SOUND AND RELATIVELY COMPLETE COEFFECT AND EFFECT REFINEMENT TYPE SYSTEMS FOR CALL-BY-PUSH-VALUE PCF

Masterarbeit im Fach Informatik
Master's Thesis in Computer Science

von / by
Maxi Wuttke

angefertigt unter der Leitung von / supervised by
Prof. Dr. Deepak Garg

begutachtet von / reviewed by
Prof. Dr. Deepak Garg
Prof. Dr. Derek Dreyer

Saarland Informatics Campus, April 2021

UNIVERSITÄT DES SAARLANDES

MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS

Eidesstattliche Erklärung
Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath
I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung
Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent
I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 15th April, 2021

Abstract

In this thesis, we study coeffect-based and effect-based refinement type systems for verification and complexity analysis of pure functional recursive programs. These type systems are relatively complete, which roughly means that they can be fine-tuned either for expressiveness or for tractability.

We consider two approaches: using coeffects and using effects. For the first approach, we generalise and simplify previous work by introducing a system that targets a call-by-push-value (CBPV) version of the programming language PCF, which supports higher-order recursive functions. We derive soundness and relative completeness for the new system. From this, these properties also follow for the old systems. In the effect-based approach, we also target CBPV.

For both approaches, we explain how the systems can be extended with features of modern programming language like polymorphism. One of the key properties of these systems is that they are compositional, which enables modular verification. We (informally) discuss efficient, sound and complete type inference algorithms that exploit this fact. Finally, we (informally) compare and combine both approaches, and we formalise a sound and relatively complete coeffect-based system from prior work in the proof assistant Coq.

Acknowledgements

I want to thank my parents and grandparents for their kindness and love.

I want to thank Deepak Garg for advising me during this project. He guided me through highs and lows and always provided much constructive feedback. He is a great source of inspiration for me.

I am grateful for the funding provided by the Max Planck Society and the Graduate School Computer Science at Saarland University.

Further, I want to thank Gert Smolka and Deepak Garg for supporting my application to the graduate programme at the Max Planck Institute for Software Systems. I am glad that I got the opportunity to continue pursuing a PhD at MPI SWS after I finished this thesis.

Last but not least, I am thankful to Derek Dreyer for agreeing to be the second reviewer of this thesis.

Contents

List of Figures

Chapter 1

Introduction

As safety and performance critical software and hardware systems are ubiquitous, verification of these systems is essential. There are various important properties that systems must fulfil. Perhaps the most well-known class of properties are safety properties, like *functional correctness.* A system is said to be correct if it fulfils a functional specification, i.e. a mathematical mapping of inputs to outputs, and if it never crashes on valid inputs. However, functional correctness is not enough in practice, since even programs that never terminate are considered (functionally) ‘correct’. Verification of other properties, like termination and efficiency, is indispensable for computing systems that ought to compute an answer in a certain amount of time and use only a certain amount of memory.

In this thesis, we discuss *type*-based approaches to verification of functional behaviour and running time of programs.

Type systems Type systems are one of the most-widely used approaches to ‘lightweight’ program verification, especially in functional programming languages. The famous slogan by Robin Milner, *well-typed programs cannot “go wrong”*[29], summarises one of the merits of type systems, namely that they exclude a (more or less wide) range of errors. In well-typed programs, type errors (like adding a truth value to a number) are excluded during program evaluation. Moreover, type systems are usually automated and many systems provide good feedback to the programmer if there are (potential) errors. One important property of most type systems is *compositionality*, which states that separate components can be typed separately. For example, if we can assign the type $\mathsf{Nat} \to \mathsf{Bool}$ to a term t_1 and the type Nat to another term t_2, then the application of the terms, written $t_1\, t_2$, can be assigned the type Bool. This means that t_1 expects as input any natural number and returns a Boolean (truth value). For example, t_1 could return true if and only if the number is even, but this specification is usually not expressed in this type of t_1.

The designers of type systems have to find a compromise between *expressiveness* and *tractability* of a type system, since safety properties, in general, are undecidable. On one extreme, there are *dependent type systems*, which can be used for specifying and verifying properties of programs *inside* the system itself, as dependent types can refer to concrete terms. For example, the dependent type $\forall n : \mathsf{Nat}.\, (\{\exists n' : \mathsf{Nat}.\, n = 2n'\} + \{\exists n' : \mathsf{Nat}.\, n =$

$2n'+1\}$) expresses that a function takes a natural number n as input and either computes a number n' such that $n = 2n'$ or a number n' such that $n = 2n' + 1$. This type suffices to functionally specify the program.

One famous implementation of such a dependent type system is Coq [38]. Coq's logic, the calculus of (co)inductive constructions, employs the *proof-as-programs* correspondence (also known as the Curry-Howard isomorphism): Types are seen as propositions and programs are seen as proofs. For example, the above dependent type states that every number is either even or odd; a (*constructive*) 'proof' of this fact is a decision procedure that either yields a proof for "n is even" or "n is odd". Moreover, one of the main strengths of Coq is that one can extend its logic (in a sound way) by defining inductive and coinductive data types. This can be used to embed other programming languages inside Coq. Moreover, we can prove theorems about embedded programming languages and propositions about embedded terms. Among many theorems in the theory of programming languages, it is possible to show that every typed program of the simply typed λ-calculus terminates (see, e.g. [5]). However, it is not possible to specify the *running time* of Coq terms inside Coq itself, although we can reason about the complexity of (deeply) embedded programs.

Refinement type systems Refinement type systems are less expressive than (fully) dependent type systems, but are more practical to implement. A well-known problem to programmers of Standard ML [30] is taking the first element (called the *head* element) of a list that is assumed to be non-empty (i.e. non-*nil*). Such an assumption is usually stated outside the program, for example in a comment. It is the obligation of the user of the function to prove that it is not called with an empty list as argument, since the function could crash otherwise. However, the type checker is not aware of such an assumption, and therefore emits a warning:

```
(* hd : list -> int *)
(* The list must be non-nil *)
fun hd (x :: xs) = x     (* Warning: match non-exhaustive *)
```

The type system in [16] overcomes this problem by making it possible to define a type of non-empty lists that is a *refinement* of the type of lists. This means, a non-empty list can be used everywhere were any list is required, but only terms of the type `nonEmptyList` can be applied to the function `hd`, which is assigned the type `nonEmptyList -> int`. In this system, it can also be expressed that if a non-empty list is appended to any list, the resulting list is non-empty.

Refinement type systems can also be used for specifying and verifying functional correctness of programs [14]. For example, Dependent ML (DML) [39] is an extension of Standard ML. In contrast to fully dependent type systems, DML has two layers of types and terms: *Index terms* (which denote natural numbers) are used to refine the types of (computational) terms. For example, the type $int(I)$ is only inhabited by the constant n that is equivalent to the meaning of the index term I. Moreover, assertions and assumptions can be added to types in order to express invariants. For example, the type

$\exists a. (\mathit{int}(a) \wedge a > 0)$ stands for the positive natural numbers. Lists can be refined with their length. For example, the application function can be assigned the following type:

$$\forall i_1\, i_2. (\mathit{list}(i_1) \to \mathit{list}(i_2) \to \mathit{list}(i_1 + i_2))$$

If we combine refined base types with quantification over index terms, we can also specify the functional behaviour of functions on integers or natural numbers. For example, the type $\forall a. (\mathit{int}(a) \to \mathit{int}(I(a)))$ stands for functions that take an integer a as argument and compute a number that is equivalent to the index term $I(a)$. Note that this approach is different from (fully) dependent types, since we only quantify over type refinements, not over arbitrary language terms.

Subtyping obligations in DML are reduced to assertions on propositions on index terms. For example, the subtyping judgement $\mathit{int}(I_1) \sqsubseteq \mathit{int}(I_2)$ holds if and only if the two index terms I_1 and I_2 are equivalent. DML is parametrised by a language $\mathcal{L}$ of index terms. Thus, if one chooses a computationally tractable language of index terms, off-the-shelf tools like SMT solvers can be used to discharge most assertions. In case the solver fails to prove the obligations, this could either mean that the subtyping does not hold or that the solver was not powerful enough to discharge the obligation. In the latter case, the user could prove the obligations, e.g. with the help of an interactive proof assistant. However, this contradicts the goal of automation, since the user would have to re-prove these obligations every time the code is changed. Yet, if we assume that $\mathcal{L}$ is sufficiently expressive and its theory is complete, DML is *complete*. This phenomena, called *relative completeness*, also holds for both families of type systems that we discuss in this thesis, namely $\mathsf{d}\ell\mathsf{PCF}$ and $\mathsf{d}f\mathsf{PCF}$. In other words, the systems are complete relative to completeness of the language of index terms.

Effects and coeffects Before we discuss how refinement type systems can be used for complexity analysis, we first discuss two fundamental concepts: *Effects* and *coeffects* are two dual views on how a program interacts with its environment. We use the word *environment* in an abstract sense here. Most programs, for example, depend on operating systems and software libraries, and some programs may even require specialised hardware. Moreover, restrictions on *time and space* are also crucial for applications of software systems, and they are thus also part of the 'environment'. In embedded systems, for example, programs can only use a constant amount of memory, which should ideally be low. Similarly, real-time and security-critical systems ought to compute an answer in predictable time.

Effects, roughly, are interactions of the program with its environment that are initiated by the program. For example, most programs produce some kind of output. In imperative programming languages it is possible to change the value of variables that are shared among different parts of the program. In particular, incrementing a global counter constitutes an effect. Moreover, many programming languages provide primitives for modifying the execution stack, for example by throwing an exception or aborting the program, or by spawning sub-processes.

On the other hand, *coeffects* [33, 32] describe how the environment affects the program. For example, a program may need to read data from a file or sensors, interact with POSIX-like environment variables, or use up certain abstract *resources*. One can think of many kinds of resources. For example, the program may only be allowed to allocate a certain amount of memory. It may have to 'pay' for certain operations, which is useful for amortised cost analysis [37].

Effect and coeffect type systems are type systems that are augmented with effects and coeffects, respectively. Effect type systems can be used, for example, to analyse to which memory cells a program may potentially write. Coeffect systems can be used to enforce that a program only refers to a variable a certain number of times.

(Co)effect type systems for cost analysis We define the *cost* of a (terminating) program as the number of times certain operations are executed during its execution. For example, the cost of an execution could be defined as the number of times a variable is accessed or a function is applied.

So how can effects and coeffects be used to analyse the cost of programs? From the perspective of effects, we view these 'costly' operations as effectful operations that increment a virtual global counter. Effect type systems for cost analysis can bound the number of times this virtual counter is incremented.

Coeffect type systems do not directly analyse the number of times costly operations are executed. Instead, every execution of such an operation constitutes a use of one abstract resource. Coeffect systems thus bound the number of times these resources can potentially be consumed. To analyse the cost of a closed program, we count how many of these resources are *allocated*, since a (closed) program can only use these resources that are part of its input or are allocated by the program itself. This idea actually comes from linear logic [18] and *bounded linear logic* [19] in particular. We will discuss the connection between the coeffect-based systems and (bounded) linear logic in more detail later.

Call-by-push-value The same program can have different costs if the programming language admits different executions. In other words, costs depend on the *evaluation strategy* (e.g. call-by-name (CBN) or call-by-value (CBV), which we will recapitulate later). This is one of the reasons why we need different type systems for different evaluation strategies.

Call-by-push-value (CBPV) [27] is a paradigm that *subsumes* the call-by-name and call-by-value strategies. It is based on the idea that "a value *is*, a computation *does*". Subsumption implies that there are two translations of programs to call-by-push-value programs – a CBN translation and a CBV translation. It can be shown that a program and its CBN/CBV translation behave observationally equivalent to its CBN/CBV semantics. Moreover, for different kinds of (concrete or abstract) semantics, call-by-name and call-by-value semantics can be 'translated' to call-by-push-value semantics. Subsumption also makes it possible that once we have proved, for example, a semantic soundness theorem for CBPV, the theorem can be 'translated back' to the respective theorems for CBN and CBV. We will discuss CBPV in more detail in Section 2.3.

In prior work, coeffect-based type systems for a call-by-name programming language ($\mathsf{d}\ell\mathsf{PCF}_\mathsf{n}$ [11]) and a call-by-value programming language ($\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$ [12]) have been developed. We introduce a new system, $\mathsf{d}\ell\mathsf{PCF}_\mathsf{pv}$, that targets a call-by-push-value language, and show that this system in fact subsumes the two prior systems in the above sense.

Contributions of this thesis First, we unify and simplify prior work on coeffect-based systems for verification and complexity analysis. More concretely, we:

- simplify the formal proof of soundness and relative completeness of $\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$;
- we introduce a new system $\mathsf{d}\ell\mathsf{PCF}_\mathsf{pv}$ that subsumes the call-by-name and call-by-value version of $\mathsf{d}\ell\mathsf{PCF}$;
- thereby, we derive proofs of the above properties for the two other systems;
- we review and generalise a type inference algorithm for $\mathsf{d}\ell\mathsf{PCF}$, and we add support for polymorphism.

In the second part, we:

- introduce new effect-based type systems (the $\mathsf{d}f\mathsf{PCF}$ family);
- and we introduce type inference algorithms for these systems.

We also (informally) compare the two approaches and discuss their strengths and weaknesses.

Structure of this thesis In the next chapter, we will first define the programming languages that we target for our type systems, namely System T and PCF [34]. In particular, we recapitulate a call-by-push-value version of PCF (which we call CBPV) in Section 2.3.

The remainder of this thesis is structured in two main parts, in which we study coeffect-based ($\mathsf{d}\ell\mathsf{PCF}$) and effect-based systems ($\mathsf{d}f\mathsf{PCF}$), respectively.

In Part I, we consider the coeffect-based approach to our problem. We first explain and motivate, in Chapter 3, the basic ideas of $\mathsf{d}\ell\mathsf{PCF}$ from the perspective of bounded linear logic [19]. Afterwards, we introduce a type system for System T, which is a total language. In this chapter, we also introduce the index term language $\mathcal{L}^\ell_{idx}$ that is used for the systems in the first part. Then, we review the call-by-value version of $\mathsf{d}\ell\mathsf{PCF}$ in Chapter 5, where we need to consider unbounded recursion. In Chapter 6, we briefly recapitulate the call-by-name version of $\mathsf{d}\ell\mathsf{PCF}$, but we do not spell out any proofs. We introduce $\mathsf{d}\ell\mathsf{PCF}_\mathsf{pv}$, in Chapter 7, and we show that it subsumes the other two versions of $\mathsf{d}\ell\mathsf{PCF}$ and derive soundness and relative completeness for all versions of $\mathsf{d}\ell\mathsf{PCF}$. We also discuss how we can extend $\mathsf{d}\ell\mathsf{PCF}$ with product and sum types. In Chapter 8, we first discuss a type inference algorithm for $\mathsf{d}\ell\mathsf{PCF}_\mathsf{pv}$, which is based on a similar algorithm in [13]. Furthermore, we extend the language with polymorphism and show how polymorphism can be used to encode bounded recursive data types. Finally, we observe that polymorphism does not pose a problem for the type inference algorithm.

In the introductory chapter of Part II, we first explain the main weak points of the first approach, and we discuss how we tackle these problems in the effect-based approach. Again, we start with a system for System T, in Chapter 10, where we also introduce a new index term language, $\mathcal{L}^f_{idx}$, and prove compositional completeness using a type inference algorithm. Then, we generalise this system to CBPV and extend the algorithm.

In the last part of this thesis, we discuss and compare the coeffect and effect-based approaches. We outline how the coeffect and effect systems can be combined, which makes the coeffect system more expressive. Finally, we summarise other approaches to verification and complexity analysis, and propose future work.

In Appendix A, we list proofs that are omitted in the main part. In Appendix B, we outline our Coq formalisation of $\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$.

Chapter 2

Programming languages preliminaries

In this chapter, we first recapitulate some simple programming languages. The coeffect-based and effect-based type systems that we will discuss in this thesis are *refinements* of the simple type systems that we present in this chapter. This means that typings in $\mathsf{d\ell PCF_v}$, for example, have the same structure as simple PCF typings, but they contain additional information. In other words, a $\mathsf{d\ell PCF_v}$ typing can be converted to a (simple) PCF typing by removing the refinements.

2.1 System T

System T[1] is a *total* (and thus Turing incomplete) programming language, which means that all well-typed programs terminate. Yet, it is very expressive (at least in an *extensional* sense): All total natural functions of intuitionistic arithmetic (equivalently, all higher-order primitive recursive functions) can be encoded in System T. These properties makes it an attractive programming language for type-based complexity analysis, as we do not have to deal with non-termination.

2.1.1 Syntax of System T

We consider a version of System T with natural numbers, λ-abstractions, higher-order iteration, and binary sums and products.

$$\begin{array}{ll} \text{Terms:} & t ::= v \mid x \mid t_1\, t_2 \mid \mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3 \mid \mathsf{Succ}(t) \mid \mathsf{Pred}(t) \\ & \quad\;\; \mid \langle t_1; t_2\rangle \mid \pi_1(t) \mid \pi_2(t) \mid \mathsf{inl}(t) \mid \mathsf{inr}(t) \mid \mathsf{case}\, t\, [\mathsf{inl}(x) \Rightarrow t_1 \mid \mathsf{inr}(y) \Rightarrow t_2] \\ \text{Values:} & v ::= \underline{n} \mid \lambda x.\, t \mid \mathsf{iter}\, t_1\, t_2 \mid \langle\rangle \mid \langle v_1; v_2\rangle \mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \end{array}$$

The meta variables i, n and k range over natural numbers, x over (term) variables, t over terms. The symbol v is used for *values*, which are a subset of terms that are already fully

---

[1] System T was introduced by Gödel in an article about proof theory in the *Dialectica* journal in 1958.

evaluated. In other words, they are terminal, or in normal form. Note that a tuple is a value if and only if both of its components are values. One can also derive n-ary products and projections as syntactic sugar.

The ifz operator first evaluates t_1 to a constant $\underline{n}$. If it is zero, the execution is continued in t_2, and in t_3 otherwise.

Free variables of terms are defined in the standard way. Terms without free variables are called *closed* (and *open* otherwise). We consider terms to be equal if they are equivalent up to variable renaming. In the entire thesis, we never substitute open terms for variables, since we never reduce *below binders*. Thus, *capturing* of variable names cannot happen, since only closed terms are executed. When we introduce new binders, we always assume that they are fresh.

Substitution of closed terms for variables is defined in the standard way:

Definition 2.1 (Substitution)**.** Let t' be a closed term and let t a term that may have the variable x free. We define $t\{t'/x\}$ by recursion on t:

$$
\begin{aligned}
y\{t'/x\} &:= \begin{cases} t' & x = y \\ y & x \neq y \end{cases} \\
(\lambda y.\, t)\{t'/x\} &:= \lambda y.\, t\{t'/x\} \\
(\mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3)\{t'/x\} &:= \mathsf{ifz}\, t_1\{t'/x\}\, \mathsf{then}\, t_2\{t'/x\}\, \mathsf{else}\, t_3\{t'/x\} \\
(\mathsf{Succ}(t))\{t'/x\} &:= \mathsf{Succ}(t\{t'/x\}) \\
(\mathsf{Pred}(t))\{t'/x\} &:= \mathsf{Pred}(t\{t'/x\}) \\
\langle t_1; t_2\rangle\, \{t'/x\} &:= \langle t_1\{t'/x\}; t_2\{t'/x\}\rangle \\
(\pi_i(t))\{t'/x\} &:= \pi_i(t\{t'/x\}) \\
(\mathsf{inl}(t))\{t'/x\} &:= \mathsf{inl}(t\{t'/x\}) \\
(\mathsf{inr}(t))\{t'/x\} &:= \mathsf{inr}(t\{t'/x\}) \\
(\mathsf{case}\, t\, [\mathsf{inl}(y_1) \Rightarrow t_1 \mid \mathsf{inr}(y_2) \Rightarrow t_2])\{t'/x\} &:= \mathsf{case}\, t\{t'/x\}\, [\mathsf{inl}(y_1) \Rightarrow t_1\{t'/x\} \mid \mathsf{inr}(y_2) \Rightarrow t_2\{t'/x\}] \\
(\mathsf{iter}\, t_1\, t_2)\{t'/x\} &:= \mathsf{iter}\, (t_1\{t'/x\})\, (t_2\{t'/x\})
\end{aligned}
$$

In the λ and case distinction cases, we assume (as usual) that the binder variables are distinct from the substituted variable. Moreover, we can do a sequence of substitutions:

$$t\{t_1/x_1, \ldots, t_n/x_n\} := t\{t_1/x_1\} \cdots \{t_n/x_n\}$$

The order of the substitutions does not matter, since we always assume that the terms $t_1, \ldots, t_n$ are closed.

2.1.2 Semantics of System T

Our variant of System T has *call-by-value* semantics. This means that in an application $t_1\, t_2$, t_1 and t_2 first have to be evaluated. In particular, t_1 has to evaluate either to a λ-abstraction or to an iteration. If t_1 evaluates to an iteration, the argument t_2 has to evaluate to a number. Similarly, before we can project a tuple or do a case analysis on a sum, it has to be fully evaluated first.

The small-step semantics is augmented with a *cost* i for each step: $t_1 \succ_i t_2$, which may either be 0 or 1. It is 1 for β-substitutions and iter unfolding, and 0 otherwise. We write $t \succ t'$ if we do not care about the cost. Both kinds of semantics are summarised in Figure 2.1. We use *program contexts* to streamline the definition of the small-step semantics:

$$\begin{aligned} C ::= {} & \bullet \mid C\,t_2 \mid v_1\,C \mid \mathsf{ifz}\,C\,\mathsf{then}\,t_2\,\mathsf{else}\,t_3 \mid \langle C; t_2\rangle \mid \langle v_1; C\rangle \\ & \mid \mathsf{inl}(C) \mid \mathsf{inr}(C) \mid \mathsf{case}\,C\,[\mathsf{inl}(x) \Rightarrow t_1 \mid \mathsf{inr}(y) \Rightarrow t_2] \end{aligned}$$

A reduction $t \succ_i t'$ is either a head reduction (with the rules depicted in Figure 2.1), or a reduction inside a program context:^{2}

$$\frac{t \succ_i t'}{C[t] \succ_i C[t']}$$

Note that the head reduction rule for iteration implies that t_1 has to be (re-)executed for every iteration. We choose these semantics in order to make it possible to define both a coeffect and an effect type system for System T. However, if we want to avoid having t_1 executed for every iteration, we can transform the code using an eta-expansion, i.e. substitute $\mathsf{iter}\,t_1\,t_2$ with $(\lambda x.\,\mathsf{iter}\,x\,t_2)\,t_1$.

We also define big-step semantics: $t \Downarrow_i v$ means that t evaluates to v, and the cost of this evaluation is i. It is easy to prove that small-step and big-step semantics agree.

