,x$ as having two parameters, while it is in fact a function with parameter x that returns another function with parameter y:

$$(\lambda x\,.\,\lambda y\,.x)\,u \rightarrow_\beta \lambda y\,.\,u$$

We can make use of repeated applications to supply all arguments:

$$(\lambda x\,.\,\lambda y\,.\,x)\,u\,v \rightarrow_\beta (\lambda y\,.\,u)\,v \rightarrow_\beta u$$

In the following, we will use $\twoheadrightarrow$ to denote multiple reductions, such that we can abbreviate the above sequence as

$$(\lambda x\,.\,\lambda y\,.\,x)\,u\,v \twoheadrightarrow u$$

We will use the same symbol to denote reductions that occur within an expression:

$$\lambda x\,.\,(\lambda y\,.\,z)\,x \twoheadrightarrow \lambda x\,.\,z$$

When a term cannot be reduced any further (it contains no redexes), it is in *normal form*. Every term has at most one normal form: normal forms are unique. Furthermore, if a term has a normal form, it can be reached by repeated reduction [Bar93]. However, not every term has a normal form. A canonical example is the term $(\lambda x\,.\,x\,x)\,(\lambda x\,.\,x\,x)$, which reduces to itself.

As we reduce a term, its meaning as a whole does not change. Therefore, if two terms have the same normal form, they have the same meaning. This leads to a notion of β-equality (sometimes called convertability):

Definition 3.2.3. β-equality:
If two terms t_1 and t_2 have the same normal form, i.e. there exists a non-reducible term t such that $t_1 \twoheadrightarrow t$ and $t_2 \twoheadrightarrow t$, then we call them β-equal:

$$t_1 =_\beta t_2$$

For example, we have $(\lambda x\,.\,x)\,y =_\beta (\lambda x\,.\,y)\,z$. Note that terms that do not have a normal form are also not equivalent to any other term.

The lambda-calculus so far may seem limited, but it is nevertheless turing complete. We will give two basic examples of programming in the untyped lambda calculus, involving booleans and natural numbers. To represent the boolean values "true" and "false", we can use the following definitions:

$$\mathit{true} \equiv \lambda x\,.\,\lambda y\,.\,x$$
$$\mathit{false} \equiv \lambda x\,.\,\lambda y\,.\,y$$

We can think of them as functions of two parameters, where *true* always returns the first, and *false* always returns the second parameter. We can define some of the usual boolean functions based on this selective behaviour as follows:

$$\begin{aligned} \mathit{not} &\equiv \lambda x.\, x\ \mathit{false}\ \mathit{true} \\ \mathit{and} &\equiv \lambda x.\, \lambda y.\, x\ y\ \mathit{false} \\ \mathit{or} &\equiv \lambda x.\, \lambda y.\, x\ \mathit{true}\ y \end{aligned}$$

If we apply *not* to *true* and *false*, we get the following reduction sequences:

$$\begin{aligned} &\mathit{not}\ \mathit{true} \to_\beta \mathit{true}\ \mathit{false}\ \mathit{true} \to_\beta (\lambda y.\, \mathit{false})\ \mathit{true} \to_\beta \mathit{false} \\ &\mathit{not}\ \mathit{false} \to_\beta \mathit{false}\ \mathit{false}\ \mathit{true} \to_\beta (\lambda y.y)\ \mathit{true} \to_\beta \mathit{true} \end{aligned}$$

The other functions work in a similar way.

Natural numbers can be encoded as so-called Church numerals:

$$\begin{aligned} 0 &\equiv \lambda f.\, \lambda n.\, n \\ 1 &\equiv \lambda f.\, \lambda n.\, f\ n \\ 2 &\equiv \lambda f.\, \lambda n.\, f\ (f\ n) \\ 3 &\equiv \lambda f.\, \lambda n.\, f\ (f\ (f\ n)) \\ &\vdots \end{aligned}$$

So a number x is represented by a function that makes a composition of x applications of the first argument f and applies that composition to the second argument n. If we view f as a "successor" and n as "zero", we can see a resemblance to the Peano representation of the natural numbers.

An actual successor function for Church numbers would be

$$\mathit{succ} \equiv \lambda x.\, \lambda f.\, \lambda n.\, f\ (x\ f\ n)$$

Given a number x, it returns another function with arguments f and n. It applies x to f and n to get the x-fold composition of f applied to n, and then f is applied to the result to get the actual successor. For example:

$$\mathit{succ}\ 0 \to_\beta \lambda f.\, \lambda n.\, f\ (0\ f\ n) \to_\beta \lambda f.\, \lambda n.\, f\ ((\lambda n.\, n)\ n) \to_\beta \lambda f.\, \lambda n.\, f\ n$$

Addition can be done in a similar manner with

$$\mathit{add} \equiv \lambda x.\, \lambda y.\, \lambda f.\, \lambda n.\, x\ f\ (y\ f\ n)$$

This function first applies the y-fold composition of f to n, and then, instead of just a single application of f, it applies the x-fold composition of f to the result, leading to the $(y+x)$-fold composition of f applied to n.

Our last demonstration of the untyped lambda calculus is the powerful concept of recursion: functions defined in terms of themselves. Famous examples are functions that calculate factorials and Fibonacci numbers:

$$\begin{aligned} \mathit{fac}\ 0 &\equiv 1 \\ \mathit{fac}\ n &\equiv n \cdot \mathit{fac}\ (n-1) \end{aligned} \qquad \begin{aligned} \mathit{fib}\ 0 &\equiv 0 \\ \mathit{fib}\ 1 &\equiv 1 \\ \mathit{fib}\ n &\equiv \mathit{fib}\ (n-2) + \mathit{fib}\ (n-1) \end{aligned}$$

In the pure lambda calculus we can't use the term we are defining within itself, but we are not prohibited to apply a term to itself. We have already seen the term

$$(\lambda x.\, x\, x)\ \ (\lambda x.\, x\, x)$$

which reduces to itself. A similar term from the mind of Alan Turing is

$$Y \equiv (\lambda y.\, \lambda f.\, f\, (y\, y f))\ (\lambda y.\, \lambda f.\, f\, (y\, y f))$$

This term is a bit more complicated, but it can be quickly checked that for every function f we have

$$Y\ f \twoheadrightarrow f\ (Y\ f) \twoheadrightarrow f\ (f\ (Y\ f)) \twoheadrightarrow \cdots$$

Now suppose that f has the form $\lambda g.\, \lambda x.\, (\cdots g \cdots)$, such that

$$Y\ f \twoheadrightarrow f\ (Y\ f) \twoheadrightarrow \lambda x.\, (\cdots (Y\ f) \cdots)$$

The term $Y\ f$ now reduces to a function that contains itself: it is a recursive function. As a more concrete example, here is a definition of the factorial function:

$$\begin{aligned} \mathit{rfac}' &\equiv \lambda f.\lambda n.\mathit{if}\ (n = 0)\ \mathit{then}\ 1\ \mathit{else}\ n \cdot f\ (n-1) \\ \mathit{rfac} &\equiv Y\ \mathit{rfac}' \end{aligned}$$

We sugar-coated it a bit with some arithmetic operators and a test for zeroness, but it is nevertheless expressible in the pure lambda calculus. The Y-function effectively replaces f in *rfac'* with *rfac*, such that *rfac* becomes the factoriol function as we know it.

3.2.2 *Simply typed lambda calculus*

The untyped lambda calculus is powerful, but it has some peculiarities. We have seen in the previous section that it is capable of representing booleans and natural numbers, but it is also very well possible to write terms like *succ add false* , which have little meaning to us. We perceive booleans, natural numbers, and the functions acting on them as different concepts, which are not necessarily interchangeable.

To fix this peculiarity we can divide terms into types, and introduce rules that limit how we can use terms of different types. The language of types, like that of the lambda calculus, is simple:

Definition 3.2.4. The syntax of simple types is:

$$T \coloneqq B \mid T \to T$$

The category B of base types contains "atomic types", which are not composed of other types. Examples are the type of booleans ($\mathbb{B}$) and the type of natural numbers ($\mathbb{N}$). Functions, in contrast, do not belong to a base type. They have a domain and range of certain types, which we put together with the *type constructor* $\to$ to form the function type. If a function takes a parameter of type α, and has a result of type β, then the type of the function is $\alpha \to \beta$. When a function accepts or returns another function, we use parentheses: $(\alpha \to \beta) \to \gamma$ takes a function of type $\alpha \to \beta$ to a result of type γ, whereas $\alpha \to (\beta \to \gamma)$ takes something of type α and delivers a function of type $\beta \to \gamma$. To enhance readability we will often omit parentheses with the convention that $\to$ associates to the right, in other words $\alpha \to \beta \to \gamma$ will mean $\alpha \to (\beta \to \gamma)$.

Every allowable term will have a type in the prescribed form. We can say that a term has some type if we can derive the type from (the types of) its subterms with the typing rules we will discuss. Because variables do not have subterms they appear at the top of the derivation tree. We can't derive the types of variables, so we have to assume their types. As a consequence, a term only has a type for a certain set of type assumptions. We call this set the *context*. To express that a term t has a type τ, we will write

$$t : \tau$$

and to say that term t has type τ in context Γ, we write

$$\Gamma \vdash t : \tau$$

During the typing derivation we will want to alter the context, so we can for example create a different context by adding a new assumption:

$$\Gamma;\ x : \sigma$$

forms a new context that consists of all assumptions from Γ and the new assumption $x : \sigma$. For a context to be valid, we require that each assumption consists of a variable and a type with proper syntax (a well formed type).

Now we come to the typing rules. We borrow the notation of natural deduction:

Definition 3.2.5. The typing rules *var*, *lam*, and *app* are:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma; x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x{:}\sigma.e\ :\ \sigma \to \tau} \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}$$

The rule for variables (var) expresses that if $x : \tau$ is an assumption in context Γ, we may conclude that given context Γ, x has type τ. The lambda abstraction rule (lam) tells us that in order to build a lambda abstraction $\lambda x : \sigma.e$ of type $\sigma \to \tau$, we need to have a term e of type τ under at least the assumption $x : \sigma$. By abstracting over x we turn it into a parameter of type σ, so we can remove $x : \sigma$ from the context. Often it is clear what the type of x should be; in such cases we will omit the type and write $\lambda x.e$. Finally, the application rule (app) states that the result of an application $e_1\ e_2$ has type τ if e_1 is a function with a domain of type σ and a range of type τ, and the type of argument e_2 matches the domain type σ of e_1, all under context Γ. Note that these typing rules only hold under the assumption that the contexts are valid (i.e. all elements of the contexts are variables with well formed types, as discussed above).

The following derivation example contains all three rules:

$$\dfrac{\dfrac{\dfrac{f : \sigma \to \tau \in \Gamma}{\Gamma \vdash f : \sigma \to \tau}\ \text{(var)} \qquad \dfrac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \text{(var)}}{\Gamma \vdash f\,x : \tau}\ \text{(app)}}{x : \sigma \vdash \lambda f.f\,x : (\sigma \to \tau) \to \tau}\ \text{(lam)}$$

where we used

$$\Gamma \equiv x : \sigma;\ f : \sigma \to \tau$$

to keep the notation clean. This derivation leads us to the type of a function which takes another function as argument and applies it to some variable from the context. At the top of the tree are the variable assumptions: $f : \sigma \to \tau$ and $x : \sigma$ occur in the context. Because the type of x matches the domain type of f, we can apply f to x

with the application rule, and subsequently abstract over f with the lambda rule, where f disappears from the context as it becomes bound.

The simply typed lambda calculus so far does not have any meaningful base types, so we will extend it. We will add the type $\mathbb{N}$ for natural numbers. Because the definitions we gave in the previous section of the values of these types are not really intuitive, we will also add a couple of new terms that seem to make more sense, namely Z for zero and S for the successor function. To make sure that these terms have the proper types, we introduce some new typing rules:

$$\frac{}{\Gamma \vdash Z : \mathbb{N}} \qquad \frac{\Gamma \vdash k : \mathbb{N}}{\Gamma \vdash S\,k : \mathbb{N}}$$

We may always conclude that zero is a natural number, and if k is a natural number, we may conclude its successor is one as well. Besides creating natural numbers, we also want to use them. For this we introduce the term *natElim*, which will allow us to perform recursion over the natural numbers. Its typing rule is

$$\frac{\Gamma \vdash e_0 : \tau \qquad \Gamma \vdash f : \mathbb{N} \to \tau \to \tau \qquad \Gamma \vdash k : \mathbb{N}}{\Gamma \vdash \mathit{natElim}\; e_0\; f\; k : \tau}$$

The term e_0 is the base case of the recursion, and f is the step function. The behaviour of *natElim* is perhaps best explained with its reduction rules:

$$\begin{aligned} &\mathit{natElim}\; e_0\; f\; Z \to e_0 \\ &\mathit{natElim}\; e_0\; f\; (S\,k) \to f\; k\; (\mathit{natElim}\; e_0\; f\; k) \end{aligned}$$

When the last argument is Z, we are in the base case and return e_0. When it is a successor of k, we return f applied to both k and the result of the previous recursion step. As an example, we can implement the factorial function (we assume that we have a multiplication on natural numbers):

$$\mathit{fac} \equiv \lambda n.\, \mathit{natElim}\; (S\,Z)\; (\lambda k.\, \lambda r.\, (S\,k) \cdot r)\; n$$

When n is Z, this reduces to $S\,Z$. When n is $S\,k$ we get

$$(\lambda k.\, \lambda r.\, (S\,k) \cdot r)\; k\; (\mathit{fac}\; k)$$

which reduces to

$$(S\,k) \cdot (\mathit{fac}\; k)$$

as we would expect from the factorial function.

3.2.3 *Propositions as types*

We started this chapter with a discussion of propositional logic. The language of this logic allowed us to build propositions using connectives like conjunction, disjunction, and implication. We explained that the constructive view requires that each proposition that we hold true is accompanied by a proof, and that such proofs have a computational nature. In the current section, we have just introduced the lambda calculus as a computational language, where we used types to restrict valid terms. An interesting observation, most famously made by Haskell Curry and William Howard, is that propositions and types behave similarly. In fact, when we regard propositions as types, we see that our deduction rules become typing rules, that proof

trees become typing derivations, and that proofs become terms. In this section we will explore this dual view.

The simply typed lambda calculus cannot yet accomodate all connectives we have described for the propositional calculus (§3.1.1), so we will have to extend it. This is partly a repetition of section 3.1.1, because we will have to describe what are valid types (formulas) and what are the typing rules (introduction and elimination rules). The new component is that we also describe new terms, which we will use to construct the proofs of the propositions, and new reduction rules that come with such terms, which describe how can we convert one proof of a proposition to another. We will trot deeper into these concepts as we go along, but first we will look at the single connective the calculus already supports: implication.

A lambda-abstraction represents a method to turn terms of one type into terms of another. This conforms exactly to the BHK-interpretation of a proof of implication, which is a method to turn the proof of one thing into a proof of another. When we read the typing rules of lambda abstraction and application as deduction rules, we see that they resemble the introduction and elimination rules. Compare for example the introduction and lambda abstraction:

$$\dfrac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \Rightarrow B}\ (\Rightarrow\text{I}) \qquad \dfrac{\Gamma; x:\sigma \vdash e:\tau}{\Gamma \vdash \lambda x.e : \sigma \to \tau}\ (\text{lam})$$

To introduce an implication $A \Rightarrow B$, we need to show that we can derive proposition B from the assumption A. To introduce a term of type $\sigma \to \tau$, we need to show that we have a term of type τ under the assumption that we have a term of type σ. The structures are equal. The main difference between the rules is that the typing rules also tell you what a proof looks like: the proof of $\sigma \to \tau$ is a lambda-abstraction, a method, which takes one proof to another.

When we look at elimination and application, we see a similar congruence: To eliminate an implication $A \Rightarrow B$, we need a proof of A, and we conclude that we have a proof of B. When we have a term of type $\sigma \to \tau$ and a term of type σ, the application of the first to the second gives us a term of type τ:

$$\dfrac{A \qquad A \Rightarrow B}{B}\ (\Rightarrow\text{E}) \qquad \dfrac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau}\ (\text{app})$$

Considering the similarities we saw, it is not a giant step to read $\to$ as the implicational connective.

Aside from the types, terms and typing rules, the lambda calculus also gives us a reduction rule:

$$(\lambda x.t_1)\, t_2 \to_\beta t_1[x := t_2]$$

This rule is a transformation between proofs: it preserves type, but changes the proof. Notice that the term $(\lambda x.t_1)\, t_2$ is a lambda-abstraction followed by an application, which means that as a proof, it introduces an implication which is subsequently eliminated, forming a sort of detour. The reduction rule translates this detoured proof into a more direct one.

Now let's introduce conjunctions in the calculus. Conjunctions are of the form $(A \wedge B)$. Our simply typed lamda calculus has no conforming type, so we will simply add $(T \wedge T)$ to definition 3.2.4. For the terms of this type we look at the constructive interpretation of a proof of $(A \wedge B)$, which says such a proof should consist of a pair of proofs, namely a proof of A and a proof of B. We can reflect this in the calculus

by adding terms of the form (t_1, t_2), which then have the type $(\sigma \wedge \tau)$. Because we can only build a proof of $(A \wedge B)$ when we have proofs of both A and B, we can only construct a term of type $(\sigma \wedge \tau)$ if we have a term of type σ and a term of type τ:

$$\frac{\Gamma \vdash t_1 : \sigma \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash (t_1, t_2) : (\sigma \wedge \tau)}$$

This rule corresponds to the $\wedge$-introduction rule. We also need the corresponding elimination rules, which say that we may extract the subproofs to prove the subpropositions: from $(A \wedge B)$ we can prove A and B individually. Since a proof of $(A \wedge B)$ is a pair of subproofs, we can extract the right subproof if we want to conclude A or B. In the lambda calculus this means we have to add two new terms $\pi_1\, t$ and $\pi_2\, t$, which, given a term $(t_1, t_2) : (\sigma \wedge \tau)$, extract the corresponding subterms t_1 and t_2. This is reflected in the reduction rules we add:

$$\pi_1\ (t_1, t_2) \to t_1$$
$$\pi_2\ (t_1, t_2) \to t_2$$

The typing rules for π_1 and π_2 are then

$$\frac{\Gamma \vdash t : (\sigma \wedge \tau)}{\Gamma \vdash \pi_1\ t : \sigma} \qquad \frac{\Gamma \vdash t : (\sigma \wedge \tau)}{\Gamma \vdash \pi_2\ t : \tau}$$

As an example, let's look at the following derivation:

$$\Gamma \equiv x : \sigma \wedge \tau;\ \ f : \sigma \to \rho$$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash f : \sigma \to \rho \qquad \dfrac{\Gamma \vdash x : \sigma \wedge \tau}{\Gamma \vdash \pi_1\ x : \sigma}}{\Gamma \vdash f(\pi_1\ x) : \rho}}{x : \sigma \wedge \tau \vdash \lambda f.f(\pi_1\ x) : (\sigma \to \rho) \to \rho}}{\vdash \lambda x.\lambda f.f(\pi_1\ x) : (\sigma \wedge \tau) \to (\sigma \to \rho) \to \rho}$$

Here we have a type derivation of a function which takes a pair x and some function f, and returns f applied to the first element of x. This is similar to having a proof that $\sigma \to \rho$ implies ρ if $\sigma \wedge \tau$ hold. This proof then constitutes the method of transforming the proof of σ, that is included in the proof x of $\sigma \wedge \tau$, to the proof of ρ, using π_1 and the method f that proves $\sigma \to \rho$.

Besides conjuntion we also have disjunction: a proposition $(A \vee B)$, which is proved by either a proof of A or a proof of B, plus an indication of which proof it is. In functional languages, we may view disjunctions as disjoint unions of datatypes, as for instance the *Either* type in Haskell:

$$\mathit{data\ Either\ a\ b} = \mathit{Left\ a} \mid \mathit{Right\ b}$$

This type represents the disjoint union of the types a and b: the contained values may be of either type, while the constructors *Left* and *Right* indicate which of the two it is.

We will include the disjunction in the extended lambda calculus as types of the form $\sigma \vee \tau$. To introduce terms of such a type we add two new forms: $in_1\, t$ and $in_2\, t$. If we regard them both of type $\sigma \vee \tau$, then the first form indicates that t is of type σ, while the second indicates that t is of type τ:

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash in_1\, t : \sigma \vee \tau} \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash in_2\, t : \sigma \vee \tau}$$

To eliminate a disjunction, we need to show we can prove another proposition from either disjunct. We had the following elimination rule:

$$\frac{A \vee B \qquad \begin{array}{c}[A]\\ \vdots \\ C\end{array} \qquad \begin{array}{c}[B]\\ \vdots \\ C\end{array}}{C}\ (\vee\mathrm{E})$$

We can express the fact that C is derivable from A (or B) as $A \Rightarrow C$ (or $B \Rightarrow C$), so the above rule is equivalent to

$$\frac{A \vee B \qquad A \Rightarrow C \qquad B \Rightarrow C}{C}$$

We can now easily translate the premisses to typing judgements, but we don't yet know exactly what a proof of C looks like. It depends on the proof of $A \vee B$: If we have $in_1\, t$, then C is proved by applying the proof of $A \Rightarrow C$ (the second premiss) to t, which is the proof of A. If $A \vee B$ is proved by $in_2\, t$, then C is proved by applying the proof of $B \Rightarrow C$ (the third premiss) to t (which is now a proof of B). Because we don't know on beforehand which case we are in, we add another term to express the options: $case\ t\ f\ g$. Here, t is our disjunction, f is the proof of $A \Rightarrow C$, and g is the proof of $B \Rightarrow C$. If t has the form $in_1\ t_1$ it will reduce to $f\ t_1$, and if it has the form $in_2\ t_2$ it will reduce to $g\ t_2$:

$$case\ (in_1\ t_1)\ f\ g \to f\ t_1$$
$$case\ (in_2\ t_2)\ f\ g \to g\ t_2$$

With this new term, we can express the elimination rule as a typing rule:

$$\frac{\Gamma \vdash t : \sigma \vee \tau \qquad \Gamma \vdash f : \sigma \to \rho \qquad \Gamma \vdash g : \tau \to \rho}{\Gamma \vdash case\ t\ f\ g : \rho}$$

The last connective we will discuss is $\bot$. This is the absurd proposition, which has no proof, and therefore no introduction rule. In the lambda calculus, this means that there is no value of type $\bot$. If any computation has type $\bot$, it can therefore never return a value: either it never halts, or it is inherently erroneous. However, we did have an elimination rule, which said that by proving $\bot$, we could prove anything. As a typing rule, this means that if we do have a term of type $\bot$, we can introduce a term of any type. A term that can have any type voids any computation that depends on it of its meaning, and we might as well abort the computation. This suggests an interpretation of the "term-of-any-type" as a primitive to raise errors, and we may use for example the following typing rule:

$$\frac{\Gamma \vdash t : \bot}{\Gamma \vdash abort\ t : \tau}$$

3.2.4 *Type polymorphism*

In the simply typed lambda calculus arguments and results of functions always have a fixed type. Even if the same term can be used to implement some functionality for different types, we still need to define separate functions. For example, the identity function for booleans and the identity function for natural numbers have to be distinct functions.

With some additions to the simply typed lambda calculus, we can introduce type polymorphism. This kind of polymorphism allows types to contain type variables, similar to types in Haskell. An identity function *id* in Haskell may have the type $a \to a$, where a is a variable. When we use *id* on a boolean or a natural number, a will be substituted with *Bool* or *Nat*, respectively. Under Haskell's hood, a function of type $a \to a$ is actually quantified, to signify that we can replace a with any type: $\forall a\,.\,a \to a$. When the function is applied, the unification algorithm will try to find a suitable choice for a. Adding type variables and universal quantification over those variables to the simply typed lambda calculus, gives us the following type syntax:

$$T \coloneqq B \mid V \mid T \to T \mid \forall V.T$$

The new elements are V, the type variables, and $\forall V.T$, the quantification over type variables. We will use lower case roman letters to represent individual type variables.

Unlike Haskell, we will make explicit what we subtitute the type variables with. We do this by adding function abstraction over types, such that we can use types as function arguments. We'll use Λ for type abstraction:

$$t \coloneqq b \mid v \mid t\,t \mid \lambda v{:}t\,.\,t \mid \Lambda V.\,t$$

As an example, we can now express an identity function for any type:

$$id \equiv \Lambda a\,.\,\lambda x\,.\,x : \forall a\,.\,a \to a$$

To use this function, we have to apply it first to a type. The abstracted type variable then gets substituted with that particular type:

$$id\ \mathbb{N} : \mathbb{N} \to \mathbb{N}$$

We can then apply the function to a term:

$$id\ \mathbb{N}\ (S\,Z) : \mathbb{N}$$

This behaviour is captured in the typing rules as follows:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Lambda a\,.\,e : \forall a\,.\,\tau}\ (a \notin FV(\Gamma)) \qquad \frac{\Gamma \vdash e : \forall a\,.\,\tau}{\Gamma \vdash e\,\sigma : \tau[a \coloneqq \sigma]}\ (\sigma \text{ is a type})$$

The first rule introduces type abstraction, which has as side condition that the type variable a is not already present as a free variable in the context. Whenever we have a term e of type τ, we can add a type abstraction. The second rule is the application rule for types. We may only apply a function e to a type σ if e is a type abstraction. According to the first rule, this is the case when it has type $\forall a.\tau$. The argument σ of course has to be a type. To continue the example of the identity function, we can derive its type as follows:

$$\dfrac{\dfrac{x : a \vdash x : a}{\vdash \lambda x.x : a \to a}}{\vdash \Lambda a.\lambda x.x : \forall a.a \to a}$$

When we apply it to a natural number, we get

$$\dfrac{\dfrac{\Gamma \vdash id : \forall a.a \to a}{\Gamma \vdash id\ \mathbb{N} : \mathbb{N} \to \mathbb{N}} \qquad \Gamma \vdash S\,Z : \mathbb{N}}{\Gamma \vdash id\ \mathbb{N}\,(S\,Z) : \mathbb{N}}$$

3.2.5 *Dependent types*

Although type polymorphism is very useful, we want something more expressive. Type polymorphism allows us to define types that quantify over other types, like for example the Haskell list type. In hardware (yes, we were concerned with hardware) we would like to specify lists of a certain length, such that we can regard binary words as lists of bits with a given length. Lengths are most naturally expressed with terms, not types, such that we can for example specify arithmetic relations between lengths of vectors (e.g. in vector concatenation, where vector lengths add up). We would like types to depend on term parameters: we would like *dependent types*.

For now, let's assume we have a type constructor *Vect* (vector), such that we can express the type of a vector containing n elements of type a with *Vect a n*. Different choices of n and a give us different types: *Vect* $\mathbb{N}\,3$ and *Vect* $\mathbb{N}\,4$ are in that regard not less different than say $\mathbb{N}$ and $\mathbb{B}$. (We will discuss the specific rules for these vectors later.) In the previous section we already quantified over types, so we can create functions that operate on vectors with arbitrary element types, but with given lengths. For example:

$$\Lambda a.\lambda x.x : \forall a.Vect\ a\ 3 \to Vect\ a\ 3$$

When we apply a function with a quantified type $\forall a.\tau$ to a type σ, the result has type $\tau[x := \sigma]$. Therefore, if we apply for example the function above to the type $\mathbb{N}$, we get the identity function from *Vect* $\mathbb{N}$ *3* to *Vect* $\mathbb{N}$ *3*. It is not a particularly contrived idea to generalise the quantification mechanism towards term variables, such that we are allowed to write

$$vId \equiv \Lambda a.\lambda n.\lambda x.x : \forall a.\forall n{:}\mathbb{N}.Vect\ a\ n \to Vect\ a\ n$$

Just as with type polymorphism, n gets replaced in the type when we apply the function:

$$vId\ \mathbb{N}\,3 : Vect\ \mathbb{N}\,3 \to Vect\ \mathbb{N}\,3$$

Note that nothing precludes us from using n in the body of the lambda-expression (as long as the body matches the return type). We could very well have

$$\Lambda a.\lambda n.\lambda x.n : \forall a.\forall n{:}\mathbb{N}.Vect\ a\ n \to \mathbb{N}$$

Quantification over terms is nice, but it does not yet give us the flexibility we will require of dependent types. One problem is that we don't have polymorphism for dependent types. Take for example a function of type

$$\forall n{:}\mathbb{N}.\,t$$

The type expression t may depend on n. As we have seen in the previous section, if we want the function to be polymorphic in its return type, we can give it the type

$$\forall a.\forall n{:}\mathbb{N}.\,a$$

However, the type we substitute for a can never depend on n, because it would have to depend on n outside the quantification, and therefore outside the scope of n. We lose the dependency of the return type on n.

Like *Vect a n*, the type expression t will represent different types for different choices of n. It behaves pretty much like a function, mapping terms to types. However, as n represents only a single number in t (though it could by any number), t represents only a single type. Imagine we can extract the function embedded in t; let's call it t'. We can then write t as the application of t' to n, or $t'\ n$, and if we replace t' with a variable a, we can quantify over it:

$$\forall a . \forall n{:}\mathbb{N} . a\ n$$

The variable a now ranges over functions from terms to types, giving us a polymorphic dependent type. Of course, this begs the question of how we construct such functions (we can't, yet), and how we know the above expression is legal (we don't, yet). However, the above generalisation introduces a key element of dependent types. When we added polymorphism to the simply typed lambda calculus, we went from *term-to-term* functions to *type-to-term* functions. With dependent types, it is not so much important that we have some handy built-in dependent types like vectors; it is that we are able to construct *term-to-type functions*.[2]

Before we go on into the details of dependent types, let's look at an example of what it means to have functions that take terms to types. Imagine a type constructor *BN* with the following behaviour:

$$BN\ n \equiv \begin{cases} \mathbb{B} & \text{if } n \text{ is } Z \\ \mathbb{N} & \text{if } n \text{ is } (S\ k) \end{cases}$$

Note that for every n, *BN n* is a type, just like *Vect a n* would be a type. However, where we regard all types of the form *Vect a n* to represent vectors, the types of the form *BN n* represent truly different types: terms of type *BN Z* are booleans, and terms of type *BN (S k)* are natural numbers. If we disregard the typing rules for a moment, we could actually define *BN* with recursion over the natural numbers:

$$BN \equiv \lambda n . natElim\ \mathbb{B}\ (\lambda k . \lambda t . \mathbb{N})\ n$$

With *BN* we can express types of functions that return either booleans or natural numbers, depending on their arguments:

$$\forall n{:}\mathbb{N} . BN\ n$$

A function of this type would return a boolean if its argument if Z, and a natural number otherwise.