Lemma 2.2 (Agreement of the small-step and big-step semantics)**.** *Let t be a term and let v be a value. Then the following propositions are equivalent:*

- $t \Downarrow_i v$
- $t \succ_i^* v$*, where* $\succ_\cdot^*$ *sums up the cost of multiple steps:*

$$\frac{}{v \succ_0^* v} \qquad\qquad \frac{t \succ_{i_1} t' \qquad t' \succ_{i_2}^* v}{t \succ_{i_1+i_2}^* v}$$

2.1.3 Simple types

We define a simple type system for System T with the following types:

$$\begin{aligned} A & ::= \mathsf{Nat} \mid A_1 \to A_2 \mid A_1 \times A_2 \mid A_1 + A_2 \\ \Gamma & ::= \emptyset \mid x : A, \Gamma \end{aligned}$$

Typing contexts (contexts for short) assign a type to every free variable of a term. The empty context ($\emptyset$) can thus only be used for closed terms. The context $x : A, \Gamma$ assigns the type A to x and otherwise behaves like Γ; we assume that x is not in the domain of Γ. The 'order' of the variables in the context is thus irrelevant. We can also see contexts as a partial mapping from variables to types: $\Gamma(x)$ is the type of x in the context Γ.

$$(\lambda x.\,t)\,v \succ_1 t\{v/x\} \qquad (\mathsf{iter}\,t_1\,t_2)\,\underline{0} \succ_1 t_2 \qquad (\mathsf{iter}\,t_1\,t_2)\,\underline{1+n} \succ_1 t_1\,(\mathsf{iter}\,t_1\,t_2\,\underline{n})$$

$$\mathsf{ifz}\,\underline{0}\,\mathsf{then}\,t_2\,\mathsf{else}\,t_3 \succ_0 t_2 \qquad \mathsf{ifz}\,\underline{1+n}\,\mathsf{then}\,t_2\,\mathsf{else}\,t_3 \succ_0 t_3 \qquad \pi_k\,\langle v_1; v_2\rangle \succ_0 v_k$$

$$\mathsf{Succ}(\underline{n}) \succ_0 \underline{1+n} \qquad \mathsf{Pred}(\underline{n}) \succ_0 \underline{n \mathbin{\dot{-}} 1}$$

$$\mathsf{case}\,\mathsf{inl}(v)\,[\mathsf{inl}(x) \Rightarrow t_1 \mid \mathsf{inr}(y) \Rightarrow t_2] \succ_0 t_1\{v/x\}$$

$$\mathsf{case}\,\mathsf{inr}(v)\,[\mathsf{inl}(x) \Rightarrow t_1 \mid \mathsf{inr}(y) \Rightarrow t_2] \succ_0 t_2\{v/y\}$$

$$\frac{t_1 \Downarrow_{i_1} \lambda x.\,t \qquad t_2 \Downarrow_{i_2} v \qquad t\{v/x\} \Downarrow_{i_3} v'}{t_1\,t_2 \Downarrow_{1+i_1+i_2+i_3} v'} \qquad \frac{t_1 \Downarrow_{i_1} \mathsf{iter}\,t_1'\,t_2 \qquad t_3 \Downarrow_{i_2} \underline{0} \qquad t_2 \Downarrow_{i_3} v}{t_1\,t_3 \Downarrow_{1+i_1+i_2+i_3} v}$$

$$\frac{t_1 \Downarrow_{i_1} \mathsf{iter}\,t_1'\,t_2 \qquad t_3 \Downarrow_{i_2} \underline{1+n} \qquad t_1'\,(\mathsf{iter}\,t_1'\,t_2\,\underline{n}) \Downarrow_{i_3} v}{t_1\;t_3 \Downarrow_{1+i_1+i_2+i_3} v} \qquad \frac{t_1 \Downarrow_{i_1} \underline{0} \qquad t_2 \Downarrow_{i_2} v}{\mathsf{ifz}\,t_1\,\mathsf{then}\,t_2\,\mathsf{else}\,t_3 \Downarrow_{i_1+i_2} v}$$

$$\frac{t_1 \Downarrow_{i_1} \underline{1+n} \qquad t_3 \Downarrow_{i_2} v}{\mathsf{ifz}\,t_1\,\mathsf{then}\,t_2\,\mathsf{else}\,t_3 \Downarrow_{i_1+i_2} v} \qquad \frac{t \Downarrow_i \underline{n}}{\mathsf{Succ}(t) \Downarrow_i \underline{1+n}} \qquad \frac{t \Downarrow_i \underline{n}}{\mathsf{Pred}(t) \Downarrow_i \underline{n \mathbin{\dot{-}} 1}}$$

$$\frac{t_k \Downarrow_{i_k} v_k \text{ for } k = 1, 2}{\langle t_1; t_2\rangle \Downarrow_{i_1+i_2} \langle v_1; v_2\rangle} \qquad \frac{t \Downarrow_i \langle v_1; v_2\rangle}{\pi_k(t) \Downarrow_i v_k} \qquad v \Downarrow_0 v$$

$$\frac{t_1 \Downarrow_{i_1} \mathsf{inl}(v) \qquad t_2\{v/x\} \Downarrow_{i_2} v'}{\mathsf{case}\,t_1\,[\mathsf{inl}(x) \Rightarrow t_2 \mid \mathsf{inr}(y) \Rightarrow t_3] \succ_0 v'} \qquad \frac{t_1 \Downarrow_{i_1} \mathsf{inr}(v) \qquad t_3\{v/y\} \Downarrow_{i_2} v'}{\mathsf{case}\,t_1\,[\mathsf{inl}(x) \Rightarrow t_2 \mid \mathsf{inr}(y) \Rightarrow t_3] \succ_0 v'}$$

Figure 2.1: Head reduction rules and big-step semantics of System T

$$\textsc{Const}\quad \Gamma \vdash \underline{n} : \mathsf{Nat} \qquad \textsc{Var}\quad x : A, \Gamma \vdash x : A$$

$$\textsc{Lam}\quad \frac{x : A, \Gamma \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \textsc{Iter}\quad \frac{\Gamma \vdash t_1 : A \to A \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash \mathsf{iter}\, t_1\, t_2 : \mathsf{Nat} \to A}$$

$$\textsc{Succ}\quad \frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathsf{Succ}(t) : \mathsf{Nat}} \qquad \textsc{Pred}\quad \frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathsf{Pred}(t) : \mathsf{Nat}} \qquad \textsc{App}\quad \frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1\, t_2 : B}$$

$$\textsc{Ifz}\quad \frac{\Gamma \vdash t_1 : \mathsf{Nat} \qquad \Gamma \vdash t_2 : B \qquad \Gamma \vdash t_3 : B}{\Gamma \vdash \mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3 : B}$$

$$\textsc{Tuple}\quad \frac{\Gamma \vdash t_k : A_k \text{ for } k = 1, \ldots, 1}{\Gamma \vdash \langle t_1; t_2 \rangle : A_1 \times A_2} \qquad \textsc{Proj}\quad \frac{\Gamma \vdash t : A_1 \times A_2}{\Gamma \vdash \pi_i(t) : A_i} \qquad \textsc{Inl}\quad \frac{\Gamma \vdash t : A_1}{\Gamma \vdash \mathsf{inl}(t) : A_1 + A_2}$$

$$\textsc{Inr}\quad \frac{\Gamma \vdash t : A_2}{\Gamma \vdash \mathsf{inr}(t) : A_1 + A_2} \qquad \textsc{CaseSum}\quad \frac{\Gamma \vdash t_1 : A_1 + A_2 \qquad x : A_1, \Gamma \vdash t_2 : B \qquad y : A_2, \Gamma \vdash t_3 : B}{\Gamma \vdash \mathsf{case}\, t_1\, [\mathsf{inl}(x) \Rightarrow t_2 \mid \mathsf{inr}(y) \Rightarrow t_3] : B}$$

Figure 2.2: Simple typing rules of System T

The typing rules (which are all standard), are showed in Figure 2.2. The following properties are standard:

Lemma 2.3 (Substitution)**.** *If* $x : A_1, \Gamma \vdash t : A_2$ *and* $\emptyset \vdash v : A_1$*, then* $\Gamma \vdash t\{v/x\} : A_2$*.*

Lemma 2.4 (Subject reduction)**.** *If* $\emptyset \vdash t : A$ *and* $t \succ t'$*, then* $\emptyset \vdash t' : A$*.*

Lemma 2.5 (Progress)**.** *If* $\emptyset \vdash t : A$*, then either* t *is a value, or there exists a successor term* $t \succ t'$*.*

A *program* is a closed term with the simple type Nat. By the above lemmas, a program either diverges or evaluates to a constant.

2.1.4 Example terms

Primitive recursive functions are those *natural functions* that can be computed by iterating over a number. For example, we can implement addition and multiplication in System T:

$$\begin{aligned} s &:= \lambda x.\, \mathsf{Succ}(x) &&: \mathsf{Nat} \to \mathsf{Nat} \\ \mathit{add} &:= \lambda x.\, \mathsf{iter}\, s\, x &&: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\ \mathit{mult} &:= \lambda x.\, \mathsf{iter}\, (\mathit{add}\, x)\, \underline{0} &&: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \end{aligned}$$

System T, however, is more expressive since we can also construct (higher-order) functions through iteration. The archetypal example for a higher-order primitive recursive function is the Ackermann function, which is defined by the following equations:

$$\begin{aligned} &\mathit{ack}\, 0 \quad\quad\quad\;\; n &&:= n+1 \\ &\mathit{ack}\, (m+1)\, 0 &&:= \mathit{ack}\, m\, 1 \\ &\mathit{ack}\, (m+1)\, (n+1) &&:= \mathit{ack}\, m\, (\mathit{ack}\, (m+1)\, n) \end{aligned}$$

To implement this function in System T, observe that $\mathit{ack}\, (m+1)\, n$ is called in the third line – with smaller n. In each of the other recursive calls in the second and third line, $\mathit{ack}\, m$ is used – with smaller m. Thus we can refactor the last two lines and store the result of $\mathit{ack}\, m$ in a temporary variable x. The last two lines then amount to an iteration over n, where x is used in each case of the iteration:

$$\begin{aligned} \mathit{ack}\, 0 &:= s \\ \mathit{ack}\, (1+m) &:= \mathit{ack}'\, (\mathit{ack}\, m) \\ \mathit{ack}'\, x &:= \lambda n.\, \begin{cases} x\, \underline{1} & n = 0 \\ x\, (\mathit{ack}'\, x\, (n-1)) & n > 0 \end{cases} \end{aligned}$$

This can now very elegantly be implemented in System T:

$$\begin{aligned} \mathit{ack} &:= \mathsf{iter}\, u\, s \\ u &:= \lambda x.\, \mathsf{iter}\, x\, (x\, \underline{1}) \end{aligned}$$

---

[2] $C[t]$ substitutes t for the $\bullet$ in C.

m\n	0	1	2	3	4
0	1; 2	2; 2	3; 2	4; 2	5; 2
1	2; 5	3; 7	4; 9	5; 11	6; 13
2	3; 10	5; 19	7; 32	9; 49	11; 70
3	5; 22	13; 113	29; 548	61; 2439	125; 10314

Table 2.1: Values and costs (separated by semicolons) for some inputs of the System T program $ack\,m\,n$

Here is an execution protocol for $ack\,\underline{1}\,\underline{1}$, which computes to $\underline{3}$ with a cost of 7:

$$\begin{aligned} ack\,\underline{1}\,\underline{1} &= (\mathsf{iter}\,u\,s\,\underline{1})\,\underline{1} \succ (u\,(\mathsf{iter}\,u\,s\,\underline{0}))\,\underline{1} \succ (u\,s)\,\underline{1} \\ &\succ (\mathsf{iter}\,s\,(s\,\underline{1}))\,\underline{1} \succ s\,(\mathsf{iter}\,s\,(s\,\underline{1})\,\underline{0}) \succ s\,(s\,\underline{1}) \succ s\,\underline{2} \succ \underline{3} \end{aligned}$$

In Table 2.1, we show some values of the Ackermann function. The second entry in each cell is the cost of the execution of *ack*; it has been computed using a simple System T interpreter implemented in Haskell. Values outside the range of the table grow extremely fast.

It is a well-known result that, although System T is quite expressive, it is often not the case that functions can be implemented efficiently. For example, an implementation of a function *min* that computes the minimum of two numbers cannot be implemented in $\mathcal{O}(\min(a, b))$; only $\mathcal{O}(a + b)$ is possible [10]. The (informal) reason for this is that we can only iterate over one number. In other words, it is not possible in the call-by-value System T to break out of a loop.

2.2 The programming language PCF

PCF (*programming computable functions*) [34] is a simple functional Turing-complete language. It has all the features of System T, but, instead of iteration, we have unbounded recursion. Of course, removing iteration is not a restriction, since iteration can be implemented using recursion. Implementing iteration as syntactic sugar also preserves the cost of an execution, which we will exploit in Section 5.6, where we will show that a coeffect-based type system for System T can be embedded inside such a system for PCF.

We consider two semantics of PCF: call-by-value (CBV) and call-by-name (CBN). Both variants have the same syntax (although we will need to introduce a syntactic restrictions on fixpoints in the CBV setting), but they have different *evaluation strategies*, as we will discuss below.

$$\begin{aligned} t ::= x \mid t_1\,t_2 \mid \mathsf{ifz}\,t_1\,\mathsf{then}\,t_2\,\mathsf{else}\,t_3 \mid \mathsf{Succ}(t) \mid \mathsf{Pred}(t) \mid \langle t_1; t_2\rangle \mid \pi_n(t) \\ \mid \mathsf{inl}(t) \mid \mathsf{inr}(t) \mid \mathsf{case}\,t\,[\mathsf{inl}(x) \Rightarrow t_1 \mid \mathsf{inr}(y) \Rightarrow t_2] \mid \underline{n} \mid \lambda x.\,t \mid \mu x.\,t \end{aligned}$$

2.2.1 Call-by-value version (CBV)

In the *call-by-value* version of PCF, abbreviated CBV, the argument t_2 of an application $t_1\, t_2$ first has to compute to a *value*. Values are the following subset of terms:

$$v ::= \underline{n} \mid \lambda x.\, t \mid \mu f.\, \lambda x.\, t \mid \langle v_1; v_2\rangle \mid \mathsf{inl}(v) \mid \mathsf{inr}(v)$$

In CBV, we only allow fixpoints of the shape $\mu f.\, \lambda x.\, t$, which we abbreviate to $\mu f x.\, t$.

In addition to the small-step and big-step operational semantics rules of System T (which also has call-by-value semantics) in Figure 2.1, CBV has the following rules:

$$(\mu f x.\, t)\, v \succ_1 t\{\mu f x.\, t/f, v/x\} \qquad \frac{t_1 \Downarrow_{i_1} \mu f x.\, t \qquad t_2 \Downarrow_{i_2} v_1 \qquad t\{\mu f x.\, t/f, v_1/x\} \Downarrow_{i_3} v_2}{t_1\, t_2 \Downarrow_{1+i_1+i_2+i_3} v_2}$$

2.2.2 Call-by-name version (CBN)

In the call-by-name semantics of PCF, arguments are not evaluated before being substituted for variables. In particular, we have the following head reduction rules:

$$(\lambda x.\, t)\, t' \succ t\{t'/x\} \qquad \pi_k\, \langle t_1; t_2\rangle \succ t_k \qquad \mu x.\, t \succ t\{\mu x.\, t/x\}$$

$$\mathsf{case}\, \mathsf{inl}(t)\, [\mathsf{inl}(x) \Rightarrow t_1 \mid \mathsf{inr}(y) \Rightarrow t_2] \succ t_1\{t/x\}$$

$$\mathsf{case}\, \mathsf{inr}(t)\, [\mathsf{inl}(x) \Rightarrow t_1 \mid \mathsf{inr}(y) \Rightarrow t_2] \succ t_2\{t/y\}$$

Closed terms evaluate to *terminal terms* (T) (we reserve the word *value* for the call-by-value setting):

$$T ::= \underline{n} \mid \lambda x.\, t \mid \langle t_1; t_2\rangle \mid \mathsf{inl}(t) \mid \mathsf{inr}(t)$$

Observe that the components of products are only evaluated after we apply projections. Furthermore, a sum is already terminal after the constructor (inl or inr) is known. In particular, $\mathsf{inl}(\mu x.\, x)$ is a terminal term, but not a value (in CBV).

The cost of a CBN execution is defined by the number of *variable lookups*. We will discuss later why this cost metric is useful. However, variable lookups cannot be counted using the ordinary substitution-based semantics. Therefore, we define big-step semantics using *environments* and *closures*.

Definition 2.6 (Closures and environments)**.** Environments and (terminal) closures are defined by mutual induction:

$$c ::= \langle t; \xi\rangle \qquad tc ::= \langle T; \xi\rangle \qquad \xi ::= \emptyset \mid x \mapsto c, \xi$$

- An *environment* ξ is a partial mapping from variables to closures.
- A *closure* $c = \langle t; \xi\rangle$ is a tuple of a term and an environment.
- A *terminal closure* $\langle T; \xi\rangle$ is a closure of which the term is terminal.

$$tc \Downarrow_0 tc \qquad \frac{\xi(x) \Downarrow_i tc}{\langle x; \xi\rangle \Downarrow_{1+i} tc} \qquad \frac{\langle t;\xi\rangle \Downarrow_i \langle \underline{k}; \xi'\rangle}{\langle \mathsf{Succ}(t);\xi\rangle \Downarrow_i \langle \underline{1+k};\xi'\rangle} \qquad \frac{\langle t;\xi\rangle \Downarrow_i \langle \underline{k}; \xi'\rangle}{\langle \mathsf{Pred}(t);\xi\rangle \Downarrow_i \langle \underline{k \dot{-} 1};\xi'\rangle}$$

$$\frac{\langle t; x \mapsto \langle \mu x.\, t;\xi\rangle, \xi\rangle \Downarrow_i tc}{\langle \mu x.\, t;\xi\rangle \Downarrow_i tc} \qquad \frac{\begin{array}{c}\langle t_1;\xi\rangle \Downarrow_{i_1} \langle \lambda x.\, t';\xi'\rangle \\ \langle t'; x \mapsto \langle t_2;\xi\rangle, \xi'\rangle \Downarrow_{i_2} tc\end{array}}{\langle t_1\, t_2;\xi\rangle \Downarrow_{i_1+i_2} tc} \qquad \frac{\langle t_1;\xi\rangle \Downarrow_{i_1} \langle \underline{0}; \_\rangle \qquad \langle t_2;\xi\rangle \Downarrow_{i_2} tc}{\langle \mathsf{ifz}\, t_1 \,\mathsf{then}\, t_2 \,\mathsf{else}\, t_3;\xi\rangle \Downarrow_{i_1+i_2} tc}$$

$$\frac{\langle t_1;\xi\rangle \Downarrow_{i_1} \langle \underline{1+k}; \_\rangle \qquad \langle t_3;\xi\rangle \Downarrow_{i_3} tc}{\langle \mathsf{ifz}\, t_1 \,\mathsf{then}\, t_2 \,\mathsf{else}\, t_3;\xi\rangle \Downarrow_{i_1+i_3} tc} \qquad \frac{\langle t;\xi\rangle \Downarrow_{i_1} \langle \langle t_1;t_2\rangle;\xi'\rangle \qquad \langle t_k;\xi'\rangle \Downarrow_{i_2} tc}{\langle \pi_k(t);\xi\rangle \Downarrow_{i_1+i_2} tc}$$

$$\frac{\langle t_1;\xi\rangle \Downarrow_{i_1} \langle \mathsf{inl}(t_1');\xi'\rangle \qquad \langle t_2; x \mapsto \langle t_1';\xi'\rangle, \xi'\rangle \Downarrow_{i_2} tc}{\langle \mathsf{case}\, t_1\, [\mathsf{inl}(x) \Rightarrow t_2 \mid \mathsf{inr}(y) \Rightarrow t_3];\xi\rangle \Downarrow_{i_1+i_2} tc}$$

$$\frac{\langle t_1;\xi\rangle \Downarrow_{i_1} \langle \mathsf{inr}(t_1');\xi'\rangle \qquad \langle t_3; y \mapsto \langle t_1';\xi'\rangle, \xi'\rangle \Downarrow_{i_3} tc}{\langle \mathsf{case}\, t_1\, [\mathsf{inl}(x) \Rightarrow t_2 \mid \mathsf{inr}(y) \Rightarrow t_3];\xi\rangle \Downarrow_{i_1+i_3} tc}$$

Figure 2.3: CBN big-step environment semantics

- A closure $\langle t;\xi\rangle$ is *closed*, if all free term variables in t are bound in ξ and if all closures for these free variables are also closed closures.

The call-by-name big-step environment semantics are shown in Figure 2.3. The relation $\cdot \Downarrow_i \cdot$ is a partial and deterministic mapping from closed closures to closed terminal closures, where i is the number of variable lookups. Note that in the variable rule, $\xi(x)$ may be a non-terminal closure, so it needs to be evaluated first, and we increment the counter.

We can unfold closed closures to closed terms:

$$\mathit{unf}(\langle t; x_1 \mapsto c_1, \ldots, x_n \mapsto c_n\rangle) := t\{\mathit{unf}(c_1)/x_1, \ldots, \mathit{unf}(c_n)/x_n\}$$

We can now define executions of closed terms: We write $t \Downarrow_k T$ if there is a terminal closure tc such that $\langle t;\emptyset\rangle \Downarrow_k tc$ such that $\mathit{unf}(tc) = T$. Note that T must be a closed terminal term.

If we are not interested in costs, we can also use similar big-step and small-step semantics as in CBV.

2.2.3 Simple types

Like System T, both variants of PCF are simply typed. In addition to the typing rules of System T (see Figure 2.2, except iteration), we introduce the following rule for fixpoints:

$$\textsc{Fix}\quad \frac{x : B, \Gamma \vdash t : B}{\Gamma \vdash \mu x.\, t : B}$$

Remember that if we are in the CBV setting, t must be a λ-abstraction, and thus B must be an arrow type.

We can also prove subject reduction and progress for both variants of PCF:

Lemma 2.7 (Subject reduction)**.** *If* $\Gamma \vdash t : A$ *and* $t \succ t'$*, then* $\Gamma \vdash t' : A$*.*

Lemma 2.8 (Progress)**.** *If* $\emptyset \vdash t : A$*, then either* t *is a (*CBV *or* CBV*) value/terminal term, or there exists a successor term* $t \succ t'$ *(in the* CBV *or* CBV *semantics, respectively).*

2.3 Call-by-push-value

In this section, we introduce a programming language based on the call-by-push-value (CBPV) paradigm [27]. Call-by-push-value is a "*subsuming paradigm*", which (roughly) means that we can use it for simulating both call-by-value and call-by-name executions and semantics. In this thesis, we will use CBPV to generalise coeffect-based and effect-based type systems. The general idea is that if we have proved a theorem about the system CBPV, we get the same result for the CBV and CBN systems (almost) for free.

2.3.1 Syntax and semantics

CBPV is based on the idea that "a value *is*, a computation *does*". *Values* and *computations* are two syntactic categories. Computations *do*, since we define the operational semantics on computations. A computation (t) evaluates to a *terminal computation* (T), which is either a returned value ($\mathsf{return}\,v$) or a λ-abstraction.

$$
\begin{array}{rrl}
\text{Values:} & u, v ::= & x \mid \underline{n} \mid \mathsf{thunk}\,t \\
\text{Computations:} & t ::= & \mathsf{force}\,v \mid \mathsf{return}\,v \mid t\,v \mid \mathsf{bind}\,x \leftarrow t_1 \,\mathsf{in}\, t_2 \\
 & & \mid \mathsf{ifz}\,v\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2 \mid \lambda x.\,t \mid \mu x.\,t \\
 & & \mid \mathsf{calc}\,x \leftarrow \mathsf{Succ}(v)\,\mathsf{in}\,t \mid \mathsf{calc}\,x \leftarrow \mathsf{Pred}(v)\,\mathsf{in}\,t \\
\text{Terminal computations:} & T ::= & \mathsf{return}\,v \mid \lambda x.\,t
\end{array}
$$

- Since variables are placeholders for values, they belong to the syntactic category of values. We can thus only substitute values for variables.
- return and bind are well-known operators in monadic programming. The terminal computation $\mathsf{return}\,v$ denotes that the computation is finished and the result of the computation is v. The computation $\mathsf{bind}\,x \leftarrow t_1 \,\mathsf{in}\, t_2$ first executes t_1. After t_1 returns a value v, v is substituted for x in t_2, which is then executed. We use $\mathsf{bind}\,x \leftarrow t_1,\, y \leftarrow t_2 \,\mathsf{in}\, t_3$ as syntactic sugar for $\mathsf{bind}\,x \leftarrow t_1 \,\mathsf{in}\, \mathsf{bind}\,y \leftarrow t_2 \,\mathsf{in}\, t_3$.
- Computations can be *thunked* (or suspended). For a computation t, $\mathsf{thunk}\,t$ is a value that can be *forced*, which means that the computation t is executed.
- Arithmetic operations like case distinction and successor require that the argument is a value v. In a well-typed program, v can either be a constant or a variable. In

$$\mathsf{force}\,\mathsf{thunk}\,t \succ_1 t \qquad \mathsf{bind}\,x \leftarrow \mathsf{return}\,v\,\mathsf{in}\,t_2 \succ_0 t_2\{v/x\}$$

$$(\lambda x.\,t)\,v \succ_0 t\{v/x\} \qquad \mu x.\,t \succ_0 t\{\mathsf{thunk}\,\mu x.\,t/x\}$$

$$\mathsf{calc}\,x \leftarrow \mathsf{Succ}(\underline{n})\,\mathsf{in}\,t \succ_0 t\{\underline{1+n}/x\} \qquad \mathsf{calc}\,x \leftarrow \mathsf{Pred}(\underline{n})\,\mathsf{in}\,t \succ_0 t\{\underline{n \dot{-} 1}/x\}$$

$$\mathsf{ifz}\,\underline{0}\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2 \succ_0 t_1 \qquad \mathsf{ifz}\,\underline{1+n}\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2 \succ_0 t_2$$

$$\frac{t \Downarrow_i T}{\mathsf{force}\,\mathsf{thunk}\,t \Downarrow_{1+i} T} \qquad \frac{t_1 \Downarrow_{i_1} \mathsf{return}\,v \qquad t_2\{v/x\} \Downarrow_{i_2} T}{\mathsf{bind}\,x \leftarrow t_1\,\mathsf{in}\,t_2 \Downarrow_{i_1+i_2} T} \qquad \frac{t \Downarrow_{i_1} \lambda x.\,t' \qquad t'\{v/x\} \Downarrow_{i_2} T}{t\,v \Downarrow_{i_1+i_2} T}$$

$$\frac{t\{\mathsf{thunk}\,\mu x.\,t/x\} \Downarrow_i T}{\mu x.\,t \Downarrow_i T} \qquad \frac{t_2\{\underline{1+n}/x\} \Downarrow_i T}{\mathsf{calc}\,x \leftarrow \mathsf{Succ}(\underline{n})\,\mathsf{in}\,t \Downarrow_i T} \qquad \frac{t_2\{\underline{n \dot{-} 1}/x\} \Downarrow_i T}{\mathsf{calc}\,x \leftarrow \mathsf{Pred}(\underline{n})\,\mathsf{in}\,t \Downarrow_i T}$$

$$\frac{t_1 \Downarrow_i T}{\mathsf{ifz}\,\underline{0}\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2 \Downarrow_i T} \qquad \frac{t_2 \Downarrow_i T}{\mathsf{ifz}\,\underline{1+n}\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2 \Downarrow_i T}$$

Figure 2.4: Head-reduction rules and big-step operational semantics of CBPV

the latter case, if the computation is closed, v will eventually be substituted with a constant. The computation $\mathsf{calc}\,x \leftarrow \mathsf{Succ}(\underline{n})\,\mathsf{in}\,t$ reduces to $t\{\underline{1+n}/x\}$.

- Note that in an application $t\,v$, the argument has to be a value. If we do not want that the argument is evaluated (e.g. as in call-by-name semantics), we can thunk it.

- Fixpoints ($\mu x.\,t$) are not terminal computations. They are unfolded, which means that they reduce to $t\{\mathsf{thunk}\,\mu x.\,t/x\}$.

The syntax for CBPV that we use in this thesis is similar to [24]. CBPV also supports product and sum types, which we will discuss later.

Operational semantics of CBPV Closed computations may diverge or evaluate to a closed terminal closure. In the small-step and big-step semantics depicted in Figure 2.4, we count the number of forcing steps, i.e. $\mathsf{force}\,\mathsf{thunk}\,t \succ_1 t$. From the syntax it is already clear that reductions can only happen below the left side of applications and below the bind operation. This suggests the following definition of evaluation contexts:

$$C ::= \bullet \;\big|\; C\,v \;\big|\; \mathsf{bind}\,x \leftarrow C\,\mathsf{in}\,t$$

$$\textsc{Const}\quad \Gamma \vdash^{\mathsf{v}} \underline{n} : \mathsf{Nat} \qquad\qquad \textsc{Var}\quad x : A, \Gamma \vdash^{\mathsf{v}} x : A$$

$$\textsc{Lam}\quad \frac{x : A, \Gamma \vdash^{\mathsf{c}} t : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \lambda x.\, t : A \to \underline{B}} \qquad\qquad \textsc{Fix}\quad \frac{x : \mathsf{U}\,\underline{B}, \Gamma \vdash^{\mathsf{c}} t : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mu x.\, t : \underline{B}}$$

$$\textsc{App}\quad \frac{\Gamma \vdash^{\mathsf{c}} t : A \to \underline{B} \qquad \Gamma \vdash^{\mathsf{v}} v : A}{\Gamma \vdash^{\mathsf{c}} t\, v : \underline{B}} \qquad\qquad \textsc{Ifz}\quad \frac{\Gamma \vdash^{\mathsf{v}} v : \mathsf{Nat} \qquad \Gamma \vdash^{\mathsf{c}} t_2 : \underline{B} \qquad \Gamma \vdash^{\mathsf{c}} t_3 : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathsf{ifz}\, v \,\mathsf{then}\, t_2 \,\mathsf{else}\, t_3 : \underline{B}}$$

$$\textsc{Succ}\quad \frac{\Gamma \vdash^{\mathsf{v}} v : \mathsf{Nat} \qquad x : \mathsf{Nat}, \Gamma \vdash^{\mathsf{c}} t : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathsf{calc}\, x \leftarrow \mathsf{Succ}(v) \,\mathsf{in}\, t : \underline{B}} \qquad \textsc{Pred}\quad \frac{\Gamma \vdash^{\mathsf{c}} t_1 : \mathsf{F}\,\mathsf{Nat} \qquad x : \mathsf{Nat}, \Gamma \vdash^{\mathsf{c}} t : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathsf{calc}\, x \leftarrow \mathsf{Pred}(t_1) \,\mathsf{in}\, t_2 : \underline{B}} \qquad \textsc{Return}\quad \frac{\Gamma \vdash^{\mathsf{v}} v : A}{\Gamma \vdash^{\mathsf{c}} \mathsf{return}\, v : \mathsf{F}\, A}$$

$$\textsc{Bind}\quad \frac{\Gamma \vdash^{\mathsf{c}} t_1 : \mathsf{F}\, A \qquad x : A, \Gamma \vdash^{\mathsf{c}} t_2 : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathsf{bind}\, x \leftarrow t_1 \,\mathsf{in}\, t_2 : \underline{B}} \qquad \textsc{Thunk}\quad \frac{\Gamma \vdash^{\mathsf{c}} t : \underline{B}}{\Gamma \vdash^{\mathsf{v}} \mathsf{thunk}\, t : \mathsf{U}\,\underline{B}} \qquad \textsc{Force}\quad \frac{\Gamma \vdash^{\mathsf{v}} v : \mathsf{U}\,\underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathsf{force}\, v : \underline{B}}$$

Figure 2.5: Simple typing rules of CBPV

2.3.2 Simple typings

There are two categories of types: value types and computation types.

Definition 2.9 (Simple CBPV types)**.**

$$\begin{aligned}
\text{Value types:} \quad & A ::= \mathsf{U}\,\underline{B} \mid \mathsf{Nat} \\
\text{Computation types:} \quad & \underline{B} ::= \mathsf{F}\, A \mid A \to \underline{B} \\
\text{Contexts:} \quad & \Gamma, \Delta ::= \emptyset \mid x : A, \Gamma
\end{aligned}$$

The simple typing rules are depicted in Figure 2.5. We can also add sum and product types to CBPV, but we will consider these types later.

2.3.3 Call-by-name translation

In the following, we show how to translate a PCF term t to a CBPV computation t^{n} that has the same behaviour as the call-by-name semantics of t. We can also translate simple typings. The general idea of the translation is that we introduce thunk at all variable bindings and force at every variable lookup. In an application $t_1\, t_2$, the argument (t_2) is thunked, because it only evaluated if it is needed by the function. This idea also suggest the following translation of PCF types to CBPV computation types:

Definition 2.10 (Translation of CBN types and contexts)**.**

$$\begin{aligned}
\mathsf{Nat}^{\mathsf{n}} &:= \mathsf{F}\,\mathsf{Nat} \\
(A \to B)^{\mathsf{n}} &:= \mathsf{U}\, A^{\mathsf{n}} \to B^{\mathsf{n}}
\end{aligned}$$

Contexts are translated pointwisely, i.e. $\emptyset^{\mathsf{n}} = \emptyset$ and $(x : A, \Gamma)^{\mathsf{n}} := x : A^{\mathsf{n}}, \Gamma^{\mathsf{n}}$.

For example, the type $\mathsf{Nat} \to \mathsf{Nat}$ is translated to $(\mathsf{U}\,\mathsf{F}\,\mathsf{Nat}) \to \mathsf{F}\,\mathsf{Nat}$. This means, the CBPV function expects as argument a thunked computation that, when forced, will eventually return a constant (or diverge).

Definition 2.11 (Translation of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ terms)**.**

$$\begin{aligned}
x^{\mathsf{n}} &:= \mathsf{force}\, x \\
\underline{k}^{\mathsf{n}} &:= \mathsf{return}\, \underline{k} \\
(\lambda x.\, t)^{\mathsf{n}} &:= \lambda x.\, t^{\mathsf{n}} \\
(\mu x.\, t)^{\mathsf{n}} &:= \mu x.\, t^{\mathsf{n}} \\
(\mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3)^{\mathsf{n}} &:= \mathsf{bind}\, x \leftarrow t_1^{\mathsf{n}}\, \mathsf{in}\, \mathsf{ifz}\, x\, \mathsf{then}\, t_2^{\mathsf{n}}\, \mathsf{else}\, t_3^{\mathsf{n}} \\
(t_1\, t_2)^{\mathsf{n}} &:= t_1^{\mathsf{n}}(\mathsf{thunk}\, t_2^{\mathsf{n}}) \\
(\mathsf{Succ}(t))^{\mathsf{n}} &:= \mathsf{bind}\, x \leftarrow t^{\mathsf{n}}\, \mathsf{in}\, \mathsf{calc}\, y \leftarrow \mathsf{Succ}(x)\, \mathsf{in}\, \mathsf{return}\, y \\
(\mathsf{Pred}(t))^{\mathsf{n}} &:= \mathsf{bind}\, x \leftarrow t^{\mathsf{n}}\, \mathsf{in}\, \mathsf{calc}\, y \leftarrow \mathsf{Pred}(x)\, \mathsf{in}\, \mathsf{return}\, y
\end{aligned}$$

Lemma 2.12 (Call-by-name typing translation)**.** *Every* PCF *typing* $\Gamma \vdash t : A$ *can be translated to a* CBPV *typing* $\Gamma^{\mathsf{n}} \vdash t^{\mathsf{n}} : A^{\mathsf{n}}$.

Proof. By induction on the PCF typing. We will see a more detailed proof in Section 7.3. □

We will later need to convert CBN (closure) executions to CBPV executions, and vice versa.[3] Therefore, we also define closure semantics for CBPV and show that the number of variable lookups in a CBN execution corresponds to the number of forcing steps in the corresponding CBPV closure execution. We omit the translation from ordinary CBPV big-step executions to CBPV closure executions.

Closures and environments are defined similarly as in Definition 2.6. As there are two syntactic categories of CBPV terms (values and computations), there are also two categories of closures.

Definition 2.13 (CBPV closures and environments)**.** An *environment* ξ is a partial mapping from term variables to value closures. A *computation closure* $c = \langle t; \xi \rangle$ is a tuple of a CBPV computation terms and an environment. A *value closure* $vc = \langle v; \xi \rangle$ is a tuple of which the term is a CBPV value term. A (computation or value) closure is closed, if all free variables occurring in the (value or computation) term are bound in ξ, and all the respective value closures in ξ are also closed. A *terminal closure* is a computation closure where the computation term is terminal (i.e. either $\mathsf{return}\, v$ or $\lambda x.\, t$).

$$c ::= \langle t; \xi \rangle \qquad tc ::= \langle \mathsf{return}\, v; \xi \rangle \;\big|\; \langle \lambda x.\, t; \xi \rangle \qquad vc ::= \langle v; \xi \rangle \qquad \xi ::= \emptyset \;\big|\; x \mapsto vc, \xi$$

---

[3] It was first shown in [27] that the translation function $\cdot^{\mathsf{n}}$ preserves operational semantics. However, it is not shown there that the number of *variable lookups* (i.e. in the CBN environment semantics) corresponds to the number of forces.

$$\frac{}{tc \Downarrow_0 tc} \qquad \frac{\langle t_1; \xi\rangle \Downarrow_{i_1} \langle \mathsf{return}\, v; \xi'\rangle \qquad \langle t_2\{v/x\}; \xi'\rangle \Downarrow_{i_2} tc}{\langle \mathsf{bind}\, x \leftarrow t_1 \,\mathsf{in}\, t_2; \xi\rangle \Downarrow_{i_1+i_2} tc}$$

$$\frac{\begin{array}{c} \mathit{unroll}\,\langle v; \xi\rangle = \langle \underline{n}; \_\rangle \\ \langle t\{\underline{1+n}/x\}; \xi\rangle \Downarrow_i tc \end{array}}{\langle \mathsf{calc}\, x \leftarrow \mathsf{Succ}(v) \,\mathsf{in}\, t; \xi\rangle \Downarrow_i tc} \qquad \frac{\begin{array}{c} \mathit{unroll}\,\langle v; \xi\rangle = \langle \underline{n}; \_\rangle \\ \langle t\{\underline{n \dot{-} 1}/x\}; \xi\rangle \Downarrow_i tc \end{array}}{\langle \mathsf{calc}\, x \leftarrow \mathsf{Pred}(v) \,\mathsf{in}\, t; \xi\rangle \Downarrow_i tc}$$

$$\frac{\langle t; x \mapsto \langle \mathsf{thunk}\, \mu x.\, t; \xi\rangle, \xi\rangle \Downarrow_i tc}{\langle \mu x.\, t; \xi\rangle \Downarrow_i tc} \qquad \frac{\begin{array}{c} \langle t; \xi\rangle \Downarrow_{i_1} \langle \lambda x.\, t'; \xi'\rangle \\ \langle t'; x \mapsto \langle v; \xi\rangle, \xi'\rangle \Downarrow_{i_2} tc \end{array}}{\langle t\, v; \xi\rangle \Downarrow_{i_1+i_2} tc}$$

$$\frac{\mathit{unroll}\,\langle v; \xi\rangle = \langle \underline{0}; \_\rangle \qquad \langle t_2; \xi\rangle \Downarrow_i tc}{\langle \mathsf{ifz}\, v \,\mathsf{then}\, t_2 \,\mathsf{else}\, t_3; \xi\rangle \Downarrow_i tc} \qquad \frac{\mathit{unroll}\,\langle v; \xi\rangle = \langle \underline{1+n}; \_\rangle \qquad \langle t_3; \xi\rangle \Downarrow_i tc}{\langle \mathsf{ifz}\, v \,\mathsf{then}\, t_2 \,\mathsf{else}\, t_3; \xi\rangle \Downarrow_i tc}$$

$$\frac{\mathit{unroll}\,\langle v; \xi\rangle = \langle \mathsf{thunk}\, t; \xi'\rangle \qquad \langle t; \xi'\rangle \Downarrow_i tc}{\langle \mathsf{force}\, v; \xi\rangle \Downarrow_{1+i} tc}$$

Figure 2.6: Environment semantics of CBPV

We define the big-step environment semantics $\cdot \Downarrow_i \cdot$ in Figure 2.6. Closed computation closures are partially mapped to a terminal closure; i is the number of forces during the execution.

Note that there is no variable lookup rule as in call-by-value PCF since variables are already considered values in CBPV. This is actually a complication for the environment semantics, since operators like Succ require the value v to be in a certain shape, (i.e. $\underline{k}$ for Succ and $\mathsf{thunk}\, t$ for force), but v could be a variable that is bound in the environment. Because of this, we need to unfold value closures until this head symbol is known:

Definition 2.14 ($\mathit{unroll}(vc)$)**.** We define the function $\mathit{unroll}(vc)$ on value closures by structural induction:

$$\begin{aligned} \mathit{unroll}\,\langle \underline{n}; \xi\rangle &:= \langle \underline{n}; \xi\rangle \\ \mathit{unroll}\,\langle \mathsf{thunk}\, t; \xi\rangle &:= \langle \mathsf{thunk}\, t; \xi\rangle \\ \mathit{unroll}\,\langle x; \xi\rangle &:= \mathit{unroll}(\xi(x)) \end{aligned}$$

Another complication of the environment semantics are the binders introduced by bind and calc. Because there are no corresponding binders in the CBN closure semantics, we choose to substitute the binders instead of adding the result of t_1 to the environment. This is very important in the proof of bisimulation of CBN closures and their CBPV translations.

We can show that there is a bisimulation between executions of closures c and their CBN translations $\cdot^{\mathsf{n}}$. First, we define how to translate CBN closures to CBPV closures.

Definition 2.15 (Translation of CBN closures)**.**

$$\begin{aligned} \langle t;\xi\rangle^{\mathsf{n}} &:= \langle t^{\mathsf{n}};\xi^{\mathsf{n}}\rangle \\ \emptyset^{\mathsf{n}} &:= \emptyset \\ (x \mapsto \langle t_x;\xi_x\rangle, \xi)^{\mathsf{n}} &:= x \mapsto \langle \mathsf{thunk}\, t_x^{\mathsf{n}};\xi_x^{\mathsf{n}}\rangle, \xi^{\mathsf{n}} \end{aligned}$$

The definition of ξ^{n} can also be stated as: $\xi^{\mathsf{n}}(x) := \langle \mathsf{thunk}\, t_x^{\mathsf{n}};\xi_x^{\mathsf{n}}\rangle$ whenever $\xi(x) = \langle t_x;\xi_x\rangle$.

The following lemma is one part of the bisimulation (the more complicated part).

Lemma 2.16 (Simulation of c^{n} by c)**.** *Let $c^{\mathsf{n}} = \langle t^{\mathsf{n}};\xi^{\mathsf{n}}\rangle \Downarrow_i tc$ be a CBPV execution. Then there exists a CBN terminal closure tc_{CBN} such that $tc = tc_{\mathsf{CBN}}^{\mathsf{n}}$ and $c \Downarrow_i tc_{\mathsf{CBN}}$.*

Proof. By induction on the lexicographic order over i and the size of t. We make a case analysis over t and inspect the executions of c^{n}.

- Cases $t = \underline{n}$ or $t = \lambda x.\, t'$ (i.e. the value cases). Then c^{n} is already a terminal closure (because the term of c^{n} is either $\mathsf{return}\,\underline{n}$ or $\lambda x.\, t'^{\mathsf{n}}$); thus $c^{\mathsf{n}} = T$ and $i = 0$. This is simulated by the empty computation $\langle t;\xi^{\mathsf{n}}\rangle \Downarrow_0 \langle t;\xi^{\mathsf{n}}\rangle$ in CBN.

- Case $t = \mu x.\, t'$. Then $t^{\mathsf{n}} = \mu x.\, t'^{\mathsf{n}}$, and the CBPV execution must be:

  $$\frac{\left\langle t'^{\mathsf{n}}; x \mapsto \left\langle \mathsf{thunk}\, \mu x.\, t'^{\mathsf{n}};\xi^{\mathsf{n}}\right\rangle, \xi^{\mathsf{n}}\right\rangle \Downarrow_i tc}{\left\langle \mu x.\, t'^{\mathsf{n}};\xi^{\mathsf{n}}\right\rangle \Downarrow_i tc}$$

  The inductive hypothesis yields $\langle t'; x \mapsto \langle \mu x.\, t';\xi\rangle, \xi\rangle \Downarrow_i tc_{\mathsf{CBN}}$, and thus $\langle \mu x.\, t';\xi\rangle \Downarrow_i tc_{\mathsf{CBN}}$.

- Case $t = x$; $t^{\mathsf{n}} = \mathsf{force}\, x$. Define $\langle c_x;\xi_x\rangle := \xi(x)$. Then the CBPV execution has the following shape:

  $$\frac{\begin{array}{c} \mathit{unroll}\,\langle x;\xi^{\mathsf{n}}\rangle = \mathit{unroll}(\xi^{\mathsf{n}}(x)) = \mathit{unroll}\,\langle \mathsf{thunk}\, c_x^{\mathsf{n}};\xi_x^{\mathsf{n}}\rangle = \langle \mathsf{thunk}\, c_x^{\mathsf{n}};\xi_x^{\mathsf{n}}\rangle \\ \langle c_x^{\mathsf{n}};\xi_x^{\mathsf{n}}\rangle \Downarrow_i tc \end{array}}{\langle \mathsf{thunk}\, x;\xi^{\mathsf{n}}\rangle \Downarrow_{1+i} tc}$$

  Now, the inductive hypothesis for $\Downarrow_i$ yields a tc_{CBN} such that $tc = tc_{\mathsf{CBN}}^{\mathsf{n}}$ and $\langle t_x;\xi_x\rangle = \xi(x) \Downarrow_i tc_{\mathsf{CBN}}$. Thus, $\langle x;\xi\rangle \Downarrow_{1+i} tc_{\mathsf{CBN}}$.

- Case $t = \mathsf{Succ}(t')$; $t^{\mathsf{n}} = \mathsf{bind}\, x \leftarrow t'^{\mathsf{n}}\, \mathsf{in}\, \mathsf{calc}\, y \leftarrow \mathsf{Succ}(x)\, \mathsf{in}\, \mathsf{return}\, y$. From inverting the CBPV execution, we know:

  $$\left\langle t'^{\mathsf{n}};\xi^{\mathsf{n}}\right\rangle \Downarrow_{i_1} \left\langle \mathsf{return}\, v;\xi'\right\rangle \qquad\qquad \mathit{unroll}\,\left\langle v;\xi'\right\rangle = \langle \underline{n};\_\rangle$$

  $$\left\langle \mathsf{calc}\, y \leftarrow \mathsf{Succ}(\underline{n})\, \mathsf{in}\, \mathsf{return}\, y;\xi'\right\rangle \Downarrow_{i_2} tc$$

  By inverting the last execution, we get $i_2 = 0$ and $tc = \langle \underline{1+n};\xi'\rangle$.

  The inductive hypothesis yields a tc_{CBN} with $tc_{\mathsf{CBN}}^{\mathsf{n}} = \langle \mathsf{return}\, v;\xi'\rangle$ and $\langle t';\xi\rangle \Downarrow_{i_1} tc_{\mathsf{CBN}}$. Because tc_{CBN} is a terminal closure (which implies that the term of tc_{CBN} is a terminal (CBN) term), we have $tc_{\mathsf{CBN}} = \langle \underline{n};\xi''\rangle$ with $\xi''^{\mathsf{n}} = \xi'$. Thus, we have $\langle \mathsf{Succ}(t);\xi\rangle \Downarrow_{i_1+i_2} \langle \underline{1+n};\xi'\rangle$.

- Case $t = \mathsf{Pred}(t')$. As above.
- Case $t = \mathsf{ifz}\, t_1 \,\mathsf{then}\, t_2 \,\mathsf{else}\, t_3$; $t^{\mathsf{n}} = \mathsf{bind}\, x \leftarrow t_1^{\mathsf{n}} \,\mathsf{in}\, \mathsf{ifz}\, x \,\mathsf{then}\, t_2^{\mathsf{n}} \,\mathsf{else}\, t_3^{\mathsf{n}}$. Like above, we invert the CBPV execution:

  $$\langle t_1^{\mathsf{n}}; \xi^{\mathsf{n}} \rangle \Downarrow_{i_1} \left\langle \mathsf{return}\, v; \xi' \right\rangle \qquad \mathit{unroll} \left\langle v; \xi' \right\rangle = \langle \underline{n}; \_ \rangle \qquad \left\langle \mathsf{ifz}\, \underline{n} \,\mathsf{then}\, t_2^{\mathsf{n}} \,\mathsf{else}\, t_3^{\mathsf{n}}; \xi' \right\rangle \Downarrow_{i_2} tc$$

  The inductive hypothesis on $\Downarrow_{i_1}$ (like above) yields a ξ'' with $\xi''^{\mathsf{n}} = \xi'$ and $\langle t_1; \xi \rangle \Downarrow_{i_1} \langle \underline{n}; \xi'' \rangle$.

  We make a case distinction over n.

  - Case $n = 0$. Then (since $\mathit{unroll}\, \langle \underline{n}; \xi' \rangle = \langle \underline{n}; \_ \rangle$), $\langle t_2^{\mathsf{n}}; \xi' = \xi''^{\mathsf{n}} \rangle \Downarrow_{i_2} tc$. The inductive hypothesis on this execution yields a tc_{CBN} such that $\langle t_2; \xi'' \rangle \Downarrow_{i_2} tc_{\mathsf{CBN}}$. This means that:

    $$\langle \mathsf{ifz}\, t_1 \,\mathsf{then}\, t_2 \,\mathsf{else}\, t_3; \xi \rangle \Downarrow_{i_1+i_2} tc_{\mathsf{CBN}}$$

  - Case $n > 0$: analogously. □

The other part of the bisimulation is the following lemma:

Lemma 2.17 (Simulation of c by c^{n})**.** *Let $c \Downarrow_i tc_{\mathsf{CBN}}$. Then $c^{\mathsf{n}} \Downarrow_i tc_{\mathsf{CBN}}^{\mathsf{n}}$.*

Proof. By induction on $c \Downarrow_i tc_{\mathsf{CBN}}$. □

We can also show that the CBPV closure semantics is a refinement of the operational semantics. For this, we define the closure unrolling functions:

Definition 2.18 ($\mathit{unf}(vc)$ and $\mathit{unf}(c)$)**.** We define the functions $\mathit{unf}(vc)$ and $\mathit{unf}(c)$ on closed value/computation closures, that return values or computations, respectively, by structural induction:

$$\begin{aligned}
\mathit{unf}\, \langle \underline{n}; \xi \rangle &:= \underline{n} \\
\mathit{unf}\, \langle \mathsf{thunk}\, t; \xi \rangle &:= \mathsf{thunk}\, \mathit{unf}\, \langle t; \xi \rangle \\
\mathit{unf}\, \langle x; \xi \rangle &:= \mathit{unf}(\xi(x)) \\
\mathit{unf}\, \langle \mathsf{force}\, v; \xi \rangle &:= \mathsf{force}\, \mathit{unf}\, \langle v; \xi \rangle \\
\mathit{unf}\, \langle \mathsf{return}\, v; \xi \rangle &:= \mathsf{return}\, \mathit{unf}\, \langle v; \xi \rangle \\
\mathit{unf}\, \langle \lambda x.\, t; \xi \rangle &:= \lambda x.\, \mathit{unf}\, \langle t; \xi \rangle
\end{aligned}$$

The remaining defining equations for $\mathit{unf}(c)$ are similar. Note that $\mathit{unf}(tc)$ returns a closed terminal computation for closed terminal closures.