As is visible in the example above, terms and types become quite mixed up in the context of dependent types. Functions can give and take terms and types, variables can range over terms and types. It becomes cumbersome to keep the two separate, and therefore we will join the syntax:

$$t \coloneqq c \mid v \mid t\ t \mid \lambda x{:}t . t \mid \forall x{:}t . t$$

[2] The programming-oriented reader might wonder if the dependency of types on terms implies dynamic typing. We will come to that when we have discussed the rules.

The category c defines constants like $\mathbb{N}$ and $\mathbb{B}$ together with their related terms (so we will for example extend this category with vector notation), and the category v contains the variables (we use the same set for both terms and types).

Instead of using two different syntaxes to distinguish terms and types, we will use the typing relation: just like terms have types, types will have *kinds*.[3] We will add two symbols to the category c of constants: the symbol $*$ for the "type of types", and the symbol $\Box$ for the the "type of types of types". We call $*$ and $\Box$ *sorts*, and we will often use the letter s to refer to either one (that is, we let $s \in \{*, \Box\}$). With the sorts, we can write

$$\mathbb{N} : * \qquad \text{and} \qquad * : \Box$$

to express that $\mathbb{N}$ is a type and that $*$ is a kind. We can also say that one variable ranges over terms and another over types, with a single notation:

$$x : \mathbb{N} \qquad \text{versus} \qquad a : *$$

We can subsequently relieve ourselves of the capital Λ, an write $\lambda a{:}* $ instead.

Before we go on to the typing rules, we have to pay some attention to the context. If we wanted to add a type assumption $x : \tau$ to the context in the simply typed lambda calculus, the type τ had to be formed according to the syntax specification. In the dependently typed system, this means that τ has to be a type or a kind, so we have to derive $\tau : s$. We reflect this in the system with the following rules, after [Bar93]:

$$\frac{\Gamma \vdash \tau : s}{\Gamma;\, x : \tau \vdash x : \tau}\ (x \notin FV(\Gamma)) \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \sigma : s}{\Gamma;\, x : \sigma \vdash e : \tau}\ (x \notin FV(\Gamma))$$

The left rule tells us that we may conclude the assumption we are making, on the premiss that the type of the assumption is legal. The rule on the right says that we can derive $e : \tau$ under the assumption $x : \sigma$, if we can also derive it without that assumption (and if σ is legal, of course). Notice that we used s, such that the assumptions can either concern term variables ($\tau : *$), or type variables ($\tau : \Box$).

We now come to the actual typing rules. The first rule formalises the relation between $*$ and $\Box$ as an axiom:

$$\frac{}{\Gamma \vdash * : \Box}$$

The following rules determine which types are legal. We often use natural numbers, so we want to be able to conclude that $\mathbb{N}$ is a legal type:

$$\frac{}{\Gamma \vdash \mathbb{N} : *}$$

We also want to use vectors types, but we will come to them later. The only other form of type left in the syntax is the quantified type:

$$\frac{\Gamma \vdash \sigma : s_1 \qquad \Gamma;\, x : \sigma \vdash \tau : s_2}{\Gamma \vdash \forall x{:}\sigma.\, \tau : s_2}$$

If we know σ is legal, and we can derive that τ is legal under the assumption $x : \sigma$, then $\forall x{:}\sigma.\tau$ is also legal. Like s, both s_1 and s_2 range over $\{*, \Box\}$. This gives us four types of function, depending on the choice of s_1 and s_2, as shown in the following table:

[3] We could turn this into an infinite hierarchy, as is done in for example Idris and Agda [Bra13, Nor07]. However, in this thesis we don't need the higher levels, so we rather keep things simple. Another option would be to let $*$ be the type of itself: $* : *$. This is also not preferable, as it leads to inconsistencies in the system [LMS10].

(s_1, s_2)	e.g.	function
$(*, *)$	$\forall x{:}\mathbb{N}.\mathbb{N}$	terms to terms
$(\Box, *)$	$\forall a{:}*.a$	types to terms
$(*, \Box)$	$\forall x{:}\mathbb{N}.*$	terms to types
$(\Box, \Box)$	$\forall a{:}*.*$	types to types

The first option corresponds to the simply typed lambda calculus, while the second adds type polymorphism. The third option gives us dependent types. We have not yet discussed the fourth option, as it is not very important in this thesis. It completes the list as the representative of functions from types to types, or type operators. It may be interesting to note that we can include or leave out any combination of these options, to get different kinds of systems. With the simply typed lambda calculus as basis, this gets us a set of eight systems, collectively known as the *lambda cube* [Bar84, Bar93].

To illustrate the type formation rule, we will derive that $\forall a{:}*.\forall n{:}\mathbb{N}.\mathit{Vect}\ a\ n$ is a type, under the assumption that we have the constructor *Vect*:

$$\dfrac{\dfrac{}{\Gamma \vdash *:\Box} \qquad \dfrac{\dfrac{}{\Gamma \vdash \mathbb{N}:*} \qquad \Gamma;\ a:*;\ n:\mathbb{N} \vdash \mathit{Vect}\ a\ n:*}{\Gamma;\ a:* \vdash \forall n{:}\mathbb{N}.\mathit{Vect}\ a\ n:*}}{\Gamma \vdash \forall a{:}*.\forall n{:}\mathbb{N}.\mathit{Vect}\ a\ n:*}$$

We start with the right-most premiss. We assume that *Vect a n* is a type under the assumptions $a:*$ and $n:\mathbb{N}$. Because we have the axiom $\mathbb{N}:*$, we are justified in assuming $n:\mathbb{N}$, and we may apply the rule with s_1 and s_2 both $*$. We are also justified in assuming $a:*$, because we have the axiom $*:\Box$. We can therefore apply the rule again, but now with $\Box$ for s_1.

The rules above give us a set of legal types, so now we want to add a corresponding set of legal terms. For $\mathbb{N}$ we can use the corresponding terms and rules from the simply typed lambda calculus (section 3.2.2), as they require no immediate revision. We will instead focus our attention on the main operations of the lambda calculus: application $(t\ t)$ and abstraction $(\lambda x{:}t.t)$. We start with abstraction, for which we have the following rule:

$$\dfrac{\Gamma;\ x:\sigma \vdash e:\tau \qquad \Gamma \vdash \forall x{:}\sigma.\tau:s}{\Gamma \vdash \lambda x{:}\sigma.e:\forall x{:}\sigma.\tau}$$

Note that both e and τ can depend on x, and that x and e can range over both terms and types, depending on σ and τ. When σ and s are both $*$ (such that $\tau:*$), the rule is equivalent to the type abstraction rule of section 3.2.4. On the other hand, when σ and τ are both types, and τ does not depend on x, we get the abstraction rule of the simply typed lambda calculus, but with $\sigma \to \tau$ written as $\forall x{:}\sigma.\tau$. The same also happens with the application rule below. This means we can regard $\forall x{:}\sigma.\tau$ as a generalisation of $\sigma \to \tau$, and for this reason we omitted arrow types from the syntax. However, we will keep it as a shorthand for the quantifier, because the arrow is slightly less cluttering.

When σ is $\Box$, we get functions from terms to types and from types to types (left and right respectively):

$$\dfrac{\Gamma;\ a:* \vdash \mathbb{N}:* \qquad \Gamma \vdash \forall a{:}*.*:*}{\Gamma \vdash \lambda a.\mathbb{N}:\forall a{:}*.*} \qquad \dfrac{\Gamma;\ n:\mathbb{N} \vdash \mathbb{N}:* \qquad \Gamma \vdash \forall n{:}\mathbb{N}.*:*}{\Gamma \vdash \lambda n.\mathbb{N}:\forall x{:}\mathbb{N}.*}$$

We see that the abstraction rule is very generic, as it can be used for all the functions from the table on page 30. The application rule is likewise very generic:

$$\frac{\Gamma \vdash e_1 : \forall x{:}\sigma.\,\tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau[x := e_2]}$$

When σ is $*$, the rule is again equivalent to the application rule we introduced in section 3.2.4, only with the side condition explicitly put in as an additional premiss. As said above, when σ and τ are both types, with τ not depending on x, the application rule becomes equivalent to the application rule of the simply typed lambda calculus.

To illustrate the application rule for type-returning functions, we continue the two examples above:

$$\frac{\Gamma \vdash \lambda a.\mathbb{N} : \forall a{:}*.* \qquad \Gamma \vdash \mathbb{B} : *}{\Gamma \vdash (\lambda a.\mathbb{N})\,\mathbb{B} : *} \qquad \frac{\Gamma \vdash \lambda n.\mathbb{N} : \forall x{:}\mathbb{N}.* \qquad \Gamma \vdash Z : \mathbb{N}}{\Gamma \vdash (\lambda n.\mathbb{N})\,Z : *}$$

On the left, we derive that a type-to-type function applied to a type forms a type. On the right, we derive that a term-to-type function applied to a term also forms a type.

Of course, when we encounter functions applied to arguments, we get the urge to evaluate. The β-reduction rule remains unaltered, so the following expressions reduce without problems:

$$(\lambda a{:}*.\mathbb{N})\ \mathbb{B} \to_\beta \mathbb{N}$$
$$(\lambda n{:}\mathbb{N}.\mathbb{N})\ Z \to_\beta \mathbb{N}$$

We also get to keep strong normalisation, such that every well-typed expression can be reduced to normal form (see [Bar93] or [Tho91] for proofs), and with that we also have β-equality, for both term and type expressions.

This leaves us to discuss one last rule to make the dependently typed system complete, at least regarding the basic rules (examples of extensions follow below). This last rule incorporates type equality into the type system. We need this, because sometimes we need to be able to derive that two types can be regarded equal. This is the case in for example the application rule, where the type of the actual argument needs to match the argument type of the function. In the simply typed lambda calculus the same types would have exactly the same syntax, and can therefore be compared relatively easy. However, when types may depend on terms, and therefore on computation, β-equality forms a better comparison. As a practical example, consider the following two vector types:

$$\mathit{Vect}\,\mathbb{N}\,(1{+}2) \qquad \text{and} \qquad \mathit{Vect}\,\mathbb{N}\,3$$

To us the types are equal, but a syntactical comparison would tell us otherwise. However, as 1+2 reduces to 3, the above types can still be regarded equal. We can encorporate this into the system with the following rule:

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash \tau' : s \qquad \tau =_\beta \tau'}{\Gamma \vdash t : \tau'}$$

When we have a term t of type τ, and τ is equivalent to τ' (which we have proven to be legal), then we can also say t is of type τ'.

We should remark that β-equivalence is not a "perfect" equivalence either. As an example we have again two vector types, but now depending on variables:

$$\mathit{Vect}\,\mathbb{N}\,(n{+}1) \qquad \text{and} \qquad \mathit{Vect}\,\mathbb{N}\,(1{+}n)$$

For these to be equivalent, n+1 and 1+n need to have the same normal form. However, this need not be the case: Imagine + is defined by recursion, for example with

the following behaviour:

$$Z + y = y$$
$$S\,x + y = S\,(x + y)$$

With such a definition, $1{+}n$ reduces to $S\,n$, while $n{+}1$ won't reduce at all. As a consequence, these expressions do not have the same normal form, and are therefore not β-equal.

Natural numbers

In the beginning of this section, we mentioned the type *BN*, and that it could have the following definition:

$$\lambda n\,.\,natElim\ \mathbb{B}\ (\lambda k\,.\,\lambda t\,.\,\mathbb{N})\ n$$

For *natElim*, we had the following rule:

$$\frac{\Gamma \vdash e_0 : \tau \qquad \Gamma \vdash f : \mathbb{N} \to \tau \to \tau \qquad \Gamma \vdash k : \mathbb{N}}{\Gamma \vdash natElim\ e_0\ f\ k : \tau}$$

This rule remains valid in the system of dependent types, and even permits the above definition of *BN*: We simply let τ be $*$, such that we get

$$\frac{\Gamma \vdash \mathbb{B} : * \qquad \Gamma \vdash \lambda k\,.\,\lambda t\,.\,\mathbb{N} : \mathbb{N} \to * \to * \qquad \Gamma \vdash k : \mathbb{N}}{\Gamma \vdash natElim\,\mathbb{B}\ f\ k : *}$$

However, we can make a more general rule for *natElim*. Notice that the first argument of f is always a natural number. In the current system, we can make the second argument and the return type of f dependent on the first argument, by using a function m that returns a type for every k:

$$\frac{\Gamma \vdash m : \forall k{:}\mathbb{N}\,.\,* \qquad \Gamma \vdash e_0 : m\,Z \qquad \Gamma \vdash f : \forall l{:}\mathbb{N}\,.\,m\,l \to m\,(S\,l) \qquad \Gamma \vdash k : \mathbb{N}}{\Gamma \vdash natElim\ m\ e_0\ f\ k : m\,k}$$

When we remember the propositions-as-types interpretation, we can view this rule as the construction of a proof by mathematical induction: we may prove $\forall x.P(x)$ if we can prove $P(0)$ and $\forall x.P(x) \Rightarrow P(x+1)$. The same structure is embodied in the premisses: e_0 is the proof of $m\,Z$, and f is the proof that $m\,l$ implies $m\,(S\,l)$, for all l. When we have completed this inductive proof, we can conclude that $m\,k$ holds for arbitrary k.

We can see that this also works in terms of types by studying the behaviour of the types during recursion (refer to the end of section 3.2.2 for refreshments, if necessary). The second argument of f is always the result of a recursive call, which is another application of f, or e_0 when l is Z. The type of the second argument of f therefore has to match the return type of f in the recursive call, or the type of e_0 when l is Z. Let's examine this for both cases, i.e. when l equals $S\,k$ and when l equals Z: When we apply f to $S\,k$, we expect the type of the recursive call to be $m\,(S\,k)$:

$$f\,(S\,k)\ :\ m\,(S\,k) \to m\,(S\,S\,k)$$

But the recursive call applies f to k, such that we have

$$f\,k\ :\ m\,k \to m\,(S\,k)$$

The return type indeed matches the expected type. When we apply f to Z, we expect the type of the recursive call to be $m\,Z$. In this case the recursive call returns e_0, which also has type $m\,Z$, as was to be shown.

We will postpone examples of the generalised rule. In the following we will discuss vectors, which are accompanied by a similar recursive structure. We will see that the dependency we introduced actually becomes a necessity, if we want to implement some of the basic operations on vectors.

Vectors

We will now expand the system with the canonical example of dependent types. We have already used the constructor *Vect* to illustrate dependent types, perhaps often enough to let its type formation rule be no surprise:

$$\frac{\Gamma \vdash a : * \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathit{Vect}\ a\ n : *}$$

To build the actual vectors, we use the terms *nil* and *cons*, representing empty and non-empty vectors respectively, similar to the list constructors of Haskell. The terms come with the following introduction rules:

$$\frac{\Gamma \vdash a : *}{\Gamma \vdash \mathit{nil}\ a : \mathit{Vect}\ a\ Z} \qquad \frac{\Gamma \vdash a : * \qquad \Gamma \vdash k : \mathbb{N} \qquad \Gamma \vdash x : a \qquad \Gamma \vdash xs : \mathit{Vect}\ a\ k}{\Gamma \vdash \mathit{cons}\ a\ k\ x\ xs : \mathit{Vect}\ a\ (S\,k)}$$

$\mathit{nil}\ \tau$ is an empty vector with (no) elements of type τ. Because it has no elements, it has type $\mathit{Vect}\ \tau\ Z$ (if τ is a type). When we have an element x of type τ and a vector xs with k elements of type τ, we can build a vector of length $S\,k$:

$$\mathit{cons}\ \tau\ k\ x\ xs : \mathit{Vect}\ \tau\ (S\,k)$$

As $\mathit{nil}\ a$ and $\mathit{cons}\ a\ k\ x\ xs$ are quite lengthy expressions, we will often suppress the length and the element type, and instead use the notation

$$[] \qquad \text{and} \qquad x :: xs$$

Like natural numbers, we can put vectors to use via recursion. We introduce the term *vElim*, which takes a base case e_0 and a step function f, and reduces roughly like this (we omit some details):

$$\begin{aligned} &\mathit{vElim}\ e_0\ f\ [] \to e_0 \\ &\mathit{vElim}\ e_0\ f\ (x :: k\ xs) \to f\ k\ x\ (\mathit{vElim}\ e_0\ f\ xs) \end{aligned}$$

This behaviour is that of a right fold. When we fold over the empty vector, we get e_0, and otherwise we get f applied to the length of the remaining vector, the first element, and the result of the recursive call.

The full typing rule for *vElim* looks like this:

$$\frac{\Gamma \vdash a : * \qquad \Gamma \vdash p : \mathbb{N} \to * \qquad \Gamma \vdash e_0 : p\,Z \qquad \Gamma \vdash f : \forall l{:}\mathbb{N}.\, a \to p\,l \to p\,(S\,l) \qquad \Gamma \vdash k : \mathbb{N} \qquad \Gamma \vdash xs : \mathit{Vect}\ a\ k}{\Gamma \vdash \mathit{vElim}\ a\ p\ e_0\ f\ k\ xs : p\,k}$$

A full application of *vElim* evidently takes quite some arguments.[4] We have a vector, xs, for which we have to specify a length k and element type a. The function p allows

[4] Some of these parameters can actually be inferred, such that a more advanced system might keep them implicit. However, we choose to leave things explicit in favour of easy implementation.

us to make the result type variable while depending on k, by giving us a return type for each vector length. When we have an empty vector, the result is of type $p\,Z$, and when we have a vector of length $S\,k$, the result is of type $p\,(S\,k)$. The type of the step function f shows the behaviour of the type during recursion: when we call f for a vector of length $S\,l$, we have to pass it the result of the recursion over the tail of the vector which has length l. Therefore, f will expect an argument of type $p\,l$, and give a result of type $p\,(S\,l)$ (for any l).

With all arguments of *vElim* explained, we can give the complete reduction rules:

$$\begin{aligned} &\mathit{vElim}\ a\ p\ e_0\ f\ Z\ (\mathit{nil}\ a) \to e_0 \\ &\mathit{vElim}\ a\ p\ e_0\ f\ (S\,k)\ (\mathit{cons}\ a\ k\ x\ xs) \to f\ k\ x\ (\mathit{vElim}\ a\ p\ e_0\ f\ k\ xs) \end{aligned}$$

To illustrate, we will study a vector concatenation function. This function takes two vectors *xs* and *ys* of lengths n and m respectively, and glues them together into a vector of length $n+m$. We therefore get a function of type

$$\forall a{:}*.\forall n{:}\mathbb{N}.\forall m{:}\mathbb{N}.\mathit{Vect}\ a\ n \to \mathit{Vect}\ a\ m \to \mathit{Vect}\ a\ (n+m)$$

We will at some point need a proof that this is indeed a legal type, but this proof is relatively straightforward, so we leave it to the reader. We proceed with the function definition, for which will use recursion over the first vector:

$$\begin{aligned} \mathit{concat}\ []\ ys &\to ys \\ \mathit{concat}\ (x :: xs)\ ys &\to x :: \mathit{concat}\ xs\ ys \end{aligned}$$

This behaviour corresponds to a fold with the following definitions:

$$\begin{aligned} p \equiv \lambda l.\,\mathit{Vect}\ a\ (l+m) &: \mathbb{N} \to * \\ e_0 \equiv ys &: \mathit{Vect}\ a\ m \\ f \equiv \lambda k.\lambda x.\lambda r.\,x :: r &: \forall l{:}\mathbb{N}.\,a \to \mathit{Vect}\ a\ (l+m) \to \mathit{Vect}\ a\ (S\,(l+m)) \end{aligned}$$

When the vector *xs* is empty, our return type should be $p\,Z$, which reduces to $\mathit{Vect}\ a\ m$. Indeed, e_0, defined to be *ys*, has this type. When *xs* has non-zero length $S\,k$, the return type is $p\,(S\,k)$, a vector of length $((S\,k)+m)$. This vector is built by first concatenating the tail of *xs* and *ys*, which gives a vector of length $l+m$. We give this vector to f, which adds the head of *xs* to it, increasing the length with one. This is expressed nicely by the type of f. However, when *xs* has length $S\,k$, f returns a vector of length $S\,(k+m)$, while we may expect a return type $p\,(S\,k)$, which reduces to $(S\,k)+m$. At this point, it is necessary for + to have the right reduction behaviour: If + is defined by recursion over its first (left) argument, then it will reduce $(S\,k)+m$ to $S\,(k+m)$, in which case all is fine. However, if + is defined by recursion over the second (right) argument, in this case m, we have a problem (we don't know anything about m), and we might need to redefine the concatenation.

Propositions as types

In section 3.2.3 we introduced the idea of propositions as types. We showed that with a few extensions, we could make the simply typed lambda calculus isomorphic with propositional calculus. With the predicate calculus we gained predicates, which allowed propositions to depend on terms. It may be clear that such propositions

correspond to dependent types. Predicates then become functions (or type constructors) that take terms to types. For example, the predicate $\leqslant$, applied to two natural numbers x and y, forms the proposition $x \leqslant y$. We may see $\leqslant$ as having type $\mathbb{N} \to \mathbb{N} \to *$.

In predicate logic we could quantify universally over a variable in a proposition with a formula of the form $\forall x.A$. We will specify the corresponding type with a formation rule. We will keep the syntax of the type more or less equal to that of the logical form, but because we are working in a typed environment, the quantification variable will also have a type:

$$\forall x : \sigma.\tau$$

This is a type under two conditions: σ has to be a type, and τ has to be a type under the assumption $x : \sigma$. The formation rule therefore looks like this:

$$\frac{\Gamma \vdash \sigma \textit{ is a type} \qquad \Gamma; x : \sigma \vdash \tau \textit{ is a type}}{\Gamma \vdash \forall x : \sigma.\tau \textit{ is a type}} \ (\forall \text{F})$$

To introduce a term of this type we need to show that we can derive a proof $p : \tau$ under the assumption $x : \sigma$, if no other assumption contains x free (see rule $\forall$I in section 3.1.2). In the constructive interpretation, a proof of $\forall x : \sigma.\tau$ is a method that takes terms t of type σ to proofs (terms) of $\tau[x := t]$. This is nothing else than a lambda abstraction:

$$\frac{\Gamma; x : \sigma \vdash p : \tau}{\Gamma \vdash \lambda x.p : (\forall x : \sigma.\tau)} \ (\forall \text{I})$$

To eliminate a term e_1 of type $\forall x : \sigma.\tau$, we can apply it to a term $e_2 : \sigma$. This gives us a term $e_1\, e_2$ of type $\tau[x := t]$:

$$\frac{\Gamma \vdash e_1 : (\forall x : \sigma.\tau) \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\, e_2 : \tau[x := t]} \ (\forall \text{E})$$

These rules are very similar to the rules for function types, and that is no coincidence. In fact, when τ does not depend on x, they behave exactly the same: $\sigma \to \tau$ is a special case of $\forall x : \sigma.\tau$ in which τ does not depend on x. We can therefore replace the rules of the function types with the dependent versions, and keep the arrow as a shorthand notation.

Existential quantification can be introduced in a similar way to universal quantification. Again we will use the same syntax as the logical variant, but with a typed variable:

$$\exists x : \sigma.\tau$$

The formation rule should also need little explanation:

$$\frac{\Gamma \vdash \sigma \textit{ is a type} \qquad \Gamma; x : \sigma \vdash \tau \textit{ is a type}}{\Gamma \vdash \exists x : \sigma.\tau \textit{ is a type}} \ (\exists \text{F})$$

Remember that a constructive proof of $\exists x : \sigma.\tau$ consists of an object of type σ, and a proof that τ holds for that object. We can therefore represent a term of type $\exists x : \sigma.\tau$ with the pair (t_1, t_2), where t_1 has type σ and t_2 has type $\tau[x := t_1]$. We can of course only do this when we indeed have both terms:

$$\frac{\Gamma \vdash t_1 : \sigma \qquad \Gamma \vdash t_2 : \tau[x := t_1]}{\Gamma \vdash (t_1, t_2) : (\exists x : \sigma.\tau)} \ (\exists \text{I})$$

We can translate the elimination rule almost directly from its logical counterpart:

$$\frac{\Gamma \vdash t : (\exists x : \sigma.\tau) \qquad \Gamma; x : \sigma; y : \tau \vdash p : \rho}{\Gamma \vdash p : \rho} \ (\exists E)$$

Note that neither p nor ρ may contain x and y free, as they are discharged. This side condition is quite restrictive, because we have a term t which we essentially just throw away. We can create a more complex rule that does make use of t:

$$\frac{\Gamma \vdash t : (\exists x : \sigma.\tau) \qquad \Gamma; x : \sigma; y : \tau \vdash p : \rho[z := (x, y)]}{\Gamma \vdash \mathit{cases}_{x,y}\, t\, p : \rho[z := t]} \ (\exists E')$$

Because we know in the conclusion what the hypothetical variables x and y should be in the conclusion, we can replace them with the corresponding constituents of t. We don't know the exact form of term t however (it might not be in normal form), so we can't use those constituents directly, and we have to use t as a whole. For the type part, we do that by making ρ dependent on a variable z of type $\exists x : \sigma.\tau$. We then replace it with a hypothetical term (x, y) in the premiss, and with t in the conclusion. For the term, we introduce the new construction $\mathit{cases}_{x,y}\, t\, p$, which has the following reduction rule:

$$\mathit{cases}_{x,y}\, (t_1, t_2)\, p \ \rightarrow \ p[x := t_1, y := t_2]$$

When t is known to be of the form (t_1, t_2), we can replace the hypothetical variables in p with t_1 and t_2. Because the conclusion may not contain x and y free, they will be bound by $\mathit{cases}_{x,y}$, which we indicate with the subscript. (Note that ρ itself may still not contain x and y free.)

Like universal quantification we can regard existential quantification as a generalisation of a type we already encountered, namely conjunction. This becomes again visible when we view the type without the dependency: Say we have the term (t_1, t_2) of type $\exists x : \sigma.\tau$, then t_1 has type σ, and t_2 has type $\tau[x := t_1]$. If τ does not depend on t_1, then t_2 has simply type τ, and we could also write $(t_1, t_2) : (\sigma \wedge \tau)$. However, the elimination of conjunction looked a bit different. We had two rules, with two projection terms π_1 and π_2. We can actually derive them for dependently typed terms from E′. For π_1 we have for instance:

$$\frac{\dfrac{\Gamma; t : (\exists x : \sigma.\tau) \vdash t : (\exists x : \sigma.\tau) \qquad \Gamma; x : \sigma; y : \tau \vdash x : \sigma}{\Gamma; t : (\exists x : \sigma.\tau) \vdash \mathit{cases}_{x,y}\, t\, x : \sigma}}{\Gamma \vdash \lambda t.\mathit{cases}_{x,y}\, t\, x : (\exists x : \sigma.\tau) \rightarrow \sigma}$$

This gives us

$$\pi_1 \equiv \lambda t.\mathit{cases}_{x,y}\, t\, x$$

For π_2, the case is a bit more difficult, because its result type depends on the first term of the pair. However, we can extract this term with π_1, such that we get the following derivation:

$$\frac{\dfrac{\Gamma; t : (\exists x' : \sigma.\tau) \vdash t : (\exists x' : \sigma.\tau) \qquad \Gamma; x : \sigma; y : \tau \vdash y : \tau[x' := \pi_1 (x, y)]}{\Gamma; t : (\exists x' : \sigma.\tau) \vdash \mathit{cases}_{x,y}\, t\, y : \tau[x' := \pi_1 (x, y)]}}{\Gamma \vdash \lambda t.\mathit{cases}_{x,y}\, t\, y : (\exists x' : \sigma.\tau) \rightarrow \tau[x' := \pi_1\, t]}$$

This gives us

$$\pi_2 \equiv \lambda t.\mathit{cases}_{x,y}\, t\, y$$

We could use the dependent versions of π_1 and π_2 to replace E′ with two slightly more compact elimination rules:

$$\frac{\Gamma \vdash t : (\exists x : \sigma.\tau)}{\Gamma \vdash \pi_1\, t : \sigma} \ (\exists E'_1) \qquad \frac{\Gamma \vdash t : (\exists x : \sigma.\tau)}{\Gamma \vdash \pi_2\, t : \tau[x := \pi_1\, t]} \ (\exists E'_2)$$

These rules are equivalent to E′ [Tho91].

4 · Timed types

In the previous chapter we discussed some basics of type theory and dependent types. In this chapter we will present a dependently typed system, based on [LMS10], which allows one to specify the timing behaviour of digital circuits on register transfer level by means of types. More specifically, we will embed a notion of time into types. This concept was first introduced in [Ott13], but it should be noted that our interpretations and definitions are different. We discuss the most important differences in section 5.1.

Because it was not feasible to develop a more elaborate system in the time given, we will restrict the scope of the system to a small subset of digital circuits. This small subset will consist of causal synchronous circuits without feedback, driven by a single clock. These circuits should also be time-invariant, in the sense that their architecture should remain fixed over time.

The structure of this chapter is as follows: First we will introduce the concept of *timed types* with a bit of its underlying philosophy. We then elaborate this concept into a formal type system, which we will implement and test. We end with a discussion, a conclusion, and some suggestions for future work.

4.1 The beginning of time in types

Imagine a synchronous circuit, processing some data. From an RTL perspective, time in this circuit is discrete and absolute: we can represent time as the number of clock cycles that have passed since the circuit started its logical operation. Figure 4.1 shows a small circuit, repeated a couple of times to portray different clock cycles in a kind of timeline. In each cycle, the circuit adds the values of two registers, and loads two new values to be added in the next cycle. Any value in the circuit "lives" at a certain place (in terms of wires) and in a certain moment (in terms of clock cycles), a location in spacetime if you will. In the figure, we made this somewhat explicit by naming a few points, and we have taken the liberty to relate points at the same location in space by giving them the same name, their location in time being indicated by their subscripts. As such, we have some points called x which are all located in space at

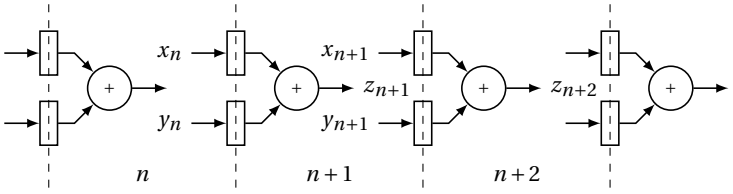


Figure 4.1: A synchronous adder circuit in consecutive clock cycles n, n+1 and n+2.

the input of the top register, but where x_n is located in moment n and x_{n+1} is located in moment $n+1$. Similar for the y's and z's. Because during the operation of a circuit each location (in space and time) corresponds to a specific value, we can use the same name to mean both the value and the location, and in the following we will do so, hopefully without causing too much confusion.

Points in space are connected to each other by combinational logic, such that a value in one place can depend on values in other places. If we assume that the combinational logic settles within one clock cycle, then it becomes physically impossible for a value to depend on values in other moments, if they are to be connected by combinational logic only. To connect points separated in time, we then require registers, which carry values from one moment to the next. (In general, points connected by registers also differ in space, since it has little use to connect the input of a register directly to its output.) In figure 4.1, the behaviour of the registers is illustrated by their position on (or across) the time boundaries.

We can describe the dependency of a value at one point on values at other points in the circuit with a purely mathematical function. For the circuit in figure 4.1 we have for example the following relation:

$$z_{n+1} = x_n + y_n = f(x_n, y_n) \qquad \text{for all moments } n \tag{4.1}$$

We are now making a transition from the concrete circuit to an abstract description: f is ignorant of space and time, a relation between pure mathematical values. It is only through the use of it in the equation above that it relates values we consider as having some location. We could also use f to relate values at other locations. For example, a different circuit is described by

$$z_{n+1} = f(x_n, x_{n+1}) \qquad \text{for all moments } n \tag{4.2}$$

The global intent underlying this thesis is to perform the above transition the other way around: we want to extract hardware descriptions from abstract functions. To be more precise, we want to extract hardware descriptions from functions defined in a functional language, with as few annotations as possible. Our approach to this is to encode some of the spatial and temporal information in types. This is not entirely unnatural. For example, if we want to regard the function f as a complete description of the architecture in figure 4.1, then we may only apply f to values that live in the same moment (as in equation 4.1), but not to values from different moments (as in equation 4.2), because that would mean a change of architecture. We could see the latter as a kind of type error: values in different moments are of different types. A related problem occurs when we require values to come from a single wire. There is a significant difference in architecture between a function that operates on a number of values from a single wire and a function that operates on the same number of values from different wires: the first function will lead to a single string of registers to delay the values, while the second function will need such a string for every value. It is sensible to form sequence types, which represent groups of values that are bound to the same wire.

For singular values, we introduce the *timed types*. As the data represented by a value will have some type itself, the timed types will be composed of both a data type and a moment. We will use the natural numbers to represent moments.[1] We will

[1] In a further development of the theory, this choice may be objectionable. For example, at some point it might be more sensible to give moments their own type. However, in the scope of this thesis the natural numbers appear adequate.

write the timed type for a value with data type t and living in moment n as

$$t\langle n\rangle$$

For example, a natural number available in the second clock cycle will have type $\mathbb{N}\langle 1\rangle$, and a boolean in the fifth cycle will have type $\mathbb{B}\langle 4\rangle$. A register takes values of some type t and some moment n to values of the same type a moment later:

$$t\langle n\rangle \to t\langle n+1\rangle$$

Of course, a register can do this for any type and any moment, so we have to quantify over both to get the actual type of the register:[2]

$$\forall t{:}*.\forall n{:}\mathbb{N}\,.\,t\langle n\rangle \to t\langle n+1\rangle$$

(where $\forall t{:}*$ means "for all types t"). The addition we used in the circuit has all its in- and outputs in the same moment, whereas the complete circuit described by f delays its input one cycle:

$$+ \;:\; \forall n{:}\mathbb{N}\,.\,\mathbb{N}\langle n\rangle \to \mathbb{N}\langle n\rangle \to \mathbb{N}\langle n\rangle$$

and

$$f \;:\; \forall n{:}\mathbb{N}\,.\,\mathbb{N}\langle n\rangle \to \mathbb{N}\langle n\rangle \to \mathbb{N}\langle n+1\rangle$$

For values that come from the same wire we introduce *timed sequences* A timed sequence is a list of values located in consecutive moments on the same wire. If the first value of the sequence lives in moment n and the last value in moment m (with $n \leqslant m$), and all values have data type t, then we write the type of that timed sequence as

$$t\langle n\,..\,m\rangle$$

For example, a sequence of two natural numbers may have type $\mathbb{N}\langle n\,..\,n+1\rangle$. These numbers come in successive moments via the same wire. If we would construct a function to sum these two values, we would give it the type

$$\forall n{:}\mathbb{N}\,.\,\mathbb{N}\langle n\,..\,n+1\rangle \to \mathbb{N}\langle n+1\rangle$$

We want to note that we do not attach high value to our choice for the form of the sequence types. Another option would have been to denote the timing with the first moment and the length of the sequence, instead of the first and last moments. This would probably save some steps during type checking, but in our opinion this conveys the meaning of a sequence less well, and that is what had our preference. We can imagine that system may evolve to include more intricate sequence types (sequences that take every n'th value, for example). In that case, other solutions may be deemed preferable.

[2] It would make a more concise language to leave these quantifications implicit, as happens in Haskell and Idris. However, we will keep them explicit for ease of implementation (it will match more closely the dependently typed theory on which we based our system). This choice also has the benefit that it makes the dependently typed nature more visible. We recommend implicit quantifications to be part of future work.

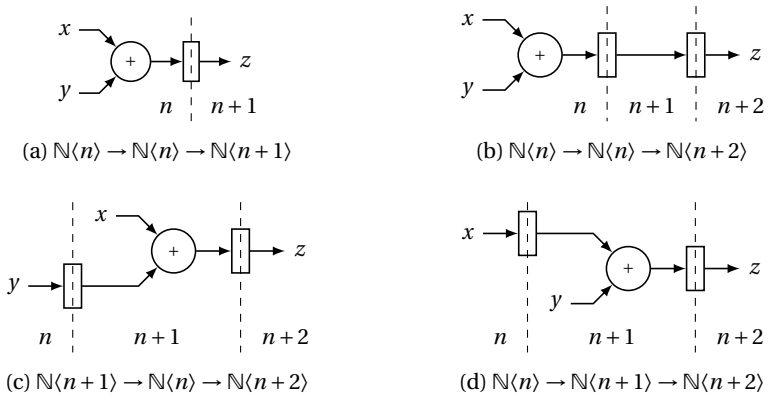


Figure 4.2: Circuits derived from $g\ n\ x\ y = f\ n\ x\ y$. Each circuit corresponds to a different type of g. In the case of (a), the type is identical to that of f, such that the circuit is equivalent to that of figure 4.1

4.1.1 *Register inference*

When we define a function, we want time to appear only in its type (at least for time-invariant functions), and accordingly we want as few time related constructions within the function body as possible. This most notably affects registers, which, in the functional realm, become identity functions with the sole purpose of casting values of type $t\langle n\rangle$ to values of type $t\langle n+1\rangle$. It is our aim to leave these registers implicit, which means they are to be inferred when the functions are translated to architectures. However, the type system still has to account for these inferred registers.

To illustrate, we take our running example of figure 4.1. We might describe the architecture with the following definition:

$$f : \forall n{:}\mathbb{N}\,.\,\mathbb{N}\langle n\rangle \to \mathbb{N}\langle n\rangle \to \mathbb{N}\langle n+1\rangle$$
$$f\ n\ x\ y\ =\ x+y$$

(Note that we use explicit quantifications, which means the moment is always an explicit parameter to the function.) The parameters x and y are both of type $\mathbb{N}\langle n\rangle$. The addition is combinational, such that $x+y$ is also of type $\mathbb{N}\langle n\rangle$. However, the function should deliver its output at $n+1$, which means we are forced to infer a register at this point. This gives us the circuit shown in figure 4.2a, which is equal (modulo retiming) to the circuit of figure 4.1.

The above example shows that we can delay if the result of a function is too early compared to the function type. It can also happen that the function is defined in such a way that the result is late, which means the computation takes longer than indicated by the type. As we are dealing with causal systems, this should incur a type error.

Aside from the temporal mismatch at the result side of the function, there can also be mismatches in the arguments of a function. In this case we can add delays regardless of whether the arguments are early or late (unless this forces the result to come late). When an argument is early, it is always possible to insert a delay. When an argument is late, we have to try to delay the function itself by shifting it to a later moment, inserting delays if it depends on values from outside the function (e.g., if it

$$
\begin{aligned}
e, \rho, k \;\coloneqq\;& * \mid \forall x{:}\rho\,.\,\rho \mid e{:}\rho \mid x \mid e\;e \mid \lambda x\,.\,e \\
\mid\;& \mathbb{N} \mid Z \mid S\;k \mid \mathit{nElim}\;e\;e\;e\;k \\
\mid\;& \mathit{Vect}\;\rho\;k \mid \mathit{vNil}\;\rho \mid \mathit{vCons}\;\rho\;k\;e\;e \mid \mathit{vElim}\;\rho\;e\;e\;e\;k\;e \\
\mid\;& \rho\langle k\,..\,k\rangle \mid \mathit{sCons}\;e\;e \mid \mathit{sElim}\;\rho\;k\;e\;e\;e\;k\;e
\end{aligned}
$$

Figure 4.3: The syntax of expressions

is a partially applied function). This is illustrated by the circuits in figures 4.2c and d, which reflect different types for the function

$$g\;n\;x\;y = f\;n\;x\;y$$

Note that f is applied to n, which means, according to the type of f, that x and y should come in moment n:

$$f\;n : \mathbb{N}\langle n\rangle \to \mathbb{N}\langle n\rangle \to \mathbb{N}\langle n+1\rangle$$

However, x is assumed to be in moment $n+1$, such that $f\;n$ needs to be shifted one moment (compare figure 4.2a), and we get[3]

$$f\;n\;x : \mathbb{N}\langle n+1\rangle \to \mathbb{N}\langle n+2\rangle$$

Now y, living in moment n, is early, and has to be delayed with a register, as shown in figure 4.2c.

In figure 4.2d the moments of x and y are swapped. Now x comes in time, but y is late. This means $f\;n\;x$ has to be shifted, and because we can't move x itself to another moment, we have to insert a register to delay it.

4.2 A SYSTEM OF TIMED TYPES

We will now introduce a type system with timed types. The language is kept to the most essential elements needed to demonstrate the timed types. There is no syntactic sugar, and there are no elaborate data types. We limit ourselves to natural numbers and sequences thereof. Although the language and implementation also provide vectors, they are not directly relevant, so we will omit them in the discussion.

The basis for the system is the theory as described in [LMS10], which is very similar to the theory of dependent types we have described in section 3.2.5. We will highlight the differences along the way. First, we focus on the syntax, which is shown in figure 4.3.

To aid the interpretation of the syntax, we use the three symbols e, ρ, and k to denote expressions. We use them with the understanding that e represents expressions of any type, ρ expressions of type $*$, and k expressions of type $\mathbb{N}$. However, one should note that these distinctions are only based on semantics, and that syntactically e, ρ, and k are equivalent. It is the type system, not the syntax, that will reject meaningless expressions. (For example, $* : Z$ is syntactically valid, but will be rejected by the type system.)

[3] Note that n remains unaltered. The type checker only checks types; it does not rewrite definitions.

The first three lines of figure 4.3 comprise the dependently typed lambda calculus (with natural numbers and vectors) as described in [LMS10]. For the bare lambda calculus we have a type of types ($*$), dependent function types ($\forall x{:}\rho\,.\,\rho$), type annotation ($e:\rho$), variables ($x$), application ($e\ e$), and lambda abstraction ($\lambda x\,.\,e$). For the natural numbers we have the type ($\mathbb{N}$), zero and successor (Z and $S\ k$), and the eliminator ($nElim\ e\ e\ k$). For the vectors we have the type ($Vect\ \rho\ k$), the empty vector ($vNil$), the cons constructor ($vCons\ \rho\ k\ e\ e$), and the eliminator ($vElim\ \rho\ e\ e\ e\ k\ e$). As said, we will not discuss vectors any further.

The language that is formed with these expressions has two differences with respect to the language described in section 3.2.5. First, we do not have the symbol $\Box$ for kinds. Instead, we use the axiom $*:*$. This makes the theory unsound (it is possible to encode a variant of Russell's paradox [Coq86]), but although this can be avoided, it does make the implementation simpler. For this thesis, we regard simplicity more important than full soundness. The second difference is that we do not state the type of the variable in lambda-abstractions. Instead we may annotate any term by writing $e:\rho$. This makes the syntax a little easier to use.

The last line of the syntax shows our contribution: the time-related expressions. We have already seen expressions of the form $\rho\langle k\,..\,k'\rangle$, which are the types of timed sequences. Types of the form $\rho\langle k\rangle$ are not of part of the syntax, but we will regard them as shorthand for $\rho\langle k\,..\,k\rangle$, the types of single-valued sequences. The terms of such single-valued sequences are the terms of the underlying data types. For example, $Z:\mathbb{N}\langle n\rangle$ is a valid typing (if $n:\mathbb{N}$). This follows from the idea that we want to think of the function as acting on pure values, such that we can define it first as a timeless function, and then turn it into a timed function by altering the types.

sCons is the constructor for sequences with more than one value. It puts a newer value in front of a sequence with older values. For example, if some value x lives in moment n, another value y in moment $n+1$, and the value z in moment $n+2$, we can write

$$sCons\ z\ (sCons\ y\ x)\ :\ \mathbb{N}\langle n\,..\,n+2\rangle$$

Notice that the last element, x, is a single-valued sequence of type $\mathbb{N}\langle n\,..\,n\rangle$. It is impossible to form a type for sequences with fewer elements, such that every sequence is terminated by a value with a singular timed type. Consequently, this means we do not need a *Nil* value to represent the empty sequence, as with vectors.

sElim is the sequence eliminator. It works much like the vector eliminator discussed in section 3.2.5, in that it performs a right fold over the sequence, where the type of the intermediate depends on the position in the sequence via a motive. The general form of an elimination is

$$sElim\ \rho\ n\ m\ z\ f\ k\ xs$$

where ρ is the underlying type of the sequence, n is the first moment of the sequence, m is the motive, z is the base case, f is the folding function, k is the length of the sequence, and xs is the sequence itself. The folding behaviour is defined as

$$sElim\ \rho\ n\ m\ z\ f\ Z\ x \twoheadrightarrow f\ Z\ x\ z$$
$$sElim\ \rho\ n\ m\ z\ f\ (S\ k)\ (sCons\ x\ xs) \twoheadrightarrow f\ (S\ k)\ x\ (sElim\ \rho\ n\ m\ z\ f\ k\ xs)$$

Notice that f takes the position in the sequence as an argument. The types of the other arguments depend on this argument, to determine the right moment for the sequence element ($\rho\langle n{+}k\rangle$), and to determine the intermediate result types:

$$f\ :\ \forall k{:}\mathbb{N}\,.\,\rho\langle n{+}k\rangle \to m\ k \to m\ (S\ k)$$

The motive m is a function that takes moments to types:

$$m : \mathbb{N} \to *$$

As an example, we define a summation over the elements of a sequence. The full application of the eliminator looks like this:

$$\begin{aligned} m &\equiv \lambda k . \mathbb{N}\langle n+k \rangle \\ f &\equiv \lambda k . \lambda x . \lambda y . x+y \\ &sElim\,\mathbb{N}\; n\; m\; Z\; f\; k\; xs \end{aligned}$$

The motive m defines that the intermediate results live in consecutive moments. The function f sums the intermediate result and the current sequence element. Figure 4.4 shows what the architecture may look like. The dotted rectangles show the first and last instantiations of the function f. If we look at the function type of each instance (that is, f applied to some k), we get

$$\begin{aligned} f\; k &: \mathbb{N}\langle n+k \rangle \to m\; k \to m\; (S\; k) \\ &: \mathbb{N}\langle n+k \rangle \to \mathbb{N}\langle n+k \rangle \to \mathbb{N}\langle n+(S\; k) \rangle \end{aligned}$$

such that each instance takes to values in one moment, and returns a result in the next moment.

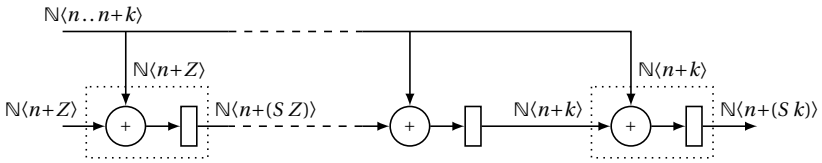


Figure 4.4: A possible architecture derived from an instance of *sElim*.

4.2.1 *The rules*

We will now present the typing and evaluation rules. We will first discuss the existing rules, and then we will add our contribution, which are the rules for timed types. The existing rules are similar to those of section 3.2, with some changes inspired by [LMS10]. A major change is that we express reductions with so-called big-step evaluation rules. Instead of defining the single steps with which terms can be reduced, as we did in section 3.2, we define how expressions evaluate to the values they represent (that is, their normal forms). We will use the symbol $\Downarrow$ for the evaluation relation, such that "e evaluates to v" is written as

$$e \Downarrow v$$

Figure 4.5 shows what are the values in our system. There is a separate category n for "neutral values", which are in a sense indeterminate, like variables for instance. As we will see, neutral values lead to special cases in the evaluation rules, hence the distinct category.

Another change is in the presentation of the typing rules. We separate the rules into ones that allow us to infer types, and ones that can only be used to check types.

$$
\begin{array}{lcl}
v,\tau,l & \coloneqq & n \mid * \mid \forall x{:}\tau\,.\,\tau \mid \lambda x\,.\,v \\
& \mid & \mathbb{N} \mid Z \mid S\ v \\
& \mid & \mathit{Vect}\ \tau\ k \mid \mathit{vNil}\ v \mid \mathit{vCons}\ v\ v\ v\ v \\
& \mid & \tau\langle l\,..\,l\rangle \mid \mathit{sCons}\ v\ v
\end{array}
\qquad
\begin{array}{lcl}
n & \coloneqq & x \mid n\ v \\
& \mid & \mathit{nElim}\ v\ v\ v\ n \\
& \mid & \mathit{vElim}\ \tau\ v\ v\ v\ l\ n \\
& \mid & \mathit{sElim}\ \tau\ l\ v\ v\ v\ l\ n
\end{array}
$$

Figure 4.5: The syntax of values

$$
\Gamma \coloneqq \varepsilon \mid \Gamma; x : \tau
\qquad
\frac{}{\text{valid}(\varepsilon)}
\qquad
\frac{\text{valid}(\Gamma) \qquad \Gamma \vdash \tau :_i *}{\text{valid}(\Gamma;\ x : \tau)}
$$

Figure 4.6: Context syntax and rules

For example, we can infer that Z has type $\mathbb{N}$, but we can't infer that $\lambda x\,.\,x$ has type $\mathbb{N} \to \mathbb{N}$. We can only check the latter. To make this distinction clear, we write

$$e :_i \rho$$

when ρ is to be inferred from e, and

$$e :_c \rho$$

when it has to be checked that e is of type ρ. In preparation of the implementation, we also explicitly mention when types are to be evaluated, such that the inferred/checked types are all in normal form.

The foundation of our system is formed by the bare dependently typed lambda calculus. Figure 4.6 shows the syntax and rules for the contexts, and figures 4.7 and 4.8 show the typing and evaluation rules respectively.

The typing rules Pi, Var, App and Lam have been discussed in chapter 3. Rule Pi governs the formation of (dependent) function types, Var the inference of variable types from the context, App the typing of applications, and Lam the typing of lambda abstractions. As we said above, we now explicitly mention when types are to be evaluated, such that we only infer or check normal forms. The axiom Star has also been discussed, in section 4.2. It defines $*$ as the type of all types. The new rules in the figure are Ann (annotation) and Chk (check). Ann tells us that we can infer the type of an annotated expression, and Chk says that we can check the type of an expression if we can infer it.

Evaluation (figure 4.8) for $*$ and variables is trivial: they do not evaluate any further. Annotated expressions, quantified types, and lambda abstractions are evaluated by reducing their constituent expressions. Annotated expressions reduce to unannotated expressions, because types are not part of values (but types can still *be* values). The evaluation of applications is split into two cases. If the applied expression evaluates to a lambda abstraction, the application can be fully evaluated by substituting the applicand in the function body and evaluating the result. If the applied expression evaluates to a neutral value, then the function is unknown, and the whole application becomes a neutral value.

Figures 4.9 and 4.10 show the typing and evaluation rules for natural numbers. The general makeup of these rules was discussed in section 3.2.5. In reading order,

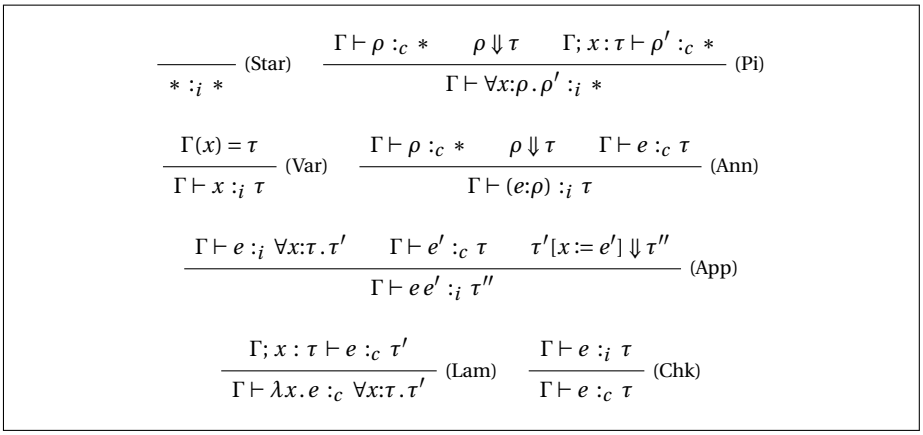


Figure 4.7: Typing rules for the bare lambda calculus

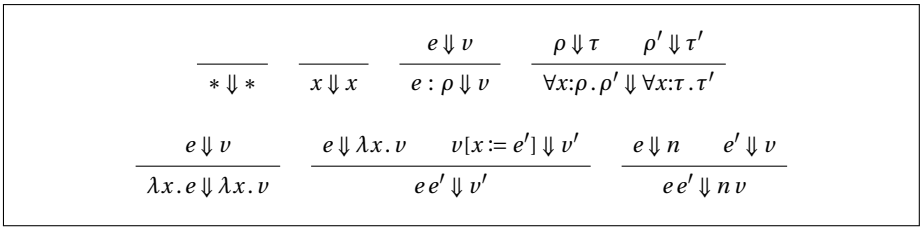


Figure 4.8: Evaluation rules for the bare lambda calculus

the typing rules show the type derivations for the type $\mathbb{N}$, the zero and successor terms, and the eliminator terms. In the same order, the evaluation rules show how we may reduce the type, the zero and successor terms, and the eliminator. Notice that we have three rules for *nElim*: one where k reduces to a neutral value (such that *nElim* can't be reduced any further), one for the base case (k reduces to Z), and one for the step case (k reduces to $S\,l$).

Figure 4.11 shows the typing rules we contribute for timed types. The rule Seq shows that $\rho\langle k\,..\,k'\rangle$ is a type if ρ is a type and k and k' are natural numbers. Because expressions of these types represent values in hardware, ρ should be a type whose values are representable. We regard it as future work to make the type system handle representable types in a decent manner. In this thesis, we use $\mathbb{N}$ as a surrogate.

The rule TChk is a timed variant of Chk, which allows us to check the type of an

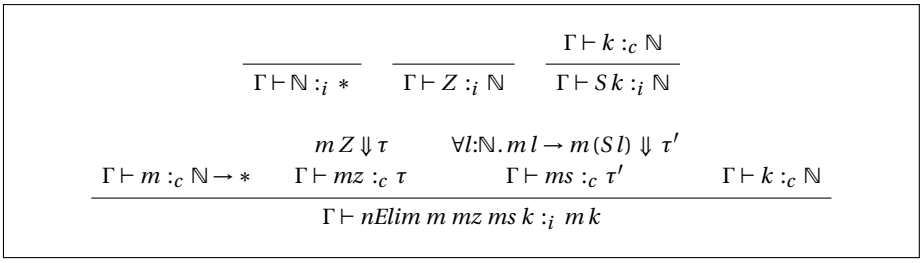


Figure 4.9: Typing rules for the natural numbers

$$\frac{}{\mathbb{N} \Downarrow \mathbb{N}} \qquad \frac{}{Z \Downarrow Z} \qquad \frac{k \Downarrow l}{S\,k \Downarrow S\,l} \qquad \frac{k \Downarrow n}{nElim\ m\ mz\ ms\ k \Downarrow nElim\ m\ mz\ ms\ n}$$

$$\frac{k \Downarrow Z \qquad mz \Downarrow v}{nElim\ m\ mz\ ms\ k \Downarrow v} \qquad \frac{k \Downarrow S\,l \qquad ms\ l\ (nElim\ m\ mz\ ms\ l) \Downarrow v}{nElim\ m\ mz\ ms\ k \Downarrow v}$$

Figure 4.10: Evaluation rules for the natural numbers

$$\frac{\Gamma \vdash \rho :_c *\qquad \Gamma \vdash k :_c \mathbb{N} \qquad \Gamma \vdash k' :_c \mathbb{N}}{\Gamma \vdash \rho\langle k..k'\rangle :_i *}\ (\text{Seq})^{\dagger}$$

$$\frac{\Gamma \vdash e :_i \tau\langle l..l'\rangle \qquad l \leqslant m \qquad m - l = m' - l'}{\Gamma \vdash e :_c \tau\langle m..m'\rangle}\ (\text{TChk}) \qquad \frac{\Gamma \vdash e :_i \tau}{\Gamma \vdash e :_c \tau\langle n..n\rangle}\ (\text{TChkL})$$

$$\frac{\begin{array}{c}\Gamma \vdash e :_i \forall x{:}\tau\langle l..l'\rangle\,.\,\tau' \qquad m - l = m' - l' \\ \Gamma \vdash e' :_i \tau\langle m..m'\rangle \qquad \tau'[x := e'] \Downarrow \tau''\end{array}}{\Gamma \vdash e\,e' :_i \Delta(l, m, \tau'')}\ (\text{TApp})$$

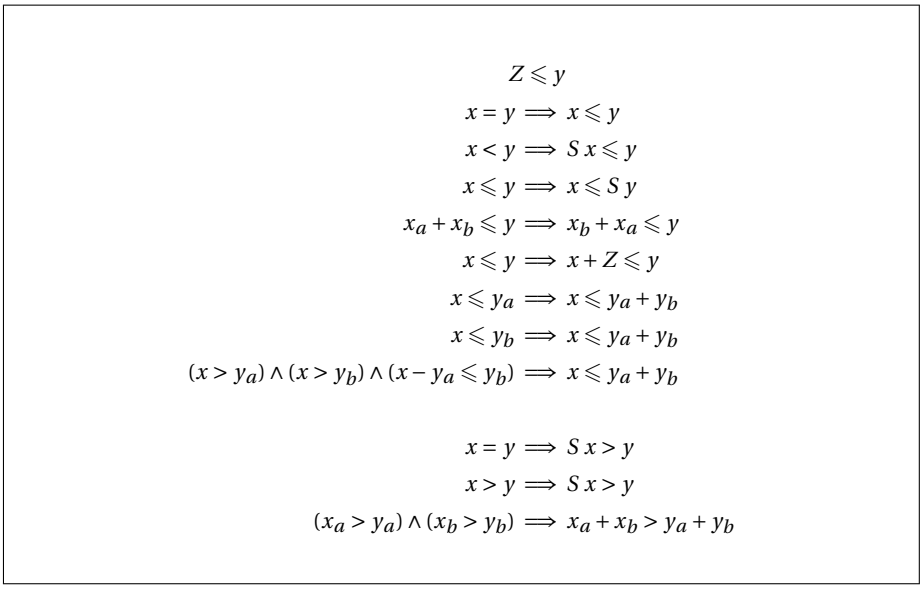
$$\frac{\Gamma \vdash e :_i \forall x{:}\tau\,.\,\tau' \qquad \Gamma \vdash e' :_i \tau\langle l\rangle \qquad \tau'[x := e'] \Downarrow \tau''}{\Gamma \vdash e\,e' :_i \Theta(l, \tau'')}\ (\text{TAppLF})$$

$$\frac{\Gamma \vdash es :_i \tau\langle l..l'\rangle \qquad \Gamma \vdash e :_c \tau\langle S\,l'\rangle}{\Gamma \vdash sCons\ e\ es :_i \tau\langle l..S\,l'\rangle}$$

$$\frac{\begin{array}{ccc} & m\,Z \Downarrow \tau & \\ \Gamma \vdash \rho :_c * & \Gamma \vdash mz :_c \tau & \\ \Gamma \vdash k :_c \mathbb{N} & \forall l{:}\mathbb{N}\,.\,\rho\langle k{+}l\rangle \to m\,l \to m\,(S\,l) \Downarrow \tau' & \Gamma \vdash l :_c \mathbb{N} \\ \Gamma \vdash m :_c \mathbb{N} \to * & \Gamma \vdash ms :_c \tau' & \Gamma \vdash e :_c \rho\langle k..k{+}l\rangle \end{array}}{\Gamma \vdash sElim\ \rho\ k\ m\ mz\ ms\ l\ e :_i m\ l}$$

† Side condition: ρ must be $\mathbb{N}$. In a more developed HDL, ρ should be representable in hardware.

Figure 4.11: Typing rules for timed types



$$
\begin{aligned}
Z &\leqslant y \\
x = y &\Longrightarrow x \leqslant y \\
x < y &\Longrightarrow S\,x \leqslant y \\
x \leqslant y &\Longrightarrow x \leqslant S\,y \\
x_a + x_b \leqslant y &\Longrightarrow x_b + x_a \leqslant y \\
x \leqslant y &\Longrightarrow x + Z \leqslant y \\
x \leqslant y_a &\Longrightarrow x \leqslant y_a + y_b \\
x \leqslant y_b &\Longrightarrow x \leqslant y_a + y_b \\
(x > y_a) \wedge (x > y_b) \wedge (x - y_a \leqslant y_b) &\Longrightarrow x \leqslant y_a + y_b \\
\\
x = y &\Longrightarrow S\,x > y \\
x > y &\Longrightarrow S\,x > y \\
(x_a > y_a) \wedge (x_b > y_b) &\Longrightarrow x_a + x_b > y_a + y_b
\end{aligned}
$$

Figure 4.12: Some examples of the properties that define the relations $\leqslant$ and $>$

expression if it is inferable. In addition, TChk requires the inferred moment to be the same as or earlier than the moment that is checked ($l \leqslant m$), and the lengths of the sequences have to be equal ($m - l = m' - l'$). If this is the case, then zero or more registers can be inserted in the hardware realisation to make delay the data. As an example of TChk, regard the following derivation:

$$
\dfrac{\dfrac{\Gamma;\, x : \mathbb{N}\langle n\rangle \vdash x :_i \mathbb{N}\langle n\rangle \qquad n \leqslant n+1 \qquad n+1-n = n+1-n}{\Gamma;\, x : \mathbb{N}\langle n\rangle \vdash x :_c \mathbb{N}\langle n+1\rangle}\ \text{(TChk)}}{\Gamma \vdash \lambda x.\, x :_c \mathbb{N}\langle n\rangle \to \mathbb{N}\langle n+1\rangle}\ \text{(Lam)}
$$

As a parameter x is assumed to have type $\mathbb{N}\langle n\rangle$. This means whatever we supply as argument must live in moment n. However, x is also the return value of the function, which must have type $\mathbb{N}\langle n+1\rangle$. Because it is possible to delay the return value ($n \leqslant n+1$) the rule TChk can be applied, and x can be checked to have type $\mathbb{N}\langle n+1\rangle$.