Lemma 2.19 (CBPV closure and normal semantics)**.** *Let t be a closed* CBPV *computation. Then the following propositions are equivalent:*

- $t \Downarrow_i T$
- $\langle t; \emptyset \rangle \Downarrow_i tc$ *with* $T = \mathit{unf}(vc)$.

2.3.4 Call-by-value translation

We now show how to translate CBV terms and typings to CBPV terms and typings.

First, we translate CBV types to CBPV value types:

Definition 2.20 (Call-by-value type translation)**.**

$$\begin{aligned} \mathsf{Nat}^{\mathsf{v}} &:= \mathsf{Nat} \\ (A \to B)^{\mathsf{v}} &:= \mathsf{U}\,(A^{\mathsf{v}} \to \mathsf{F}\,B^{\mathsf{v}}) \\ (x : A, \Gamma)^{\mathsf{v}} &:= x : A^{\mathsf{v}}, \Gamma^{\mathsf{v}} \end{aligned}$$

To translate terms t, we have to do a case distinction whether t is a value. (Remember that values are a subcategory of PCF terms.) Values v are translated to CBPV values v^{val}, and all terms t can be translated to CBPV computation terms t^{v}. In particular, a value v will be translated to a returner – $v^{\mathsf{v}} = \mathsf{return}\, v^{\mathsf{val}}$.

Definition 2.21 (Translation of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ terms)**.**

$$\begin{aligned} \underline{k}^{\mathsf{val}} &:= \underline{k} \\ (\lambda x.\, t)^{\mathsf{val}} &:= \mathsf{thunk}\, \lambda x.\, t^{\mathsf{v}} \\ (\mu f x.\, t)^{\mathsf{val}} &:= \mathsf{thunk}\, \mu f.\, \lambda x.\, t^{\mathsf{v}} \\ v^{\mathsf{v}} &:= \mathsf{return}\, v^{\mathsf{val}} \\ x^{\mathsf{v}} &:= \mathsf{return}\, x \\ (\mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3)^{\mathsf{v}} &:= \mathsf{bind}\, x \leftarrow t_1^{\mathsf{v}}\, \mathsf{in}\, \mathsf{ifz}\, x\, \mathsf{then}\, t_2^{\mathsf{v}}\, \mathsf{else}\, t_3^{\mathsf{v}} \\ (t_1\, t_2)^{\mathsf{v}} &:= \mathsf{bind}\, x \leftarrow t_1^{\mathsf{v}},\, y \leftarrow t_2^{\mathsf{v}}\, \mathsf{in}\, (\mathsf{force}\, x)\, y \\ (\mathsf{Succ}(t))^{\mathsf{v}} &:= \mathsf{bind}\, x \leftarrow t^{\mathsf{v}}\, \mathsf{in}\, \mathsf{calc}\, y \leftarrow \mathsf{Succ}(x)\, \mathsf{in}\, \mathsf{return}\, y \\ (\mathsf{Pred}(t))^{\mathsf{v}} &:= \mathsf{bind}\, x \leftarrow t^{\mathsf{v}}\, \mathsf{in}\, \mathsf{calc}\, y \leftarrow \mathsf{Pred}(x)\, \mathsf{in}\, \mathsf{return}\, y \end{aligned}$$

To translate CBV typings, we again have two cases:

Lemma 2.22 (Call-by-value typing translation)**.**
- *Every* PCF *typing* $\Gamma \vdash v : A$, *where* v *is a value, can be translated into a* CBPV *value typing* $\Gamma^{\mathsf{v}} \vdash^{\mathsf{v}} v^{\mathsf{val}} : A^{\mathsf{v}}$.
- *Every* PCF *typing* $\Gamma \vdash t : A$ *can be translated into a* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ *computation typing* $\Gamma^{\mathsf{v}} \vdash^{\mathsf{c}} t^{\mathsf{v}} : \mathsf{F}\, A^{\mathsf{v}}$.

Proof. By mutual induction on the PCF typings. We will see a more detailed proof in Section 7.4. ☐

The bisimulation between t and t^{v} is easier than in the call-by-name case, because we can use the normal big-step semantics (using small-step semantics is also possible). We first show that t^{v} is simulated by t.

Lemma 2.23 (Call-by-value simulation (big step))**.** *Let* $t^{\mathsf{v}} \Downarrow_i T$ *(where* t *is a closed* CBV *term and* T *denotes a terminal* CBPV *computation). Then there exists a* CBV *value* v *with* $v^{\mathsf{v}} = T$ *and* $t \Downarrow_i v$. *(Note that* $v^{\mathsf{v}} = \mathsf{return}\, v^{\mathsf{val}}$*.)*

Proof. By induction on i and t, as in the proof of Lemma 2.16.

- Case $t = v$; $t^{\mathsf{v}} = \mathsf{return}\, v^{\mathsf{val}}$. Then $T = t^{\mathsf{v}}$ and $i = 0$; also $v \Downarrow_0 v$.
- Case $t = t_1\, t_2$; $t^{\mathsf{v}} = \mathsf{bind}\, x \leftarrow t_1^{\mathsf{v}},\, y \leftarrow t_2^{\mathsf{v}} \,\mathsf{in}\, (\mathsf{force}\, x)\, y$. We partially invert the CBPV execution:

  $$\dfrac{t_1^{\mathsf{v}} \Downarrow_{i_1} \mathsf{return}\, u_1 \qquad \dfrac{t_2^{\mathsf{v}} \Downarrow_{i_2} \mathsf{return}\, u_2 \qquad \dfrac{\mathsf{force}\, u_1 \Downarrow_{i_3} \lambda z.\, t' \qquad t'\{u_2/z\} \Downarrow_{i_4} T}{(\mathsf{force}\, u_1)\, u_2 \Downarrow_{i_3+i_4} T}}{\mathsf{bind}\, y \leftarrow t_2 \,\mathsf{in}\, (\mathsf{force}\, u_1)\, y \Downarrow_{i_2+i_3+i_4} T}}{\mathsf{bind}\, x \leftarrow t_1^{\mathsf{v}},\, y \leftarrow t_2^{\mathsf{v}} \,\mathsf{in}\, (\mathsf{force}\, x)\, y \Downarrow_{i_1+i_2+i_3+i_4} T}$$

  The first inductive hypothesis yields a v_1 such that $v_1^{\mathsf{v}} = \mathsf{return}\, v^{\mathsf{val}} = \mathsf{return}\, u_1$ (and thus $v_1^{\mathsf{val}} = u_1$) and $t_1 \Downarrow_{i_1} v_1$. The second inductive hypotheses yields a v_2 such that $v_2^{\mathsf{val}} = u_2$ and $t_2 \Downarrow_{i_2} v_2$.

  Now we make a case distinction over v_1.

  - Case $v_1 = \underline{n}$. This case is not possible, since $\mathsf{force}\, u_1 = \mathsf{force}\, \underline{n}$ is stuck.
  - Case $v_1 = \lambda z.\, t''$; $u_1 = v_1^{\mathsf{val}} = \mathsf{thunk}\, \lambda z.\, t''^{\mathsf{v}}$. Hence, $t' = t''^{\mathsf{v}}$ and $i_3 = 1$. Thus, the last part of the CBPV execution is:

    $$t'\{u_2/z\} = t''^{\mathsf{v}}\{v_2^{\mathsf{val}}/z\} = (t''\{v_2/z\})^{\mathsf{v}} \Downarrow_{i_4} T$$

    On this we can apply the inductive hypothesis once more and get a v_3 such that $T = v_3^{\mathsf{v}} = \mathsf{return}\, v_3^{\mathsf{val}}$ and $t''\{v_2/z\} \Downarrow_{i_4} v_3$. Thus, we have $t_1\, t_2 \Downarrow_{1+i_1+i_2+i_4} v_3$.
  - Case $v_1 = \mu f z.\, t''$; $u_1 = v_1^{\mathsf{val}} = \mathsf{thunk}\, \mu f.\, \lambda z.\, t''^{\mathsf{val}}$. The computation $\mathsf{force}\, u_1$ terminates after one ($i_3 = 1$) step into:

    $$\lambda z.\, t' = \lambda z.\, t''^{\mathsf{val}}\{\mathsf{thunk}\, \mu f.\, \lambda z.\, t''/f\}$$

    The last part of the computation has the shape:

    $$t'\{u_2/z\} = t''^{\mathsf{v}}\{v_2^{\mathsf{val}}/z, \mathsf{thunk}\, \mu f.\, \lambda z.\, t''/f\} = (t''\{v_2/z, \mu f x.\, t''\})^{\mathsf{v}} \Downarrow_{i_4} T$$

    On this we can apply the inductive hypothesis once more and get a v_3 such that $T = v_3^{\mathsf{v}} = \mathsf{return}\, v_3^{\mathsf{val}}$ and $t''\{v_2/z, \mu f x.\, t''\} \Downarrow_{i_4} v_3$. Thus, we have $t_1\, t_2 \Downarrow_{1+i_1+i_2+i_4} v_3$.
- Case $t = \mathsf{Succ}(t')$; $t^{\mathsf{v}} = \mathsf{bind}\, x \leftarrow t'^{\mathsf{v}} \,\mathsf{in}\, \mathsf{calc}\, y \leftarrow \mathsf{Succ}(x) \,\mathsf{in}\, \mathsf{return}\, y$. Then $t'^{\mathsf{v}} \Downarrow_i \mathsf{return}\, \underline{n}$ and $T = \mathsf{return}\, \underline{1+n}$. The inductive hypothesis yields $t' \Downarrow_i \underline{n}$, and thus $\mathsf{Succ}(t') \Downarrow_i \underline{1+n}$.
- Case $t = \mathsf{Pred}(t')$. As above.
- Case $t = \mathsf{ifz}\, t_1 \,\mathsf{then}\, t_2 \,\mathsf{else}\, t_3$, and thus $t^{\mathsf{v}} = \mathsf{bind}\, x \leftarrow t_1^{\mathsf{v}} \,\mathsf{in}\, \mathsf{ifz}\, x \,\mathsf{then}\, t_2^{\mathsf{v}} \,\mathsf{else}\, t_3^{\mathsf{v}}$. We have $t_1^{\mathsf{v}} \Downarrow_{i_1} \mathsf{return}\, u$ and $\mathsf{ifz}\, u \,\mathsf{then}\, t_2^{\mathsf{v}} \,\mathsf{else}\, t_3^{\mathsf{v}} \Downarrow_{i_2} T$. For the second part of the execution, there are two cases:

- $u = \underline{0}$. Then $t_2^{\mathsf{v}} \Downarrow_{i_2} T$.
- $u = \underline{1+n}$. Then $t_3^{\mathsf{v}} \Downarrow_{i_2} T$.

In both cases, using the two inductive hypotheses, we have $\mathsf{ifz}\, t_1 \,\mathsf{then}\, t_2 \,\mathsf{else}\, t_3 \Downarrow_{i_1+i_2} v$ for a v with $v^{\mathsf{v}} = T$. ☐

Note that the cases Succ, Pred and ifz are similar to the corresponding cases in the proof of Lemma 2.16, since these terms are translated in the same way.

Again, the second part of the bisimulation is easier:

Lemma 2.24 (Simulation of t by t^{v}). *Let* $t \Downarrow_i v$. *Then* $t^{\mathsf{v}} \Downarrow_i v^{\mathsf{v}} = \mathsf{return}\, v^{\mathsf{val}}$.

Proof. By induction on $t \Downarrow_i v$. ☐

Part I

Coeffect systems

Chapter 3

Introduction

In this part of the thesis, we discuss a *coeffect*-based approach to complexity analysis with refinement type systems. In general, coeffect systems are about how the *environment* affects the program. Compare this to the dual notation, *effects*, which consider how the program affects the environment. There are many possible applications of coeffects. For example, it is discussed in [32] how coeffects can be used to track implicit variables, bound variable reuse, and analyse liveness of variables. Since we are interested in analysing the complexity of programs, we want to analyse how a program uses certain abstract non-duplicateable *resources*. Ultimately, the number of resources that are (potentially) consumed can be seen as an upper bound on the *dynamic cost* of a program (i.e. the cost of its execution). However, what exactly constitutes a resource varies from system to system.

$\mathsf{d}\ell\mathsf{PCF}$ is a family of conceptually similar coeffect-based refinement type systems. There are different variants – each of them is sound and relatively complete – but they target different execution strategies. $\mathsf{d}\ell\mathsf{PCF}_\mathsf{n}$ [11] is the original version, which targets the call-by-name version of PCF (without pairs), and $\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$ [12] targets the call-by-value strategy. To understand why there are different versions for different evaluation strategies, and to understand the underlying ideas of $\mathsf{d}\ell\mathsf{PCF}$ better, it is helpful to understand *bounded linear logic* BLL [19] – a logical calculus. The variants of $\mathsf{d}\ell\mathsf{PCF}$ are inspired by different *computational interpretations* of proofs in (variants of) BLL. The correspondence between proofs and programs in general is known as the *Curry-Howard isomorphism.*

3.1 A brief primer on BLL

Here we give a short summary of intuitionistic linear logic (ILL) and bounded linear logic (BLL) following [19]. Readers that are familiar with BLL can skip this section.

The reader should first recall *intuitionistic logic* (IL). A proof of a *sequent* in IL is a derivation from the (standard) rules in Figure 3.1. Here, A and B denote formulae, which are built from atomic formulae (α), logical conjunction ($\wedge$), and implication ($\rightarrow$). *Contexts* (Γ or Δ) are multisets of formulae.[1] The meaning of a sequent is that the

---

[1] A *multiset* is a set where every member has a count, but the order does not matter. For example, the

$$\dfrac{}{A \vdash A}\,\textsc{Axiom} \qquad \dfrac{A, A, \Gamma \vdash C}{A, \Gamma \vdash C}\,\textsc{Contraction} \qquad \dfrac{\Gamma \vdash C}{A, \Gamma \vdash C}\,\textsc{Weakening} \qquad \dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,\wedge\mathrm{R} \qquad \dfrac{A, B, \Gamma \vdash C}{A \wedge B, \Gamma \vdash C}\,\wedge\mathrm{L}$$

$$\dfrac{A, \Gamma \vdash B}{\Gamma \vdash A \to B}\,\to\mathrm{R} \qquad \dfrac{\Gamma \vdash A \qquad B, \Gamma \vdash C}{A \to B, \Gamma \vdash C}\,\to\mathrm{L} \qquad \dfrac{\Gamma \vdash A \qquad A, \Gamma \vdash B}{\Gamma \vdash B}\,\textsc{Cut}$$

Figure 3.1: Rules of intuitionistic logic (IL)

$$\dfrac{}{A \vdash A}\,\textsc{Axiom} \qquad \dfrac{!A, !A, \Gamma \vdash C}{!A, \Gamma \vdash C}\,\textsc{Contraction} \qquad \dfrac{\Gamma \vdash C}{!A, \Gamma \vdash C}\,\textsc{Weakening} \qquad \dfrac{A, \Gamma \vdash B}{!A, \Gamma \vdash B}\,\textsc{Dereliction}$$

$$\dfrac{\Delta_1 \vdash A \qquad \Delta_2 \vdash B}{\Delta_1, \Delta_2 \vdash A \otimes B}\,\otimes\mathrm{R} \qquad \dfrac{A, B, \Gamma \vdash C}{A \otimes B, \Gamma \vdash C}\,\otimes\mathrm{L} \qquad \dfrac{A, \Gamma \vdash B}{\Gamma \vdash A \multimap B}\,\multimap\mathrm{R} \qquad \dfrac{\Delta_1 \vdash A \qquad B, \Delta_2 \vdash C}{A \multimap B, \Delta_1, \Delta_2 \vdash C}\,\multimap\mathrm{L}$$

$$\dfrac{\Delta_1 \vdash A \qquad A, \Delta_2 \vdash B}{\Delta_1, \Delta_2 \vdash B}\,\textsc{Cut} \qquad \dfrac{!\Gamma \vdash B}{!\Gamma \vdash !B}\,\textsc{Promotion}$$

Figure 3.2: Rules of intuitionistic linear logic (ILL)

truth of the formula C follows from truth of the formulae in Γ. In intuitionistic logic, there are no restrictions on how often an assumption may be used. For example, in the sequent $A, A \to B \vdash A \wedge B$, the assumption A is used twice and $A \to B$ is used once. The *contraction* and *weakening* rules allow duplication and discarding of assumptions, respectively. If we want to show a conjunction of two formulae $A \wedge B$, we simply have to prove both of them with the same assumptions. Dually, if we have $A \wedge B$ as an assumption, we can split it into two assumptions. To show an implication $A \to B$, we have to show B with A as an additional assumption. The dual rule allows us to use an assumed implication.

The *cut* rule allows us to reuse a proof: If we have already shown A, we can add it as an assumption when we prove another formula B. Every proof can be converted into a proof that does not use the cut rule by a process called *cut elimination*; a proof is said to be in *normal form* if it does not use the cut rule.

Formulae of *intuitionistic linear logic* (ILL, see Figure 3.2 for the standard rules) are built using the following grammar:

$$A, B ::= \alpha \mid A \otimes B \mid A \multimap B \mid !A$$

---

equality $\{A, B, A\} = \{B, A, A\}$ holds, but $\{A, A\} \neq \{A\}$. A, Γ denotes the multiset where A appears one time more often than in Γ. Δ_1, Δ_2 denotes multiset union.

$$\dfrac{\vdash I_2 \sqsubseteq I_1 \qquad \vdash A \sqsubseteq B}{\vdash\, !_{a<I_1} A \sqsubseteq\, !_{a<I_2} B} \qquad \dfrac{\vdash A_2 \sqsubseteq A_1 \qquad \vdash B_1 \sqsubseteq B_2}{\vdash A_1 \multimap B_1 \sqsubseteq A_2 \multimap B_2} \qquad \dfrac{\vdash A_1 \sqsubseteq A_2 \qquad \vdash B_1 \sqsubseteq B_2}{\vdash A_1 \otimes B_1 \sqsubseteq A_2 \otimes B_2}$$

$$\overset{\textsc{Axiom}}{\dfrac{\vdash A \sqsubseteq A'}{A \vdash A'}} \qquad \overset{\textsc{Contraction}}{\dfrac{!_{a<I_1} A,\, !_{a<I_2} A\{a + I_1/a\}, \Gamma \vdash C}{!_{a<I_1+I_2+I_3} A, \Gamma \vdash C}} \qquad \overset{\textsc{Weakening}}{\dfrac{\Gamma \vdash C}{!_{a<I} A, \Gamma \vdash C}} \qquad \overset{\textsc{Dereliction}}{\dfrac{A\{0/a\}, \Gamma \vdash B}{!_{a<1+I} A, \Gamma \vdash B}}$$

$$\overset{\otimes\mathrm{R}}{\dfrac{\Delta_1 \vdash A \qquad \Delta_2 \vdash B}{\Delta_1, \Delta_2 \vdash A \otimes B}} \qquad \overset{\otimes\mathrm{L}}{\dfrac{A, B, \Gamma \vdash C}{A \otimes B, \Gamma \vdash C}} \qquad \overset{\multimap\mathrm{R}}{\dfrac{A, \Gamma \vdash B}{\Gamma \vdash A \multimap B}} \qquad \overset{\multimap\mathrm{L}}{\dfrac{\Delta_1 \vdash A \qquad B, \Delta_2 \vdash C}{A \multimap B, \Delta_1, \Delta_2 \vdash C}}$$

$$\overset{\textsc{Cut}}{\dfrac{\Delta_1 \vdash A \qquad A, \Delta_2 \vdash B}{\Delta_1, \Delta_2 \vdash B}} \qquad \overset{\textsc{Promotion}}{\dfrac{\Delta \vdash B \qquad \vdash \Gamma \sqsubseteq \sum_{a<I} \Delta}{\Gamma \vdash\, !_{a<I} B}}$$

where in Promotion: $\Delta = !_{c<J_1} A_1\{c + \sum_{d<a} J_1\{d/a\}/b\}, \ldots, !_{c<J_n} A_n\{c + \sum_{d<a} J_n\{d/a\}/b\}$

and $\sum_{a<I} \Delta := !_{c<(\sum_{a<I} J_1)} A_1, \ldots, !_{c<(\sum_{a<I} J_n)} A_n$

Figure 3.3: Rules of bounded linear logic (BLL)

ILL is a refinement of intuitionistic logic, where assumptions are seen as *resources.* These resources are normally neither duplicateable nor disposable, unless they are marked with '!'. Thus, in the contraction and weakening rules of ILL, we may only duplicate or forget banged assumptions. In other words, banged resources may be (re)used arbitrarily often. To prove a multiplicative conjunction $A \otimes B$, we also have to show A and B; however, we have to distribute the resources among the proofs of A and B.[2] For linear implications $A \multimap B$, we add A to the multiset of assumptions. This implies that we have to use A exactly once, unless A is itself a banged formula. For example, we can show $A \to B, A \to C \vdash\, !A \to B \otimes C$. The *promotion* rule makes a formula $!B$ arbitrarily often reusable. For this, the assumptions of B also have to be banged ($\Gamma!$ stands for a context of banged formulae). For example, we can show $!A, !(A \multimap B), C \vdash\, !B \otimes C$:

$$\dfrac{\dfrac{(2\times\text{dereliction})\ \dfrac{\dfrac{\vdots}{A, (A \multimap B) \vdash B}}{!A, !(A \multimap B) \vdash B}}{!A, !(A \multimap B) \vdash\, !B} \qquad \dfrac{}{C \vdash C}}{!A, !(A \multimap B), C \vdash\, !B \otimes C}$$

---

[2] This is why $\otimes$ is called *multiplicative conjunction.* There also is an *additive* conjunction (&), which we will discuss later.

The *substructural* control that we gain with bangs in ILL is coarse: We can only specify whether a resource may be used exactly once or arbitrarily often. Accordingly, *bounded linear logic* (BLL, see Figure 3.3) is a refinement of ILL that increases the expressive power. It has three main changes to ILL:

- There is a language of *index terms* (called *resource polynomials* in [19]).[3] Atomic formulae $\alpha(I_1, \ldots, I_n)$ may depend on a list index terms. Otherwise, the grammar of formulae is the same as in ILL.[4]
- The logic is *affine*, which means that resources may always be thrown away. In particular, if $\vdash A \sqsubseteq A'$ (which roughly says that A' is *weaker* than A), we can convert a proof of $\Gamma \vdash A$ into a proof of $\Gamma \vdash A'$.
- Banged formulae ($!A$) are refined to $!_{a<I} A$, where I is an index term and a is an index variable that may occur free in the formula A. Such a formula may be used at most I times, each time with a different value for a. Thus, the formula $!_{a<I} A$ is morally equivalent to $A\{0/a\} \otimes \cdots \otimes A\{I-1/a\}$.

In the *contraction* rule, we may contract the banged formulae $!_{a<I_1} A$ and $!_{a<I_2} A\{a + I_1/a\}$.[5] This means that if the first formula is equivalent to $A\{0/a\} \otimes \cdots \otimes A\{I_1 - 1/a\}$ and the second formula is equivalent to $A\{I_1/a\} \otimes \cdots \otimes A\{I_1 + I_2 - 1/a\}$, then the contracted formula, which we also write as $(!_{a<I_1} A) \uplus (!_{a<I_2} A\{a + I_1/a\}) := !_{a<I_1+I_2} A$, is morally equivalent to $A\{0/a\} \otimes \cdots \otimes A\{I_1 + I_2 - 1/a\}$.[6] Finally, since BLL is affine, we may throw away I_3 more 'instances' of A.

The *dereliction* rule allows us to access the first 'instance' of a banged type, under the assumption that the bound is positive. Similarly, the weakening rule allows us to throw away a banged resource, regardless of the bound. In a linear (also called 'precise') version of BLL, we may only throw away resources with bound 0.