To judge the timings, TChk relies on a proof of $l \leqslant m$. Ideally, the system should be able to derive such proofs itself with an automated theorem prover. As this falls beyond the scope of this thesis, we make do with an ad-hoc solution. We add two sets of axioms to the system, which are the universal closures of statements as shown in figure 4.12 (the full set of statements can be found in B). One set is used to derive $l \leqslant m$, and the other to derive $l > m$ (which is used below). In the implementation, a separate function is used to construct the proofs by trying the axioms one by one. There is no particular reasoning behind the choice of these particular axioms, other than that they were either obvious or a quick fix to make some example work.

Besides TChk there is another check rule: TChkL. This rule says that an expression can be checked to have a timed type if the underlying data type can be inferred from it. The reasoning behind this rule is that nontimed values may represent constants in hardware, and can therefore live in any moment.

When functions and arguments have fully matched types (and therefore matched time signatures), no register inference is needed, and the application rule of the bare lambda calculus can be used. When the time signatures do not match registers may be inferred, but then the application rule does not suffice. Instead, the rule TApp may be applied:

$$\frac{\Gamma \vdash e :_i \forall x{:}\tau\langle l\,..\,l'\rangle\,.\,\tau' \qquad m - l = m' - l' \qquad \Gamma \vdash e' :_i \tau\langle m\,..\,m'\rangle \qquad \tau'[x := e'] \Downarrow \tau''}{\Gamma \vdash e\,e' :_i \Delta(l, m, \tau'')}\ \text{(TApp)}$$

Notice that TApp infers the argument type while the original App rule checks it, because the timing of the argument is not known on beforehand.

Because applicands may be early or late, the only additional premiss required by TApp (with respect to App) is that the lengths of the timed sequences are equal ($m - l = m' - l'$). As discussed in section 4.1.1, when the applicand is late, the computation represented by the applied function is shifted to make up. This means the time signature of the result has to be adjusted, which is done with the function Δ (figure 4.13). For function types, Δ shifts the result type, and the argument type if it is not a function type. Non-function types are shifted only if they are timed. An example:

If

$$\begin{aligned} \tau &\equiv \mathbb{N} \to (\forall m{:}\mathbb{N}\,.\,\mathbb{N}\langle m\rangle) \to \mathbb{N}\langle n+4\rangle \\ f &: \mathbb{N}\langle n+2\rangle \to \tau \\ x &: \mathbb{N}\langle n+3\rangle \end{aligned}$$

then

$$\begin{aligned} f\,x &: \Delta(n+2, n+3, \tau) \\ &: \mathbb{N} \to (\forall n{:}\mathbb{N}\,.\,\mathbb{N}\langle n\rangle) \to \mathbb{N}\langle n+5\rangle \end{aligned}$$

Besides TApp there is another application rule for timed types, namely TAppLF:

$$\frac{\Gamma \vdash e :_i \forall x{:}\tau\,.\,\tau' \qquad \Gamma \vdash e' :_i \tau\langle l\rangle \qquad \tau'[x := e'] \Downarrow \tau''}{\Gamma \vdash e\,e' :_i \Theta(l, \tau'')}\ \text{(TAppLF)}$$

This rule allows timeless functions to be applied to arguments with timed types. The intuition behind this is that timeless functions may represent pure combinational hardware, which, not being driven by a clock, is also timeless. Combinational circuits can be combined with synchronous circuits to form new synchronous circuits, and this is what the rule represents. As a consequence of the application, the type of the result needs to be converted to account for time: if the argument lives in moment n, then the result will be in moment n, and any other argument that can possibly be an input of the circuit must also live in moment n. This conversion is done with the function Θ, shown in figure 4.13. For our purposes, we only convert $\mathbb{N}$ (our surrogate representable type), and recursively the constituents of function types. Another example:

If

$$\begin{aligned} (+) &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ x &: \mathbb{N}\langle n\rangle \end{aligned}$$

Time shifting

$$\Delta(l,l',\tau) \equiv \begin{cases} \tau'\langle k+l'-l\,..\,k'+l'-l\rangle & \text{if } l \leqslant l' \text{ and } \tau = \tau'\langle k\,..\,k'\rangle \\ \forall x{:}\sigma.\,\Delta(l,l',\tau') & \text{if } \tau = \forall x{:}\sigma.\,\tau' \text{ and } \sigma = \forall\ldots \\ \forall x{:}\Delta(l,l',\sigma).\,\Delta(l,l',\tau') & \text{if } \tau = \forall x{:}\sigma.\,\tau' \text{ and } \sigma \neq \forall\ldots \\ \tau & \text{if } l > l' \text{ or } \tau = \mathbb{N} \text{ or } \tau = \mathit{Vect}\ \tau'\ k \end{cases}$$

Implicit timing

$$\Theta(k,\tau) \equiv \begin{cases} \mathbb{N}\langle k\rangle & \text{if } \tau = \mathbb{N} \\ \mathit{Vect}\ \tau'\ l & \text{if } \tau = \mathit{Vect}\ \tau'\ l \\ \forall x{:}\sigma.\,\Theta(k,\tau') & \text{if } \tau = \forall x{:}\sigma.\,\tau' \text{ and } \sigma = \forall\ldots \\ \forall x{:}\Theta(k,\sigma).\,\Theta(k,\tau') & \text{if } \tau = \forall x{:}\sigma.\,\tau' \text{ and } \sigma \neq \forall\ldots \end{cases}$$

Intralingual functions

$$(+) \equiv \lambda n.\,nElim\ (\lambda k.\mathbb{N}\to\mathbb{N})\ (\lambda x.\,x)\ (\lambda l.\,\lambda f.\,\lambda n.\,S\,(f\,n))\ n : \mathbb{N}\to\mathbb{N}\to\mathbb{N}$$
$$pred \equiv \lambda n.\,nElim\ (\lambda k.\mathbb{N})\ Z\ (\lambda k.\,\lambda r.\,k)\ n : \mathbb{N}\to\mathbb{N}$$

Figure 4.13: Auxiliary definitions for the type system

then

$$\begin{aligned}(+)\,x &: \Theta(n,\mathbb{N}\to\mathbb{N}) \\ &: \mathbb{N}\langle n\rangle \to \mathbb{N}\langle n\rangle\end{aligned}$$

The last two typing rules for timed types deal with the constructor *sCons* and the eliminator *sElim*. The rule for *sCons* was

$$\frac{\Gamma \vdash es :_i \tau\langle l\,..\,l'\rangle \qquad \Gamma \vdash e :_c \tau\langle S\,l'\rangle}{\Gamma \vdash sCons\ e\ es :_i \tau\langle l\,..\,S\,l'\rangle}$$

An element e can be attached to a sequence es with *sCons* if it has the same underlying type and lives in the next moment. That is, if es has type $\tau\langle l\,..\,l'\rangle$, then e must have type $\tau\langle S\,l'\rangle$. The resulting sequence then has type $\tau\langle l\,..\,S\,l'\rangle$.

The rule for *sElim* is a bit more elaborate:

$$\frac{\begin{array}{ccc} & m\,Z \Downarrow \tau & \\ \Gamma \vdash \rho :_c * & \Gamma \vdash mz :_c \tau & \\ \Gamma \vdash k :_c \mathbb{N} & \forall l{:}\mathbb{N}.\,\rho\langle k{+}l\rangle \to m\,l \to m\,(S\,l) \Downarrow \tau' & \Gamma \vdash l :_c \mathbb{N} \\ \Gamma \vdash m :_c \mathbb{N}\to * & \Gamma \vdash ms :_c \tau' & \Gamma \vdash e :_c \rho\langle k\,..\,k{+}l\rangle \end{array}}{\Gamma \vdash sElim\ \rho\ k\ m\ mz\ ms\ l\ e :_i m\,l}$$

First of all, we have some premisses related to the sequence e to which *sElim* is applied. The type of e is passed to *sElim* as the arguments ρ, k and l. If they have the right type ($\rho : *$, $k : \mathbb{N}$ and $l : \mathbb{N}$), then the type of e ($\rho\langle k\,..\,k{+}l\rangle$) can be checked. The folded function is represented by *ms*, and the initial accumulation value by *mz*. The types of the accumulations are determined by m, which should be a function from $\mathbb{N}$

$$\frac{\rho \Downarrow \tau \qquad k \Downarrow l \qquad k' \Downarrow l'}{\rho\langle k..k'\rangle \Downarrow \tau\langle l..l'\rangle} \qquad \frac{e \Downarrow v \qquad es \Downarrow vs}{sCons\ e\ es \Downarrow sCons\ v\ vs}$$

$$\frac{e \Downarrow n}{sElim\ \rho\ m\ mz\ ms\ k\ e \Downarrow sElim\ \rho\ m\ mz\ ms\ k\ n}$$

$$\frac{k \Downarrow Z \qquad e \Downarrow v \qquad mz \Downarrow vz \qquad ms\ Z\ v\ vz \Downarrow v'}{sElim\ \rho\ m\ mz\ ms\ k\ e \Downarrow v'}$$

$$\frac{k \Downarrow S\ l \qquad e \Downarrow sCons\ v\ vs \qquad ms\ l\ v\ (sElim\ \rho\ m\ mz\ ms\ l\ vs) \Downarrow v'}{sElim\ \rho\ m\ mz\ ms\ k\ e \Downarrow v'}$$

Figure 4.14: Evaluation rules for timed types

to $*$. For the initial accumulation value the position in the list is Z, and so *mz* should have type $m\ Z$. This is shown in the first two premisses of the second column. The last two premisses of that column specify the type of the function *ms*. The arguments of this function are the position l in the sequence, the element at that position, which then has type $\rho\langle k+l\rangle$, and an accumulation value, of type $m\ l$. The function returns the next accumulation value, which is of type $m\ (S\ l)$.

The evaluation rules for timed types are displayed in figure 4.14. Type expressions are evaluated through their constituents. Because there is no specific constructor for terms of singular timed types ($\tau\langle l..l\rangle$), these terms are evaluated with the rules of the underlying data type τ. Terms constructed with *sCons* are evaluated by separately evaluating the head and the tail.

For *sElim* there are three rules. The first rule in figure 4.14 is used when the sequence is a neutral value. In that case the whole term becomes a neutral value. The second rule applies if the sequence consists of a single value. Because sequences cannot be empty, this is the base case. k, which is the length of the tail of the sequence, is now Z. The sequence e evaluates to the single value v, and the initial accumulation value *mz* evaluates to *vz*. The folded function *ms* is then applied to Z, v and *vz*, and evaluates to the value of the complete term.

The last rule of figure 4.14 shows the step case of *sElim*. The index k now evaluates to the successor of some l, and the sequence e evaluates to $sCons\ v\ vs$. The folded function *ms* is then applied to l, the current element value v, and the recursive call containing l and the tail *vs*. The full application of *ms* evaluates to the final value of the whole *sElim*-term.

4.3 Implementation

In this section we will discuss an implementation of the system of section 4.2. Like that system, the implementation is inspired by [LMS10]. We will first describe the global structure, and then we will go into the data structures and functions more deeply.

As explained in section 4.2.1, type checking is split into checking and inference. This is reflected in the implementation: the abstract syntax consists of two corresponding parts which are handled separately. The type-checkable part is checked

```
1  data Name : Type where
2    Global : String → Name
3    Local  : Nat → Name
4    Quote  : Nat → Name
```

Figure 4.15: The data structure for names of variables.

by a function that needs an expected type as additional input and returns the unit value if the term matches the type. The type-inferable part is fed into a function that returns the inferred type as a value (that is, evaluated as far as possible). Both functions accept an environment with type-checked definitions (which might be needed in evaluation) and a context with the names and types of variables. We use a locally nameless representation of variables, which means that bound variables are represented by De Bruijn indices, while free variables are named.

Like type checking/inference, evaluation is also handled by two functions, which also accept an environment and context. Both functions return values which are evaluated as far as possible. To ease the implementation, we use host language functions to represent evaluated functions and pi-types. This means we let the host language do proper substitution, instead of implementing it ourselves. As a consequence, we cannot compare values, so we also need a quotation function which translates values back into terms.

4.3.1 *Representation of terms and values*

As said above, we use a locally nameless representation of variables, which makes a distinction between bound and free variables. Bound variables are represented by De Bruijn indices[4], while free variables have names. A name can be of three kinds: global, local or quote, as shown in figure 4.15. A global name is the name given to a variable by the user. A local name is generated by the type checker when it moves under a lambda-abstraction. In that case the bound variable is added to the context, and all occurrences of it in the body of the abstraction are replaced by free variables with a local name. Quote names are used by the quotation functions, and are also generated by the type checker.

The syntax tree is constructed with the two data types in figure 4.16 (see figure 4.3 for the formal syntax). Inferable terms are of type *ITerm*, and checkable terms are of type *CTerm*. This partioning matches the check/infer indications in the typing rules.

IStar is the type of types, *IPi* is Pi-type, *IAnn* is an annotated term, and *IApp* is an application. *IBound* and *IFree* represent the variables. For the natural numbers there is the type *INat* and the terms *IZ*, *IS* and *INElim*. For the timed types we have the type *ISeq* and the terms *ISCons* and *ISElim*. Finally, we have the checkable terms, which are either lambda abstractions (*CLam*) or inferable terms (*CInf*).

Before we move on to the representation of values, we will give some examples. The term $\lambda x\,.\,x$ is constructed as

[4] A De Bruijn index is the distance between a variable occurrence and the binding, as measured by the number of abstractions between the two. For example, $\lambda x\,.\,\lambda y\,.\,(\lambda z\,.\,z)\ x\ y$ is represented with De Bruijn indices as $\lambda\,.\,\lambda\,.\,(\lambda\,.\,0)\ 1\ 0$. Notice that the variable names all disappear.

```
1   data ITerm : Type where
2     IStar   : ITerm
3     IPi     : CTerm → CTerm → ITerm
4     IBound  : Nat → ITerm
5     IFree   : Name → ITerm
6     IAnn    : CTerm → CTerm → ITerm
7     IApp    : ITerm → CTerm → ITerm
8     INat    : ITerm
9     IZ      : ITerm
10    IS      : CTerm → ITerm
11    INElim  : CTerm → CTerm → CTerm → CTerm → ITerm
12    ISeq    : CTerm → CTerm → CTerm → ITerm
13    ISCons  : CTerm → ITerm → ITerm
14    ISElim  : CTerm → CTerm → CTerm → CTerm
15              → CTerm → CTerm → CTerm → ITerm

16  data CTerm : Type where
17    CInf  : ITerm → CTerm
18    CLam  : CTerm → CTerm
```

Figure 4.16: The data structures for inferable terms (*ITerm*) and checkable terms (*CTerm*).

CLam (*CInf* (*IBound* 0))

Note that the variable x has been replaced with a De Bruijn index, *IBound* 0. The term $\lambda x\,.\,y$, where y is a free variable, becomes

CLam (*CInf* (*IFree* (*Global* "y")))

We can annotate this expression with a type, for example $\forall x{:}a\,.\,t\ x$. The type itself will be

IPi (*CInf* (*IFree* (*Global* "a")))
  (*IApp* (*IFree* (*Global* "t")) (*CInf* (*IBound* 0)))

The forall-quantifier binds x, and therefore the variable becomes a De Bruijn index in the return type. Combining the last two expressions as an annotated term gives

CInf (*IAnn* (*CLam* (*CInf* (*IFree* (*Global* "y"))))
        (*IPi* (*CInf* (*IFree* (*Global* "a")))
          (*IApp* (*IFree* (*Global* "t")) (*CInf* (*IBound* 0))))))

The values to which terms may evaluate are represented with the data structures shown in figure 4.17 (see figure 4.5 for the formal syntax definition). As we mentioned earlier, we use the host language to perform substitutions during function application. These substitutions occur in both the types and the bodies of functions, which means *VPi* and *VLam* both take a host language function as argument. The neutral values

```
1   data Value : Type where
2     VNeu    : Neutral → Value
3     VStar   : Value
4     VPi     : Value → (Value → Result Value) → Value
5     VLam    : (Value → Result Value) → Value
6     VNat    : Value
7     VZ      : Value
8     VS      : Value → Value
9     VSeq    : Value → Value → Value → Value
10    VSCons  : Value → Value → Value


11  data Neutral : Type where
12    NFree   : Name → Neutral
13    NApp    : Neutral → Value → Neutral
14    NNElim  : Value → Value → Value → Neutral → Neutral
15    NSElim  : Value → Value → Value → Value
16              → Value → Value → Neutral → Neutral
```

Figure 4.17: The data structures for values.

```
1   data PDecl : Type where
2     Ass  : Name → CTerm → PDecl
3     Def  : Name → ITerm → PDecl
```

Figure 4.18: The data structure for declarations.

are represented with their own data stucture *Neutral*, and are embedded in the *Value* structure with the *VNeu* constructor.

Finally, figure 4.18 shows a data structure for declarations. The constructor *Ass* declares a type assumption, and the constructor *Def* a definition. For example,

```
Ass (Global "a") (CInf (IStar))
```

declares a variable a of type $*$, and

```
Def (Global "x") IZ
```

defines x to be Z. The term of a definition is always an inferable type, which forces a top-level type annotation for lambda abstractions. The evaluation and type checking functions accept a list of declarations, which can used to define substitutes for the free variables in a term that is being evaluated.

```
1   ieval : List PDecl → Vect n Value → ITerm → Result Value
2   ieval D G (IStar)      = return VStar
3   ieval D G (IPi s t)    = do vs ← ceval D G s
4                               return $ VPi vs (λx ⇒ ceval D (x :: G) t)
5   ieval D G (IBound i)   = index i G
6   ieval D G (IFree x)    = case fvlookup x D of
7                               Just (Def _ x', D') ⇒ ieval D' G x'
8                               _                   ⇒ return (VNeu (NFree x))
9   ieval D G (IAnn e _)   = ceval D G e
10  ieval D G (IApp f x)   = do vf ← ieval D G f
11                              vx ← ceval D G x
12                              vapp vf vx


13  ceval : List PDecl → Vect n Value → CTerm → Result Value
14  ceval D G (CLam e) = return $ VLam (λx ⇒ ceval D (x :: G) e)
15  ceval D G (CInf e) = ieval D G e


16  vapp : Value → Value → Result Value
17  vapp (VLam f) x = f x
18  vapp (VNeu e) x = return $ VNeu (NApp e x)
19  vapp v        x = Left "int. error: vapp"
```

Figure 4.19: The evaluation functions

4.3.2 *Evaluation*

For evaluation, we have two functions: *ceval*, which handles type-checkable terms, and *ieval*, which handles type-inferable terms. The necessary evaluation steps for each term can mostly be read off the rules discussed in section 4.2. We will start with the evaluation rules of the bare lambda calculus (figure 4.8), shown in figure 4.19.

For both functions, the first two arguments are a list D of declarations, and a vector G with values. The list of declarations contains prior definitions of terms. The vector G contains values that are to be substituted for bound variables. The index of a value in the vector will be the same as the De Bruijn index of the corresponding bound variable, such that bound variables can be evaluated by returning the value with the right index (see line 5 in the figure). Both functions return a *Result*, which is defined as

```
Result : Type → Type
Result = Either String
```

so that the functions can also return with errors.

The first case of *ieval* is trivial: $*$ evaluates to $*$. The second case handles pi-types. As is shown in the corresponding rule in figure 4.8, first the argument type is evaluated (line 3), and then the return type (the *ceval* call in line 4). To facilitate further evaluation of the return type when the argument is known, the return type is

```
1   ieval D G (INat)                = return VNat
2   ieval D G (IZ)                  = return VZ
3   ieval D G (IS k)                = do vk ← ceval D G k
4                                       return $ VS vk
5   ieval D G (INElim m mz ms k)    = do vmz ← ceval D G mz
6                                        vms ← ceval D G ms
7                                        vk  ← ceval D G k
8                                        rec vmz vms vk
9     where
10      rec : Value → Value → Value → Result Value
11      rec vmz vms (VZ)      = return vmz
12      rec vmz vms (VS l)    = do r ← rec vmz vms l
13                                 v ← vapp vms l
14                                 vapp v r
15      rec vmz vms (VNeu e)  = do vm ← ceval D G m
16                                 return $ VNeu (NNElim vm vmz vms e)
17      rec vmz vms _         = Left "int. error: eval natElim"
```

Figure 4.20: Evaluation of natural numbers

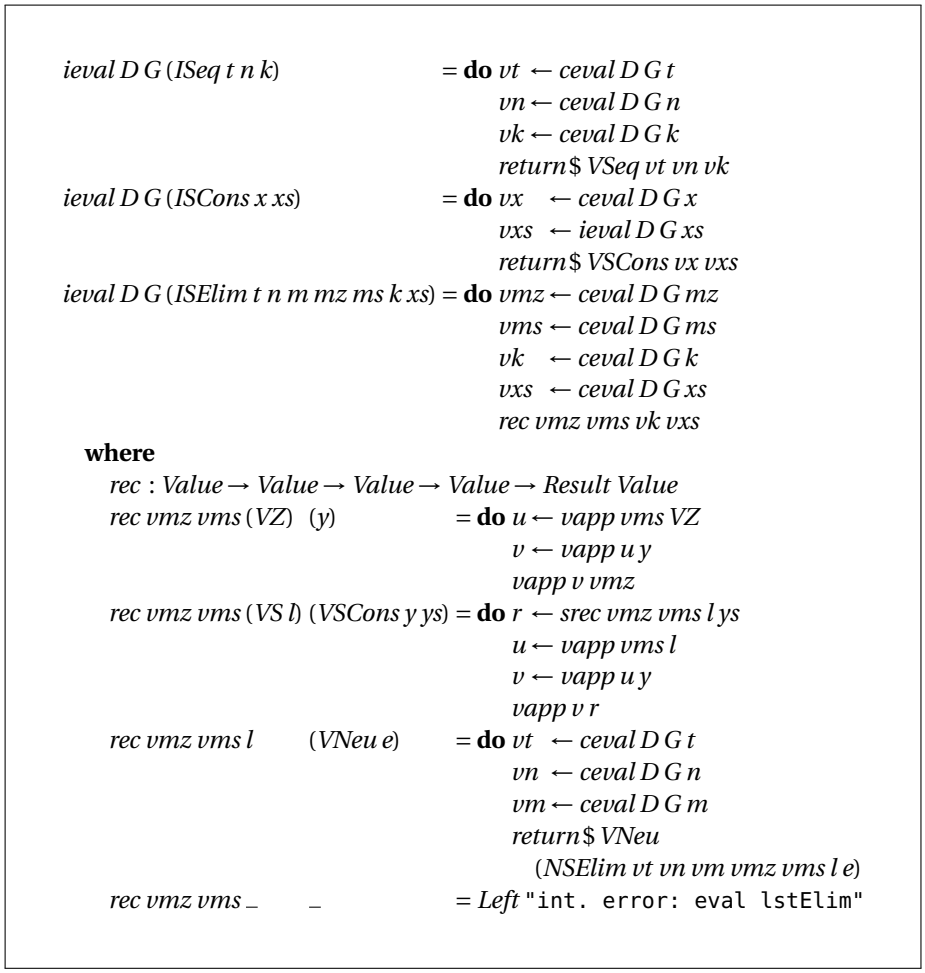
represented as an Idris function which extends the context with this argument, and evaluates the return type under the extended context. A similar construction is used for the evaluation of lambda abstractions in line 14.

As said above, bound variables are looked up in the context by their index. Free variables (line 5) are looked up in the environment *D*. If there is a definition for the variable, then it is evaluated (line 7). Otherwise, the variable is regarded as a neutral value (line 8).

Annotations (line 9) are again trivial: the annotated expression is evaluated, and the annotation is discarded. For applications (line 10), the function and argument are evaluated, and subsequently applied with the helper function *vapp* (shown in lines 16 to 19). If the function is a lambda abstraction, *vapp* returns the actual application. If the function is a neutral value, then *vapp* returns a neutral value. This is in accordance with the two application rules shown in figure 4.8.

Evaluation of the natural numbers, shown in figure 4.20, proceeds in a similar fashion as the above. The cases for *INat*, *IZ* and *IS* can easily be understood by comparing them with their corresponding rules in figure 4.10. The case for *INElim* is also more or less a direct translation of the rules, but it is a bit more involved. To be able to differentiate between the base, step, and neutral cases, the argument *k* has to be evaluated first, as shown in line 7. The function *rec* is then called to perform the actual recursion. Note in lines 12 to 13 the multiple uses of *vapp* to get the value of *ms l* (*nElim m mz ms l*).

Finally, figure 4.21 shows the evaluation for terms related to timed types, corresponding to the rules of figure 4.14. The cases for *ISeq* and *ISCons* are straightforward. The case for *ISElim*, like *INElim*, makes use of a helper function *rec* to implement the recursive behaviour defined by the rules for the base case, the step case and the neutral case.



```
1  ieval D G (ISeq t n k)                  = do vt ← ceval D G t
2                                              vn ← ceval D G n
3                                              vk ← ceval D G k
4                                              return $ VSeq vt vn vk
5  ieval D G (ISCons x xs)                 = do vx  ← ceval D G x
6                                              vxs ← ieval D G xs
7                                              return $ VSCons vx vxs
8  ieval D G (ISElim t n m mz ms k xs) = do vmz ← ceval D G mz
9                                              vms ← ceval D G ms
10                                             vk  ← ceval D G k
11                                             vxs ← ceval D G xs
12                                             rec vmz vms vk vxs
13   where
14     rec : Value → Value → Value → Value → Result Value
15     rec vmz vms (VZ)   (y)              = do u ← vapp vms VZ
16                                              v ← vapp u y
17                                              vapp v vmz
18     rec vmz vms (VS l) (VSCons y ys) = do r ← srec vmz vms l ys
19                                              u ← vapp vms l
20                                              v ← vapp u y
21                                              vapp v r
22     rec vmz vms l      (VNeu e)      = do vt ← ceval D G t
23                                              vn ← ceval D G n
24                                              vm ← ceval D G m
25                                              return $ VNeu
26                                                (NSElim vt vn vm vmz vms l e)
27     rec vmz vms _      _             = Left "int. error: eval lstElim"
```

Figure 4.21: Evaluation of timed types.

4.3.3 *Type checking*

We will now discuss the type checking and inference rules. The type checking rules are implemented in the function *ctype*:

$$ctype : Nat \rightarrow List\ PDecl \rightarrow Vect\ n\ (Name, Ty) \rightarrow CTerm \rightarrow Ty \rightarrow Result\ ()$$

The first argument is the abstraction level, which increases every time the checker moves under a binding. The second and third arguments are the environment and context, as discussed. The last arguments are the term to be checked, and the type against we check it. The function returns with a *Result*, which is an error string on failure or the unit value () on success. The type inference rules are implemented in the function *itype*, which has the same type, except that it returns the type of the term as a result, instead of accepting it as an argument:

$$itype : Nat \rightarrow List\ PDecl \rightarrow Vect\ n\ (Name, Ty) \rightarrow ITerm \rightarrow Result\ Ty$$

We will start again with the lambda calculus and the natural numbers, and focus on the timed types afterwards.

Non-timed types

Figure 4.22 shows the type checking and inference functions for the rules of figure 4.7. We will go trough them one by one, in the order they are depicted. The first rule is the axiom $* : *$, which is trivial to implement. The next rule is that for pi-types. Conform with the rule, the argument type s is first checked to really be a type, and it is evaluated to s' (lines 4 and 5). Then, the return type t is checked in line 6. In doing so, the checker moves under the binding, and the bound variable becomes locally free. All bound variables with De Bruijn index 0 are accordingly replaced by a free variable with the name *Local n*, and this variable is added to the context G with the assumption that it has type s'. The substitution of the bound variable is done in line 3 with the call to *csubst*, which we will discuss later on.

The next rule is Var. Because all bound variables are replaced with free variables when the checker moves under a binder, the checker should never have to check a bound variable, so this case gives an error (line 8). The type assumptions of free variables reside in the context, so we simply look those up when necessary (line 9).

For annotations (line 10) the checker first needs to check and evaluate the given type, and then check the annotated term against the result.

The application rule is a bit more involved, because it also accounts for the application rules for timed types. The rules have in common that the type of the function is inferred first, and the type of the argument is then compared to the inferred type. This comparison is for all application rules embodied in one function *unify*, which we will discuss when we go into the details of checking timed types.

Lines 21 to 27 of figure 4.22 show the checking clauses for lambda abstractions and inferable terms, corresponding to the rules Lam and Chk of figure 4.7. To check lambda abstractions, the bound variables in both the return type t and the lambda body e are replaced with locally free variables. For the lambda body, this is done in line 22 with the substitution function *csubst*. For the return type, it is done by applying t to the free variable (remember that t is an Idris function that evaluates and returns the term with the variable substituted). Finally, to check an inferable term e,

```
1  itype : Nat → List PDecl → Vect n (Name, Ty) → ITerm → Result Ty
2  itype n D G (IStar)      = return VStar
3  itype n D G (IPi s t)    = let t' = csubst 0 (IFree (Local n)) t in
4                               do ctype n D G s VStar
5                                  s' ← ceval D [ ] s
6                                  ctype (n + 1) D ((Local n, s') :: G) t' VStar
7                                  return VStar
8  itype n D G (IBound i) = Left "bound variable not bound"
9  itype n D G (IFree x)    = lookup x G
10 itype n D G (IAnn e t)  = do ctype n D G t VStar
11                              t' ← ceval D [ ] t
12                              ctype n D G e t'
13                              return t'
14 itype n D G (IApp f x)  = do tf ← itype n D G f
15                              case tf of
16                                VPi s t ⇒ case unify n D G s t x of
17                                            Right t' ⇒ return t'
18                                            Left er  ⇒ Left er
19                                _       ⇒ Left "illegal application"


20 ctype : Nat → List PDecl → Vect n (Name, Ty) → CTerm → Ty → Result ()
21 ctype n D G (CLam e) (VPi s t) =
22   let e' = csubst 0 (IFree (Local n)) e in
23   do t' ← t (VNeu (NFree (Local n)))
24      ctype (n + 1) D ((Local n, s) :: G) e' t'
25 ctype n D G (CInf e) tt =
26   do ts ← itype n D G e
27      valEq ts tt


28 valEq : Value → Value → Result ()
29 valEq x y = do x' ← quote0 $ Right x
30                y' ← quote0 $ Right y
31                if (x' ≡ y')
32                  then return ()
33                  else Left "can't convert types"
```

Figure 4.22: Type inference/checking of non-timed types

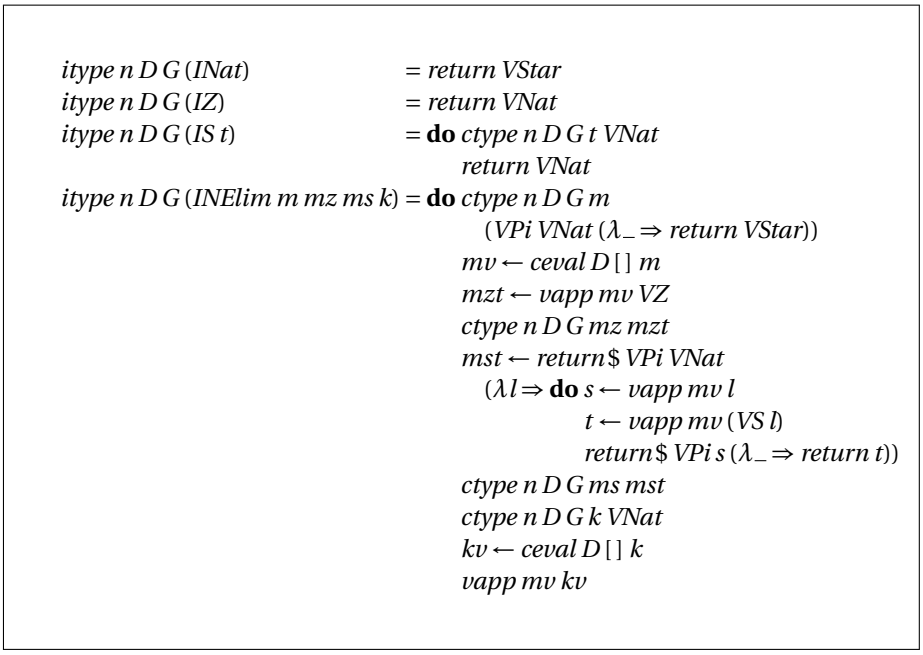


Figure 4.23: Type inference for the natural numbers.

the type of *e* is inferred and compared to the given type with the function *valEq*. This function makes use of the quotation, which we will describe below.

The natural numbers, whose typing rules are depicted in figure 4.9, have the implementation shown in figure 4.23. The cases for *INat*, *IZ*, and *IS* are trivial reflections of their corresponding rules. The case for *INElim* is more involved, due to the two pi-types that need to be checked. The first one (in lines 5 and 6) is *m*, the function that generates the types for each stage of the recursion. This function goes from $\mathbb{N}$ to $*$, so it's type is *VPi VNat* ($\lambda\_ \Rightarrow$ *return VStar*). If *m* has the right type, it can be evaluated and applied to *VZ* to give the type of the initial value *mz* (lines 7 and 8). Subsequently, *mz* can be checked to have type *mzt*. The type of the folded function *ms* should be *mst*, which is the second pi-type, namely $\forall l{:}\mathbb{N}.\, m\ l \to m\ (S\ l)$. Notice the applications of *m* (*mv* in the code) in lines 11 and 13.

Substitution and quoting

In the above we used the functions *csubst* and *quote0*. Figure 4.24 shows part of the definitions of *csubst* and its sibling *isubst*. This substitution is generally straightforward, as it is a recursion on the structure of terms. The parameter *n* indicates the abstraction level, so in the cases of *CLam* and *IPi* it is increased for the corresponding subterms. The actual substitution is performed when *isubst* encounters a bound variable (line 6).

The function *quote0* translates values back into terms. Its definition is (partially) shown in figure 4.25. *quote0* is a small wrapper for *quote*, which performs the actual translation. For most values, this translation is trivial. For neutral values (*VNeu*), quotation is delegated to the *neutralQuote* function. Pi-type values (*VPi*) and lambda abstraction values (*VLam*) carry functions, which are quoted by applying them to fresh *Quote* variables (lines 9 and 12 respectively), and quoting the result. These

```
1   csubst : Nat → ITerm → CTerm → CTerm
2   csubst n r (CInf e)   = CInf (isubst n r e)
3   csubst n r (CLam e) = CLam (csubst (n+1) r e)

4   isubst : Nat → ITerm → ITerm → ITerm
      ⋮
5   isubst n r (IPi s t)      = IPi (csubst n r s) (csubst (n+1) r t)
6   isubst n r (IBound m) = if (m ≡ n) then r else IBound m
7   isubst n r (IFree x)      = IFree x
8   isubst n r (IAnn e t)     = IAnn (csubst n r e) (csubst n r t)
      ⋮
```

Figure 4.24: Substitution of bound variables.

variables are indexed with the abstraction level at which the binding occurs, such that when the quotation function reaches that variable, it can calculate the corresponding De Bruijn index (line 23).

Timed types

Now we come to the implementation of timed types. Figure 4.26 shows the *itype* and *ctype* clauses for the rules of figure 4.11, except for TApp and TAppL (which are implemented in the *unify* function we will discuss below). The *itype* clauses are again pretty much direct translations of the rules, and they are implemented in the same manner as the rules we discussed above, so we won't elaborate on them. We will discuss the *ctype* clause, because it combines the rules TChk and TChkL, and it is the first case where moments have to be compared.

Both TChk and TChkL allow type checking of inferable terms. They extend the Chk rule, implemented in figure 4.22 in lines 25 to 27. Which of TChk and TChkL is applicable is determined by the inferred type: TChk applies if it is a timed type (line 40), and TChkL applies otherwise (line 44). If TChk applies, the first requirement is that the underlying data types are equal, which is checked with the statement *valEq vts vtt* (*valEq* is defined in figure 4.22). The next requirement is that the moments in the inferred type are earlier than those of the expected type. This is checked with the case statement in line 41. The function *comp* performs the comparison, and we will discuss it below. Line 42 shows that it may return a *CLTE* value, which indicates that the first argument of *comp* is smaller than or equal to the second argument, by some amount. The final requirement for TChk is that the sequences are of equal length, which is the case when the differences k and l are equal (*valEq vkt vlt*).

For rule TChkL there are two requirements: the inferred type has to be equal to the expected underlying data type (*valEq vts vtt*), and the expected sequence should have only one element, such that its first and last moments are equal (*valEq vkt vlt*).

Finally, we turn to the application rules: App from figure 4.7, and TApp and TAppLF from figure 4.11. As we explained above, they are handeld by the same *itype* clause (line 14 in 4.22), which infers the function type, and calls *unify* to check the argument type and compute the return type. The definition of *unify* is shown

```
1   quote0 : Result Value → Result CTerm
2   quote0 rv = do v ← rv
3                  quote 0 v

4   quote : Nat → Value → Result CTerm
5   quote n (VNeu v)  = do e ← neutralQuote n v
6                          return (CInf e)
7   quote n (VStar)   = return (CInf IStar)
8   quote n (VPi v f) = do s  ← quote n v
9                          t  ← f (VNeu (NFree (Quote n)))
10                         t' ← quote (n + 1) t
11                         return (CInf (IPi s t'))
12  quote n (VLam f) = do v ← f (VNeu (NFree (Quote n)))
13                        e ← quote (n + 1) v
14                        return (CLam e)
      ⋮

15  neutralQuote : Nat → Neutral → Result ITerm
16  neutralQuote n (NFree x) =
17    return (boundfree n x)
18  neutralQuote n (NApp vf vx) =
19    do x ← quote n vx
20    f ← neutralQuote n vf
21    return (IApp f x)
      ⋮

22  boundfree : Nat → Name → ITerm
23  boundfree n (Quote k) = IBound (n − k − 1)
24  boundfree n x         = IFree x
```

Figure 4.25: Quotation of values.

```
itype n D G (ISeq t l k) =
  do ctype n D G l VNat
     ctype n D G k VNat
     ctype n D G t VStar
     t' ← ceval D [] t
     case t' of
       VNat ⇒ return VStar
       _    ⇒ Left "type is not Nat"
itype n D G (ISCons x xs) =
  do txs ← itype n D G xs
     case txs of
       VSeq vt vl vk ⇒ do ctype n D G x (VSeq vt (VS vk) (VS vk))
                          return $ VSeq vt vl (VS vk)
       _ ⇒ Left "SCons tail not of a timed type"
itype n D G (ISElim t l m mz ms k xs) =
  do ctype n D G t VStar
     vt ← ceval D [] t
     ctype n D G l VNat
     vl ← ceval D [] l
     ctype n D G m (VPi VNat (λ_ ⇒ return VStar))
     vm ← ceval D [] m
     vmz ← vapp vm VZ
     ctype n D G mz vmz
     ctype n D G ms
       (VPi  VNat
             (λvk ⇒ do d   ← vplus vk vl
                       mk  ← vapp vm vk
                       msk ← vapp vm (VS vk)
                       pi1 ← return (λ_ ⇒ return msk)
                       pi2 ← return (λ_ ⇒ return $ VPi mk pi1)
                       return $ VPi (VSeq vt d d) pi2))
     ctype n D G k VNat
     vk ← ceval D [] k
     vd ← vplus vk vl
     ctype n D G xs (VSeq vt vl vd)
     vapp vm (VS vk)


ctype n D G (CInf e) (VSeq vtt vkt vlt) =
  do s ← itype n D G e
     case s of
       VSeq vts vks vls ⇒ do valEq vts vtt
                             case (comp vks vkt, comp vls vlt) of
                               (Right (CLTE k), Right (CLTE l)) ⇒ valEq k l
                               _ ⇒ Left "timing mismatch"
       vts              ⇒ do valEq vts vtt; valEq vkt vlt
```

Figure 4.26: Type checking for timed types.

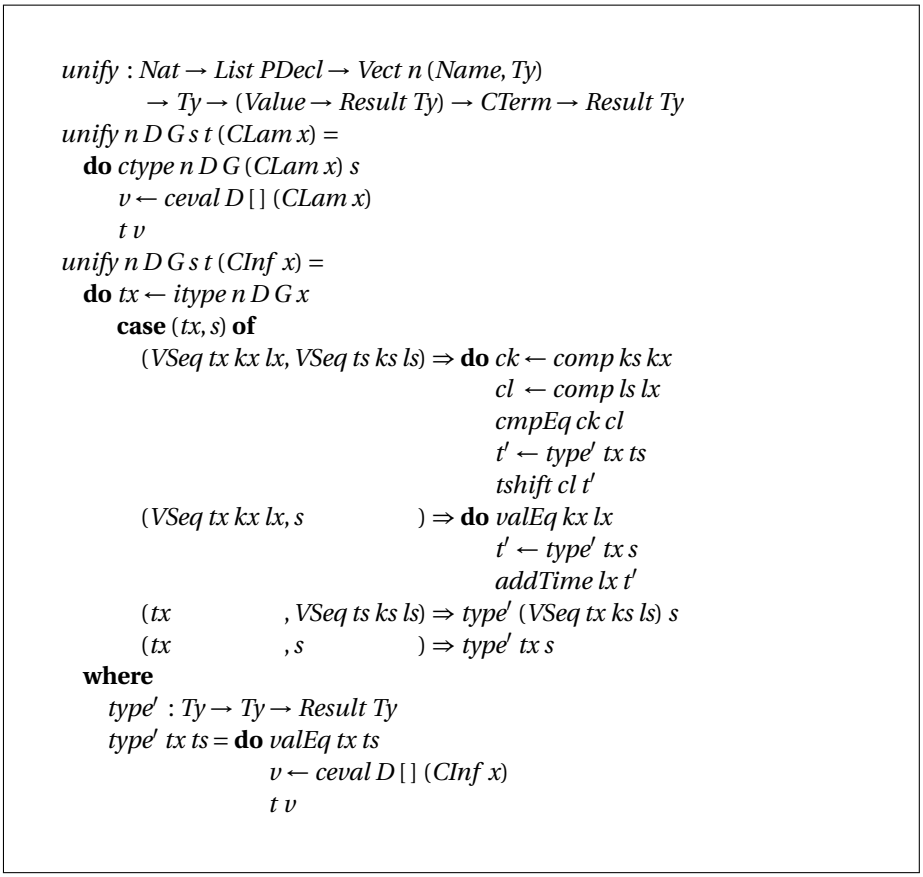
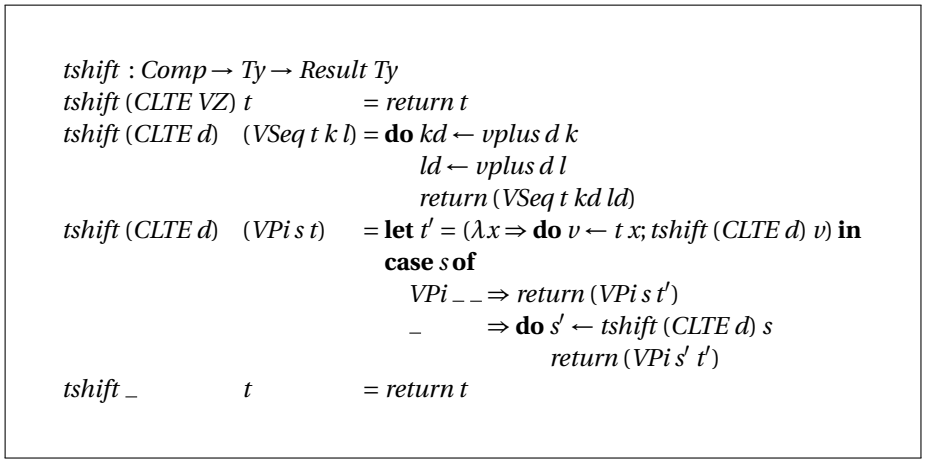


Figure 4.27: Unification of argument types.

in figure 4.27. The first three arguments of the function are the abstraction level, environment and context. The next arguments are the argument and return types that were embedded in the function type, and the argument of the application.

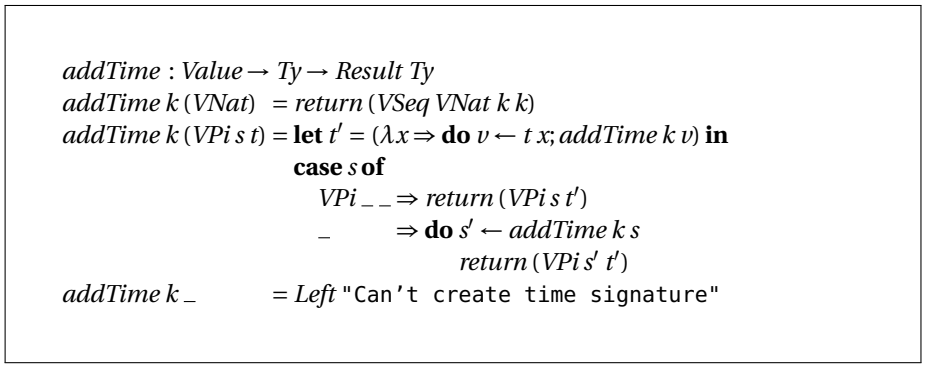
unify has two clauses: one for when the argument of the application is a lambda-abstraction, and one for when it is an inferable type. When the argument is a lambda-abstraction, it can never have a sequence type, so it is handled in accordance with the regular application rule App (lines 3 to 6). When the argument is an inferable term (line 7), its type is inferred, such that its timing can be inspected if it is a timed type. When the type has been inferred, a **case**-statement is used to select the rule that is applied. The first value of the case pattern is the inferred type of the argument, and the second value is the type that the applied function expects.

If both the argument type and the expected type are timed types, the rule TApp is applied (lines 10 to 14). The moments in the types are compared, giving two differences, *ck* and *cl*. If *ck* and *cl* are equal (line 12), the sequences are of the same length. The underlying data types are then checked for equality, and the return type t' is calculated (this is done with the function *type*′). Finally, the return type is adjusted to accomodate delays, with the function *tshift*. This function implements the adjustment function Δ from the typing rules (defined in figure 4.13), and is shown in figure 4.28.



```
1   tshift : Comp → Ty → Result Ty
2   tshift (CLTE VZ) t              = return t
3   tshift (CLTE d)  (VSeq t k l) = do kd ← vplus d k
4                                      ld ← vplus d l
5                                      return (VSeq t kd ld)
6   tshift (CLTE d)  (VPi s t)    = let t' = (λx ⇒ do v ← t x; tshift (CLTE d) v) in
7                                    case s of
8                                      VPi _ _ ⇒ return (VPi s t')
9                                      _       ⇒ do s' ← tshift (CLTE d) s
10                                                  return (VPi s' t')
11  tshift _        t               = return t
```

Figure 4.28: The time shifting function.



```
1   addTime : Value → Ty → Result Ty
2   addTime k (VNat)   = return (VSeq VNat k k)
3   addTime k (VPi s t) = let t' = (λx ⇒ do v ← t x; addTime k v) in
4                         case s of
5                           VPi _ _ ⇒ return (VPi s t')
6                           _       ⇒ do s' ← addTime k s
7                                        return (VPi s' t')
8   addTime k _         = Left "Can't create time signature"
```

Figure 4.29: This function adds time signatures to untimed types.

If the argument type of the application is a sequence, but the expected type is not (line 15), it might be possible to apply TAppLF. If the sequence contains only one element (*valEq kx lx*), then the underlying type of the sequence is compared to the expected type. If they are equal, the return type t' is calculated, and an attempt is made to add timing to it with the function *addTime*, which implements the function Θ from the rules (also defined in figure 4.13), shown in figure 4.29.

If the argument type is not a timed type while the function expects one (line 18), it might be possible to apply the rule TChkL (figure 4.11) to the argument, before applying the regular application rule (figure 4.7). This means the argument type *tx* is turned into a timed type, after which it is compared to the expected type *s*, and the return type is again calculated.

Finally, if neither the argument type nor the expected type are timed, then the regular application rule from figure 4.7 may be tried. This means checking for equality of the types, and calculating the return type.

As we explained in section 4.2.1, and as we saw in the implementation above, the rules for timed types rely on comparisons between natural numbers within the language. Currently, these comparisons are performed by a crude function *comp* that encodes the axioms in figure 4.12. Another way to view *comp* is that it essentially performs a "smart" subtraction, which takes in to account neutral values and some

```
1   data Comp : Type where
2     CLTE : Value → Comp
3     CGT  : Value → Comp

4   retGT k = return (CGT k)
5   retLTE k = return (CLTE k)

6   comp : Value → Value → Result Comp
7   comp x y  =  comp' x y
8             <|> case x of
9                   VNeu x' ⇒ do (xa, xb) ← isPlus x'
10                               x'' ← vplus xb xa
11                               comp' x'' y   -- (a+b ⩽ y) → (b+a ⩽ y)
12                  _        ⇒ Left "comparison failed"

13  comp' : Value → Value → Result Comp
14  comp' VZ     y = retLTE y
15  comp' (VS x) y = do d ← comp x y
16                      case d of
17                        CGT k      ⇒ retGT (VS k)     -- (x > y) → (Sx > y)
18                        CLTE VZ    ⇒ retGT (VS VZ)    -- (x = y) → (Sx > y)
19                        CLTE (VS k) ⇒ retLTE k        -- (x < y) → (Sx ⩽ y)
20                        CLTE _     ⇒ Left "comparison failed"
    ⋮
```

Figure 4.30: Comparison of natural numbers

arithmetical properties of the functions *plus* and *pred* that are defined in the language itself. Figure 4.30 shows a part of the definition of the comparison. We refer to appendix C for the full listing.

The result of a comparison is a *Result Comp*, where *Comp* is defined on lines 1 to 3 of the figure. If the first argument of *comp* is greater than the second it returns *Right* (*CGT k*), if it is less it returns *Right* (*CLTE k*), where *k* is the difference between the two arguments. It may be that the two values cannot be compared (when they are two distinct variables for instance), in which case *comp* returns a *Left* value.

The comparison is split into two cases. The first case (line 7) compares the arguments as they are given. The second case (lines 8 to 12) tries to apply the commutative property of addition to the first argument: The function *isPlus* tests if the argument is indeed the addition defined in figure 4.13, and returns the two operands if it is. The operands are then added together again in reverse order in line 10, and the result is compared to the second argument in line 11. Both cases call the secondary function *comp'*, which handles all other axioms. This separation prevents the function from infinitely commuting the operands of the addition.

To give a flavour of the rest of the comparison, lines 13 and further show a part

of *comp*′. Line 14 for example implements the axiom $\forall y\,.\,Z \leqslant y$, and lines 17 to 20 implement the axioms that say something about the case when the first argument is a successor of something. To determine the relation between some $S\ x$ and y, x and y are compared, resulting in the four different cases. Notice that the comparison fails if x is less than y by an undetermined amount (a neutral value), because that amount could be zero, making $S\ x$ larger than y.

4.4 Examples

We will now discuss some examples that can be (and have been) checked and evaluated in our implementation. First, we have a single value at some moment:

$$1 : \mathbb{N}\langle 1\rangle$$

This may be represented with

```
IAnn (CInf (IS (CInf IZ)))
     (CInf (ISeq (CInf INat)
                 (CInf (IS (CInf IZ)))
                 (CInf (IS (CInf IZ)))))
```

If we infer the type of this expression, we get $\mathbb{N}\langle 1\rangle$:

```
VSeq VNat (VS VZ) (VS VZ)
```

and if we evaluate the expression, we get

```
VS VZ
```

To define a value for every moment, we can build an abstraction:

$$s \equiv \lambda n.n\ :\ \forall n{:}\mathbb{N}.\mathbb{N}\langle n\rangle$$

Notice that this function converts the moment (a non-timed value) to a timed value (this is justified by the rule TChkL). The expression is represented as

```
(IAnn (CLam (CInf (IBound 0)))
     (CInf (IPi (CInf INat)
                (CInf (ISeq (CInf INat)
                            (CInf (IBound 0))
                            (CInf (IBound 0)))))))
```

With this definition in the environment, we can use it to get a value at some particular moment:

$$s\,3$$

which is represented with

```
IApp (IFree (Global "s"))
     (CInf (IS (CInf (IS (CInf (IS (CInf IZ)))))))
```

Accordingly, the type of this expression infers as $\mathbb{N}\langle 3\rangle$:

```
VSeq VNat (VS (VS (VS VZ))) (VS (VS (VS VZ)))
```

The expression itself evaluates to

$$VS\,(VS\,(VS\,VZ))$$

Of course, it is also possible to create functions that accept timed values. Consider the following "identity circuit":

$$id \equiv \lambda n\,.\,\lambda x\,.\,x\;:\;\forall n{:}\mathbb{N}\,.\,\mathbb{N}\langle n\rangle \to \mathbb{N}\langle n\rangle$$

represented as

$$\begin{array}{l}(IAnn\,(CLam\,(CLam\,(CInf\,(IBound\,0))))\\
\quad(CInf\,(IPi\,(CInf\,INat)\\
\qquad(CInf\,(IPi\,(CInf\,(ISeq\,(CInf\,INat)\\
\qquad\qquad(CInf\,(IBound\,0))\\
\qquad\qquad(CInf\,(IBound\,0))))\\
\qquad\quad(CInf\,(ISeq\,(CInf\,INat)\\
\qquad\qquad(CInf\,(IBound\,1))\\
\qquad\qquad(CInf\,(IBound\,1)))))))))\end{array}$$

As this definition shows, the data representations become quite cumbersome. Since they can easily be derived from the mathematical statements, we will omit them in the following, and show only the data representations of the inferred types and the evaluation results.

We can take the *id* function, and apply it to values from different moments. For example, we can apply the function to a value from moment 2:

$$id\;2\;(3:\mathbb{N}\langle 2\rangle)$$

The type checker infers the type of this expression ($\mathbb{N}\langle 2\rangle$) as

$$VSeq\;VNat\;(VS\,(VS\,VZ))\;(VS\,(VS\,VZ))$$

and the expression evaluates to 3:

$$VS\,(VS\,(VS\,VZ))$$

We can also apply *id* to a value from a moment that is too early, like

$$id\;2\;(3:\mathbb{N}\langle 1\rangle)$$

(notice that the function will expect a value from moment 2, but gets one from moment 1). We get the same type and evaluation result as above, because the checker assumes the value will be delayed. However, if we apply *id* to a value from a moment that is too late, like

$$id\;2\;(3:\mathbb{N}\langle 3\rangle)$$

the checker will assume any other inputs (in this case none) are delayed, and infers the type $\mathbb{N}\langle 3\rangle$:

$$VSeq\;VNat\;(VS\,(VS\,(VS\,VZ)))\;(VS\,(VS\,(VS\,VZ)))$$

Of course, the evaluation result again remains the same.

The system has a built-in definition of addition, which is untimed:

$$(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

By applying the TAppLF rule this function can be used as a combinational circuit, such that we can define a synchronous adder with delayed inputs. For example:

$$dplus \equiv \lambda n.\lambda x.\lambda y.x + y : \forall n{:}\mathbb{N}.\mathbb{N}\langle 1+n\rangle \to \mathbb{N}\langle n\rangle \to \mathbb{N}\langle 2+n\rangle$$

The arguments x and y are timed, such that the checker will turn the addition into a timed function. We can again apply the function to some values:

$$dplus\ 2\ 3\ 1$$

The checker will turn the natural numbers into timed values. Because $n = 2$, the result will come at moment 4, and indeed the inferred type is $\mathbb{N}\langle 4\rangle$:

$$VSeq\ VNat\ (VS\ (VS\ (VS\ (VS\ VZ))))\ (VS\ (VS\ (VS\ (VS\ VZ))))$$

The expression evaluates to 4:

$$VS\ (VS\ (VS\ (VS\ VZ)))$$

The order of the arguments with respect to their moments does not matter. We can switch around the timings in *dplus* and still have it pass the checker:

$$dplus' \equiv \lambda n.\lambda x.\lambda y.x + y : \forall n{:}\mathbb{N}.\mathbb{N}\langle n\rangle \to \mathbb{N}\langle 1+n\rangle \to \mathbb{N}\langle 2+n\rangle$$

In the current system, we do enforce causality, such that the arguments never come later than the result. The following definition will therefore *not* pass the checker:

$$dplus'' \equiv \lambda n.\lambda x.\lambda y.x + y : \forall n{:}\mathbb{N}.\mathbb{N}\langle n\rangle \to \mathbb{N}\langle 2+n\rangle \to \mathbb{N}\langle 1+n\rangle$$

Aside from constant delays in the timed types, we can also use (non-timed) variables:

$$delay \equiv \lambda n.\lambda m.\lambda x.x : \forall n{:}\mathbb{N}.\forall m{:}\mathbb{N} \to \mathbb{N}\langle n\rangle \to \mathbb{N}\langle m+n\rangle$$

Applying this function to different values of m gives us different moments for the result type:

$$delay\ 1\ 1\ 4$$

gives us the inferred type $\mathbb{N}\langle 2\rangle$:

$$VSeq\ VNat\ (VS\ (VS\ VZ))\ (VS\ (VS\ VZ))$$

while

$$delay\ 1\ 2\ 4$$

gives us $\mathbb{N}\langle 3\rangle$:

$$VSeq\ VNat\ (VS\ (VS\ (VS\ VZ)))\ (VS\ (VS\ (VS\ VZ)))$$

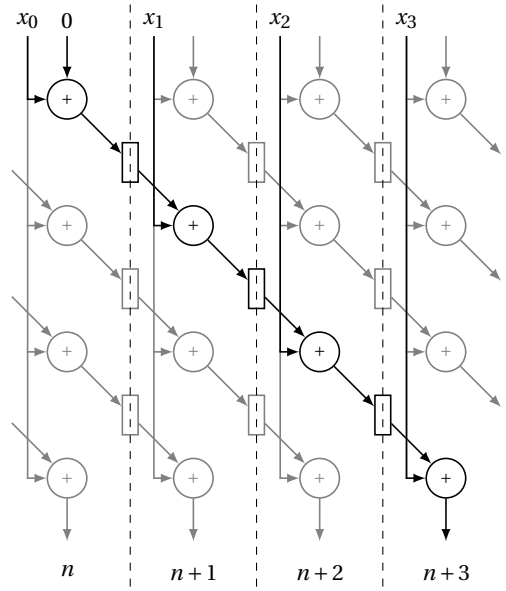
Up to now we used the singular timed types. Of course, we can also form sequences, like the following:

$$seq \equiv sCons\ 3\ (sCons\ 3\ (sCons\ 3\ (3 : \mathbb{N}\langle 0\rangle)))$$

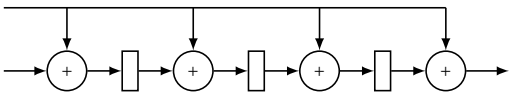
This sequence contains four elements, of which the first (in time) occurs at moment 0. This means the sequence has type $\mathbb{N}\langle 0..3\rangle$, and indeed the inferred type is $\mathbb{N}\langle 0..3\rangle$:

$$VSeq\ VNat\ VZ\ (VS\ (VS\ (VS\ VZ)))$$

Next we will use *sElim* to define a summation over the sequence elements. Figure 4.31 shows an example of the circuits this definition will represent. Figure 4.31a shows a timeline representation, while 4.31b shows the hardware realisation. Note that the realisation shows a single wire for the sequence, which corresponds to the notion that sequences live at different moments at a single location.



(a) Timeline representation.



(b) Hardware realisation.

Figure 4.31: Summation of a sequence with four elements.

Before we invoke *sElim*, we must first determine the "motive", the function that defines the types of the accumulation argument and the result for each iteration. Figure 4.31a shows that the last iteration receives the result from moment $n+2$ and returns in moment $n+3$. According to the typing rule of *sElim*, the adders have a type of the form

$$\forall l{:}\mathbb{N}.\,\mathbb{N}\langle k+l\rangle \to m\,l \to m\,(S\,l)$$

(where m is the motive), such that for the last iteration (where $l=3$) we have that $m\,3$ should evaluate to $\mathbb{N}\langle n+2\rangle$ and $m\,4$ should evaluate to $\mathbb{N}\langle n+3\rangle$. A possible definition for the motive is

$$m \equiv \lambda l.\,\mathbb{N}\langle (pred\ l)+n\rangle$$

The summation of a sequence *xs* of natural numbers can then be defined as

$$\begin{aligned} sum &\equiv \lambda n\,.\,\lambda d\,.\,\lambda xs\,.\,sElim\ \mathbb{N}\ n\ m\ 0\ (\lambda l\,.\,\lambda x\,.\,\lambda acc\,.\,acc{+}x)\ d\ xs \\ &: \forall n{:}\mathbb{N}\,.\forall d{:}\mathbb{N}\,.\,\mathbb{N}\langle n\,..\,d{+}n\rangle \to \mathbb{N}\langle d{+}n\rangle \end{aligned}$$

The elimination works on a sequence of type $\mathbb{N}$, starting at moment n, with the motive m we have just defined. The initial value is 0, and the values of the following iterations are determined with

$$(\lambda l\,.\,\lambda x\,.\,\lambda acc\,.\,acc{+}x)$$

(remember that *sElim* also provides the iteration number as an argument, in this case l). The elimination acts on the sequence *xs* with length d (plus one).