The *promotion* rule is perhaps the most interesting rule, which allows duplicating a resource B. As in ILL, the assumptions of B must also be duplicated. Note that the index variable a may be free in Δ. Essentially, we prove B I-times, and we therefore have to build a sum over the bounds. In particular, if the assumption A_i has the bound J_i in Δ, then it will have the bound $\sum_{a<I} J_i$ in the context $\sum_{a<I} \Delta$, which is equivalent to:

$$\Delta\{b/c, 0/a\} \uplus \Delta\{(b + J\{0/a\})/c, 1/a\} \uplus \cdots \uplus \Delta\{(b + \textstyle\sum_{a<I-1} J)/c, I-1/a\}$$

3.2 From BLL to dℓPCF

BLL is a logical calculus, but what does this has to do with dℓPCF, which is a family of type systems? The Curry-Howard isomorphism relates proofs of intuitionistic logic to terms of

---

[3] In [19], the index terms must be *polynomials*. We remove this restriction here and assume an abstract language of index terms. This is essential to attain (relative) completeness of the dℓPCF calculi.

[4] Actually, BLL also features quantification over formulae ($\forall\alpha$). We will consider this feature later.

[5] $A\{J/a\}$ means that we substitute the index term J for the index variable a in A.

[6] The notation $\uplus$ was used in [11], and is called *modal sum*. Although modal sums are conceptually similar to the multiplicative conjunctive ($\otimes$), they should not be confused. The former is an operation on banged formulae with the same shape, while the latter is a type constructor.

the simply typed λ-calculus (i.e. PCF without fixpoints): A formula is provable if and only if there is a term of the corresponding type. For example, $\lambda x.\, \lambda y.\ \langle x; x(y)\rangle$ has type $(\sigma \to \tau) \to (\sigma \to \sigma \times \tau)$, which corresponds to a proof of the formula $(\alpha \to \beta) \to (\alpha \to \alpha \wedge \beta)$. The cut rule corresponds to the substitution lemma (see e.g. Lemma 2.3 for System T): If we can type $x : \sigma, \Gamma \vdash t_1 : \tau$ and $\Gamma \vdash t_2 : \sigma$, then we can type $\Gamma \vdash t_1\{t_2/x\} : \tau$.[7]

Now, there are two possible interpretations to translate formulae of IL to formulae of ILL [1]: In an intuitionistic proof of the implication $A \to B$, it could be the case that the premise is not used at all, or perhaps it is used more than once. This idea leads to the first translation – known as the call-by-name interpretation: $A \to B$ is translated to $!A' \multimap B'$, where A' and B' are the translations of A and B. In the corresponding λ-term, the argument is only evaluated when needed. Thus, we use call-by-name semantics.

The second interpretation is a call-by-value translation. Arrows $A \to B$ are translated to $!(A' \multimap B')$, where A' and B' are again the translations of A and B. Here, the idea is that the corresponding λ-term expects a 'proof' of A in normal form, i.e. the arguments to the function are *values*, following the call-by-value evaluation strategy.

We now want to introduce affine type systems where the types correspond to formulae of BLL. To this end, we will sketch out two variants of the types systems, one for the call-by-name and one for the call-by-value interpretation. In the first type system, types have the following grammar:

$$\sigma, \tau ::= b(I_1, \ldots, I_n) \bigm| ([a < I] \cdot \sigma) \multimap \tau$$

Here, b stands for refined base types, e.g. $\mathsf{Nat}[I]$. $[a < I]$ corresponds to $!_{a<I}$ in BLL. Note that we only have quantifiers at negative positions. For example, the type $([a < I]\cdot\sigma) \multimap \tau$ means that the argument may be evaluated at most I times, where I is an index term.

The call-by-value type system has the following syntax of types:

$$\sigma, \tau ::= b(I_1, \ldots, I_n) \bigm| [a < I] \cdot (\sigma \multimap \tau)$$

We may apply functions of type $[a < I] \cdot (\sigma \multimap \tau)$ I-times – each time possibly with different arguments and results.

The type systems that we have just sketched lead to $\mathsf{d\ell PCF_n}$ and $\mathsf{d\ell PCF_v}$, which target the call-by-name and call-by-value version of PCF, respectively. The austere reader will wonder whether these type systems are sound, since PCF features unbounded recursion: The simple typing $x : \tau \vdash \mu x.\, x : \tau$ is clearly unsound from the logical perspective. In our versions of $\mathsf{d\ell PCF}$, it is in fact possible to type diverging terms.[8] *Soundness* of our systems, however, ensures that typings of diverging terms also have diverging index terms. On the other hand, *(relative) completeness* implies that terminating programs can be typed with terminating index terms as annotations.

In Section 2.3, we recapitulated a variant of PCF called call-by-push value (CBPV). One of the main contribution in this part of the thesis is that we define a type system called $\mathsf{d\ell PCF_{pv}}$ that targets CBPV and subsumes $\mathsf{d\ell PCF_v}$ and $\mathsf{d\ell PCF_n}$.

---

[7] This holds for a suitable definition of substitution. For simplicity, however, we always assume that the terms that are substituted for values are closed.

[8] This is not possible in the versions of $\mathsf{d\ell PCF}$ in [11, 12].

3.3 Costs and weights

The last question that we should address before we can continue is: How can we analyse the *complexity* of programs using the type systems inspired by BLL? The answer is quite simple: If we sum up the indexes at the promotion rule, we get an upper bound on how many resources can be consumed in the proof of a closed formula. If the initial context is empty, we can only use those resources that are allocated and maybe pushed to the context afterwards. This sum, which is an index term, is called the *weight* of a proof.

Accordingly, in the variants of dℓPCF, the weight of a typing is a (static) upper bound on the (dynamic) *execution cost* of the corresponding program. In Section 2.2, we have defined, not without coincidence, that the cost of an CBN execution is the number of *variable lookups* – each variable lookup corresponds to the use of one resource. Furthermore, the cost of a CBV execution is the number of β-substitutions – each application consumes one resource. In dℓPCF$_{\mathsf{pv}}$, the weight will be an upper bound on the number of times thunked computation is forced during the execution.

3.4 Organisation of the remainder of this part

The first sound and relatively complete type system that we discuss in this part is dℓT in Chapter 4, which targets System T. We also use this chapter to introduce basic concepts that are also in common with dℓPCF$_{\mathsf{v}}$, like the concrete syntax of the types. Also all type systems in this part feature a set of *constraints* over index terms ($\mathcal{L}^{\ell}_{idx}$), which we will introduce there. In Chapters 5 and 6, we will discuss dℓPCF$_{\mathsf{v}}$ (call-by-value) and dℓPCF$_{\mathsf{n}}$ (call-by-name), which first appeared in [12] and [11], respectively. For the former, we will give arguably simpler proofs of soundness and completeness, but we omit proofs for the latter. Since both dℓT and dℓPCF$_{\mathsf{v}}$ target languages with call-by-value semantics, we show that dℓT can be embedded in dℓPCF$_{\mathsf{v}}$. In Chapter 7, we discuss a new type system called dℓPCF$_{\mathsf{pv}}$, which subsumes both dℓPCF$_{\mathsf{n}}$ and dℓPCF$_{\mathsf{v}}$. We prove that dℓPCF$_{\mathsf{pv}}$ is sound and (relatively) complete, and from this we derive the same results for dℓPCF$_{\mathsf{v}}$ and dℓPCF$_{\mathsf{n}}$. In the soundness and completeness proofs for dℓPCF$_{\mathsf{pv}}$, we use the same techniques as in Chapter 5. In Chapter 8, we discuss an algorithm for creating composable dℓPCF$_{\mathsf{pv}}$ typings, and we add polymorphism to dℓPCF$_{\mathsf{pv}}$.

Chapter 4

Index terms ($\mathcal{L}^{\ell}_{idx}$) and $\mathsf{d}\ell\mathsf{T}$

In this chapter, we introduce a coeffect system called $\mathsf{d}\ell\mathsf{T}$ that targets System T. This system is an extension of a system (with the same name) published in [3], but it can also be seen as a stripped-down version of $\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$ [12] (which will be the subject of the next chapter) with an additional rule for higher-order iteration. Thus, $\mathsf{d}\ell\mathsf{T}$ is not novel on its own. Instead, we also use the present chapter to introduce the index term language $\mathcal{L}^{\ell}_{idx}$ (which is used throughout the first part of this thesis) and the syntax and meaning of $\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$ types and modal sums (which are the same in $\mathsf{d}\ell\mathsf{T}$ and $\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$). We do not present formal proofs here, as we will discuss more general proofs in Chapter 5 and Chapter 7. Finally, we will type some first-order functions, and we will compare our version with the version published in [3].

Although all variants of System T and PCF support product and sum types, we will not consider these types here. Adding these types is straightforward, as we will show in Section 7.7.

4.1 Types of $\mathsf{d}\ell\mathsf{T}$ (and $\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$)

There are two syntactic categories of types in $\mathsf{d}\ell\mathsf{T}$ (and $\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$), *modal types* and *linear types*, which are defined by mutual induction. Modal types are the types that occur in contexts, and they are also the types that terms are assigned to.

Types are annotated with *index terms* (e.g. I, J), which we will define in the next section. For now, it suffices to know that index terms are expressions that (may) evaluate to natural numbers, and *index variables* (e.g. a) may appear free in index terms.[1]

$$\begin{aligned} \text{Modal types:}\quad & \sigma, \tau, \rho ::= \mathsf{Nat}[I] \;\big|\; [a < I] \cdot A \\ \text{Linear types:}\quad & A, B ::= \sigma \multimap \tau \\ \text{Contexts:}\quad & \Gamma, \Delta ::= \emptyset \;\big|\; x : \tau, \Gamma \end{aligned}$$

---

[1] The syntax of types also comes from [12]. However, this work also considers interval types, $\mathsf{Nat}[I_1; I_2]$, which we do not consider in this thesis.

Notationally, $\multimap$ binds stronger than $[a < I]$, which means that we can write $[a < I]\cdot\sigma \multimap \tau$ for $[a < I]\cdot(\sigma \multimap \tau)$. To avoid confusion, however, we will often use full parentheses, since it is exactly the opposite in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$. Furthermore, we write $[\_ < I]\cdot A$ if the index variable does not appear in A.

$\mathsf{Nat}[I]$ stands for the type of constants $\underline{n}$ that are equivalent to the index term I.

As $\mathsf{d}\ell\mathsf{T}$ targets System T, which has call-by-value semantics, we bound how often abstractions may be applied. The type $[a < I]\cdot(\sigma \multimap \tau)$ means that a term of this type may be applied I times. Here, $[a < I]$ also acts as a *binder* for the index variable a. This means that the index variable a may occur free in the index terms of σ and τ. For example, if we have a typing of a function with type $[a < 2]\cdot(\mathsf{Nat}[a] \multimap \mathsf{Nat}[1 + a])$, we can apply this function twice: once each with an argument of type $\mathsf{Nat}[0]$ and $\mathsf{Nat}[1]$, respectively.

Note that in contrast to BLL, contexts are not multisets.[2] This means that when we write $x : \tau, \Gamma$, we implicitly assume that x is not already in the domain of Γ. As usual, we assume that contexts assign a type to every free variable. The types $\Gamma(y)$ for variables y that are not free in a term t are irrelevant, and can be removed from the context.

The types of $\mathsf{d}\ell\mathsf{T}$ and $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ can be seen as decorated PCF types. The *erasure* function $(\!|\cdot|\!)$ removes these decorations and returns the PCF type with the same shape. Note that we overload the function for modal and linear types.

Definition 4.1 (Type erasure)**.** By mutual recursion on the modal and linear types, we define: $(\!|\mathsf{Nat}[I]\,|\!) := \mathsf{Nat}$, $(\!|[a < I]\cdot A|\!) := (\!|A|\!)$, and $(\!|\sigma \multimap \tau|\!) := (\!|\sigma|\!) \to (\!|\tau|\!)$. We call $(\!|\sigma|\!)$ the *shape* of σ.

4.2 Index terms ($\mathcal{L}^{\ell}_{idx}$) and constraints

We now define the language $\mathcal{L}^{\ell}_{idx}$ of index terms for $\mathsf{d}\ell\mathsf{T}$ and the $\mathsf{d}\ell\mathsf{PCF}$ family. From the above section, it should be clear that index terms serve two purposes:

- To bounds how often an (arrow) type may be used, and
- to refine the numerical values of the simple type Nat.

$\mathcal{L}^{\ell}_{idx}$ is a generalisation of a similar language in [11, 12]. As in these works, we will later need to extend the language to show (relative) completeness. In particular, to handle unbounded recursion for the systems in the $\mathsf{d}\ell\mathsf{PCF}$ family, we need to include a non-total construct.

$$\begin{array}{lr}
\text{Index terms:} & I, J, K, L, M ::= \bot \mid n \mid a \mid I + J \mid I \mathbin{\dot-} J \mid \sum_{a<I} J \mid \mathsf{if}\, C \,\mathsf{then}\, J \,\mathsf{else}\, K \mid \cdots \\
\text{Constraints:} & C ::= I \sqsubseteq J \mid I \equiv J \mid I < J \mid I \leq J \mid I \gtrsim J \mid I\downarrow \\
\text{Constr. list:} & \Phi := \emptyset \mid C, \Phi
\end{array}$$

---

[2] In terms of BLL, the contraction rule is applied implicitly in all multiplicative operations to ensure that every variable only appears once in the typing context.

$$
\begin{aligned}
\llbracket \bot \rrbracket &= \bot\\
\llbracket n \rrbracket &= n\\
\llbracket I + J \rrbracket &= \begin{cases} m + n & \llbracket I \rrbracket = m \wedge \llbracket J \rrbracket = n\\ \bot & \llbracket I \rrbracket = \bot \vee \llbracket J \rrbracket = \bot \end{cases}\\
\llbracket I \mathbin{\dot{-}} J \rrbracket &= \begin{cases} m \mathbin{\dot{-}} n & \llbracket I \rrbracket = m \wedge \llbracket J \rrbracket = n\\ \bot & \llbracket I \rrbracket = \bot \vee \llbracket J \rrbracket = \bot \end{cases}\\
\llbracket \textstyle\sum_{a<I} J \rrbracket &= \begin{cases} 0 & \llbracket I \rrbracket = 0\\ \llbracket J\{0/a\} + \sum_{a<I \mathbin{\dot{-}} 1} J\{a+1/a\} \rrbracket & \llbracket I \rrbracket > 0\\ \bot & \llbracket I \rrbracket = \bot \end{cases}\\
\llbracket \mathsf{if}\ C\ \mathsf{then}\ I\ \mathsf{else}\ J \rrbracket &= \begin{cases} \llbracket I \rrbracket & \vDash C\\ \llbracket J \rrbracket & \nvDash C \end{cases}
\end{aligned}
$$

$$
\frac{\exists n : \mathsf{Nat}.\ \llbracket I \rrbracket = n}{\vDash I \downarrow}
\qquad
\frac{\forall n : \mathsf{Nat}.\ \llbracket J \rrbracket = n \Rightarrow \llbracket I \rrbracket = n}{\vDash I \sqsubseteq J}
\qquad
\frac{\vDash I \sqsubseteq J \qquad \vDash J \sqsubseteq I}{\vDash I \equiv J}
$$

$$
\frac{\begin{array}{c}\exists m : \mathsf{Nat}.\ \llbracket I \rrbracket = m \wedge\\ \forall n : \mathsf{Nat}.\ \llbracket J \rrbracket = n \Rightarrow m < n\end{array}}{\vDash I < J}
\qquad
\frac{\vDash I < J \text{ or } \vDash I \sqsubseteq J}{\vDash I \leq J}
\qquad
\frac{\begin{array}{c}\forall n : \mathsf{Nat}.\ \llbracket J \rrbracket = n \Rightarrow\\ \exists m : \mathsf{Nat}.\ \llbracket I \rrbracket = m \wedge m \geq n\end{array}}{\vDash I \gtrsim J}
$$

Figure 4.1: Semantics of closed $\mathcal{L}^{\ell}_{idx}$ terms and constraints

Here, n stands for a constant, and a, b, c are index variables (from a list ϕ of index variables). Index variable *substitution* is defined in the standard way. For example, $I\{a+J/a\}$ is the index term where all occurrences of a are replaced with $a + J$.[3]

The main addition to the language in [11, 12] is that we add support for *undefined* index terms ($\bot$). This will allows us to embed the simple type systems inside dℓT and the variants of dℓPCF. In particular, we will also be able to type diverging programs. However, our soundness theorems ensure that if the index terms that occur in a refinement terminate, so do the typed terms. (This is not relevant for dℓT, since all simply typed System T programs terminate.)

The semantics of the language of index terms and constraints is given in Figure 4.1. For closed index terms I, we write $\llbracket I \rrbracket = k$ if the index term I is *defined* and has value k. We write $\llbracket I \rrbracket = \bot$ if the index term I is *undefined.*[4]

The constraints $\sqsubseteq$ and $\gtrsim$ are only used to compare Nat-refinements. Thus, they trivi-

---

[3] Note that in contrast to term substitution, we often substitute non-closed index terms for variables.

[4] Since we will also allow recursive index terms later, formally, we need to define the relation $\llbracket I \rrbracket = k$ inductively or using a term rewriting system. In the latter case, $\llbracket I \rrbracket = \bot$ means that one cannot reduce $\llbracket I \rrbracket$ to a constant. Furthermore, we define $m \mathbin{\dot{-}} n$ as $m - n$ if $m \geq n$ and 0 otherwise.

ally hold if the right hand side is $\bot$.[5]

The constraint $I \downarrow$ simply asserts that the index term I is defined.

We can prove the following facts about the semantics of the constraints:

Fact 4.2.

- *The relation $\vDash \cdot < \cdot$ is a partial order (antisymmetric and transitive), where $\bot$ is the largest element.*
- *The relations $\vDash \cdot \leq \cdot$ and $\vDash \cdot \sqsubseteq \cdot$ are preorders (reflexive and transitive).*
- *The relation $\vDash \cdot \equiv \cdot$ is an equivalence (reflexive, symmetric, transitive).*
- *For all closed index terms I and J, we either have $\vDash I < J$ or $\vDash J \leq I$.*

We use the meta variable ϕ to denote lists of index variables. A *valuation* ν of ϕ is a substitution that maps all index variables of ϕ to a constant (i.e. not $\bot$). We write $val(\phi)$ for the set of such valuations and define $[\![I]\!](\nu) := [\![I\nu]\!]$ for non-closed index terms.

Finally, if C is a constraint that only has the variables in ϕ free, and Φ is a list of such constraints, then $\phi; \Phi \vDash C$ is an *assertion*:

$$\frac{}{\vDash \emptyset} \qquad \frac{\vDash C \qquad \vDash \Phi}{\vDash C, \Phi} \qquad \frac{\forall \nu \in val(\phi).\ \vDash \Phi\nu \Rightarrow\ \vDash C\nu}{\phi; \Phi \vDash C}$$

Note that if Φ is unsatisfiable (e.g. if it contains the constraints $1 < 0$ or $\bot < 1$), then the assertion holds vacuously.

Interpretations of undefined index terms The original versions of dℓPCF [11, 12] do not support ‘undefined’ index terms. Thus, only terminating programs can be typed in these systems, since they (implicitly) add constraints $\phi; \Phi \vDash I \downarrow$ for all appearing index terms and the respective Φ. In our generalisation of the systems, the index term $\bot$ has different meaning for bounds and Nat-refinements:

- As a bound, $\bot$ can be intuitively thought as ‘infinite’. The sub-exponential $[a < \bot]$ is equivalent to ! in linear logic. This means that the abstraction can be applied arbitrarily often. Terms that have $\bot$ as the annotation of a bound at a positive position thus also have the *weight* $\bot$. We will show that for a *precise* typing (which we will define in Section 5.4) of a closed program, this means that the program diverges, since all allocated resources must be used in such a program. In a non-precise typing, we may always weaken a finite weight to $\bot$.
- The type $\mathsf{Nat}[\bot]$ is equivalent to the simple type Nat. Thus, $\mathsf{Nat}[\bot]$ is inhabited by all constants. We may subtype $\mathsf{Nat}[I] \sqsubseteq \mathsf{Nat}[\bot]$, but only in a non-precise typing.

---

[5] We only use $0 \gtrsim J$ in the case-distinction rule. There, the constraint should hold vacuously if $[\![J]\!] = \bot$.

4.3 Modal sums

Binary modal sum From Chapter 3, it should already be clear why we need modal sums: Different parts of a program may need to share common variables. For example, in an application $t_1\, t_2$, both terms may need to use a function x from the context. The term t_1 may need the first I_1 'instances' of the type of x, and t_2 may need the remaining I_2 instances. To type $t_1\, t_2$, we need all $I_1 + I_2$ 'instances' of the type of x.

Note that the 'order' of these instances does not correspond to the order in which they are consumed, but with the syntactic order. For example, in an application $t_1\, t_2$, t_1 may apply x I_1-times before evaluating to a λ-abstraction, then t_2 applies x I_2-times, and the body of the λ-abstraction applies x another I_3-times. In this case, the type of x will consist of the $I_1 + I_3$ instances by t_1 and then the I_2 instances by t_2.

There are also some seemingly nonsensical modal sums. For example, in the following typing, each application of the variable x yields a different result:

$$x : [a < 2] \cdot (\mathsf{Nat}[0] \multimap \mathsf{Nat}[a]) \vdash_4 \mathit{add}\, (x\, \underline{0})\, (x\, \underline{0}) : \mathsf{Nat}[0+1]$$

Here, the type of x is split using the following modal sum: $([a < 1] \cdot (\mathsf{Nat}[0] \multimap \mathsf{Nat}[a])) \uplus ([a < 1] \cdot (\mathsf{Nat}[0] \multimap \mathsf{Nat}[a+1]))$. The first/second application of x is typed with the first/second type, respectively, and each of the types can be used at most once. Note that such a type cannot be constructed by a closed program (because of the absence of side effects), but we will not exclude this kind of modal type.

Variables of natural types like $\mathsf{Nat}[I]$ can always be shared among different parts of a program. However, in the definition of binary modal sum $\mathsf{Nat}[I_1] \uplus \mathsf{Nat}[I_2]$, we assume that I_1 and I_2 are equal.

Definition 4.3 (Binary modal sum)**.** We define the ternary relation $\sigma_1 \uplus \sigma_2 = \tau$ inductively:

$$\frac{\sigma_1 = \mathsf{Nat}[I] \qquad \sigma_2 = \mathsf{Nat}[I]}{\sigma_1 \uplus \sigma_2 = \sigma_1} \qquad \frac{\sigma_1 = [a < I_1] \cdot A \qquad \sigma_2 = [a < I_2] \cdot A\{a + I_1/a\}}{\sigma_1 \uplus \sigma_2 = [a < I_1 + I_2] \cdot A}$$

Note that we slightly abuse notation here, in a way that is common in mathematics. For example, if mathematicians write $1 + \lim_{x\to\infty} f(x)$, they implicitly assume that this limit is defined. Here, whenever we write $\sigma_1 \uplus \sigma_2$, we implicitly assume that the types fulfil the syntactic restrictions in the above definition. However, we will later show that if the types have the same shape, then we can always construct *equivalent* types such that the sum is defined (see Lemma 5.36).

Bounded modal sum There is another kind of sum, which we will need for the λ and iteration rules. In these rules, variables may be reused along multiple uses of the same function. The definition of sum of quantified types should be familiar from Chapter 3.

Definition 4.4 (Bounded modal sums)**.** Let σ be a type that may have a free, and let I an index term. We define the binary relation $\sum_{a<I} \sigma = \tau$ inductively:

$$\frac{\sigma = \mathsf{Nat}[I] \qquad a \text{ not free in } I}{\sum_{a<I} \sigma = \mathsf{Nat}[I]} \qquad\qquad \frac{\sigma = [c < J] \cdot A\{c + \sum_{d<a} J\{d/a\}/b\}}{\sum_{a<I} \sigma = [b < \sum_{a<I} J] \cdot A}$$

Note that in the second rule, A has b as one additional free variable (but a is not free). In that rule, the substitution introduces two free variables (a and c). J and σ have a free, but not b and c.

If $\sigma = \mathsf{Nat}[I]$, we again have a syntactic restriction, requiring that the index variable a may not occur free in σ. The reason for this is the same reason for which $\mathsf{Nat}[I_1] \uplus \mathsf{Nat}[I_2]$ is only defined if $I_1 = I_2$.