We can now try to sum the sequence we defined earlier:

$$tot \equiv sum\ 0\,3\ seq$$

The inferred type of this expression is $\mathbb{N}\langle 3\rangle$:

$$VSeq\ VNat\ (VS\ (VS\ (VS\ VZ)))\ (VS\ (VS\ (VS\ (VZ))))$$

The value of the sum should be $3 \times 4 = 12$, and indeed the expression evaluates to 12:

$$VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ VZ))))))))))$$

The sequence used above was defined to start at moment 0, but we can also construct a more generic sequence:

$$gseq \equiv \lambda n\,.\,sCons\,3\ (sCons\,3\ (sCons\,3\ (3 : \mathbb{N}\langle n\rangle)))$$

A sum over this sequence at any moment should have the same result, but a different type. Indeed,

$$sum\ 12\,3\ (gseq\ 12)$$

has the type $\mathbb{N}\langle 15\rangle$:

$$\begin{aligned} &VSeq\ VNat \\ &\quad (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ VZ)))))))))))))) \\ &\quad (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ VZ)))))))))))))) \end{aligned}$$

and evaluates to 12:

$$VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ (VS\ VZ))))))))))$$

5 · Discussion, conclusions, future work

In chapter 4 we introduced a new system of timed types, together with an implementation and some examples. In this section we will discuss some global properties of the system, followed by a conclusion and some suggestions for future work.

5.1 Discussion

In section 4.4 we showed some examples of what our system can do. In essence, we can describe parallel and serial compositions of combinational circuits interspersed with registers, where the timing behaviour of the circuits is specified in their types. In other words, our we can combine multi-cycle circuits in a pipelined and/or parallel fashion, and have the type system check whether this composition is allowed. In that sense our system supports a way of hardware design that is more focused on multi-cycle functionality than on single-cycle behaviour. In addition, the system we developed fits in the approach that is used by CλaSH, the functional hardware specification system developed in the chair CAES.

However, our system also has several limitations. First of all, to realise an initial implementation we have omitted some features that are common among functional languages, like pattern matching, general recursion, and user definable types. As a dependently typed language our system is limited as well, because it has no way to assist the type checker in terms of theorem proving (for example with user-supplied proofs of equality). Although these deficiencies are not essential limitations of our language, they make the system cumbersome to use, and restrict the number of functions we can describe substantially. To give an example of how limited the system is: It is currently impossible to define the *tail*-function for vectors. The system cannot just return the tail via pattern matching, and recursion over the complete sequence does not work because the type checker cannot reason about decisions. That is, if we want to implement the tail function with the vector eliminator, we need a function that decides if it should keep appending elements of the list. The result type of the function depends on this choice, and because the type checker cannot reason about this choice, it will reject the expression. This is a consequence of the omissions we mentioned above, not a fundamental problem.

More specific to our language is the fact that the focus of this project was on the timed types themselves, and not on how specifications with timed types translate to hardware. As a result, the interpretation of the language as a hardware specification language is not yet well defined. A part of this can be attributed to the lack of distinction between clocks, time values and circuit parameters. Consider for example the

expression

$$\lambda n.\lambda m.m : \forall n{:}\mathbb{N}.\forall m{:}\mathbb{N}.\mathbb{N}\langle n+m\rangle$$

In one interpretation, we regard n as a clock, and m as a circuit parameter. In that case, the function returns a constant value (parameter m). However, we can also regard n as the parameter, while m is the clock. In that case, the output changes over time, and we can regard the circuit as unrealisable[1] (though the expression may be useful for simulation purposes). If the function would return a constant, either interpretation would result in a realisable circuit.

Related to the above property is that we silently assume that non-timed arguments remain constant relative to a clock argument. For example, if we have a delay function

$$\lambda n.\lambda m.\lambda x.x : \forall n{:}\mathbb{N}.\forall m{:}\mathbb{N}.\mathbb{N}\langle n\rangle \to \mathbb{N}\langle n+m\rangle$$

we regard n as the clock, and m as a circuit parameter that defines the number of registers between the in- and output. We assume that m remains constant for different n. If m varies with n (which the system allows), the corresponding architecture varies over time, and is therefore unrealisable (we assumed that realisable architectures are time-invariant).

Another aspect of the non-welldefinedness of interpretation of the language is caused by a lack of checks on the time expressions. Time variant architectures can not only be constructed via non-timed function arguments, but also via the time expressions. The system currently has no checks on whether all time delays are constant, such that we can typecheck for instance the following function:

$$\lambda n.\lambda x.x : \forall n{:}\mathbb{N}.\mathbb{N}\langle n\rangle \to \mathbb{N}\langle n+n\rangle$$

This function requires ever increasing delays, and is therefore unrealisable.

Folding behaviour of sequences

In section 4.2 we introduced sequences as a means to restrict values that are distributed in time to one location in space. The framework we constructed for these sequences induces a few interesting observations.

The first observation is that the way we have defined our timed sequences determines the folding direction of the eliminator: We have defined the sequences such that the newest value is placed at the head, and older values are placed in the tail. This order suggests that the results in the fold should propagate from the tail towards the head, which corresponds to a right fold. Conversely, if the sequences had been defined with the opposite direction of time (oldest value at head), it would have been more appropriate to have the results propagate towards the tail with a left fold.

To allow the result types in the fold to depend on the position in the sequence, our current eliminator requires a motive which maps natural numbers to types. The folded function is then required to have a type of the following form (where m is the motive, and n and ρ are arguments of *sElim*):

$$\forall l{:}\mathbb{N}.\rho\langle n+l\rangle \to m\,l \to m\,(S\,l)$$

The main reason why we chose this form is that it allows both the timing as well as the underlying data types of the intermediate results to vary along the fold. We can

[1] It might be tempting to view the expression as a counter circuit, but then it is the question how the ever increasing clock count is represented in hardware.

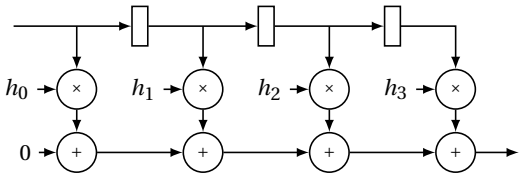


Figure 5.1: An FIR filter architecture.

therefore interpret l to be an offset in time, like for example in the type

$$\forall l{:}\mathbb{N}\,.\,\rho\langle n{+}l\rangle \to \rho\langle n{+}l\rangle \to \rho\langle n{+}(S\,l)\rangle$$

but we can also use it to increase for instance the vector size for each intermediate result:

$$\forall l{:}\mathbb{N}\,.\,\rho\langle n{+}l\rangle \to (\mathit{Vect}\;\sigma\;l)\langle n{+}l\rangle \to (\mathit{Vect}\;\sigma\;(S\,l))\langle n{+}(S\,l)\rangle$$

Although the current form of the function type allows for some freedom in the choice of result types, it does have its limitations. The type of the second argument ($\rho\langle n{+}l\rangle$) says that the argument has moment $n{+}l$, where l can be any number. Under the assumption of causality the intermediate results can be produced no earlier than the second argument, which means their moment has to be defined in terms of l. As an example, we shall attempt to use *sElim* to define an FIR filter, the architecture of which is shown in figure 5.1. We define the function used during the folding to be one multiplier and one addition, and we assume that we can determine the coefficients from the position in the sequence. Because the architecture has no delays between the additions, all intermediate results appear in the same moment as the latest sequence element, namely $n{+}N$. This suggests we define the motive of the fold as $\lambda l\,.\,\mathbb{N}\langle n{+}N\rangle$, such that the type of the function becomes

$$\forall l{:}\mathbb{N}\,.\,\mathbb{N}\langle n{+}l\rangle \to \mathbb{N}\langle n{+}N\rangle \to \mathbb{N}\langle n{+}N\rangle$$

This type violates our assumption of causality ($n{+}l \leqslant n{+}N$ does not hold for all l), and using it for our fold will result in a type error.

It seems that the above problem does not need to occur when sequences are defined with the reverse timing order. In that case we can use a left fold, and the function type takes the form

$$\forall l{:}\mathbb{N}\,.\,\rho\langle n'{-}l\rangle \to m\,(S\,l) \to m\,l$$

where n' is the latest moment of the sequence. The sequence element moment ($n'{-}l$) now decreases with l, to match the time direction of the sequence. Because $n'{-}l \leqslant n'$ for every l, there is no problem in defining a fold with a combinational folding function, such as the FIR filter discussed above. The corresponding type for the FIR filter function would be

$$\forall l{:}\mathbb{N}\,.\,\mathbb{N}\langle n'{-}l\rangle \to \mathbb{N}\langle n'\rangle \to \mathbb{N}\langle n'\rangle$$

The eliminators above both use the motive to define the time signature of the function. If this signature remains the same for every instance of the function, it is also possible to have only the underlying data type depend on the position in the sequence. The general type of such a function could for example be

$$\forall l{:}\mathbb{N}\,.\,\rho\langle k\rangle \to (m\,l)\langle k'\rangle \to (m\,(S\,l))\langle k''\rangle$$

where k, k' and k'' are single moments not depending on l. With such solutions, the type system cannot compute the moment in which the final result will be produced directly from the motive. Instead, it needs to derive it from the time signature of the function. For instance, consider a function with the type

$$\forall l{:}\mathbb{N}\,.\,\mathbb{N}\langle n\rangle \to (m\,l)\langle n\rangle \to (m\,(S\,l))\langle n{+}2\rangle$$

The moment of the final result can be computed from the delay between each intermediate result, and the moment of the initial value. In this case the delay is 2, so if the initial value is produced at moment n, the final result is produced at moment $n{+}N{*}2$ (where N is the length of the sequence). In a similar way, the system would also need to derive the delays between the sequence input and each function instance.

Comparison with [Ott13]

We have already mentioned that the notion of timed types was previously explored in [Ott13] with the introduction of the system λ_t. Although our syntax is quite similar, the underlying systems are very different. We will now discuss some of the differences, where we will use λ_m to denote our own system (the subscript m comes from "moments").

A notable difference between λ_t and λ_m is in the interpretation of the timed types of values and functions. In λ_t, values are initially assumed to exist at *any* moment in time. That is, an expression like $x : \tau\langle t{+}a\rangle$ should be read as $x : \forall t{:}\mathbb{N}.\tau\langle t{+}a\rangle$, and should be interpreted as saying that the value x is available for any moment later than (and including) a. Functions are similarly quantified, and when we apply a function to a timed function, the result is also quantified over time:

$$\begin{aligned} x &: \forall t{:}\mathbb{N}.\tau\langle t\rangle \\ f &: \forall t{:}\mathbb{N}.\tau\langle t\rangle \to \sigma\langle t+2\rangle \\ fx &: \forall t{:}\mathbb{N}.\sigma\langle t+2\rangle \end{aligned}$$

During typechecking, a set of constraints is gathered which limits the moments in which values are available. These constraints form a set of equations, which is solved to prove the correctness of the expression.

The perspective in λ_m is that values live at a *specific* moment in time. In describing a circuit, we may quantify over a set of possible values to form inputs, and we may quantify over time to allow the values to change over time. Because values live at a single moment, the application of a function to a timed value denotes a specific computation distributed over a specific set of moments, and is therefore not quantified over time. In contrast to λ_t, λ_m determines the correctness of the synchronisation directly during typechecking: The system always knows in which moment a value is expected and in which moment it is actually produced, and therefore it can immediately accept or reject the expression.

Because λ_m is based on dependent types, descriptions of circuits can be flexible. For example, the time expressions are regular terms of the language. Together with dependent function types, this allows us to parametrise circuits in the time domain. λ_t is based on the simply typed lambda calculus, and adds time expressions as a distinct syntactical group. No interaction between the arguments and types of functions is possible. This also has its consequences for sequences: in λ_m, sequences are first class. With the aid of dependent function types, we can construct functions that take sequences of arbitrary length, to describe parametrised architectures. In λ_t,

sequences are not first class. Functions that act on sequences are constructed from functions with a separate argument for each sequence element. Functions therefore only take sequences of fixed length, which is not very practical. Furthermore, it does not allow us to go over sequences with fold-like constructions.

Finally, λ_t requires function arguments to be placed in order of increasing time. The following type is not allowed in λ_t, while it is no problem in λ_m:

$$f : \tau\langle t+2\rangle \to \tau\langle t\rangle \to \tau\langle t+2\rangle$$

5.2 Conclusion

In this project, we set out to investigate if we can regard timed types as a specific form of dependent types. To this end, we successfully constructed a type system in which the type of a value depends on the moment in which it lives, and in which we can abstract computations over these moments, to form circuit descriptions. The terms with which we express moments are not different from the terms with which we express any other computation in the language, so in that sense, we may say that timed types depend on values, and we may answer the question positively.

We did have to make additional provisions to allow for register inference. Most importantly, we had to add a mechanism to automatically prove the inequality of time expressions. Because time values can depend on arbitrary computations, this involves proving properties of such computations. This is not always a trivial task. One would like to avoid doing this manually, and more so in a hardware description language. Ideally this would be solved by the addition of an automated theorem prover, which in itself forms another area of research.

Although the system we developed is limited in its expression (certain useful functions cannot be described), it does allow flexible circuit descriptions. The dependently typed nature allows circuits to be parametrised with regards to its behaviour in time, such as parametrised delay lines. This enables clear and concise descriptions and promotes code reuse. This incidentally also allows time-variant systems, which at the least are useful for simulation purposes (e.g., signal generators that convert time to input values), but which may also lead to more intuitive circuit descriptions (e.g., for circuits that do different computations on odd and even cycles).

5.3 Future work

As we have seen, the system we presented in this thesis is not yet practical as a hardware description language. Its shortcomings suggest several topics for future work.

Type safety. Although we believe our system to obey the properties of subject reduction (type preservation) and progress, we have no formal proof of this.

Pattern matching, implicit arguments, and general recursion. We suspect that these constructs will mostly reflect their dependently typed counterparts, for which solutions are available.

Sequences. In section 5.1 we showed that the definition of sequences affects the folding behaviour. More research into the possibilities and their consequences is needed.

Feedback circuits. We currently do not have a mechanism to describe feedback, where output values are fed back into the circuit that produces them (such that output values may depend on earlier output values). This should not be confused with general recursion, where a function calls itself (a subtle difference).

Time variance. Currently, we assume that circuits are time invariant. However, the system does allow time variant descriptions. There are two aspects to this problem. An obvious one is that we want to be able to reject descriptions of time variant architectures, in which the actual circuit changes over time. Another aspect is that we do want to describe time variant circuit *behaviour*. One can think of counter circuits, or circuits that switch operations on input samples periodically.

User-definable and bit-representable types. Currently, we use the natural numbers as a surrogate for bit-representable types. At some point, they should be replaced by actual bit-representable types. It is also desirable to allow users to define their own types, in terms of timed types.

A · Languages

A.1 IDRIS

Idris ([Bra13], official website `http://idris-lang.org`) is a general purpose dependently typed programming language. It tries to combine the benefits of precise typing (dependent types) with high level language features. Idris takes Haskell as its main influence, and syntactically they are quite similar. Functions look like Haskell functions (including **let** bindings, pattern matching, and **where**-clauses), and data declarations look like Haskell declarations for (generalised) algebraic data types. Idris also supports do-notation and typeclasses, although at the moment of writing there is no equivalent for Haskell's **deriving** statement. There are a couple of primitive types, including *Int*, *Float*, *String*, and *Char*, and the standard library also defines a number of types, including *Bool* and *Nat*.

One of the goals of Idris is to be a practical language, and there are a couple of features that support this goal: In order to support the construction of EDSL's (Embedded Domain Specific Languages), it is possible to extend the syntax (more or less like preprocessed macros), and to overload parts of it. There is also a foreign function interface, allowing Idris programs to communicate with external libraries, and there are several different compilation targets (including Javascript and LLVM). Idris is eager by default, to make reasoning about performance easier.

In the context of this thesis, we are mostly interested in Idris as a dependently typed language. We will therefore not go into the details of the features mentioned above, but we will focus on what basic dependently typed programming in Idris looks like, both in terms of actual programs, and in terms of proofs. We also tried to keep to global programming patterns which are not specific to Idris, because Idris is a research vehicle, and not a fully mature language. It has happend on multiple occasions during this project that certain aspects of the language were changed, leading to defective source code. We refer the reader to the Idris tutorial (available at `http://idris-lang.org`) for more specific information on the language. We assume the reader has some basic knowlede of functional programming in Haskell, and has read chapter 3 of this thesis.

A.1.1 *Basic Idris*

We will start with a basic description of dependent types in Idris, but before that we will define some simple types:

data *Bool* = *True* | *False*
data *Nat* = *Z* | *S Nat*

data *List a* = *Nil* | (::) *a* (*List a*)

We assume these structures need no explanation. Note that the cons constructor for lists is defined as the double colon, instead of the single colon. Unlike Haskell, Idris uses the single colon for type annotation.

Of course, we can define constants and functions for these types. For example the constant *two*:

two : *Nat*
two = *S* (*S Z*)

and the add function, which we can define as infix operator:

(+) : *Nat* → *Nat* → *Nat*
(+) *Z* *y* = *y*
(+) (*S x*) *y* = *S* (*x* + *y*)

The first lines of these definitions show type declarations, which are mandatory for every top level definition. Note the use of the single colon for type annotations.

In Haskell, terms and types are strictly separated notions. In Idris, types are first class citizens, and they have their own type, *Type*[1]. We can often use types in the same way we use terms. For example, we can ask Idris for the type of *Nat*, and it would return *Type*. We can also define constants of type *Type*, just as we did above for type *Nat*:

NatType : *Type*
NatType = *Nat*

Apart from the lack of arguments, this definition is no different from a function definition, and we can indeed also define proper functions that take and/or return types:

typefun : *Nat* → *Type* → *Type*
typefun Z *a* = *a*
typefun (*S k*) _ = *Nat*

Subsequently, we can evaluate types. The constant *NatType* evaluates to *Nat*, and the application *typefun Z Bool* evaluates to *Bool*. Both expressions are themselves of course of type *Type*. However, this does not mean that Idris has dynamic typing! Typechecking occurs only during compilation, and it is at this time that types are evaluated. In fact, Idris will try to erase the type information after typechecking in order to optimise the program. Because functions in general can be partial or non-halting, Idris will also do a conservative check for these properties before evaluating types, to make sure that the typechecker will always finish.

The type that results from the application of *typefun* to a term *k* and a type *t* (i.e., *typefun k t*), depends on the value of *k*. Because *k* is a term, and not a type, we call *typefun k t* a dependent type. It is still a type, and we can use it just as well to construct other types. For example, we can use it in a function type:

[1] *Type* is a bit more complicated than just a "type of types", but for the purposes of this thesis (and probably for most practical cases) it suffices to maintain this naive view. More information about *Type* can be found in [Bra13].

$$
\begin{aligned}
&f : (\mathit{typefun}\ Z\ \mathit{Bool}) \to \mathit{Nat} \\
&f\ \mathit{False} = 0 \\
&f\ \mathit{True}\ = 1
\end{aligned}
$$

Because type *typefun Z Bool* evaluates to *Bool*, the typechecker can determine that the function is allowed to match on the boolean constructors, and because in every clause the function returns a natural number, the whole function is typed correctly. We could use it as if it were a function from *Bool* to *Nat*. (Note that in this way, we can use constants like *NatType* as type aliases. In Haskell, one would write **type** *NatType* = *Nat*.)

Aside from the common function types like that of f, Idris also has dependent function types. They are a generalisation of the regular function type, and provide a powerful way to use dependent types: In a dependent function type, the result type may depend on the value of the argument of the function. For example, we can define a function whose argument is passed to *typefun* to compute the result type of that same function. To do that, we give the argument a name, and use that name in the result type:

$$
\begin{aligned}
&g : (n : \mathit{Nat}) \to \mathit{typefun}\ n\ \mathit{Bool} \\
&g\ Z\ \ \ \ = \mathit{True} \\
&g\ (S\ k) = k
\end{aligned}
$$

If we apply g to Z, the return type is *Bool*, and if we apply g to something else, the return type is *Nat*. The typechecker will be able to check this function, because the patterns of each clause give enough information to determine the respective result types. For the first clause n is equal to Z, such that *typefun n Bool* evaluates to *Bool*, which is also the type of *True*. For the second clause, n is equal to $S\ k$, such that *typefun n Bool* evaluates to *Nat*, which is the type of k. Because the types of the clauses are correct, we can conclude that the type of the function is correct, even though we have no definite value for n.

We can use g as any other function that takes a *Nat* as argument, provided of course that the result type matches that which the environment expects. For example, $1 + (g\ 2)$ would work because $g\ 2$ has type *Nat*, but $1 + (g\ 0)$ would not work, because $(g\ 0)$ has type *Bool*. In particular, if k is a variable, then $1 + (g\ k)$ would also not pass the typechecker, because it has not enough information to determine that $g\ k$ evaluates to *Nat*.

The mechanism of dependent function types can also be used to define data types, in a style similar to Haskell's generalised algebraic data types. An example that is often used is the vector type, which represents lists with a certain length. Vectors can be defined as follows:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Vect} : \mathit{Nat} \to \mathit{Type} \to \mathit{Type}\ \textbf{where} \\
&\quad \mathit{Nil} : \mathit{Vect}\ O\ a \\
&\quad (::)\ : a \to \mathit{Vect}\ k\ a \to \mathit{Vect}\ (S\ k)\ a
\end{aligned}
$$

The first line defines the type constructor, which takes an argument of type *Type* and an argument of type *Nat*, for example *Vect* 3 *Nat* (a vector with three elements of type *Nat*). The second and third lines define the data constructors. Similar to lists, there is an empty vector *Nil*, which has type *Vect* 0 a (a is a type variable), and a cons constructor (::) which takes an element of type a and a vector type *Vect k a*, and which produces a vector with type *Vect* $(S\ k)$ a. Notice that the type indicates that the length of a vector $x :: xs$ depends on the length of the vector xs.

Because the length of a vector is embedded in its type, we can use vector types to write more detailed specifications than with ordinary lists. Take for instance vector concatenation. In Idris, we can define this operation as follows:

$$
\begin{aligned}
&(++) : \mathit{Vect}\ n\ a \to \mathit{Vect}\ m\ a \to \mathit{Vect}\ (n+m)\ a \\
&(++)\ \mathit{Nil}\quad\ \ ys = ys \\
&(++)\ (x :: xs)\ ys = x :: (xs ++ ys)
\end{aligned}
$$

The type of this function shows not only that it takes two vectors to produce a third, but also that the length of this third vector is the sum of the lengths of the other two. Notice that the lengths are of type *Nat*, and the addition in the type declaration is therefore just the term-level addition. Because of the dependent types, it is not necessary to add special type-level constructs for the natural numbers.

To see why (++) passes the typechecker, we can again look at each clause individually. In the first clause, the first vector is *Nil*. This constructor is defined to have type $\mathit{Vect}\ 0\ a$, which allows us to conclude n is 0, and consequently, that the expected result type is $\mathit{Vect}\ (0+m)\ a$. Because the clause returns the second vector, of which we know nothing more than that it has type $\mathit{Vect}\ m\ a$, we are left to determine whether $\mathit{Vect}\ m\ a$ matches $\mathit{Vect}\ (0+m)\ a$. The latter type evaluates to $\mathit{Vect}\ m\ a$, so we can answer in the affirmative. A similar reasoning can be followed for the second clause: We can deduce that n is equal to some number $S\ k$, making the expected result type $\mathit{Vect}\ ((S\ k)+m)\ a$. The actual result type is $\mathit{Vect}\ (S\ (k+m))\ a$, and if we evaluate the expected result type, both become equal. We may therefore conclude that both clauses are well-typed.

In this example we have made use of the facts that $0+m$ evaluates to m, and that $(S\ k)+m$ evaluates to $S\ (k+m)$. We needed these facts in order to conclude that two types were equivalent. This means that if $(S\ k)+m$ would not have evaluated to $S\ (k+m)$, we would not have been able to establish this equivalence, and consequently, the well-typedness of the whole function. This poses a problem, because expressions do not always evaluate as conveniently as in our example. For instance, the expression $m+(S\ k)$ does not evaluate any further, because m, being a variable, doesn't give enough information to be able to select a clause in (+). While it is often not too difficult to write the expressions in the correct form, it is not always possible to do so, and we need to give Idris a little help: we need to provide Idris with a proof that one expression is equivalent to the other, and tell it may use that prove to rewrite the type. For example, we can prove that the commutativity law holds for (+), and we can tell Idris to use that proof to rewrite $m+(S\ k)$ into $(S\ k)+m$. We will elaborate on proofs below.

Aside from making result types more detailed, as we did with (++), we can also use dependent types to restrict arguments. For example, we can define a *head* function which only operates on non-empty vectors:

$$
\begin{aligned}
&\mathit{head} : \mathit{Vect}\ (S\ n)\ a \to a \\
&\mathit{head}\ (x :: xs) = x
\end{aligned}
$$

The type of *head* specifies that the length of the argument should have the form $S\ n$, such that it is at least one. Because the length can't be Z, *head* can't be used on empty vectors, and it doesn't need to supply a special case for *Nil*.

Implicit arguments

As is visible in the examples above, function types may contain variables. Idris will automatically bind these variables as additional arguments to the function. Because

it is often possible to derive the values of these arguments, we do not have to state these values explicitly: they are implicit arguments. However, sometimes it is useful to make them temporarily explicit. To do this, they can be put into curly braces. For example, the type of (++) is equivalent to

$$(++) : \{a : Type\} \to \{n, m : Nat\} \to Vect\ n\ a \to Vect\ m\ a \to Vect\ (n + m)\ a$$

If we want to give one of the arguments explicitly, we can write for example

$$(++)\ \{a = Integer\}\ (1 :: 2 :: Nil)\ (3 :: 4 :: Nil)$$

It is also possible to pattern match on implicit arguments using the same notation:

$$\begin{aligned}
&(++) : \{n : Nat\} \to Vect\ n\ a \to Vect\ m\ a \to Vect\ (n + m)\ a \\
&(++)\ \{n = Z\}\quad Nil\qquad ys = ys \\
&(++)\ \{n = S\ k\}\ (x :: xs)\ ys = x :: (xs ++ ys)
\end{aligned}$$

The with-rule

Dependent types make it possible to refine types with the values of function arguments, but this can also work the other way around: we can discover information about the arguments of a function through the dependent types. Take for instance the concatenation example that shows pattern matching on the implicit arguments. When the first vector argument is *Nil*, its type necessarily has the form *Vect Z a* (c.f. the constructor definition), and we can deduce that n must be Z. Similarly, if that argument is $x :: xs$, n must have the form $S\ k$.

With this mechanism, we can also learn about the form of values from intermediate computations. An example that is sometimes used in the literature is that of a parity type, which describes the property of a number being even or odd. This type can be defined as

$$\begin{aligned}
&\textbf{data}\ Parity : Nat \to Type\ \textbf{where} \\
&\quad even : Parity\ (k + k) \\
&\quad odd\ : Parity\ (S\ (k + k))
\end{aligned}$$

When a value of type *Parity n* is *even*, n must be a a multiple of two. If it is *odd*, n must be the successor of a multiple of two. To actually find out what the parity of n is, we have a function that can compute this for every n:

$$\begin{aligned}
&parity : (n : Nat) \to Parity\ n \\
&parity\ Z = even\ \{k = Z\} \\
&parity\ (S\ n)\ \textbf{with}\ (parity\ n) \\
&\quad parity\ (S\ (j + j))\qquad \mid even\ = odd\ \{k = j\} \\
&\quad parity\ (S\ (S\ (j + j)))\ \mid odd\ ?= even\ \{k = S\ j\}
\end{aligned}$$

If the number is Z, then its parity is *even*. We have to assign the implicit arguments explicitly, because it is not possible to derive $k = Z$ from the expected type *Parity Z* (Idris would have to know how to invert the operation $k + k$ in the type of *even*). If the argument of *parity* is $S\ n$, then its parity is the opposite of that of n. This is expressed using the **with** construct. This construct allows you to extend the left-hand side of a clause with patterns (written after the vertical bars) that match on the result of

an intermediate calculation (following the **with** keyword), resulting in an additional set of sub-clauses. In our example, we pattern matcho the result of *parity n*. This result can be either *even* or *odd*, giving us two sub-clauses that return *odd* and *even*, respectively (we'll come to the question mark shortly).

The benefit of the **with** construct (over for example **case** statements) is that it can extract information about the *form* of the arguments of the function from the *types* of the **with** patterns, if those types depend on the arguments. In our example, the result of *parity n* has type *Parity n*. If this result is *even*, the constructor definition tells us that *n* must have the form $j+j$ (*even* always has a type of the form *Parity* $(k+k)$), and that means that the pattern $S\,n$ can be refined to $S\,(j+j)$. Similarly, if *parity n* gives us the result *odd*, the constructor definition tells us that *n* must have the form $S\,(j+j)$, such that the pattern of $S\,n$ becomes $S\,(S\,(j+j))$.

The **with** construct allows us not only to infer information about arguments from intermediate computations, but it may also be used to refine the expected result type, if the **with** expression occurs in it. For example, we can define a function that returns the longest of two vectors as follows:

```
max : Nat → Nat → Nat
max x y = if (x > y) then x else y

longest : Vect n a → Vect m a → Vect (max n m) a
longest {n} {m} xs ys with (n > m)
  | True = xs
  | False = ys
```

(We may omit the initial part of the sub-clauses, because we learn nothing about the arguments from **with**.) *longest* returns the vector *xs* when *n* is larger than *m*, and *ys* when *m* is larger than (or equal to) *n*. Because $(n>m)$ also occurs in *max n m* (after reducing both to normal form), Idris can substitute the former in the latter with the result of $(n>m)$ as indicated by the clauses. For example, in the first clause $(n>m)$ equals *True*. Conceptually, this gives us the following sequence of substitutions and reductions:

```
Vect (max n m) a
Vect (if (n > m) then n else m) a
Vect (if True then n else m) a
Vect n a
```

This works because Idris defines **if** to be strict only in its first argument, and can therefore always reduce when given the value *True* (or *False*). At this point, the expected type is *Vect n a*, and since *xs* has the same type, this clause passes the type check. This works similarly for *False*. It is therefore possible to use the **with** statement to refine not only argument patterns, but also result types. Could we have defined *longest* by using the **if** statement to determine the result? That is, could we have defined *longest* as

```
longest : Vect n a → Vect m a → Vect (max n m) a
longest {n} {m} xs ys = if (n > m) then xs else ys
```

The answer is no. In Idris, **if** is defined with the type $Bool \to a \to a \to a$, which means the **then** and the **else** branch need to have the same type. This clearly isn't the case.

Now what about the question mark in the last line of *parity*? In short, it signifies a *provisional definition*, which means that Idris may fail to typecheck the definition on

its own, but that we will supply a proof later on. Idris will assume that the proof exists in the form of a metavariable, such that the definition still passes the check, and it will notify us of this assumption until we provide a suitable definition (i.e., the proof) for the metavariable. But why do we need to write the proof ourselves? If we look at the actual types and the expected types, we get a hint of the answer:

```
parity (S (S (j + j))) : Parity (S (S (j + j)))   -- expected type of the clause's r.h.s.
even {s = S j}         : Parity ((S j) + (S j))   -- actual type of the clause's r.h.s.
```

The typechecker can't determine that $S\,(S\,(j+j)) = (S\,j) + (S\,j)$, because the right hand side can't be reduced any further than $S\,(j + S\,j)$ (remember that the Idris definition of $(+)$ recurses over its first argument). This can be solved by proving and applying the theorem that $(x + S\,y)$ equals $(S\,(x+y))$. We will discuss proofs in the next section.

A.1.2 *Programs and proofs*

The Curry-Howard correspondence tells us that we can regard propositions as types, and proofs as programs. This duality is also embedded in Idris: it is often more natural to reason in terms of proofs, but the implementations will often look more like programs. For example, one may want to prove that a function or function argument obeys certain properties, perhaps as part of formal verification. Due to the Curry-Howard correspondence and the power of dependent types, this involves little else than adding some functions and data types. In this section, we will take a look at Idris from a theorem-proving point of view.

While we regard types as propositions, it is natural to regard type constructors that take arguments as predicates. The corresponding data constructors are then interpreted as axioms. For example, a predicate that is defined in the standard Idris library is the "less-than-or-equal-to" predicate on natural numbers:

```
data LTE : (n, m : Nat) → Type where
  lteZero : Z right
  lteSucc : LTE left right → LTE (S left) (S right)
```

A value of type *LTE n m* is a proof that *n* is less than or equal to *m*. To construct such a proof, one can use *lteZero*, which expresses that *Z* is always less then or equal to any number, or *lteSucc*, which expresses that if the proposition holds for two numbers *left* and *right*, it also holds for their successors. For example, we can write the following proofs in Idris:

```
zeroLTEtwo : LTE 0 2
zeroLTEtwo = lteZero

oneLTEtwo : LTE 1 2
oneLTEtwo = lteSucc lteZero
```

Using the axioms, we can prove *LTE n m* for appropriate *n* and *m*, and because they are the only axioms for the predicate *LTE*, we may assume that it is impossible to give a value of type *LTE n m* if *n* is not actually less than or equal to *m*. We can use this to impose requirements on the arguments of a function. Take for example a function that controls pressure in some system, and we want to make sure that the pressure is never set above 2500 PSI. We could define the function as follows:

```
setPressure : (x : Nat) → LTE x 2500 → IO ()
setPressure x p = unsafeSetPressure x
```

Because it is impossible to construct a proof of *LTE x* 2500 if *x* is larger than 2500, we may safely assume *x* is less than (or equal to) 2500. Of course, the burden of proof now lies with the caller. This means we cannot simply put *setPressure* in an **if**-statement that tests *x*, because that gives us no proof. If we do not already have a proof (in which case we can call *setPressure* directly), we must test *x* in a way that gives us one. For example:

```
...
case (maybeLTE x 2500) of
  Just p    ⇒ setPressure x p
  Nothing ⇒ reportError
...
```

The function *maybeLTE* returns *Nothing* if *x* is larger than 2500, and the calling function would then return an error. However, if *x* is not larger than 2500, *maybeLTE* returns the value *Just p*, which contains a proof that can be passed to *setPressure*.

Now we still have not seen a way to actually prove that *LTE n m*, that is, the definition of a function of type $(n, m : Nat) \rightarrow LTE\ n\ m$. Of course, such a function does not exist (at least, not one that covers all its inputs and halts), because *LTE n m* can't be proved for all combinations of *n* and *m*. That is why in our example above, we used *maybeLTE*: if there is a proof, it returns it with *Just*, and otherwise it returns *Nothing*. Here is its definition:

```
maybeLTE : (n, m : Nat) → Maybe (LTE n m)
maybeLTE Z     m     = Just lteZero
maybeLTE (S n) Z     = Nothing
maybeLTE (S n) (S m) = case (maybeLTE n m) of
                         Just p    ⇒ Just (lteSucc p)
                         Nothing ⇒ Nothing
```

We can see the recursive construction of the proofs: *lteZero* for the base case, *lteSucc* for the step case, and *Nothing* when there is no proof.

Helping the typechecker

The propositions in the example above are used to impose restrictions on the function arguments. However, as mentioned at the beginning of this section, we may also want to declare and prove propositions in order to tell something about our programs.

Perhaps the most important predicate used to reason about programs in Idris is the built-in equality predicate. Conceptually, this predicate has the following definition:

```
data (=) : a → b → Type where
  refl : x = x
```

The axiom *refl* states that if *p* and *q* are two definitionally equal terms (i.e., equal according to Idris' internal notion of equality), then the proposition $p = q$ holds. For example, $0 + 1$ and 1 are definitionally equal, so we can prove $0 + 1 = 1$ by directly invoking *refl*:

```
ZeroPlusOne : 0 + 1 = 1
ZeroPlusOne = refl
```

The reason why this predicate is the most important predicate, is that we can use this predicate to rewrite types. For this, Idris supplies the built-in function *replace*:

```
replace : x = y → P x → P y
replace refl prf = prf
```

Given that $x = y$, we can rewrite the type $P\ x$ to $P\ y$. To illustrate this, we will use the commutative property of (+). Like many other properties, it is defined in the standard library, and it has the following type:

```
plusCommutative : (left : Nat) → (right : Nat) → left + right = right + left
```

In other words, *left* + *right* is equal to *right* + *left* for any two natural numbers *left* and *right*. Now, consider we have a value *xs* of type *Vect* (*n* + *m*) *a*, while the typechecker expects it to have type *Vect* (*m* + *n*) *a*. With *replace*, we can rewrite the type:

```
xs : Vect (n + m) a
replace (plusCommutative n m) xs : Vect (m + n) a
```

In the previous section, we had an example (*parity*) where we told Idris to assume a proof, in order to make it pass the typechecker. We had the following function:

```
parity : (n : Nat) → Parity n
parity Z = even {k = Z}
parity (S n) with (parity n)
  parity (S (j + j))      | even  = odd {k = j}
  parity (S (S (j + j))) | odd ?= even {k = S j}
```

The question mark in the last line indicates that we supply our own proof of the type correctness of *even* {*k* = *S j*}. To refresh our memory, these were the actual and the expected types of the last clause:

```
parity (S (S (j + j))) : Parity (S (S (j + j)))   -- expected type of the clause's r.h.s.
even {s = S j}         : Parity ((S j) + (S j))   -- actual type of the clause's r.h.s.
```

When Idris encounters ?=, it will introduce a metavariable, which is assumed to prove the type correctness. For our example, Idris will add the following variable:

```
parity_lemma_1 : (j : Nat) → Parity ((S j) + (S j)) → Parity (S (S (j + j)))
```

The metavariable represents a lemma which proves that the actual type of the last clause implies the expected type, for all *j*. Computationally, the metavariable is just a function that transforms the actual type into the expected type (given the parameter *j*). If we therefore want to prove the lemma, we can provide a definition that rewrites the type. In the example, the difficulty was a reduction involving the second argument of (+): We need a way to transform (*S j*) + (*S j*) into *S* ((*S j*) + *j*) (which will reduce normally). The standard library contains a theorem called *plusSuccRightSucc*, which states that these two expressions are equivalent:

```
plusSuccRightSucc : (l, r : Nat) → S (l + r) = l + (S r)
```

We can use this theorem in conjunction with *replace* to rewrite our types. Unfortunately, *replace* always rewrites the l.h.s. of the equality into the r.h.s., while in our case we need to do it the other way around. We have to use the symmetry property of equality, which is also defined in the standard library:

$$sym : \{l, r : a\} \to l = r \to r = l$$

We can now give a definition for *parity_lemma_1* that proves the type correctness of the clause:

$$parity\_lemma\_1 : (j : Nat) \to Parity\ ((S\ j) + (S\ j)) \to Parity\ (S\ (S\ (j + j)))$$
$$parity\_lemma\_1\ j\ x = replace\ (sym\ (plusSuccRightSucc\ (S\ j)\ j))\ x$$

It should be noted that Idris does not *need* this definition to typecheck *parity*: it is only an assumption. This makes it possible to delay the proofs during development and focus on the actual program. However, Idris will not compile the program until all metavariables have corresponding definitions, and it will notitfy the programmer of any variables that do not.

Idris' tactics and proof assistant

Constructing proofs the way we did for *parity_lemma_1* can be a bit cumbersome. To make proving theorems easier, Idris has a built-in proof assistant. This assistant will keep track of the environment and the proof goal, such that at any point it is clear what derivations you already have, and where you need to go. The proof can constructed by applying *tactics*. We will not give a detailed description of the assistant and the tactics, but we will discuss an example just to give an idea of the concept.

The example is an alternative proof of *parity_lemma_1*. When we start the proof assistant in the REPL, we see the goal of our proof, and a hole where our proof should go, as shown in the following output:

```
*parity> :p parity_lemma_1
----------                  Goal:                  ----------
{hole0} : (j : Nat) ->
          Parity (plus (S j) (S j)) -> Parity (S (S (plus j j)))
```

The first thing we might do, is turn the left hand sides of the implications into assumptions, and add them to the context. We do this with the tactic *intros*:

```
-Main.parity_lemma_1> intros
----------              Assumptions:              ----------
 j : Nat
 value : Parity (plus (S j) (S j))
----------                  Goal:                  ----------
{hole2} : Parity (S (S (plus j j)))
```

Here, *value* represents our clause. We can now use the *rewrite* tactic and the theorem *plusSuccRightSucc* to rewrite the goal:

```
-Main.parity_lemma_1> rewrite (sym (plusSuccRightSucc j j))
----------              Assumptions:              ----------
 j : Nat
 value : Parity (plus (S j) (S j))
----------                  Goal:                  ----------
{hole3} : Parity (S (plus j (S j)))
```

If we would reduce the type of *value*, we get the goal type. That means we can fill the hole with the exact term *value*, and Idris will be able to unify the types. We can do this with *trivial* tactic:

```
-Main.parity_lemma_1> trivial
parity_lemma_1: No more goals.
```

After closing the proof assistant with the command *qed*, we get to see the full proof:

```
-Main.parity_lemma_1> qed
Proof completed!
Main.parity_lemma_1 = proof
  intros
  rewrite (sym (plusSuccRightSucc j j))
  trivial
```

This is a valid definition of *parity_lemma_1* that can be added to the source file verbatim.

A.2 AGDA

In this section, we will make a few short remarks about an alternative to Idris called Agda. Agda ([Nor07], website `http://wiki.portal.chalmers.se/agda`) is a dependently typed functional programming language and proof assistant. Its underlying type system is in most aspects the same as Idris', and it shares a lot of features with Idris. However, Agda is not inspired by Haskell as much as Idris. There is no **do**-notation, or typeclasses, and the notation is through the use of unicode exceedingly mathematical. The latter has as consequence that Agda looks more like a proof language than like a regular programming language. Reading Agda programs is not an easy task for the uninitiated. Agda does support common types like *Int*, *Float*, *String*, *Char* and *Bool*. It also supports constructs like pattern matching, recursion, **let**-bindings, and **where**- and **with**-clauses. Data types are declared in a similar ways as in Idris. Furthermore, there is a foreign function interface, and compilation is possible (though the documentation is not very clear on how one accomplishes this). Like Idris, Agda has an assistant for theorem proving, but while Idris' assistant can be used stand-alone (via the REPL), Agda's is tightly integrated in the Emacs editor. Emacs users may find this convenient, others may not.

B · The type system

Expressions

$$\begin{aligned} e, \rho, k \;\coloneqq\;& * \mid \forall x{:}\rho\,.\,\rho \mid e{:}\rho \mid x \mid e\;e \mid \lambda x\,.\,e \\ \mid\;& \mathbb{N} \mid Z \mid S\;k \mid \mathit{nElim}\;e\;e\;e\;k \\ \mid\;& \mathit{Vect}\;\rho\;k \mid \mathit{vNil} \mid \mathit{vCons}\;e\;e \mid \mathit{vElim}\;\rho\;e\;e\;e\;k\;e \\ \mid\;& \rho\langle k\,..\,k\rangle \mid \mathit{sCons}\;e\;e \\ \mid\;& \mathit{sElim}\;\rho\;k\;e\;e\;e\;k\;e \mid \mathit{tFold}\;\ldots \mid \mathit{zipWith}\;\ldots \end{aligned}$$

Values

$$\begin{aligned} v, \tau, l \;\coloneqq\;& n \mid * \mid \forall x{:}\tau\,.\,\tau \mid \lambda x\,.\,v \\ \mid\;& \mathbb{N} \mid Z \mid S\;v \\ \mid\;& \mathit{Vect}\;\tau\;k \mid \mathit{vNil} \mid \mathit{vCons}\;v\;v \\ \mid\;& \tau\langle l\,..\,l\rangle \mid \mathit{sCons}\;v\;v \end{aligned} \qquad \begin{aligned} n \;\coloneqq\;& x \mid n\;v \\ \mid\;& \mathit{nElim}\;v\;v\;v\;n \\ \mid\;& \mathit{vElim}\;\tau\;v\;v\;v\;l\;n \\ \mid\;& \mathit{sElim}\;\rho\;l\;v\;v\;v\;l\;n \end{aligned}$$

Contexts

$$\Gamma \;\coloneqq\; \varepsilon \mid \Gamma;\, x : \tau \qquad \frac{}{\mathrm{valid}(\varepsilon)} \qquad \frac{\mathrm{valid}(\Gamma) \qquad \Gamma \vdash \tau :_i *}{\mathrm{valid}(\Gamma;\, x : \tau)}$$

Evaluation rules

Bare lambda calculus

$$\frac{}{* \Downarrow *} \qquad \frac{\rho \Downarrow \tau \qquad \rho' \Downarrow \tau'}{\forall x{:}\rho\,.\,\rho' \Downarrow \forall x{:}\tau\,.\,\tau'} \qquad \frac{e \Downarrow v}{e : \rho \Downarrow v} \qquad \frac{}{x \Downarrow x}$$

$$\frac{e \Downarrow \lambda x\,.\,v \qquad v[x \coloneqq e'] \Downarrow v'}{e\;e' \Downarrow v'} \qquad \frac{e \Downarrow n \qquad e' \Downarrow v}{e\;e' \Downarrow n\;v} \qquad \frac{e \Downarrow v}{\lambda x\,.\,e \Downarrow \lambda x\,.\,v}$$

Timed types

$$\frac{\rho \Downarrow \tau \qquad k \Downarrow l \qquad k' \Downarrow l'}{\rho\langle k\,..\,k'\rangle \Downarrow \tau\langle l\,..\,l'\rangle} \qquad \frac{e \Downarrow v \qquad es \Downarrow vs}{\mathit{sCons}\;e\;es \Downarrow \mathit{sCons}\;v\;vs}$$

$$\frac{e \Downarrow n}{\mathit{sElim}\;\rho\;m\;mz\;ms\;k\;e \Downarrow \mathit{sElim}\;\rho\;m\;mz\;ms\;k\;n} \qquad \frac{k \Downarrow Z \qquad e \Downarrow v \qquad mz \Downarrow vz \qquad ms\;Z\;v\;vz \Downarrow v'}{\mathit{sElim}\;\rho\;m\;mz\;ms\;k\;e \Downarrow v'}$$

$$\frac{k \Downarrow S\;l \qquad e \Downarrow \mathit{sCons}\;v\;vs \qquad ms\;l\;v\;(\mathit{sElim}\;\rho\;m\;mz\;ms\;l\;vs) \Downarrow v'}{\mathit{sElim}\;\rho\;m\;mz\;ms\;k\;e \Downarrow v'}$$

Natural numbers

$$\frac{}{\mathbb{N} \Downarrow \mathbb{N}} \qquad \frac{}{Z \Downarrow Z} \qquad \frac{k \Downarrow l}{S\,k \Downarrow S\,l} \qquad \frac{k \Downarrow n}{nElim\ m\ mz\ ms\ k \Downarrow nElim\ m\ mz\ ms\ n}$$

$$\frac{k \Downarrow Z \qquad mz \Downarrow v}{nElim\ m\ mz\ ms\ k \Downarrow v} \qquad \frac{k \Downarrow S\,l \qquad ms\ l\ (nElim\ m\ mz\ ms\ l) \Downarrow v}{nElim\ m\ mz\ ms\ k \Downarrow v}$$

Vectors

$$\frac{\rho \Downarrow \tau \qquad k \Downarrow l}{Vect\ \rho\ k \Downarrow Vect\ \tau\ l} \qquad \frac{}{vNil \Downarrow vNil} \qquad \frac{e \Downarrow v \qquad es \Downarrow vs}{vCons\ e\ es \Downarrow vCons\ v\ vs}$$

$$\frac{e \Downarrow n}{vElim\ \rho\ m\ mz\ ms\ k\ e \Downarrow vElim\ \rho\ m\ mz\ ms\ k\ n} \qquad \frac{e \Downarrow vNil \qquad mz \Downarrow v}{vElim\ \rho\ m\ mz\ ms\ k \Downarrow v}$$

$$\frac{e \Downarrow vCons\ v\ vs \qquad k \Downarrow S\,l \qquad ms\ l\ v\ (vElim\ \rho\ m\ mz\ ms\ l\ vs) \Downarrow v'}{vElim\ \rho\ m\ mz\ ms\ k\ e \Downarrow v'}$$

Typing rules

Bare lambda calculus

$$\frac{}{* :_i *}\ (\text{Star}) \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x :_i \tau}\ (\text{Var}) \qquad \frac{\Gamma \vdash \rho :_c * \qquad \rho \Downarrow \tau \qquad \Gamma; x : \tau \vdash \rho' :_c *}{\Gamma \vdash \forall x{:}\rho\,.\,\rho' :_i *}\ (\text{Pi})$$

$$\frac{\Gamma \vdash \rho :_c * \qquad \rho \Downarrow \tau \qquad \Gamma \vdash e :_c \tau}{\Gamma \vdash (x{:}\rho) :_i \tau}\ (\text{Ann}) \qquad \frac{\Gamma \vdash e :_i \forall x{:}\tau\,.\,\tau' \qquad \Gamma \vdash e' :_c \tau \qquad \tau'[x := e'] \Downarrow \tau''}{\Gamma \vdash e\,e' :_i \tau''}\ (\text{App})$$

$$\frac{\Gamma; x : \tau \vdash e :_c \tau'}{\Gamma \vdash \lambda x\,.\,e :_c \forall x{:}\tau\,.\,\tau'}\ (\text{Lam}) \qquad \frac{\Gamma \vdash e :_i \tau}{\Gamma \vdash e :_c \tau}\ (\text{Chk})$$

Timed types

$$\frac{\Gamma \vdash \rho :_c * \qquad \Gamma \vdash k :_c \mathbb{N} \qquad \Gamma \vdash k' :_c \mathbb{N}}{\Gamma \vdash \rho\langle k\,..\,k'\rangle :_i *}\ (\text{Seq})^1$$

$$\frac{\Gamma \vdash e :_i \tau\langle l\,..\,l'\rangle \qquad l \leqslant m \qquad m - l = m' - l'}{\Gamma \vdash e :_c \tau\langle m\,..\,m'\rangle}\ (\text{TChk}) \qquad \frac{\Gamma \vdash e :_i \tau}{\Gamma \vdash e :_c \tau\langle n\,..\,n\rangle}\ (\text{TChkL})$$

$$\frac{\Gamma \vdash e :_i \forall x{:}\tau\langle l\,..\,l'\rangle\,.\,\tau' \qquad \Gamma \vdash e' :_i \tau\langle m\,..\,m'\rangle \qquad m - l = m' - l' \qquad \tau'[x := e'] \Downarrow \tau''}{\Gamma \vdash e\,e' :_i \Delta(l, l', \tau'')}\ (\text{TApp})$$

$$\frac{\Gamma \vdash e :_i \forall x{:}\tau\,.\,\tau' \qquad \Gamma \vdash e' :_i \tau\langle l\rangle \qquad \tau'[x := e'] \Downarrow \tau''}{\Gamma \vdash e\,e' :_i \Theta(l, \tau'')}\ (\text{TAppLF})$$

$$\frac{\Gamma \vdash es :_i \tau\langle l\,..\,l'\rangle \qquad \Gamma \vdash e :_c \tau\langle S\,l'\rangle}{\Gamma \vdash sCons\ e\ es :_i \tau\langle l\,..\,S\,l'\rangle}$$

$$\frac{\begin{array}{llll} \Gamma \vdash \rho :_c * & & m\,Z \Downarrow \tau & \forall l{:}\mathbb{N}\,.\,\rho\langle k{+}l\rangle \to m\,l \to m\,(S\,l) \Downarrow \tau' \qquad \Gamma \vdash l :_c \mathbb{N} \\ \Gamma \vdash k :_c \mathbb{N} & \Gamma \vdash m :_c \mathbb{N} \to * & \Gamma \vdash mz :_c \tau & \Gamma \vdash ms :_c \tau' \qquad \Gamma \vdash e :_c \rho\langle k\,..\,k{+}l\rangle \end{array}}{\Gamma \vdash sElim\ \rho\ k\ m\ mz\ ms\ l\ e :_i m\ l}$$

Natural numbers

[1] Side condition: ρ must be $\mathbb{N}$. In a more developed HDL, ρ should be a type representable in hardware.

$$\frac{}{\Gamma \vdash \mathbb{N} :_i *} \qquad \frac{}{\Gamma \vdash Z :_i \mathbb{N}} \qquad \frac{\Gamma \vdash k :_c \mathbb{N}}{\Gamma \vdash S\,k :_i \mathbb{N}}$$

$$\frac{\Gamma \vdash m :_c \mathbb{N} \to * \qquad \Gamma \vdash mz :_c \tau \qquad m\,Z \Downarrow \tau \qquad \Gamma \vdash ms :_c \tau' \qquad \forall l{:}\mathbb{N}.\, m\,l \to m\,(S\,l) \Downarrow \tau' \qquad \Gamma \vdash k :_c \mathbb{N}}{\Gamma \vdash nElim\ m\ mz\ ms\ k :_i m\,k}$$

Vectors

$$\frac{\Gamma \vdash \rho :_c * \qquad \Gamma \vdash k :_c \mathbb{N}}{\Gamma \vdash Vect\ \rho\ k :_i *} \qquad \frac{\Gamma \vdash \rho :_c *}{\Gamma \vdash vNil\ \rho :_i Vect\ \rho\ Z} \qquad \frac{\Gamma \vdash es :_i Vect\ \rho\ k \qquad \Gamma \vdash e :_c \rho}{\Gamma \vdash vCons\ e\ es :_i Vect\ \rho\ (S\,k)}$$

$$\frac{\Gamma \vdash m :_c \mathbb{N} \to * \qquad \Gamma \vdash \rho :_c * \qquad \Gamma \vdash mz :_c \tau \qquad m\,Z \Downarrow \tau \qquad \Gamma \vdash ms :_c \tau' \qquad \forall l{:}\mathbb{N}.\, \rho \to m\,l \to m\,(S\,l) \Downarrow \tau' \qquad \Gamma \vdash e :_c Vect\ \rho\ k \qquad \Gamma \vdash k :_c \mathbb{N}}{\Gamma \vdash vElim\ \rho\ m\ mz\ ms\ k\ e :_i m\,k}$$

Auxiliary definitions

Time shifting

$$\Delta(l, l', \tau) \equiv \begin{cases} \tau & \text{if } l > l' \text{ or } \tau = \mathbb{N} \text{ or } \tau = Vect\ \tau'\ k \\ \tau'\langle k + l' - l\,..\,k' + l' - l\rangle & \text{if } l \leqslant l' \text{ and } \tau = \tau'\langle k\,..\,k'\rangle \\ \forall x{:}\sigma.\, \Delta(l, l', \tau') & \text{if } \tau = \forall x{:}\sigma.\, \tau' \text{ and } \sigma = \forall\ldots \\ \forall x{:}\Delta(l, l', \sigma).\, \Delta(l, l', \tau') & \text{if } \tau = \forall x{:}\sigma.\, \tau' \text{ and } \sigma \neq \forall\ldots \end{cases}$$

Implicit timing

$$\Theta(k, \tau) \equiv \begin{cases} \mathbb{N}\langle k\rangle & \text{if } \tau = \mathbb{N} \\ Vect\ \tau'\ l & \text{if } \tau = Vect\ \tau'\ l \\ \forall x{:}\sigma.\, \Theta(k, \tau') & \text{if } \tau = \forall x{:}\sigma.\, \tau' \text{ and } \sigma = \forall\ldots \\ \forall x{:}\Theta(k, \sigma).\, \Theta(k, \tau') & \text{if } \tau = \forall x{:}\sigma.\, \tau' \text{ and } \sigma \neq \forall\ldots \end{cases}$$

Intralingual functions

$$(+) \equiv \lambda n.\, nElim\ (\lambda k.\mathbb{N} \to \mathbb{N})\ (\lambda x.\, x)\ (\lambda l.\lambda f.\lambda n.\, S\,(f\,n))\ n : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

$$pred \equiv \lambda n.\, nElim\ (\lambda k.\mathbb{N})\ Z\ (\lambda k.\lambda r.\, k)\ n : \mathbb{N} \to \mathbb{N}$$

The relations $\leqslant$ and $>$ are based on the following rules (they are implemented as a single function that tries to find a term representing $x - y$)

$$x = y \implies x \leqslant y$$

$$Z \leqslant y$$

$$x < y \implies S\,x \leqslant y$$

$$x \leqslant y \implies x \leqslant S\,y$$

$$S\,x \leqslant y \implies x \leqslant pred\ y$$

$$x \leqslant S\,y \implies pred\ x \leqslant y$$

$$x_a + x_b \leqslant y \implies x_b + x_a \leqslant y$$

$$x \leqslant y \implies x + Z \leqslant y$$

$$x \leqslant y_a \implies x \leqslant y_a + y_b$$

$$x \leqslant y_b \implies x \leqslant y_a + y_b$$

$$(x > y_a) \wedge (x > y_b) \wedge (x - y_a \leqslant y_b) \implies x \leqslant y_a + y_b$$

$$(x_a \leqslant y_a) \wedge (x_b \leqslant y_b) \implies x_a + x_b \leqslant y_a + y_b$$

$$(x_a \leqslant y_a) \wedge (x_b > y_b) \wedge (x_b - y_b \leqslant y_a - x_a) \implies x_a + x_b \leqslant y_a + y_b$$

$$(x_a > y_a) \wedge (x_b \leqslant y_b) \wedge (x_a - y_a \leqslant y_b - x_b) \implies x_a + x_b \leqslant y_a + y_b$$

$$x = y \implies S\,x > y$$

$$x > y \implies S\,x > y$$

$$(x_a > y_a) \wedge (x_b > y_b) \implies x_a + x_b > y_a + y_b$$

$$(x_a \leqslant y_a) \wedge (x_b > y_b) \wedge (x_b - y_b > y_a - x_a) \implies x_a + x_b > y_a + y_b$$

$$(x_a > y_a) \wedge (x_b \leqslant y_b) \wedge (x_a - y_a \leqslant y_b - x_b) \implies x_a + x_b > y_a + y_b$$