Bounded modal sums can be informally described using the following equation:

$$\sum_{a<I} \sigma = \sigma\{0/a\} \uplus \cdots \uplus \sigma\{I-1/a\}$$

For example, consider the following modal type:

$$\sigma = [b < a] \cdot (\mathsf{Nat}\left[b + \textstyle\sum_{d<I} d\right] \multimap \mathsf{Nat}\left[1 + b + \textstyle\sum_{d<I} d\right])$$

Then, the sum $\sum_{a<I} \sigma = [b < \sum_{a<I} a] \cdot (\mathsf{Nat}[b] \multimap \mathsf{Nat}[1+b])$ can be understood as the following (informal) modal sum:

$$\begin{aligned}
&([c < 0] \cdot (\mathsf{Nat}[0] \multimap \mathsf{Nat}[1])) \\
\uplus\ &([c < 1] \cdot (\mathsf{Nat}[c+0] \multimap \mathsf{Nat}[1+c+0])) \\
\uplus\ &([c < 2] \cdot (\mathsf{Nat}[c+0+1] \multimap \mathsf{Nat}[1+c+0+1])) \\
\uplus\ &([c < 3] \cdot (\mathsf{Nat}[c+0+1+2] \multimap \mathsf{Nat}[1+c+0+1+2])) \\
\uplus\ &\cdots \\
\uplus\ &([c < I-1] \cdot (\mathsf{Nat}\left[c + \textstyle\sum_{d<I-1} d\right] \multimap \mathsf{Nat}\left[1 + c + \textstyle\sum_{d<I-1} d\right]))
\end{aligned}$$

Modal sums are lifted to contexts pointwise, i.e. $\emptyset \uplus \emptyset = \emptyset$, $x : \tau \uplus \emptyset = x : \tau$, and $(x : \sigma_1, \Delta_1) \uplus (x : \sigma_2, \Delta_2) = x : (\sigma_1 \uplus \sigma_2), \Delta_1 \uplus \Delta_2$.

4.4 Typing rules

$\mathsf{d\ell T}$ typing judgements (and also $\mathsf{d\ell PCF_v}$ typing judgements) have the shape $\phi; \Phi; \Gamma \vdash_K t : \tau$. Here, ϕ is a list of index variables, and Φ a list of constraints over these variables, Γ is the typing context of the typing, and K is the weight. All index terms in Φ, Γ, K, and τ must be closed in ϕ.

$$\frac{\phi;\Phi \vDash I \sqsubseteq J}{\phi;\Phi \vdash \mathsf{Nat}[I] \sqsubseteq \mathsf{Nat}[J]} \qquad \frac{\phi;\Phi \vdash \sigma_2 \sqsubseteq \sigma_1 \qquad \phi;\Phi \vdash \tau_1 \sqsubseteq \tau_2}{\phi;\Phi \vdash \sigma_1 \multimap \tau_1 \sqsubseteq \sigma_2 \multimap \tau_2}$$

$$\frac{\phi;\Phi \vDash J \leq I \qquad \phi; a < J, \Phi \vdash A \sqsubseteq B}{\phi;\Phi \vdash [a < I]\cdot A \sqsubseteq [a < J]\cdot B} \qquad \frac{\begin{array}{c}\phi;\Phi \vdash \sigma \sqsubseteq \tau\\ \phi;\Phi \vdash \tau \sqsubseteq \sigma\end{array}}{\phi;\Phi \vdash \sigma \equiv \tau} \qquad \frac{\begin{array}{c}\phi;\Phi \vdash A \sqsubseteq B\\ \phi;\Phi \vdash B \sqsubseteq A\end{array}}{\phi;\Phi \vdash A \equiv B}$$

$$\textsc{Sub}\quad \frac{\begin{array}{c}\phi;\Phi;\Gamma' \vdash^{\mathsf{v}}_{K_1} t : A_1 \qquad \phi;\Phi \vdash A_1 \sqsubseteq A_2\\ \phi;\Phi \vdash \Gamma \sqsubseteq \Gamma' \qquad \phi;\Phi \vDash K_1 \leq K_2\end{array}}{\phi;\Phi;\Gamma \vdash_{K_2} t : A_2} \qquad \textsc{Var}\quad \phi;\Phi; x:\sigma,\Gamma \vdash_0 x : \sigma \qquad \textsc{Const}\quad \phi;\Phi;\emptyset \vdash_0 \underline{n} : \mathsf{Nat}[n]$$

$$\textsc{Succ}\quad \frac{\phi;\Phi;\Gamma \vdash_M t : \mathsf{Nat}[J]}{\phi;\Phi;\Gamma \vdash_M \mathsf{Succ}(t) : \mathsf{Nat}[1+J]} \qquad \textsc{Pred}\quad \frac{\phi;\Phi;\Gamma \vdash_M t : \mathsf{Nat}[J]}{\phi;\Phi;\Gamma \vdash_M \mathsf{Pred}(t) : \mathsf{Nat}[J \dot{-} 1]}$$

$$\textsc{Lam}\quad \frac{a,\phi; a < I, \Phi; x:\sigma, \Delta \vdash_K t : \tau}{\phi;\Phi; \sum_{a<I} \Delta \vdash_{I + \sum_{a<I} K} \lambda x.\, t : [a < I]\cdot(\sigma \multimap \tau)} \qquad \textsc{App}\quad \frac{\begin{array}{c}\phi;\Phi;\Delta_1 \vdash_{K_1} t_1 : [a<1]\cdot(\sigma \multimap \tau)\\ \phi;\Phi;\Delta_2 \vdash_{K_2} t_2 : \sigma\{0/a\}\end{array}}{\phi;\Phi;\Delta_1 \uplus \Delta_2 \vdash_{K_1+K_2} t_1\, t_2 : \tau\{0/a\}}$$

$$\textsc{Ifz}\quad \frac{\begin{array}{c}\phi;\Phi;\Delta_1 \vdash_{K_1} t_1 : \mathsf{Nat}[J]\\ \phi; 0 \gtrsim J, \Phi; \Delta_2 \vdash_{K_2} t_2 : \tau\\ \phi; 0 < J, \Phi; \Delta_2 \vdash_{K_2} t_3 : \tau\end{array}}{\phi;\Phi;\Delta_1 \uplus \Delta_2 \vdash_{K_1+K_2} \mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3 : \tau}$$

$$\textsc{iter}\quad \frac{\begin{array}{c}a,b,\phi; b < K, a < I, \Phi; \Delta_1 \vdash_{M_1} t_1 : [c<1]\cdot(\sigma \multimap \sigma\{1+a/a\})\\ b,\phi; b < K, \Phi; \Delta_2 \vdash_{M_2} t_2 : \sigma\{0/a, 0/c\}\end{array}}{\phi;\Phi; \sum_{b<K}((\sum_{b<I} \Delta_1\{I \dot{-} 1 \dot{-} a/a\}) \uplus \Delta_2) \vdash_M \mathsf{iter}\, t_1\, t_2 : [b<K]\cdot(\mathsf{Nat}[I] \multimap \sigma\{I/a, 0/c\})}$$

$$\text{with } M := K + \sum_{b<K}(I + (\sum_{a<I} M_1) + M_2)$$

Figure 4.2: Subtyping and typing rules of dℓT. All rules except ITER are also rules of dℓPCF$_{\mathsf{v}}$.

The index variables in ϕ can be thought of as universally quantified variables. Indeed, the following equality can be shown:[6]

$$\phi; \Phi; \Gamma \vdash_M t : \tau \iff \forall \nu \in val(\phi).\ \vDash \Phi\nu \Rightarrow \emptyset; \emptyset; \Gamma\nu \vdash_{M\nu} t : \tau\nu$$

In particular, if the constraint list Φ is unsatisfiable, we can convert any simple typing into a $\mathsf{d}\ell\mathsf{T}$ typing with the same 'shape'. This rule is called the *explosion rule* and it also holds for the systems of the $\mathsf{d}\ell\mathsf{PCF}$ and $\mathsf{d}f\mathsf{PCF}$ families.

The typing and subtyping rules are depicted in Figure 4.2. We will explain them one-by-one; the rules CONST, SUCC, and PRED are clear.

Subtyping $\phi; \Phi \vdash \sigma \sqsubseteq \tau$ means that τ is *weaker* than σ. For example, the type $[a < I_2] \cdot A$ is a subtype of $[a < I_1] \cdot A$ (relative to a constraint set Φ) if and only if the assertion $\phi; \Phi \vDash I_2 \leq I_1$ holds.

Subsumption The subsumption rule always allows us to derive a weaker (or equivalent) typing. For example, it lets us increase the weight, and it lets us replace types by weaker types. We can also assign the undefined weight ($\bot$) to the new typing, since, by definition, $\phi; \Phi \vDash K' \leq \bot$. Similarly, we can weaken $\mathsf{Nat}[I] \sqsubseteq \mathsf{Nat}[\bot]$. However, both kinds of weakening are only allowed in *non-precise* typing. In a *precise* typing, subsumption is only allowed with $\equiv$.

Variable We can assign the type $\Gamma(x)$ to variables x.

Lambda We want to build a typing for $\lambda x.\, t$ that can be used I times. This means that we have to type t I-times, which is accomplished by adding a free index variable a to ϕ and the constraint $a < I$ to Φ. We build the sum over the contexts and over the weights. Additionally, we add I to the weight, since it accounts for the cost of the *potential* applications of this function.

Application We first type t_1 with type $[a < 1] \cdot (\sigma \multimap \tau)$ (possibly after subtyping). This allows us to use the function type once; speaking of function applications as resources, we consume one of these resources. Because the cost for the application rule has already been paid for in the lambda (or iteration) rule, we do not have to increment the weight; the weight is just the sum of the weights of t_1 and t_2.

Case distinction We first type t_1 with the type $\mathsf{Nat}[J]$. This means that t_1 will terminate to a constant $\underline{n}$ such that $\phi; \Phi \vDash n \sqsubseteq J$. After this, we type t_2 *and* t_3 with the final type τ, where we add the constraints $0 \gtrsim J$ and $0 < J$, respectively. Using these constraints, we add static information to the typing: The information on the result of t_1 can be used in the typings of t_2 and t_3. For example, we can type $a; \emptyset; x : \mathsf{Nat}[a] \vdash_0 \mathsf{ifz}\, a\, \mathsf{then}\, (\mathsf{ifz}\, a\, \mathsf{then}\, \underline{0}\, \mathsf{else}\, t)\, \mathsf{else}\, (\mathsf{ifz}\, a\, \mathsf{then}\, t\, \mathsf{else}\, \underline{0}) : \mathsf{Nat}[0]$ for any program t, since t will never be executed. Note that if J is a constant, then one of the typings holds trivially (using the explosion rule).

---

[6] This is a corollary of the *uniformisation* lemma and *index term substitution*, which we will discuss in the next chapter. The direction '$\Leftarrow$' only holds if the language of index terms is sufficiently expressive.

In the special case $J = \bot$, the constraints $0 \gtrsim \bot$ and $0 < \bot$ are tautological and can thus be removed. Morally, this means that we do not gain any static information: Since the result of t_1 is unknown, we cannot express which of the two branches is taken. Note that this is the only rule where $\gtrsim$ is used, and we deliberately defined its semantics such that the constraint $\vDash 0 \gtrsim \bot$ holds.

Moreover, note that in a precise typing, J can only be undefined if t_1 diverges. The typing rules in Figure 4.2, however, do not exploit this fact. We will discuss admissible changes to the rules for precise typings in Section 5.4.

Iteration We want to make K applications of $\mathsf{iter}\, t_1\, t_2$, each of them (for $b < K$) gets a value of type $\mathsf{Nat}[I]$ as argument. This means that t_1 is executed $\sum_{c<K} I$ times in total, and t_2 is executed K times. For each of the calls of t_1 (for $b < K$ and $a < I$), we need to type t_1 once. The term t_2 only needs to be typed K-times; it is evaluated once at the end of every application of $\mathsf{iter}\, t_1\, t_2$.

The type $\sigma\{0/c\}$, which has a and b as free index variables, describes a 'chain': For $b < K$, $\sigma\{0/a, 0/c\}$ is the type of t_2, $\sigma\{1/a, 0/c\}$ is the type of $t_1\, t_2$, …, and finally, $\sigma\{I/a, 0/c\}$ is the result type of $\mathsf{iter}\, t_1\, t_2$ (with a value of type $\mathsf{Nat}[I]$ as argument).

The weight of the typings of t_1 already accounts for the cost of the applications of t_1. We also add $\sum_{b<K} I$ to the weight to account for the costs of the 'iteration unfolding' steps (i.e. $\mathsf{iter}\, t_1\, t_2\, (\underline{1+n}) \succ_1 t_1\, (\mathsf{iter}\, t_1\, t_2\, \underline{n})$). We build a similar sum over the contexts. However, for technical reasons, the order of Δ_1 is reversed.

Explicit or implicit subsumption All typing rules except the subsumption rule are syntax directed. Therefore, care must be taken when inverting a typing. Instead of having an explicit subsumption rule, we can also add subtyping judgements to the premises of the typing rules. For example, the following is an invertible rule for λ-abstractions, with subsumption 'built in':

$$\frac{\phi; \Phi \vdash \Gamma \sqsubseteq \sum_{a<I} \Delta \qquad \begin{array}{c} a, \phi; a < I, \Phi; x : \sigma, \Delta \vdash_K t : \tau \\ \phi; \Phi \vDash I + \sum_{a<I} K \le M \end{array} \qquad \phi; \Phi \vdash [a < I] \cdot (\sigma \multimap \tau) \sqsubseteq \rho}{\phi; \Phi; \Gamma \vdash_M \lambda x.\, t : \rho}$$

Having an explicit subsumption rule or not is a purely cosmetic design choice – the subsumption rule will be admissible in any case. For comparison, the rules of dℓPCF$_\mathsf{n}$ in Chapter 6 are presented without an explicit subsumption rule.

4.5 Meta theory

The two key properties of dℓT are *soundness* and *completeness*.

It can be shown that every simply typed System T term terminates, but we do not know in how many steps. If a System T program, that is a closed term t with the simple type Nat, is also typed in dℓT with weight k, we can show that k is an upper bound on the cost of the execution.

Theorem 4.5 (Soundness of $\mathsf{d\ell T}$ for programs)**.** *Let t be a closed program (i.e. a System T term with simple type Nat). Then we can show:*

- *Let $\emptyset;\emptyset;\emptyset \vdash^{\mathsf{c}}_{k} t : \mathsf{Nat}[I]$ be a $\mathsf{d\ell T}$ typing. Then there is a $k' \leq k$ and a constant n such that $t \Downarrow_{k'} \underline{n}$ and $\models n \sqsubseteq I$. In particular, if $\models I \equiv m$, then $m = n$.*
- *Let $\emptyset;\emptyset;\emptyset \vdash^{\mathsf{c}}_{K} t : \mathsf{Nat}[I]$ be a precise typing and $t \Downarrow_{k} \underline{n}$. Then $\models K \equiv k$ and $\models I \equiv n$.*

The key lemma of the soundness proof is *subject reduction.* This lemma states that if t has $\mathsf{d\ell T}$ type τ with weight K, and $t \succ_i t'$, then t' also has type τ, but with weight $K \dot{-} i$. The following lemma is one of interesting cases of subject reduction:

Lemma 4.6 (Subject reduction, case iter)**.** *If $\phi;\Phi;\emptyset \vdash_M \mathsf{iter}\, t_1\, t_2\, \underline{1+n} : \rho$, then there exists an index term M^* such that $\phi;\Phi \vdash_{M^*} t_1\, (\mathsf{iter}\, t_1\, t_2\, \underline{n}) : \rho$ and $\phi;\Phi \models 1 + M^* \leq M$.*

Proof. We first invert the typing of the application and the constant:

$$\begin{aligned}
&\phi;\Phi;\emptyset \vdash_{M'} \mathsf{iter}\, t_1\, t_2 : [b < 1] \cdot (\mathsf{Nat}[1+k] \multimap \rho') \\
&\phi;\Phi;\emptyset \vdash_{M_3} \underline{1+k} : \mathsf{Nat}[1+k] \\
&\quad \phi;\Phi \vdash \rho'\{0/b\} \sqsubseteq \rho \\
&\quad \phi;\Phi \models M' + M_3 \leq M
\end{aligned}$$

Now we invert the typing of the iteration, and we get:

$$a, b, \phi; b < 1, a < 1 + k, \Phi; \emptyset \vdash_{M_1} t_1 : [c < 1] \cdot (\sigma \multimap \sigma\{1 + a/a\}) \tag{4.1}$$

$$b, \phi; b < 1, \Phi; \emptyset \vdash_{M_2} t_2 : \sigma\{0/a\} \tag{4.2}$$

$$\phi;\Phi \models 1 + \sum_{b<1}(1 + k + \sum_{a<1+k} M_1 + M_2) \leq M'$$

$$\phi;\Phi \vdash \sigma\{0/c, 1 + k/a\} \sqsubseteq \rho'$$

By *weakening* the constraint $a < 1 + k$ to $a < k$, we get:

$$a, b, \phi; b < 1; a < k, \Phi; \emptyset \vdash_{M_1\{k/a\}} t_1 : [c < 1] \cdot (\sigma \multimap \sigma\{1 + a/a\})$$

Together with (4.2), we can then type:

$$\frac{\phi;\Phi;\emptyset \vdash_{1+\sum_{b<1}(k+\sum_{a<k} M_1+M_2)} \mathsf{iter}\, t_1\, t_2 : [b < 1] \cdot (\mathsf{Nat}[k] \multimap \sigma\{0/c, k/a\}) \qquad \phi;\Phi;\emptyset \vdash_{M_3} \underline{k} : \mathsf{Nat}[k]}{\phi;\Phi;\emptyset \vdash_{1+\sum_{b<1}(1+(\sum_{a<k} M_1)+M_2)+M_3} \mathsf{iter}\, t_1\, t_2\, \underline{k} : \sigma\{k/a, 0/b, 0/c\}}$$

We can also substitute k for the index variable a and 0 for b in (4.1), and we get:

$$\phi; \cancel{0 < 1}, \cancel{k < 1 + k}, \Phi; \emptyset \vdash_{M_1\{k/a,0/b\}} t_1 : [c < 1] \cdot (\sigma\{k/a, 0/b\} \multimap \sigma\{1 + k/a, 0/b\})$$

From this, we remove the constraints $0 < 1$ and $k < 1 + k$, since they are tautologies.

Finally, we apply the rule App:

$$\begin{aligned}
&\phi;\Phi;\emptyset \vdash_{M^* := 1+\sum_{b<1}(k+(\sum_{a<k} M_1)+M_2)+M_3+M_1\{k/b\}} \\
&\qquad\qquad t_1\, (\mathsf{iter}\, t_1\, t_2\, \underline{k}) : \sigma\{k + 1/a, 0/b, 0/c\} \sqsubseteq \rho'\{0/b\} \sqsubseteq \rho
\end{aligned}$$

It is easy to show that $\phi;\Phi \models M^* + 1 \leq M$. □

Relative completeness says that every program t that terminates in k steps to $\underline{n}$ can be assigned the type $\mathsf{Nat}[n]$ and weight k.

Theorem 4.7 (Relative completeness of $\mathsf{d\ell T}$ for programs)**.** *Let* $\emptyset \vdash t : \mathsf{Nat}$ *be a simply typed System T term, and assume* $t \Downarrow_k \underline{n}$*. Then we can type* $\emptyset; \emptyset; \emptyset \vdash_k t : \mathsf{Nat}[n]$*.*

In the proof of relative completeness, we build a typing, and thus have to show a set of assertions. These assertions are all *true*, but we have to assume that the *theory of index terms* is strong enough that we can prove these obligations *inside* this theory. We also need to extend the index term language with a certain operator (*findSlot*). Thus, we say that $\mathsf{d\ell T}$ is *relatively* complete.

We omit the proofs of the theorems in this chapter. We will discuss more general proofs in the next two chapters. In particular, $\mathsf{d\ell T}$ can be embedded in $\mathsf{d\ell PCF_v}$ without changing the weight.

4.6 Typing example

In this section, we give two example typings that build on top of each other. We first type the addition function and then we use instances of that typing to type the multiplication function. Recall that since in $\mathsf{d\ell T}$ (and also in $\mathsf{d\ell PCF_v}$) we have to know the arguments to a function, we abstract over these arguments by introducing index variables. In particular, we derive a typing for *add* that can be applied to any constant by simply instantiating the index variables.

4.6.1 Addition

First, we will give a typing for $add := \lambda x.\, \mathsf{iter}\, s\, x$ with $s := \lambda y.\, \mathsf{Succ}(y)$. We will give a typing that provides only one *instance*, i.e. a typing that allows one application to one argument. This suffices for typing multiplication. However, this is in contrast to the following example, where the typing that we construct is not general enough:

$$(\lambda f.\, f\ (f\ \underline{1}\ \underline{2})\ \underline{3})\ add$$

We only need to abstract over the arguments by introducing index variables c and d. Thus, the type that we want to assign to *add* is the following:

$$[b' < 1] \cdot (\mathsf{Nat}[c] \multimap [b < 1] \cdot (\mathsf{Nat}[d] \multimap \mathsf{Nat}[c + d]))$$

First, we type the iteration using the rule ITER with the parameters $\sigma := \mathsf{Nat}[c + a]$, $K := 1$, $M_1 := 1$, $M_2 := 0$, and $I := d$. This means, there will be one 'instance' of the iteration, which consists of d loops.

$$\frac{\begin{array}{c} a, b, c, d; b < 1, a < d; x : \mathsf{Nat}[c] \vdash_1 s : [\_ < 1] \cdot \mathsf{Nat}[c + a] \multimap \mathsf{Nat}[c + a + 1] \\ b, c, d; b < 1; x : \mathsf{Nat}[c] \vdash_0 x : \sigma\{0/a\} \end{array}}{c, d; \emptyset; x : \mathsf{Nat}[c] \vdash_{1+2d} \mathsf{iter}\, s\, x : [b < 1] \cdot \mathsf{Nat}[d] \multimap \mathsf{Nat}[c + d]}$$

The final subtypings hold, since $1 + \sum_{b<1}(d + (\sum_{b<d} 1) + 0) = 1 + 2b$ and $\sigma\{d/a\} = \mathsf{Nat}[c + d]$. Now, we can type *add* using Lam. For this, we have to add the fresh index variable b' with the constraint $b' < 1$ to the above typing of $\mathsf{iter}\, s\, x$.

$$\frac{b', c, d; b' < 1; \emptyset \vdash_{1+2d} \mathsf{iter}\, s\, x : [b < 1] \cdot \mathsf{Nat}[d] \multimap \mathsf{Nat}[c + d]}{c, d; \emptyset; \emptyset \vdash_{2+2d} \mathit{add} : [b' < 1] \cdot \mathsf{Nat}[c] \multimap [b < 1] \cdot \mathsf{Nat}[d] \multimap \mathsf{Nat}[c + d]}$$

The above weight already accounts for the cascaded application with two arguments. Therefore, given two constants m and n, we can substitute m for c and n for d. Then we can derive $\vdash_{2+2n} \mathit{add}\, \underline{m}\, \underline{n}$ by using the application rule twice. Therefore, $2 + 2n$ is an upper bound on the cost of the application. Moreover, since the typing is *precise*, it can also be shown that $2 + 2n$ is a tight bound. We will discuss precise typings in the next chapter.

4.6.2 Multiplication

Recall the definition $\mathit{mult} := \lambda x.\, \mathsf{iter}\, \mathit{add}\, x\, \underline{0}$. Again, we introduce two new index variables c and d. We have to type $\mathsf{iter}\, \mathit{add}\, x\, \underline{0}$ in the context $x : \mathsf{Nat}[c]$. As before, we choose the parameters $K := 1$, $I := d$, and $M_2 := 0$. Moreover, we choose $\sigma := \mathsf{Nat}[ac]$ and $M_1 := ac$. To type $\mathit{add}\, x$, we simply have to substitute ac for d in the above typing of *add* and use App once.

$$\frac{\begin{array}{c} a, b, c, d; b < 1, a < d; x : \mathsf{Nat}[c] \vdash_{2+2ac} \mathit{add}\, x : [\_ < 1] \cdot \mathsf{Nat}[ac] \multimap \mathsf{Nat}[(a + 1)c] \\ b, c, d; b < 1; x : \mathsf{Nat}[c] \vdash_0 \underline{0} : \sigma\{0/a\} \end{array}}{c, d; \emptyset; x : \mathsf{Nat}[c] \vdash_{1+3d+cd^2-cd} \mathsf{iter}\, \mathit{add}\, x\, \underline{0} : [b' < 1] \cdot \mathsf{Nat}[d] \multimap \mathsf{Nat}[cd]}$$

The weight can be justified using easy arithmetic:

$$1 + \sum_{b<1}(d + (\sum_{a<d}(2 + 2ac)) + 0) = 1 + d + 2d + 2c \sum_{a<d} a = 1 + 3d + cd^2 - cd$$

Finally, introducing the λ-abstraction over x increments the weight once more, and we derive:

$$c, d; \emptyset; \emptyset \vdash_{2+3d+cd^2-cd} \mathit{mult} : [b' < 1] \cdot \mathsf{Nat}[c] \multimap [b < 1] \cdot \mathsf{Nat}[d] \multimap \mathsf{Nat}[cd]$$

4.7 Related work

A system called dℓT was first introduced in [3]. However, the type system there is *not complete* (w.r.t. System T), since it does not feature bounded exponentials ($[a < I]$), and they also do not refine the type of natural numbers. Their variant of System T has similar operational semantics as our variant. However, they also consider *side effects* (global store), and their type systems tracks the set of locations that the program will read.

Our extension of dℓT is strongly inspired by dℓPCF$_\mathsf{v}$ [12], which is the subject of the next chapter. In fact, the only difference between dℓT and dℓPCF$_\mathsf{v}$ is that we replace iteration with unbounded recursion. We will show that the typing rule ITER can be recovered as an admissible typing rule in dℓPCF$_\mathsf{v}$ by treating iteration as syntactic sugar.

Support for non-defined index terms is not present in [11, 12]. On its own, this is not needed for dℓT (since all simply typed terms terminate). However, supporting diverging PCF terms will come very handy in the type inference algorithm that we will discuss in Chapter 8.

Chapter 5

Review of $\mathsf{d\ell PCF_v}$

In this chapter we discuss $\mathsf{d\ell PCF_v}$, an extension of $\mathsf{d\ell T}$ from the above chapter targeting the call-by-value variant of PCF.

The system $\mathsf{d\ell PCF_v}$ was first published in [12]. The main contribution of this chapter is that we simplify the soundness and completeness proofs. We give the first spelled-out account of completeness (some parts of it are in Appendix A). Also, most of these results have been formally verified in Coq, see Appendix B. In [12], a stack machine is introduced, which is an overhead in the proofs. We show that soundness and completeness can also be shown using small-step operational semantics. Furthermore, we show that our version of $\mathsf{d\ell T}$ can be embedded in $\mathsf{d\ell PCF_v}$. Using our refined semantics of index terms from Section 4.2, we can also type diverging programs.

5.1 Forest Cardinality

The difference between $\mathsf{d\ell T}$ from the previous chapter and $\mathsf{d\ell PCF_v}$ is that $\mathsf{d\ell PCF_v}$ features unrestricted recursion. Since $\mathsf{d\ell PCF_v}$ is an affine type system, we can bound how often the function variable f in the body of a fixpoint $\mu f x.\, t$ can be applied. We count the nodes of the *recursion forest* of the function in pre-order depth-first traversal order, and we encode this forest using an index term I. Consider the b^{th} node in this forest: There, f can be (recursively) applied I times. Thus, I (with b as free variable) denotes the number of children of the b^{th} node in the recursion forest.

Forest cardinality is an operator on index terms that counts the size of the first K trees of the forest described by an index term I. Of course, it is only defined if I actually describes a finite forest. We extend our index term language $\mathcal{L}^{\ell}_{idx}$ with an operator $\triangle_a^K I$ that has the following semantics:

Definition 5.1 (Forest cardinality)**.** Let K and I be index terms; b may occur free in I

but not in K. We define the operator $\triangle_a^K I$ with the following two equations:

$$
\begin{aligned}
&[\![\overset{0}{\underset{b}{\triangle}} I]\!] = 0 \\
&[\![\overset{1+K}{\underset{b}{\triangle}} I]\!] = 1 + h + [\![\overset{K}{\underset{b}{\triangle}} I\{h+b/b\}]\!] \qquad \text{if } [\![\overset{I\{0/b\}}{\underset{b}{\triangle}} I\{1+b/b\}]\!] = h
\end{aligned}
$$

Note that forest cardinality is only partially defined. In particular, $[\![\triangle_b^1 1]\!] = \bot$.

The first equation means that the empty forest has size 0. In the second line, we want to compute the cardinality of $1 + K$ trees. For this, we recursively compute the size of children of the first tree, and then compute the size of the next K trees.[1]

We will make use of certain operations on forests, like splitting and merging. All of these operations can of course be defined using the index term descriptions. For example, in the following lemma, we state that we can split a forest into two forests:

Fact 5.2 (Forest Splitting)**.** *Let $\phi; \Phi \vDash H \equiv \triangle_b^{K_1+K_2} I$ be defined. Then there exist index terms H_1, H_2, such that:*

- $\phi; \Phi \vDash H_1 \equiv \triangle_b^{K_1} I$,
- $\phi; \Phi \vDash H_2 \equiv \triangle_b^{K_2} I\{H_1 + b/b\}$, *and*
- $\phi; \Phi \vDash H \equiv H_1 + H_2$.

Proof. If we see the index terms as ordinary numbers and functions of our meta theory, this statement can be proved by induction on the value of K_1. □

Fact 5.3 ((Non)empty forest)**.** *Let $\phi; \Phi \vDash H \equiv \triangle_b^K I$. If $\phi; \Phi \vDash 0 < K$, then $\phi; \Phi \vDash 0 < H$. Furthermore, if $\phi; \Phi \vDash 0 \equiv H$, then $\phi; \Phi \vDash 0 \equiv K$.*

The following lemma formalises the intuitive fact that the a^{th} child of node number b is in the same forest.

Fact 5.4. *Let $a < I$ and $b < H := \triangle_b^K I$. Then $b + H' < H$ with $H' := \triangle_c^a I\{1 + b + c/b\}$.*

Proof. By induction on the definition of H, for an arbitrary b.

- Case $K = 0$. We have $H = 0$, which contradicts $b < H$.
- Case $1 + K$. This means that $H = 1 + H_1 + H_2$ with $H_1 := \triangle_b^{I\{0/b\}} I\{1 + b/b\}$ and $H_2 := \triangle_b^K I\{1 + H_1 + b/b\}$. (In other words, H_1 is the cardinality of the children of the first tree in the forest, and H_2 is the cardinality of the remaining K trees in the forest.) By the inductive hypotheses, we can assume that the property holds for H_1 and H_2. Case analysis on b:

---

[1] This definition differs slightly from the definition in [11, 12]. We only changed the order in which nodes are counted, to make some inductive proofs over forest cardinality easier. Also, they have an additional shifting parameter: $\triangle_a^{J,K} I := \triangle_a^K I\{a + J/a\}$.

- Case $b = 0$. This means, b is the very first node in the forest, and $b + H'$ is the a^{th} child node of this forest (with $a < I\{0/b\}$). We have to show: $0 + H' < H_1 + H_2$. We can write $I\{0/b\} = a + (I\{0/a\} - a)$. Therefore, we can split the cardinality H_1 using Fact 5.2: We have $H_1 = H' + H_3$ with $H_3 := \triangle_b^{I\{0/b\}-a} I\{1 + H' + b/b\}$. It remains to show $H' < H' + H_2 + H_3$. This holds, since $0 < H_3$ and $I\{0/b\} - a > 0$.
- Case $0 < b \leq H_1$. This means that b is in one of the child tress in the first forest. The goal follows from the first inductive hypothesis with $K := I\{0/b\}$, $b := b - 1$, $I := I\{1 + b/b\}$.
- Case $0 < b$ and $H_1 < b$: b is not in the first tree. The goal follows from the second inductive hypothesis with $K := K$, $I := I\{1 + H_1 + b/b\}$ and $b := b - 1 - H_1$. □

5.2 Typing Rules

The types of dℓPCF$_\mathsf{v}$ are exactly the same as the types of dℓT. We also use the same definitions of subtyping, binary and bounded modal sums, as well as subtyping. dℓPCF$_\mathsf{v}$ also has the typing rules as dℓT (depicted in Figure 4.2), except that the fixpoint rule is substituted for the iteration rule. The fixpoint rule, which is depicted in Figure 5.1, deserves an explanation. Recall that $\mu f x.\, t$ is syntactic sugar for $\mu f.\, \lambda x.\, t$.

The index term I (with b as free variable) describes the *recursion forest* for the K main applications of the fixpoint.[2] This means, the b^{th} (self)application recursively calls the function again I-times.

As an abbreviation, we introduce an index term H with the assertion $\phi; \Phi \vDash H \equiv \triangle_b^K$. Thus, H denotes the total size of the recursion forests (which consists of K trees). If we want to enforce that all index terms are terminating (as in [12]), we would have to add the assertion $\phi; \Phi \vDash H \downarrow$ as a premise. Note that the function recursively calls itself $H - K$ times, and the fixpoint can (potentially) be called K times.

In the first hypothesis of the rule, we type the underlying abstraction $\lambda x.\, t$ for each of the $b < H$ applications. Here we may call f I-times, hence $f : [a < I] \cdot A$ is included in the context (note that A may have a and b as free variables). Each of these applications is always only used once (i.e. in the next recursive call or in the main application), so the type of $\lambda x.\, t$ is $[a < 1] \cdot B$, for some type B.

The types A and B have the same PCF shape, but they may have different index terms. A describes the type of f in the I child nodes, and B is the typing at the node b. In the second line, we formalise an 'invariant' between the types A and B: The index term $1 + b + (\triangle_c^a I\{1 + b + c/b\})$ can be informally described as the node number of the a^{th} child of node number b. Using the first line, we have already shown that node number b can be typed with B *if* all the children can be typed with A. In the second line, we show that all children can also be typed with A, because B (for the a^{th} child of b) is a subtype of the corresponding A. Summarised, the first two hypotheses say that if all the

---

[2] The recursion forest can be infinite, but only finitely branching trees and forests can be encoded.

$$\textsc{Fix}\quad \dfrac{\begin{array}{c} b, \phi; b < H, \Phi; f : [a < I] \cdot A, \Delta \vdash_J \lambda x.\, t : [a < 1] \cdot B \\ a, b, \phi; a < I, b < H, \Phi \vdash B\{0/a, 1 + b + \left(\mathop{\triangle}\limits_{c}^{a} I\{1 + b + c/b\}\right)/b\} \sqsubseteq A \\ \phi; \Phi \vDash H \equiv \mathop{\triangle}\limits_{b}^{K} I \qquad I,\ J, \text{ and } \Delta \text{ may have } b \text{ free (but } \textit{not } a\text{)} \\ A \text{ and } B \text{ may have } a \textit{ and } b \text{ free} \qquad \text{the rest has } \textit{neither } a \textit{ nor } b \text{ free} \end{array}}{\Phi; \sum_{b<H} \Delta \vdash_{\sum_{b<H} J} \mu f x.\, t : [a < K] \cdot B\{0/a, \mathop{\triangle}\limits_{b}^{a} I/b\}}$$

Figure 5.1: The fixpoint typing rule of dℓPCF$_\mathsf{v}$. All other rules are as in Figure 4.2. (The rule ITER is not present in dℓPCF$_\mathsf{v}$.)

leaf nodes can be typed with B, then – by induction on the forest – all K root nodes of the forest can also be typed with B.

The final weight of the fixed point is just the sum of the weights of all typings. Finally, we 'export' the typings B of the K roots of the forest.

Changes from [12] In contrast to the original presentation in [12], we do not add H in the weight of the fixpoint. This is because we use slightly different semantics in which the fixpoint application $(\mu f x.\, t)v \succ_1 t\{\mu f x.\, t/f, v/x\}$ takes only one step instead of two. The cost of this step is already accounted for in the rule LAM, because $(\lambda x.\, t\{\mu f x.\, t/f\})\, v$ makes the same step.

5.3 Soundness

The following lemma (or admissible typing) rule is crucial in the meta theory of dℓPCF$_\mathsf{v}$. It holds for all variants of dℓPCF (and corresponding lemmas also hold for dfPCF in Part II), and states that we can instantiate an index variable a with an index term I. Note that I does not need to be closed itself; it could introduce additional free variables (or re-introduce the variable a).

Lemma 5.5 (Index term substitution)**.** *Let $a, \phi_1; \Phi; \Gamma \vdash_M t : \tau$ be a typing and let I be an index term closed in ϕ_2. Then we can substitute I for a to derive a typing $\phi_1, \phi_2; \Phi\{I/a\}; \Gamma\{I/a\} \vdash_{M\{I/a\}} t : \tau\{I/a\}$. Moreover, this operation preserves the structure of the typing.*

We show soundness of dℓPCF$_\mathsf{v}$ using subject reduction on the small-step semantics. Unlike in [12], we show that the weight of a typing reduces by one for every β-substitution step. From this, we will conclude that the weight is an upper bound on the *cost* of an execution of a closed program. This also entails that if the weight is a terminating index term, the term also terminates.

One of the key lemmas of the soundness proof are the *splitting lemmas.* The binary splitting lemma says that a $\mathsf{d\ell PCF_v}$ typings of values can be split into two parts:

Lemma 5.6 (Binary splitting)**.** *Assume a typing* $\phi; \Phi; \emptyset \vdash_M v : \rho_1 \uplus \rho_2$*. From this, we can derive two typings* $\phi; \Phi; \emptyset \vdash_{M_i} v : \rho_i$ *(for* $i = 1, 2$*) with* $\phi; \Phi \vDash M_1 + M_2 \leq M$*.*

There is also a 'parametric' version of this lemma for bounded modal sums:

Lemma 5.7 (Parametric splitting)**.** *Assume the typing* $\phi; \Phi; \emptyset \vdash_M v : \sum_{c<J} \rho$*, where the index variable* $c \notin \phi$ *may appear free in* ρ*. Then we can derive a typing* $c, \phi; c < J, \Phi; \emptyset \vdash_N v : \rho$ *with* $\phi; \Phi \vDash \sum_{c<J} N \leq M$*.*

We will show similar lemmas in Chapter 7. Proofs of these lemmas are also outlined in [12].

Key to subject reduction is *substitution.* It is a corollary of the splitting lemmas:

Lemma 5.8 (Substitution)**.** *Let* $\phi; \Phi; x : \sigma_x, \Gamma \vdash_{M_1} t : \rho$ *and* $\phi; \Phi; \emptyset \vdash_{M_2} v : \sigma_x$*, where* v *is a closed value. Then* $\phi; \Phi; \Gamma \vdash_{M_1+M_2} t\{v/x\} : \rho$*.*

Proof. By induction on the typing of t.

- Case $t = \underline{n}$. Trivial, since the term is closed.
- Case $t = y$ and hence $\phi; \Phi \vdash (x : \sigma_x, \Gamma)(y) \sqsubseteq \rho$. If $x = y$, then $(x : \sigma_x, \Gamma)(y) = \sigma_x \sqsubseteq \rho$ and thus $\phi; \Phi; \Gamma \vdash_{M_2} v : \rho$. Otherwise, $(x : \sigma_x, \Gamma)(y) = \Gamma(y) \sqsubseteq \rho$, and thus $\phi; \Phi; \Gamma \vdash_{M_1} y : \rho$.
- Case $t = t_1\, t_2$; we have:

$$\phi; \Phi; \Delta_1 \vdash_{K_1} t_1 : [a < 1] \cdot (\sigma \multimap \tau) \qquad \phi; \Phi; \Delta_2 \vdash_{K_2} t_2 : \sigma\{0/a\}$$

$$\phi; \Phi \vdash \tau\{0/a\} \sqsubseteq \rho \qquad \phi; \Phi \vDash K_1 + K_2 \leq M_1 \qquad \phi; \Phi \vdash x : \sigma_x, \Gamma \sqsubseteq \Delta_1 \uplus \Delta_2$$

  We split off the type of x in the contexts: $\Delta_i = \Delta_i(x), \Delta'_i$ (for $i = 1, 2$); and we have $\Delta_1 \uplus \Delta_2 = x : (\Delta_1(x) \uplus \Delta_2(x)), \Delta'_1 \uplus \Delta'_2$. This means that, by subsumption, v can be typed as $\phi; \Phi; \emptyset \vdash_{M_2} v : \Delta_1(x) \uplus \Delta_2(x)$. We split this typing (using Lemma 5.6), and we obtain two typings of v:

$$\phi; \Phi; \emptyset \vdash_{M_{21}} v : \Delta_1(x) \qquad \phi; \Phi; \emptyset \vdash_{M_{22}} v : \Delta_2(x) \qquad \phi; \Phi \vDash M_{21} + M_{22} \leq M_2$$

  Using inductive hypotheses on the typings of t_i and the i^{th} typing of v, we can type:

$$\frac{\begin{array}{c}\phi; \Phi; \Delta'_1 \vdash_{K_1+M_{21}} t_1\{v/x\} : [a < 1] \cdot (\sigma \multimap \tau) \qquad \phi; \Phi; \Delta'_2 \vdash_{K_2+M_{22}} t_2\{v/x\} : \sigma\{0/a\} \\ \phi; \Phi \vdash \Gamma \sqsubseteq \Delta'_1 \uplus \Delta'_2 \qquad \phi; \Phi \vDash (K_1 + M_{21}) + (K_2 + M_{22}) \leq M_1 + M_2\end{array}}{\phi; \Phi; \Gamma \vdash_{M_1+M_2} (t_1\, t_2)\{v/x\} : \rho}$$

- Case $\lambda y.\, t$, where $x \neq y$; we have:

$$a, \phi; a < I, \Phi; y : \sigma, \Delta \vdash_K t : \tau \qquad \phi; \Phi \vdash x : \sigma_x, \Gamma \sqsubseteq \textstyle\sum_{a<I} \Delta$$

$$\phi; \Phi \vDash I + \textstyle\sum_{a<I} K \leq M_1 \qquad \phi; \Phi \vdash [a < I] \cdot (\sigma \multimap \tau) \sqsubseteq \rho$$

Similar to above, we split Δ into $\Delta(x), \Delta'$, and using subsumption, we have: $\phi; \Phi; \emptyset \vdash_{M_2} v : \sum_{a<I} \Delta(x)$. Using parametric splitting (Lemma 5.7), we get:

$$a, \phi; a < I, \Phi; \emptyset \vdash_{M_2'} v : \Delta(x) \qquad\qquad \phi; \Phi \vDash \textstyle\sum_{a<I} M_2' \leq M_2$$

The inductive hypothesis yields $a, \phi; a < I, \Phi; y : \sigma, \Gamma \vdash_{M_1 + M_2'} t\{v/x\} : \tau$. From this, the goal follows from the typing rule Lam (in Figure 4.2).

- Case $\mu f y.\, t$. Similarly to the above case (also with parametric splitting).
- Case $t = \mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3$; we have:

$$\phi; \Phi; \Delta_1 \vdash_{K_1} t_1 : \mathsf{Nat}[J] \qquad \phi; J \gtrsim 0, \Phi; \Delta_2 \vdash_{K_2} t_2 : \rho \qquad \phi; 0 < J, \Phi; \Delta_2 \vdash_{K_2} t_3 : \rho$$

$$\phi; \Phi \vdash x : \sigma_x, \Gamma \sqsubseteq \Delta_1 \uplus \Delta_2 \qquad\qquad \phi; \Phi \vDash K_1 + K_2 \leq M_1$$

  Similarly to the application case, we split Δ_i and get two typings for v. We use the inductive hypothesis on the typing of t_1 and the first typing of v. We also use the inductive hypotheses of t_2 and t_3 with the second typing of v. Then, the goal follows from Ifz.

- Cases $\mathsf{Succ}(t)$ and $\mathsf{Pred}(t)$. Follows from the inductive hypothesis and the respective typing rule. ☐

Subject reduction states that the weight decreases after every β-substitution step:

Theorem 5.9 (Subject reduction of $\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$)**.** *Let $\phi; \Phi; \emptyset \vdash_M t : \rho$, and let $t \succ_i t'$ be a step. Then there exists an index term M' such that $\phi; \Phi; \emptyset \vdash_{M'} t' : \rho$ and $\phi; \Phi \vDash i + M' \leq M$.*

Proof (sketch). By induction on the small step. The context reduction cases are trivial. We outline the interesting head reduction cases in the lemmas below. ☐

Lemma 5.10 (Subject reduction, case λ-application)**.** *Let $\phi; \Phi; \emptyset \vdash_M (\lambda x.\, t)\, v : \rho$. Then there exists an index term M' such that $\phi; \Phi; \emptyset \vdash_{M'} t\{v/x\} : \rho$ and $\phi; \Phi \vDash 1 + M' \leq M$.*

Proof. By inversion, we get:

$$a, \phi; a < I, \Phi; x : \sigma \vdash_{M_1} t : \tau \qquad \phi; \Phi; \emptyset \vdash_{M_2} v : \sigma\{0/a\} \qquad \phi; \Phi \vDash 1 \leq I$$

$$\phi; \Phi \vdash \tau\{0/a\} \sqsubseteq \rho \qquad\qquad \phi; \Phi \vDash (I + \sum_{a<I} M_1) + M_2 \leq M$$

We can substitute 0 for a in the first typing and remove the constraint $0 < I$. With the substitution lemma (Lemma 5.8), we can type:

$$\phi; \Phi; \emptyset \vdash_{M' := M_1\{0/a\} + M_2} t\{v/x\} : \tau\{0/a\} \sqsubseteq \rho$$

Finally, it is easy to see that this weight is less than M. ☐

Lemma 5.11 (Subject reduction, case fixpoint application)**.** *Let* $\phi; \Phi; \emptyset \vdash_M (\mu f x.\, t)\, v : \rho$*. Then there exists an index term* M' *such that* $\phi; \Phi; \emptyset \vdash_{M'} t\{\mu f x.\, t/f, v/x\} : \rho$ *and* $\phi; \Phi \vDash 1 + M' \leq M$*.*

Proof (sketch). Part of the proof can be reduced to the previous case: Since

$$(\lambda x.\, t\{\mu f x.\, t/x\})\, v \succ_1 t\{\mu f x.\, t/f, v/x\}$$

it suffices to show that the left term has type ρ. By inverting the typing of the fixpoint application, we get:

$$\phi; \Phi; \emptyset \vdash_{K_1} \mu f x.\, t : [a < 1] \cdot (\sigma \multimap \tau) \qquad \phi; \Phi; \emptyset \vdash_{K_2} v : \sigma\{0/a\}$$

$$\phi; \Phi \vdash \tau\{0/a\} \sqsubseteq \rho \qquad \phi; \Phi \vDash K_1 + K_2 \leq M$$

Thus, we already have a typing for v and it suffices to show: $\phi; \Phi; \emptyset \vdash_{K_1} \lambda x.\, t\{\mu f x.\, t/x\} : [a < 1] \cdot (\sigma \multimap \tau)$. We can now forget everything about v, and we proceed in the following lemma. ☐

Lemma 5.12 (Subject reduction, case fixpoint application, auxiliary)**.** *If* $\phi; \Phi; \emptyset \vdash_M \mu f x.\, t : [a < 1] \cdot (\sigma \multimap \tau)$*, then* $\phi; \Phi; \emptyset \vdash_M \lambda x.\, t\{\mu f x.\, t/f\} : [a < 1] \cdot (\sigma \multimap \tau)$*.*

Proof (sketch). Inverting the fixpoint typing yields a recursion forest I consisting of at least one tree:

$$b, \phi; b < H, \Phi; f : [a < I] \cdot A \vdash_J \lambda x.\, t : [a < 1] \cdot B \tag{5.1}$$

$$a, b, \phi; a < I, b < H, \Phi \vdash B\{0/a, 1 + b + \left(\underset{c}{\overset{a}{\triangle}} I\{1 + b + c/b\}\right)/b\} \sqsubseteq A \tag{5.2}$$

$$\phi; \Phi \vdash [a < K] \cdot B\{0/a, \underset{b}{\overset{a}{\triangle}} I/b\} \sqsubseteq [a < 1] \cdot (\sigma \multimap \tau) \tag{5.3}$$

with $\phi; \Phi \vDash H \equiv \triangle_b^K I$ and $\phi; \Phi \vDash \sum_{b<H} J \leq M$. Substituting 0 for b in (5.1) and (5.2), yields:

$$\phi; \Phi; f : [a < I\{0/b\}] \cdot A\{0/b\} \vdash_{J\{0/b\}} \lambda x.\, t : [a < 1] \cdot B\{0/b\} \tag{5.4}$$

$$a, \phi; a < I\{0/b\}, \Phi \vdash B\{0/a, 1 + 0 + \left(\underset{c}{\overset{a}{\triangle}} I\{1 + c/b\}\right)/b\} \sqsubseteq A\{0/b\} \tag{5.5}$$

With the substitution lemma and (5.4), it suffices to show:

$$\phi; \Phi; \emptyset \vdash_{M^*} \mu f x.\, t : [a < I\{0/a\}] \cdot A\{0/a\}$$

We apply Fix with $I^* := I\{1 + b/b\}$, $M^* := \sum_{a<H^*} J\{1 + b/b\}$, $K^* := I\{0/b\}$, $H^* := \triangle_b^{K^*} I^*$, $A^* := A\{1 + b/b\}$, and $B^* := B\{1 + b/b\}$. Visually, these parameters means that we throw away all trees except for the first tree, and we 'chop off' the root node of that tree.