C · Implementation

C.1 Syntax

```
module tree

mutual
  data Name : Type where
    Global  : String → Name
    Local   : Nat → Name
    Quote   : Nat → Name
  data ITerm : Type where
    IStar   : ITerm
    IPi     : CTerm → CTerm → ITerm
    IAnn    : CTerm → CTerm → ITerm
    IBound  : Nat → ITerm
    IFree   : Name → ITerm
    IApp    : ITerm → CTerm → ITerm
    INat    : ITerm
    IZ      : ITerm
    IS      : CTerm → ITerm
    INElim  : CTerm → CTerm → CTerm → CTerm → ITerm
    IVect   : CTerm → CTerm → ITerm
    IVNil   : CTerm → ITerm
    IVCons  : CTerm → CTerm → CTerm → CTerm → ITerm
    IVElim  : CTerm → CTerm → CTerm → CTerm → CTerm → CTerm → ITerm
    ISeq    : CTerm → CTerm → CTerm → ITerm
    ISCons  : CTerm → ITerm → ITerm
    ISElim  : CTerm → CTerm → CTerm → CTerm → CTerm → CTerm → CTerm → ITerm
  data CTerm : Type where
    CInf  : ITerm → CTerm
    CLam  : CTerm → CTerm


  data Neutral : Type where
    NFree   : Name → Neutral
    NApp    : Neutral → Value → Neutral
    NNElim  : Value → Value → Value → Neutral → Neutral
    NVElim  : Value → Value → Value → Value → Value → Neutral → Neutral
    NSElim  : Value → Value → Value → Value → Value → Value → Neutral → Neutral
  data Value : Type where
    VLam    : (Value → Result Value) → Value
    VStar   : Value
    VPi     : Value → (Value → Result Value) → Value
    VNeu    : Neutral → Value
    VNat    : Value
    VZ      : Value
    VS      : Value → Value
    VVect   : Value → Value → Value
```

```
  VVNil  : Value → Value
  VVCons : Value → Value → Value → Value → Value
  VSeq   : Value → Value → Value → Value
  VSCons : Value → Value → Value

data PDecl : Type where
  Ass : Name → CTerm → PDecl
  Def : Name → ITerm → PDecl

getName : PDecl → Name
getName (Ass n t) = n
getName (Def n t) = n

Result : Type → Type
Result t = Either String t

Ty : Type
Ty = Value

{-instance Eq -}

instance Eq Name where
  (==) (Global s) (Global s') = s == s'
  (==) (Local n)  (Local m)   = n == m
  (==) (Quote n)  (Quote m)   = n == m
  (==) _          _           = False

mutual
  -- little helper for constructors with lots of arguments
  argCmp : (a → b → Bool) → (x : List a) → (y : List b) → {auto p : length x = length y} → Bool
  argCmp f x y {p} = foldr (∧) True$ List.zipWith f x y p

  iteq : ITerm → ITerm → Bool
  iteq (IStar)                (IStar)                  = True
  iteq (IPi s t)              (IPi s' t')              = (cteq s s') ∧ (cteq t t')
  iteq (IAnn e t)             (IAnn e' t')             = (cteq e e') ∧ (cteq t t')
  iteq (IBound n)             (IBound n')              = n == n'
  iteq (IFree n)              (IFree n')               = n == n'
  iteq (IApp f x)             (IApp f' x')             = (iteq f f') ∧ (cteq x x')
  iteq (INat)                 (INat)                   = True
  iteq (IZ)                   (IZ)                     = True
  iteq (IS t)                 (IS t')                  = cteq t t'
  iteq (INElim m mz ms k)     (INElim m' mz' ms' k')   = argCmp cteq
                                                           [m, mz, ms, k]
                                                           [m', mz', ms', k']
  iteq (IVect t k)            (IVect t' k')            = (cteq t t') ∧ (cteq k k')
  iteq (IVNil t)              (IVNil t')               = cteq t t'
  iteq (IVCons t k x xs)      (IVCons t' k' x' xs')    = argCmp cteq [t, k, x, xs] [t', k', x', xs']
  iteq (IVElim t m mz ms k xs) (IVElim t' m' mz' ms' k' xs') = argCmp cteq
                                                           [t, m, mz, ms, k, xs]
                                                           [t', m', mz', ms', k', xs']
  iteq (ISeq t n k)           (ISeq t' n' k')          = argCmp cteq [t, n, k] [t', n', k']
  iteq (ISCons x xs)          (ISCons x' xs')          = (cteq x x') ∧ (iteq xs xs')
  iteq (ISElim t n m mz ms k xs) (ISElim t' n' m' mz' ms' k' xs') = argCmp cteq
                                                           [t, n, m, mz, ms, k, xs]
                                                           [t', n', m', mz', ms', k', xs']
  iteq _                      _                        = False

  cteq : CTerm → CTerm → Bool
  cteq (CInf e)  (CInf e')  = iteq e e'
  cteq (CLam e)  (CLam e')  = cteq e e'
  cteq _         _          = False

instance Eq ITerm where
  (==) = iteq

instance Eq CTerm where
  (==) = cteq
```

```
{-instance Show -}
instance Show Name where
  show (Global s) = s
  show (Local n)  = "Local " ++ show n
  show (Quote n)  = "Quote " ++ show n
mutual
    -- helper for ishow
  argCols : List CTerm → String
  argCols = unwords ∘ (map cshow)
  ishow : ITerm → String
  ishow (IStar)                = "∗"
  ishow (IPi a b)              = "(" ++ cshow a ++ " -> " ++ cshow b ++ ")"
  ishow (IAnn a b)             = cshow a ++ " : " ++ cshow b
  ishow (IBound k)             = "(Bound " ++ show k ++ ")"
  ishow (IFree n)              = show n
  ishow (IApp f x)             = "(" ++ ishow f ++ " " ++ cshow x ++ ")"
  ishow (INat)                 = "Nat"
  ishow (IZ)                   = "Z"
  ishow (IS n)                 = "(S " ++ cshow n ++ ")"
  ishow (INElim m mz ms k)     = "(nElim " ++ argCols [m, mz, ms, k] ++ ")"
  ishow (IVect a n)            = "(Vect " ++ cshow a ++ " " ++ cshow n ++ ")"
  ishow (IVNil a)              = "(vNil " ++ cshow a ++ ")"
  ishow (IVCons a k x xs)      = "(vCons " ++ (argCols [a, k, x, xs]) ++ ")"
  ishow (IVElim a m mz ms k xs) = "(vElim " ++ (argCols [a, m, mz, ms, k, xs]) ++ ")"
  ishow (ISeq t n k)           = cshow t ++ "<" ++ cshow n ++ ".." ++ cshow k ++ ">"
  ishow (ISCons x xs)          = "(sCons " ++ (cshow x) ++ (ishow xs) ++ ")"
  ishow (ISElim t n m mz ms k xs) = "(sElim " ++ (argCols [t, n, m, mz, ms, k, xs]) ++ ")"
  cshow : CTerm → String
  cshow (CInf t)  = ishow t
  cshow (CLam t) = "(\\." ++ cshow t ++ ")"
instance Show ITerm where
  show = ishow
instance Show CTerm where
  show = cshow
instance Show PDecl where
  show (Ass n t)   = show n ++ " : " ++ show t
  show (Def n t)   = show n ++ " = " ++ show t
instance Show a ⇒ Show (Result a) where
  show (Left e)    = e
  show (Right e)   = show e


{-instance Alternative -}
instance Alternative (Either String) where
  empty = Left "No alternatives"
  (<|>) (Right a) _          = Right a
  (<|>) (Left ea) (Right b) = Right b
  (<|>) (Left ea) (Left eb)  = Left $ ea ++ "\nor " ++ eb
```

C.2 EVALUATION / TYPE CHECKING

```
module checker
import tree
len : {n : Nat} → Vect n t → Nat
len {n} _ = n
fvlookup : Name → List PDecl → Maybe (PDecl, List PDecl)
fvlookup n []       = Nothing
```

```
fvlookup n (x :: xs) = if (n == getName x) then
                         Just (x, xs)
                       else
                         let r = fvlookup n xs in
                         map (λy ⇒ (fst y, x :: snd y)) r


 {-built-in definitions for plus and pred, also for use in comp -}
plus' : CTerm → CTerm → ITerm
plus' x y = IApp (IApp (IFree (Global "plus")) x) y

nplusMotive : CTerm   -- λk.∀x:ℕ.ℕ
nplusMotive = CLam (CInf (IPi (CInf INat)
                              (CInf INat)))

nplusBase : CTerm   -- λx.x
nplusBase = CLam (CInf (IBound 0))

nplusStep : CTerm   -- λl.λf.λn.S (f n)
nplusStep = CLam (CLam (CLam (CInf (IS (CInf (IApp (IBound 1) (CInf (IBound 0))))))))

nplus' : CTerm   -- λn.NElim m b s n
nplus' = CLam (CInf (INElim nplusMotive nplusBase nplusStep (CInf (IBound 0))))

nplus : PDecl
nplus = Def (Global "plus")
            (IAnn nplus' (CInf (IPi (CInf INat) (CInf (IPi (CInf INat) (CInf INat))))))


npredMotive : CTerm   -- λk.ℕ
npredMotive = CLam (CInf INat)

npredBase : CTerm   -- Z
npredBase = CInf IZ

npredStep : CTerm   -- λk.λr.k
npredStep = CLam (CLam (CInf (IBound 1)))

npred' : CTerm   -- λn.NElim m b s n
npred' = CLam (CInf (INElim npredMotive npredBase npredStep (CInf (IBound 0))))

npred : PDecl
npred = Def (Global "pred") (IAnn npred' (CInf (IPi (CInf INat) (CInf INat))))


pre : List PDecl
pre = [nplus, npred]

mutual
  showVal : Value → String
  showVal x = case quote0 (Right x) of
    Left err ⇒ err
    Right t ⇒ show t

  showCtxt : Vect n (Name, Ty) → String
  showCtxt = let f = (λs, ss ⇒ case quote0 (Right $ snd s) of
                 Left err ⇒ show (fst s) ++ ":  " ++ err ++ ";  " ++ ss
                 Right t ⇒ show (fst s) ++ ":  " ++ show t ++ ";  " ++ ss)
             in foldr f ""


 {-evaluation -}
ieval : List PDecl → Vect n Value → ITerm → Result Value
ieval D G (IStar)          = return VStar
ieval D G (IPi s t)        = do s' ← ceval D G s
                                return $ VPi s' (λx ⇒ ceval D (x :: G) t)
ieval D G (IAnn e _)       = ceval D G e
ieval D G (IBound i)       = do i' ← maybeToEither
                                     ("local variable " ++ show i ++ " not bound")
                                     (natToFin i (len G))
```

```
                                    return $ index i' G
ieval D G (IFree x)                 = case fvlookup x D of
                                      Just (Def _ t, D') ⇒ ieval D' G t
                                      _                  ⇒ return (VNeu (NFree x))
ieval D G (IApp f x)                = do vf ← ieval D G f
                                         vx ← ceval D G x
                                         vapp vf vx
ieval D G (INat)                    = return VNat
ieval D G (IZ)                      = return VZ
ieval D G (IS t)                    = do v ← ceval D G t
                                         return $ VS v
ieval D G (INElim m mz ms k)        = do mzv ← ceval D G mz
                                         msv ← ceval D G ms
                                         kv  ← ceval D G k
                                         rec mzv msv kv
                                      where
                                         rec : Value → Value → Value → Result Value
                                         rec vmz vms (VZ)      = return vmz
                                         rec vmz vms (VS l)    = do r ← rec vmz vms l
                                                                    v ← vapp vms l
                                                                    vapp v r
                                         rec vmz vms (VNeu e) = do vm ← ceval D G m
                                                                    return $ VNeu (NNElim vm vmz vms e)
                                         rec vmz vms _         = Left "internal error: eval natElim"
ieval D G (IVect t n)               = do vt ← ceval D G t
                                         vn ← ceval D G n
                                         return $ VVect vt vn
ieval D G (IVNil t)                 = do vt ← ceval D G t
                                         return $ VVNil vt
ieval D G (IVCons t k x xs)         = do vt ← ceval D G t
                                         vk ← ceval D G k
                                         vx ← ceval D G x
                                         vxs ← ceval D G xs
                                         return $ VVCons vt vk vx vxs
ieval D G (IVElim t m mz ms k xs)   = do vmz ← ceval D G mz
                                         vms ← ceval D G ms
                                         vk  ← ceval D G k
                                         vxs ← ceval D G xs
                                         vrec D G t m vmz vms vk vxs
ieval D G (ISeq t n k)              = do vt ← ceval D G t
                                         vn ← ceval D G n
                                         vk ← ceval D G k
                                         return $ VSeq vt vn vk
ieval D G (ISCons x xs)             = do vx ← ceval D G x
                                         vxs ← ieval D G xs
                                         return $ VSCons vx vxs
ieval D G (ISElim t n m mz ms k xs) = do vt ← ceval D G t
                                         vn ← ceval D G n
                                         vmz ← ceval D G mz
                                         vms ← ceval D G ms
                                         vk ← ceval D G k
                                         vxs ← ceval D G xs
                                         srec D G m vt vn vmz vms vk vxs

ceval : List PDecl → Vect n Value → CTerm → Result Value
ceval D G (CInf e)  = ieval D G e
ceval D G (CLam e) = return $ VLam (λx ⇒ ceval D (x :: G) e)



vapp : Value → Value → Result Value
vapp (VLam f) t = f t
vapp (VNeu e) t = return $ VNeu (NApp e t)
vapp v        t = Left $ "illegal application of " ++ show (quote0 (Right v))
```

```
vplus : Value → Value → Result Value
vplus x y = do u ← ceval [] [] nplus'
               v ← vapp u x
               vapp v y


vrec : List PDecl → Vect n Value → CTerm
       → Value → Value → Value → Value → Result Value
vrec D G t m vmz vms (VZ)   (VVNil _)          = return vmz
vrec D G t m vmz vms (VS l) (VVCons _ _ y ys) = do u ← vapp vms l
                                                   v ← vapp u y
                                                   r ← vrec D G t m vmz vms l ys
                                                   vapp v r
vrec D G t m vmz vms l      (VNeu n)          = do vt ← ceval D G t
                                                   vm ← ceval D G m
                                                   return $ VNeu (NVElim vt vm vmz vms l n)
vrec D G t m vmz vms _      _                 = Left "internal error: eval vecElim"


srec : List PDecl → Vect n Value → CTerm
       → Value → Value → Value → Value → Value → Result Value
srec D G m vt vn vmz vms (VZ)   (y)          = do u ← vapp vms VZ
                                                  v ← vapp u y
                                                  vapp v vmz
srec D G m vt vn vmz vms (VS l) (VSCons y ys) = do r ← srec D G m vt vn vmz vms l ys
                                                  u ← vapp vms l
                                                  v ← vapp u y
                                                  vapp v r
srec D G m vt vn vmz vms l      (VNeu e)      = do vm ← ceval D G m
                                                  return $ VNeu (NSElim vt vn vm vmz vms l e)
srec D G m vt vn vmz vms _      _             = Left "internal error: eval lstElim"


 {-typechecking -}
valEq : Value → Value → Result ()
valEq x y = do x' ← quote0 $ Right x
               y' ← quote0 $ Right y
               if (x' == y')
                 then return ()
                 else Left $ "Can't convert\n  " ++ show x' ++ "\nwith\n  " ++ show y'


nElimArg : CTerm → CTerm → CTerm → Neutral → Result Value
nElimArg m b s (NNElim vm vb vs nk) = do m' ← quote0 (Right vm)
                                         b' ← quote0 (Right vb)
                                         s' ← quote0 (Right vs)
                                         if ((m == m') ∧ (b == b') ∧ (s == s'))
                                           then return (VNeu nk)
                                           else Left "NNelim argument mismatch"
nElimArg m b s _ = Left "Not an instance of natElim"


isPlus : Neutral → Result (Value, Value)
isPlus (NApp f l) = do k ← nElimArg nplusMotive nplusBase nplusStep f
                       return (k, l)
isPlus _ = Left "Not an instance of plus"
isPred : Neutral → Result Value
isPred e = nElimArg npredMotive npredBase npredStep e


data Comp : Type where
  CLTE : Value → Comp
  CGT  : Value → Comp
cmpEq : Comp → Comp → Result ()
```

```
cmpEq (CLTE x) (CLTE y) = valEq x y
cmpEq (CGT x)  (CGT y)  = valEq x y
cmpEq _        _        = Left $ "Sequences not of equal length"


compErr : Value → Value → Result Comp
compErr x y = Left $ "Cannot compare\n   " ++ showVal x ++ "\nwith\n   " ++ showVal y
retLTE : Value → Result Comp
retLTE k = return $ CLTE k
retGT : Value → Result Comp
retGT k = return $ CGT k


  -- make sure each recursive call to comp (not comp') has a smaller x
comp' : Value → Value → Result Comp
comp' VZ       y        = retLTE y                                    -- (Z ⩽ y)
comp' (VS x)   y        = do d ← comp x y
                             case d of
                               CGT k       ⇒ retGT (VS k)             -- (x > y) → (Sx > y)
                               CLTE VZ     ⇒ retGT (VS VZ)            -- (x = y) → (Sx > y)
                               CLTE (VS k) ⇒ retLTE k                 -- (x < y) → (Sx ⩽ y)
                               CLTE _      ⇒ compErr (VS x) y
comp' (VNeu x) VZ       = compErr (VNeu x) VZ
comp' (VNeu x) (VS y)   = do d ← comp' (VNeu x) y
                             case d of
                               CLTE k ⇒ retLTE (VS k)                 -- (x ⩽ y) → (x ⩽ Sy)
                               CGT k  ⇒ compErr (VNeu x) (VS y)
comp' (VNeu x) (VNeu y) =    do valEq (VNeu x) (VNeu y)
                               return $ CLTE VZ                       -- (x = y) → (x ⩽ y)
                          <|> do y' ← isPred y
                               comp' (VS (VNeu x)) y'                 -- (Sx ⩽ y) → (x ⩽ pred y)
                          <|> do x' ← isPred x
                               comp x' (VS (VNeu y))                  -- (x ⩽ Sy) → (pred x ⩽ y)
                          <|> do (ya, yb) ← isPlus y
                               (do (xa, xb) ← isPlus x
                                   case (comp xa ya, comp xb yb) of
                                     (Left ea          , Left eb          ) ⇒ Left $ ea ++ "\nand\n" ++ eb
                                     (_                , Left eb          ) ⇒ Left eb
                                     (Left ea          , _                ) ⇒ Left ea
                                     (Right (CLTE a), Right (CLTE b)) ⇒ (vplus a b) >>= retLTE
                                                                          -- (xa⩽ya ∧ xb⩽yb) → (xa+xb ⩽ ya+yb)
                                     (Right (CLTE a), Right (CGT b) ) ⇒ comp b a
                                                                          -- (xa⩽ya ∧ xb>yb) → ...
                                     (Right (CGT a) , Right (CLTE b)) ⇒ comp a b
                                                                          -- (xa>ya ∧ xb⩽yb) → ...
                                     (Right (CGT a) , Right (CGT b) ) ⇒ (vplus a b) >>= retGT
                                                                          -- (xa>ya ∧ xb>yb) → (xa+xb > ya+yb)
                                     (_                , _                ) ⇒ compErr (VNeu x) (VNeu y))
                                <|>
                                 (case (comp' (VNeu x) ya, comp' (VNeu x) yb) of
                                   (Left ea          , Left eb          ) ⇒ Left $ ea ++ "\nand\n" ++ eb
                                   (Right (CLTE a), _                ) ⇒ (vplus a yb) >>= retLTE
                                                                          -- (x ⩽ y) → (x ⩽ y+z)
                                   (_                , Right (CLTE b)) ⇒ (vplus ya b) >>= retLTE
                                                                          -- (x ⩽ z) → (x ⩽ y+z)
                                   (Right (CGT a) , Right (CGT b) ) ⇒ comp a yb
                                                                          -- (x>y ∧ x>z) → (x-y ⩽ z) → (x ⩽ y+z)
                                   (_                , _                ) ⇒ compErr (VNeu x) (VNeu y))
                          <|> do (xa, xb) ← isPlus x
                               valEq VZ xb
                               comp xa (VNeu y)                       -- (x ⩽ y) → (x+Z ⩽ y)
comp' x        y        = compErr x y
comp : Value → Value → Result Comp
```

```
comp x y = comp' x y
        <|> case x of
              VNeu x' ⇒ do (xa, xb) ← isPlus x'
                           x'' ← vplus xb xa
                           comp' x'' y   -- (a+b ⩽ y) → (b+a ⩽ y)
              _       ⇒ compErr x y


  -- shift the time signature (i.e. Nat<n> -> Nat<n> becomes Nat<n+1> -> Nat<n+1>)
tshift : Comp → Ty → Result Ty
tshift (CLTE VZ t)       = return t
tshift (CLTE d) VNat     = return VNat
tshift (CLTE d) (VSeq t k l) = do kd ← vplus d k
                                  ld ← vplus d l
                                  return (VSeq t kd ld)
tshift (CLTE d) (VPi s t)    = case s of
                                  VPi ss st ⇒ return $ VPi s (λx ⇒ do v ← t x; tshift (CLTE d) v)
                                  s         ⇒ do s' ← tshift (CLTE d) s
                                                 return $ VPi s' (λx ⇒ do v ← t x; tshift (CLTE d) v)
tshift _ t               = return t


  -- turn untimed functions (Nat -> Nat) into timed functions (Nat<n> -> Nat<n>)
addTime : Value → Ty → Result Ty
addTime k (VNat)     = return (VSeq VNat k k)
addTime k (VVect t k) = return (VVect t k)
addTime k (VPi s t)  = let t' = (λx ⇒ do v ← t x; addTime k v) in
                       case s of
                         VPi _ _ ⇒ return (VPi s t')
                         _       ⇒ do s' ← addTime k s
                                      return (VPi s' t')
addTime k _          = Left "Can't create time signature"


unify : Nat → List PDecl → Vect n (Name, Ty)
      → ITerm → Ty → (Value → Result Ty) → CTerm → Result Ty
unify n D G f s t (CLam x) = do ctype n D G (CLam x) s
                                v ← ceval D [] (CLam x)
                                t v
unify n D G f s t (CInf x)  = do tx ← itype n D G x
                                 case (tx, s) of
                                   (VSeq tx kx lx, VSeq ts ks ls) ⇒ do ck ← comp ks kx
                                                                       cl ← comp ls lx
                                                                       cmpEq ck cl
                                                                       t' ← type' tx ts
                                                                       tshift cl t'
                                   (VSeq tx kx lx, s            ) ⇒ do t' ← type' tx s
                                                                       addTime lx t'
                                   (tx           , VSeq ts ks ls) ⇒ type' (VSeq tx ks ls) s
                                   (tx           , s            ) ⇒ type' tx s
                                 where
                                   type' : Ty → Ty → Result Ty
                                   type' tx ts = do valEq tx ts
                                                    v ← ceval D [] (CInf x)
                                                    t v


 {– the checker -}
itype : Nat → List PDecl → Vect n (Name, Ty) → ITerm → Result Ty
itype n D G (IStar)                = return VStar
itype n D G (IPi s t)              = let t' = csubst 0 (IFree (Local n)) t in
                                     do ctype n D G s VStar
                                        s' ← ceval D [] s
                                        ctype (n + 1) D ((Local n, s') :: G) t' VStar
```

```
                                  return VStar
itype n D G (IBound i)          = Left $ "illegal bound variable " ++ show i
itype n D G (IFree x)           = do maybeToEither ("unkown identifier " ++ show x)
                                       (lookup x G)
itype n D G (IAnn e t)          = do ctype n D G t VStar
                                     t' <- ceval D [] t
                                     ctype n D G e t'
                                     return t'
itype n D G (IApp f x)          = do tf <- itype n D G f
                                     case tf of
                                       VPi s t -> case unify n D G f s t x of
                                                    Right t' -> return t'
                                                    Left er  -> Left er
                                       _       -> Left $ "illegal application"
itype n D G (INat)              = return VStar
itype n D G (IZ)                = return VNat
itype n D G (IS t)              = do ctype n D G t VNat
                                     return VNat
itype n D G (INElim m mz ms k)  = do ctype n D G m (VPi VNat (\_ -> return VStar))
                                     mv  <- ceval D [] m
                                     mzt <- vapp mv VZ
                                     ctype n D G mz mzt
                                     mst <- return $ VPi VNat
                                       (\l -> do s  <- vapp mv l
                                                 t  <- vapp mv (VS l)
                                                 return $ VPi s (\_ -> return t))
                                     ctype n D G ms mst
                                     ctype n D G k VNat
                                     kv <- ceval D [] k
                                     vapp mv kv
itype n D G (IVect t k)         = do ctype n D G t VStar
                                     ctype n D G k VNat
                                     return VStar
itype n D G (IVNil t)           = do ctype n D G t VStar
                                     vt <- ceval D [] t
                                     return $ VVect vt VZ
itype n D G (IVCons t k x xs)   = do ctype n D G t VStar
                                     vt <- ceval D [] t
                                     ctype n D G k VNat
                                     vk <- ceval D [] k
                                     ctype n D G x vt
                                     ctype n D G xs (VVect vt vk)
                                     return $ VVect vt (VS vk)
itype n D G (IVElim t m mz ms k xs) = do ctype n D G t VStar
                                         vt <- ceval D [] t
                                         ctype n D G m (VPi VNat (\_ -> return VStar))
                                         vm <- ceval D [] m
                                         vmz <- vapp vm VZ
                                         ctype n D G mz vmz
                                         ctype n D G ms (VPi VNat
                                           (\vk -> do mk  <- vapp vm vk
                                                      msk <- vapp vm (VS vk)
                                                      pi1 <- (\_ -> return msk)
                                                      pi2 <- (\_ -> return $ VPi mk pi1)
                                                      return $ VPi vt pi2))
                                         ctype n D G k VNat
                                         vk <- ceval D [] k
                                         ctype n D G xs (VVect vt vk)
                                         vapp vm vk
itype n D G (ISeq t l k)        = do ctype n D G l VNat
                                     ctype n D G k VNat
                                     ctype n D G t VStar
```

```
                                    t' ← ceval D [] t
                                    case t' of     -- check if t is a representable type
                                      VNat ⇒ return VStar
                                      _    ⇒ Left "type is not Nat"
itype n D G (ISCons x xs)         = do txs ← itype n D G xs
                                    case txs of
                                      VSeq vt vl vk ⇒ do ctype n D G x (VSeq vt (VS vk) (VS vk))
                                                         return $ VSeq vt vl (VS vk)
                                      _             ⇒ Left "Tail of SCons is not of a timed type"
itype n D G (ISElim t l m mz ms k xs) = do ctype n D G t VStar
                                    vt ← ceval D [] t
                                    ctype n D G l VNat
                                    vl ← ceval D [] l
                                    ctype n D G m (VPi VNat (λvk ⇒ return VStar))
                                    vm ← ceval D [] m
                                    vmz ← vapp vm VZ
                                    ctype n D G mz vmz
                                    ctype n D G ms (VPi VNat
                                      (λvk ⇒ do d   ← vplus vk vl
                                                mk  ← vapp vm vk
                                                msk ← vapp vm (VS vk)
                                                pi1 ← return (λ_ ⇒ return msk)
                                                pi2 ← return (λ_ ⇒ return $ VPi mk pi1)
                                                return $ VPi (VSeq vt d d) pi2))
                                    ctype n D G k VNat
                                    vk ← ceval D [] k
                                    vd ← vplus vk vl
                                    ctype n D G xs (VSeq vt vl vd)
                                    vapp vm (VS vk)

ctype : Nat → List PDecl → Vect n (Name, Ty) → CTerm → Ty → Result ()
ctype n D G (CInf e)   (VSeq vtt vkt vlt) = do s ← itype n D G e
                                    case s of
                                      VSeq vts vks vls ⇒
                                        do valEq vts vtt
                                           case (comp vks vkt, comp vls vlt) of
                                             (Right (CLTE k), Right (CLTE l)) ⇒ do valEq k l
                                             (Right _       , Right _       ) ⇒ Left "Delay mismatch"
                                             (_             , _             ) ⇒ Left er
                                      vts ⇒
                                        do valEq vts vtt
                                           valEq vkt vlt
ctype n D G (CInf e)   tt         = do ts ← itype n D G e
                                       valEq ts tt
ctype n D G (CLam e) (VPi s t)    =  let e' = (csubst 0 (IFree (Local n)) e) in
                                     do t' ← t (VNeu (NFree (Local n)))
                                        ctype (n+1) D ((Local n, s) :: G) e' t'
ctype n D G _          _          = Left "Type mismatch"


isubst : Nat → ITerm → ITerm → ITerm
isubst n r (IStar)                = IStar
isubst n r (IPi s t)              = IPi (csubst n r s) (csubst (n+1) r t)
isubst n r (IAnn e t)             = IAnn (csubst n r e) (csubst n r t)
isubst n r (IBound m)             = if (m == n) then r else IBound m
isubst n r (IFree x)              = IFree x
isubst n r (IApp f x)             = IApp (isubst n r f) (csubst n r x)
isubst n r (INat)                 = INat
isubst n r (IZ)                   = IZ
isubst n r (IS t)                 = IS (csubst n r t)
isubst n r (INElim m mz ms k)     = let s = csubst n r in INElim (s m) (s mz) (s ms) (s k)
isubst n r (IVect t k)            = IVect (csubst n r t) (csubst n r k)
isubst n r (IVNil t)              = IVNil (csubst n r t)
isubst n r (IVCons t k x xs)      = let s = csubst n r in IVCons (s t) (s k) (s x) (s xs)
```

```
isubst n r (IVElim t m mz ms k xs)  = let s = csubst n r in IVElim (s t) (s m) (s mz) (s ms) (s k) (s xs)
isubst n r (ISeq t l k)             = let s = csubst n r in ISeq (s t) (s l) (s k)
isubst n r (ISCons x xs)            = ISCons (csubst n r x) (isubst n r xs)
isubst n r (ISElim t l m mz ms k xs) = let s = csubst n r in ISElim (s t) (s l) (s m) (s mz) (s ms) (s k) (s xs)

csubst : Nat → ITerm → CTerm → CTerm
csubst n r (CInf e)  = CInf (isubst n r e)
csubst n r (CLam e) = CLam (csubst (n + 1) r e)


quote0 : Result Value → Result CTerm
quote0 rv = do v ← rv
               quote 0 v

quote : Nat → Value → Result CTerm
quote n (VLam f)              = do v ← f (VNeu (NFree (Quote n)))
                                   e ← quote (n + 1) v
                                   return (CLam e)
quote n (VStar)               = return (CInf IStar)
quote n (VPi v f)             = do s  ← quote n v
                                   t  ← f (VNeu (NFree (Quote n)))
                                   t' ← quote (n + 1) t
                                   return (CInf (IPi s t'))
quote n (VNeu v)              = do e ← neutralQuote n v
                                   return (CInf e)
quote n (VNat)                = return (CInf INat)
quote n (VZ)                  = return (CInf IZ)
quote n (VS v)                = do t ← quote n v
                                   return (CInf (IS t))
quote n (VVect vt vk)         = do t ← quote n vt
                                   k ← quote n vk
                                   return (CInf (IVect t k))
quote n (VVNil vt)            = do t ← quote n vt
                                   return (CInf (IVNil t))
quote n (VVCons vt vk vx vxs) = do t  ← quote n vt
                                   k  ← quote n vk
                                   x  ← quote n vx
                                   xs ← quote n vxs
                                   return (CInf (IVCons t k x xs))
quote n (VSCons vx vxs)       = do x  ← quote n vx
                                   xs ← quote n vxs
                                   case xs of
                                     CInf xs' ⇒ return (CInf (ISCons x xs'))
                                     _        ⇒ Left $ "internal error in quote"


neutralQuote : Nat → Neutral → Result ITerm
neutralQuote n (NFree x)                 = return (boundfree n x)
neutralQuote n (NApp vf vx)              = do x ← quote n vx
                                              f ← neutralQuote n vf
                                              return (IApp f x)
neutralQuote n (NNElim vm vmz vms vk)    = do m  ← quote n vm
                                              mz ← quote n vmz
                                              ms ← quote n vms
                                              k  ← neutralQuote n vk
                                              return (INElim m mz ms (CInf k))
neutralQuote n (NVElim vt vm vmz vms vk vxs) = do t  ← quote n vt
                                                  m  ← quote n vm
                                                  mz ← quote n vmz
                                                  ms ← quote n vms
                                                  k  ← quote n vk
                                                  xs ← neutralQuote n vxs
                                                  return (IVElim t m mz ms k (CInf xs))
neutralQuote n (NSElim vt vl vm vmz vms vk vxs) = do t  ← quote n vt
                                                     l  ← quote n vl
```

```
    m  ← quote n vm
    mz ← quote n vmz
    ms ← quote n vms
    k  ← quote n vk
    xs ← neutralQuote n vxs
    return (ISElim t l m mz ms k (CInf xs))
```

```
boundfree : Nat → Name → ITerm
boundfree n (Quote k) = IBound (n − k − 1)
boundfree n x         = IFree x
```

Bibliography

[Baa09] Christiaan Baaij. Clash, from haskell to hardware. Master's thesis, University of Twente, 2009.

[Bar84] Henk P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North Holland, 1984.

[Bar93] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, vol. 2*, pages 117–309. Oxford University Press, 1993.

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. *SIGPLAN Not.*, 34(1):174–184, September 1998.

[Bra13] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[CH09] F. Cardone and J. R. Hindley. History of lambda-calculus and combinatory logic. In *Handbook of the History of Logic, vol. 5*. North Holland, 2009.

[Coq86] Thierry Coquand. An analysis of girard's paradox. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 227–236. IEEE Computer Society Press, 1986.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[End01] Herbert B. Enderton. *A mathematical introduction to logic*. Academic Press, San Diego, CA, USA, 2001.

[Jon03] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[Koo09] M. Kooijman. Haskell as a higher order structural hardware description language. Master's thesis, University of Twente, 2009.

[LMS10] Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently-typed lambda calculus. *Fundamentae Informatica*, 102:177–207, April 2010.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[Ott13] Gerald Otter. Timed typed for synchronous hardware. Master's thesis, University of Twente, 2013.

[SJ04] Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in forsyde. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23:1, s. 17-32, 2004.

[Tho91] Simon Thompson. *Type Theory and Functional Programming*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1991.