We have to show the following typing and subtyping judgements:

$$b, \phi; b < H^*, \Phi; f : [a < I^*] \cdot A^* \vdash_J \lambda x.\, t : [a < 1] \cdot B^*$$

$$a, b, \phi; a < I^*, b < H^*, \Phi \vdash B^*\{0/a, 1 + b + \left(\overset{a}{\underset{c}{\triangle}} I^*\{1 + b + c/b\} \right) /b\} \sqsubseteq A^*$$

They follow by substituting $1 + b$ for b in (5.1) and (5.2). Finally, we have to show:

$$\phi; \Phi \vdash [a < K^*] \cdot B^*\{0/a, \overset{a}{\underset{b}{\triangle}} I/b\} \sqsubseteq [a < I\{0/b\}] \cdot A\{0/b\}$$

This follows by inverting (5.3) and substituting 0 for a. □

We have similar, easy cases for the head reduction cases, for example:[3]

Lemma 5.13. *Let* $\phi; \Phi; \emptyset \vdash_M \mathsf{ifz}\, \underline{0}\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2 : \rho$. *Then* $\phi; \Phi; \emptyset \vdash_{M'} t_1 : \rho$.

Proof. By inversion of the typing. We get $\phi; \Phi \vdash \underline{0} : \mathsf{Nat}[J]$ and hence $\phi; \Phi \vDash 0 = J$. Therefore, we can remove the true constraint $J = 0$ from the resulting typing of t_1. □

Now that we have proved Theorem 5.9, it is easy to show termination of closed $\mathsf{d\ell PCF_v}$ terms.[4] To prove this, we first define a size function on terms:

Definition 5.14 (Size of terms)**.**

$$|x| := 1 \qquad |\lambda x.\, t| := 1 + |t|$$

$$|\underline{n}| := 1 \qquad |\mu f x.\, t| := 1 + |t|$$

$$|t_1\, t_2| := 1 + |t_1| + |t_2| \qquad |\mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3| := 1 + |t_1| + |t_2| + |t_3|$$

Lemma 5.15 (Soundness of $\mathsf{d\ell PCF_v}$)**.** *Let* $\emptyset; \emptyset; \emptyset \vdash_k t : \tau$. *Then there exists a value* v *and a number* k'*, such that* $t \Downarrow_{k'} v$ *and* $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{k-k'} v : \tau$.

Proof. We prove the lemma by well-founded induction on the lexicographical order of k and the size of t. If t is a value, we are done. Otherwise, let $t \succ_i t'$ be the first step of t.[5] Using Theorem 5.9, we get an index term M' such that $\emptyset; \emptyset \vDash M' + i \leq k$ and $\emptyset; \emptyset; \emptyset \vdash_{M'} t' : \tau$. Since M' must also be closed and defined, we can write it as a constant $k' := [\![M']\!]$ (that is, $\vDash M' \equiv k'$) and type $\emptyset; \emptyset; \emptyset \vdash_{k'} t' : \tau$. Now, we do a case distinction on the cost i of the step. If $i = 1$ (that is, the step was a β-substitution), we can apply the inductive hypothesis on t' since $k' - i < k$. Otherwise ($i = 0$), we know that the size of t' is smaller than the size of t, so we can also apply the inductive hypothesis on t'. □

---

[3] In [12], only a lemma like Lemma 5.12 (but with decreasing weight) would be needed, since their semantics for fixpoint is defined by the rule $(\mu x.\, t)\, v \succ_1 t\{\mu x.\, t/x\}\, v$, but they do not prove this lemma. Unrelated to that, they also do not restrict fixpoints to have two binders. So t could be any term that evaluates to a function.

[4] In [12], the proof of this theorem is much more complicated. Instead of using the small-step semantics, the authors define a stack-based closure machine, lift typings to machine configurations, and prove subject reduction on configurations.

[5] By the progress lemma of PCF (see Lemma 2.8), it is guaranteed that such a step exists (and can be computed).

Corollary 5.16 (Soundness of $\mathsf{d\ell PCF_v}$ programs)**.** *Theorem 4.5 (1) holds for* $\mathsf{d\ell PCF_v}$*: Let* $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{k} t : \mathsf{Nat}[I]$ *be a* $\mathsf{d\ell PCF_v}$ *typing. Then there is a* $k' \leq k$ *and a constant* n *such that* $t \Downarrow_{k'} \underline{n}$ *and* $\vDash n \sqsubseteq I$*. In particular, if* $\vDash I \equiv m$*, then* $m = n$*.*

5.4 Tight bounds and precise typings

As claimed in Chapter 3, we have shown that the (static) weight of a typing of a closed term is an upper bound on its (dynamic) cost. From the perspective of BLL, this holds since only those *resources* can be *consumed* (i.e. in applications) that have been *allocated* before (i.e. in λ-abstractions and fixpoints). However, the subsumption rule allows us to increase the weight arbitrarily. In a *precise* (or *linear*) typing, we disallow wasting of resources. For example, in the following typing, we allocate 42 resources, which are then pushed into the context but never used:

$$\dfrac{\dfrac{\dfrac{}{\phi; \Phi; x : [b < 42] \cdot \cdots \vdash_0 \underline{0} : \mathsf{Nat}[0]}}{\phi; \Phi; \emptyset \vdash_1 (\lambda x.\, \underline{0}) : [a < 1] \cdot (([b < 42] \cdot \cdots) \multimap \mathsf{Nat}[0])} \qquad \dfrac{}{\phi; \Phi; \emptyset \vdash_{42} \lambda x.\, x : [b < 42] \cdot (\mathsf{Nat}[b] \multimap \mathsf{Nat}[b])}}{\phi; \Phi; \emptyset \vdash_{43} (\lambda x.\, \underline{0})\, (\lambda x.\, x) : \mathsf{Nat}[0]}$$

To ensure that exactly that many resources are allocated as are actually consumed, we restrict subsumption to $\equiv$ (and $=$) instead of $\sqsubseteq$ (and $\leq$). Furthermore, the contexts of closed terms must be empty or consist of *disposable* types, i.e. those types that do not carry any resources. Formally, we define:[6]

Definition 5.17 (Disposable types)**.** The following (modal) types are disposable:

- *ground types*, i.e. $\mathsf{Nat}[I]$, and
- modal types with bound zero ($[a < 0] \cdot A$).

A context is disposable if it only assigns disposable types to variables. In particular, $\emptyset$ is disposable.

Definition 5.18 (Precise typing)**.** A typing $\phi; \Phi; \Gamma \vdash_M t : \tau$ is precise, if:

- In all uses of subsumption, $\equiv$ is used instead of $\sqsubseteq$ and $\leq$,
- The rules for variables and constants are changed such that unused types in the contexts are disposable:

$$\dfrac{\Gamma \text{ disposable}}{\phi; \Phi; \Gamma \vdash_0 \underline{n} : \mathsf{Nat}[n]} \qquad \dfrac{\Gamma \text{ disposable}}{\phi; \Phi; x : \sigma, \Gamma \vdash_0 x : \sigma}$$

---

[6] In [11, 12], the second restriction on precise typings is not made. Subject reduction does not hold for their definition of precise typings. The above example is a counter-example: We cannot assign the weight $43 - 1$ to the successor term $\underline{0}$. However, this weaker definition suffices to show completeness.

We can now show that the weight of a precise typing of a closed program is a tight bound on its execution cost. For this, we need to re-prove the substitution and subject reduction lemmas (Lemma 5.8 and Theorem 5.9, respectively). In particular, we want to show that subject reduction preserves precision. Note that in the substitution case $\underline{n}\{v/x\}$, the weight of the typing should be $M_1 + M_2$ (where $\phi; \Phi \vDash M_1 \equiv 0$), but the overall weight must be 0. Thus, we have to show the following lemma, which implies that $\phi; \Phi \vDash M_2 \equiv 0$ since σ_x is disposable.

Lemma 5.19. *For a precise typing* $\phi; \Phi; \Gamma \vdash_M v : \tau$ *where* τ *is disposable, we have* $\phi; \Phi \vDash M \equiv 0$.

Proof. By induction (or case analysis) on the precise value typing. ☐

Theorem 5.20 (Precise subject reduction of $\mathsf{d\ell PCF_v}$)**.** *Let* $\phi; \Phi; \emptyset \vdash_M t : \rho$ *be a precise typing, and let* $t \succ_i t'$ *be a step. Then there exists an index term* M' *such that* $\phi; \Phi; \emptyset \vdash_{M'} t' : \rho$ *and* $\phi; \Phi \vDash i + M' \equiv M$.

Corollary 5.21 (Precise soundness of $\mathsf{d\ell PCF_v}$)**.** *Let* $\emptyset; \emptyset; \emptyset \vdash_K t : \tau$ *be a precise typing and* $t \Downarrow_k v$. *Then* $\emptyset; \emptyset; \emptyset \vdash_{K \dot{-} k} v : \tau$ *is a precise typing.*

Corollary 5.22 (Precise soundness of $\mathsf{d\ell PCF_v}$)**.** *Let* $\emptyset; \emptyset; \emptyset \vdash_K t : \tau$ *be a precise typing and* $t \Downarrow_k v$*, and let* τ *be disposable. Then* $\vDash K \equiv k$ *and* $\emptyset; \emptyset; \emptyset \vdash_0 v : \tau$.

Corollary 5.23 (Precise soundness of $\mathsf{d\ell PCF_v}$ for programs)**.** *Theorem 4.5 (2) holds for* $\mathsf{d\ell PCF_v}$*: Let* $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_K t : \mathsf{Nat}[I]$ *be a precise typing and* $t \Downarrow_k \underline{n}$. *Then* $\vDash K \equiv k$ *and* $\vDash I \equiv n$.

The above corollaries entail that the weight and Nat-refinements of terminating programs must be defined. Thus, it is sound to add the constraint $\phi; \Phi \vDash K_1 \downarrow$ to the rule App (in Figure 4.2), where K_1 is the weight of t_1, but only for precise typings: If t_1 diverges, the application $t_1\, t_2$ also diverges, and thus t_2 does not need to be typed. Similarly, in the rule Ifz, we can add the constraint $J \equiv 0$ to the typing of t_2 instead of $0 \gtrsim J$.

5.5 Completeness

Completeness can be proved by means of *subject expansion.* The key lemmas will be the *joining lemmas* and *converse substitution.*

Subject expansion roughly states:

> Let t be a simply typed PCF term, and let $t \succ t'$. Furthermore, assume a $\mathsf{d\ell PCF_v}$ typing $\phi; \Phi; \emptyset \vdash_M t' : \tau$. Then we can show $\phi; \Phi; \emptyset \vdash_M t : \tau$.

However, this does *not* hold in general! Consider the following counter-example:

$$(\lambda x.\, x\, \underline{0} + x\, (\lambda y.\, \underline{0})\, \underline{1})\, (\lambda z.\, z) \succ (\lambda z.\, z)\, \underline{0} + (\lambda z.\, z)\, (\lambda y.\, \underline{0})\, \underline{1}$$

Clearly, the successor term t' can be typed in $\mathsf{d\ell PCF_v}$; however, it is not possible to type t: We would have to type $\lambda z.\, z$ with a type that has the shape $\mathsf{Nat} \to \mathsf{Nat}$ *and* also has the shape $(\mathsf{Nat} \to \mathsf{Nat}) \to (\mathsf{Nat} \to \mathsf{Nat})$. As t is not even typeable in PCF, it is also not typeable in $\mathsf{d\ell PCF_v}$.

To overcome this problem, we make some restrictions on the backward step and the $\mathsf{d\ell PCF_v}$ typing of t'. Intuitively, we only allow successor terms t' that are the result of applying subject reduction on a simply typed term t. The *skeleton* (shape) of the $\mathsf{d\ell PCF_v}$ typing of t' must be exactly such a shape. In the next section, we formally define skeletons of typings.[7]

5.5.1 PCF skeletons

Skeletons of PCF or $\mathsf{d\ell PCF}$ typings are a data structure that describes all *structural choices* that can be made in a typing derivation. There is only one such choice, namely the type τ_1 in the PCF application typing rule:

$$\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1\, t_2 : B}$$

Definition 5.24 (Skeletons)**.** Skeletons are labelled trees, where each node is labelled by the name of a PCF typing rule. For the rule App, we additionally store the type τ_1.

$$s ::= \mathsf{Var} \mid \mathsf{Const} \mid \mathsf{Succ}\, s \mid \mathsf{Pred}\, s \mid \mathsf{Lam}\, s \mid \mathsf{Fix}\, s \mid \mathsf{Ifz}\, s_1\, s_2\, s_3 \mid \mathsf{App}\, A\, s_1\, s_2$$

We define the skeleton of PCF typings in the obvious way. We write $\Gamma \vdash t : \tau \mathbin{@} s$ for a (simple) PCF typing with skeleton s.

Fact 5.25. *Two* PCF *typings* $\Gamma \vdash t : \tau$ *are* equal *if and only if their skeletons are equal.*

A similar lemma for $\mathsf{d\ell PCF_v}$ – the *joining lemma* – will be the subject of the next section.

PCF subject reduction and skeletons

We can extend the step relation $t \succ_i t'$ to pairs of terms and skeletons: $(t; s) \succ_i (t'; s')$ says that the closed term t steps to t', and if t has a typing with skeleton s, then t' has a typing with skeleton s'. See Figure 5.2 for the reduction rules.

We abbreviate $(t; s_1)\{(v; s_2)/x\} := (t\{v/x\}; \mathit{subst}(x; t; s_1; s_2))$, where $\mathit{subst}(x; t; s_1; s_2)$ returns a new skeleton where we substitute s_2 for all Var in s_1 that correspond to x in t:

Definition 5.26 (Term and skeleton substitution)**.**

$$\begin{aligned}
\mathit{subst}(x; y; \mathsf{Var}; s_2) &:= \begin{cases} s_2 & x = y \\ \mathsf{Var} & x \neq y \end{cases} \\
\mathit{subst}(x; \underline{n}; s_1; s_2) &:= s_1 \\
\mathit{subst}(x; \mathsf{Succ}(t); \mathsf{Succ}\, s_1; s_2) &:= \mathsf{Succ}\, (\mathit{subst}(x; t; s_1; s_2))
\end{aligned}$$

---

[7] The idea of typing skeletons comes from [11, 12]. However, they do not define skeletons as inductive data types. Our explicit definition of skeletons was very helpful in our Coq formalisation of $\mathsf{d\ell PCF_v}$.

$$(\mathsf{Succ}(\underline{n}); \mathsf{Succ\,Const}) \succ_0 (\underline{1+n}; \mathsf{Const}) \qquad (\mathsf{Pred}(\underline{n}); \mathsf{Pred\,Const}) \succ_0 (\underline{n \dot{-} 1}; \mathsf{Const})$$

$$\frac{(t; s) \succ_i (t'; s')}{(\mathsf{Succ}(t); \mathsf{Succ}\, s) \succ_i (\mathsf{Succ}(t'); \mathsf{Succ}\, s')} \qquad \frac{(t; s) \succ_i (t'; s')}{(\mathsf{Pred}(t); \mathsf{Pred}\, s) \succ_i (\mathsf{Pred}(t'); \mathsf{Pred}\, s')}$$

$$(\mathsf{ifz}\, \underline{0}\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3; \mathsf{Ifz\,Const}\, s_2\, s_3) \succ_0 (t_2; s_2) \qquad (\mathsf{ifz}\, \underline{1+n}\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3; \mathsf{Ifz\,Const}\, s_2\, s_3) \succ_0 (t_3; s_3)$$

$$\frac{(t_1; s_1) \succ_i (t_1'; s_1')}{(\mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3; \mathsf{Ifz}\, s_1\, s_2\, s_3) \succ_i (\mathsf{ifz}\, t_1'\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3; \mathsf{Ifz}\, s_1'\, s_2\, s_3)}$$

$$((\lambda x.\, t)\, v; \mathsf{App}\, A\, (\mathsf{Lam}\, s_1)\, s_2) \succ_1 (t; s_1)\{(v; s_2)/x\}$$

$$\frac{((\lambda x.\, t)\, v; \mathsf{Lam}\, s_1)\{(\mu f x.\, t; \mathsf{Fix}\, (\mathsf{Lam}\, s_1))/f\} \succ_1 (t'; s')}{((\mu f x.\, t)\, v; \mathsf{App}\, A\, (\mathsf{Fix}\, (\mathsf{Lam}\, s_1))\, s_2) \succ_1 (t'; s')}$$

$$\frac{(t_1; s_1) \succ_i (t_1'; s_1')}{(t_1\, t_2; \mathsf{App}\, A\, s_1\, s_2) \succ_i (t_1'\, t_2; \mathsf{App}\, A\, s_1'\, s_2)} \qquad \frac{(t_2; s_2) \succ_i (t_2'; s_2')}{(v_1\, t_2; \mathsf{App}\, A\, s_1\, s_2) \succ_i (v_1\, t_2'; \mathsf{App}\, A\, s_1\, s_2')}$$

Figure 5.2: Small-step reduction rules with skeletons

$$\begin{aligned}
\mathit{subst}(x; \mathsf{Pred}(t); \mathsf{Pred}\, s_1; s_2) &:= \mathsf{Pred}\, (\mathit{subst}(x; t; s_1; s_2)) \\
\mathit{subst}(x; t_1\, t_2; \mathsf{App}\, A\, s_1\, s_1'; s_2) &:= \mathsf{App}\, A\, (\mathit{subst}(x; t; s_1; s_2))\, (\mathit{subst}(x; t; s_1'; s_2)) \\
\mathit{subst}(x; \mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3; \mathsf{Ifz}\, s_1\, s_2\, s_3; s) &:= \mathsf{Ifz}\, (\mathit{subst}(x; t_1; s_1; s))\, (\mathit{subst}(x; t_2; s_2; s)) \\
&\qquad (\mathit{subst}(x; t_3; s_3; s))
\end{aligned}$$

We can extend the standard PCF subject reduction proof with skeletons:

Lemma 5.27 (PCF substitution with skeletons)**.** *Let* $x : A, \Gamma \vdash t : B@s_1$ *and* $\emptyset \vdash v : A@s_2$*. Then* $\Gamma \vdash t\{v/x\} : B \mathbin{@} \mathit{subst}(x; t; s_1; s_2)$*.*

Lemma 5.28 (PCF subject reduction with skeletons)**.** *If* $\Gamma \vdash t : B \mathbin{@} s$ *for a closed term* t*, and* $(t; s) \succ (t'; s')$*, then* $\Gamma \vdash t' : B \mathbin{@} s'$*.*

In the completeness proof, as in [12], we only produce precise typings.[8] We write $\phi; \Phi; \Gamma \vdash_M t : \tau \mathbin{@} s$ for a precise typing with skeleton s.

Precise typing are needed in the joining lemmas, since the following fact does not hold for non-precise subtypings:

Fact 5.29 (Sums commute over type equivalence)**.** *Let* $\rho_1 = \sigma_1 \uplus \sigma_2$ *and* $\rho_2 = \tau_1 \uplus \tau_2$*. If* $\Phi \vdash \sigma_i \equiv \tau_i$ *for* $i = 1, 2$*, then* $\Phi \vdash \rho_1 \equiv \rho_2$*.*

It is easy lift the index term substitution lemma (Lemma 5.8) to precise typings with skeletons. We will also see that inversion of precise typings gets slightly easier.

---

[8] For completeness, it would suffice to only restrict subsumption to equivalences, as in [12].

5.5.2 The explosion typing rule

The following lemma states that we can always construct a $\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$ typing given a simple typing, if the constraint is unsatisfiable. As a corollary, we can type every simply typed PCF value with a 'trivial' type.

Lemma 5.30 (Explosion subtyping rule)**.** *Let* $(\!|\sigma|\!) = (\!|\tau|\!)$ *and let* Φ *be contradictory, i.e.* $\phi; \Phi \vDash \bot$*. Then* $\phi; \Phi \vdash \sigma \sqsubseteq \tau$*. (By symmetry we also have* $\phi; \bot \vdash \sigma \equiv \tau$*.) The same holds for linear types* $(\!|A|\!) = (\!|B|\!)$*.*

Proof (sketch). By induction on the shape of the two types. All semantic entailments $\phi; \Phi \vDash \cdots$ follow by *ex falso quodlibet.* ☐

Lemma 5.31 (Explosion typing rule)**.** *Let* $Ctx \vdash t : A \mathbin{@} s$ *be a simple typing, and let* $(\!|\tau|\!) = A$*. Furthermore, let* $\phi; \Phi \vDash \bot$*. Then we have* $\phi; \Phi; \Gamma \vdash_M t : \tau \mathbin{@} s$*.*

Proof (sketch). By induction on the simple typing. All subtyping obligations are discharged by Lemma 5.30.

In the fixpoint case, we choose $K = 0$ and hence $H = 0$. The typing obligation $b, \phi; b < H, \Phi; x : [a < I] \cdot A, \Delta \vdash_J \lambda x.\, t : [a < 1] \cdot B$ follows by induction, since the constraint contains (even two) contradictions. A and B are arbitrary linear types with the right shape; J is arbitrary.

All other cases are similar. ☐

Lemma 5.32 (Trivial typings for values)**.** *Let* $\hat{\Gamma}$ *be a simple context and let* $\hat{\Gamma} \vdash v : A \mathbin{@} s$ *be a simple* PCF *typing. Then we can construct a 'trivial' precise* $\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$ *typing* $\phi; \Phi; \Gamma \vdash_0 v : \tau \mathbin{@} s$ *with* $(\!|\Gamma|\!) = \hat{\Gamma}$ *and* $(\!|\tau|\!) = A$*.*

Proof (sketch). Case distinction on the value v.

- Case $v = \underline{n}$. Use rule Const (with an arbitrarily context).
- Case $v = \lambda x.\, t$. Let $(\!|\sigma \multimap \tau|\!) = A$ be arbitrarily chosen, such that a is a fresh index variable in $\sigma \multimap \tau$. We type v as $[a < 0] \cdot (\sigma \multimap \tau)$ using rule Lam and Lemma 5.31.
- Case $v = \mu f x.\, t$. As above. We define an empty recursion tree (I arbitrary, $K = 0$ and thus $H = 0$). ☐

5.5.3 Creating (bounded) sums

In the 'joining lemmas', which are essential for the converse substitution, we need to construct binary and bounded sums. Recall that there are syntactic restrictions in the definition of sums. For example, $\mathsf{Nat}[I_1] \uplus \mathsf{Nat}[I_2]$ is only defined if $I_1 = I_2$. For quantified types, the second linear component must be equal to the first but shifted by the first bound. For quantified types, it is easy though to define types that are not equal but equivalent ($\equiv$).[9]

---

[9] These lemmas are mentioned in [11] (see *Lemma 5.1* there). However, the 'proof' given there is wrong. They also make the unnecessary assumption that the language of index terms $\mathcal{L}^{\ell}_{idx}$ is *universal* in some

Creating binary sums

To create binary sums of quantified types, we need a 'case distinction' operation on types. We always assume that the two types have the same PCF structure.

Definition 5.33 (Case distinction for types)**.** Let C be a constraint and assume that the (modal/linear) types in the *if* and *else* branches below have the same shape. We define:

$$\begin{aligned}\mathsf{if}\,C\,\mathsf{then}\,\sigma_1 \multimap \tau_1\,\mathsf{else}\,\sigma_2 \multimap \tau_2 &:= (\mathsf{if}\,C\,\mathsf{then}\,\sigma_1\,\mathsf{else}\,\sigma_2) \multimap (\mathsf{if}\,C\,\mathsf{then}\,\tau_1\,\mathsf{else}\,\tau_2)\\ \mathsf{if}\,C\,\mathsf{then}\,\mathsf{Nat}[I_1]\,\mathsf{else}\,\mathsf{Nat}[I_2] &:= \mathsf{Nat}[\mathsf{if}\,C\,\mathsf{then}\,I_1\,\mathsf{else}\,I_2]\\ \mathsf{if}\,C\,\mathsf{then}\,[a < I_1]\cdot A_1\,\mathsf{else}\,[a < I_2]\cdot A_2 &:= [a < \mathsf{if}\,C\,\mathsf{then}\,I_1\,\mathsf{else}\,I_2]\cdot(\mathsf{if}\,C\,\mathsf{then}\,A_1\,\mathsf{else}\,A_2)\end{aligned}$$

Lemma 5.34 (Correctness of case distinction)**.** *The equivalence* $\phi; I < J \vdash \mathsf{if}\,I < J\,\mathsf{then}\,\tau_1\,\mathsf{else}\,\tau_2 \equiv \tau_1$ *holds. Also, the converse holds for* $J \leq I$*.*

Lemma 5.35 (Type case distinction and substitution)**.** *For every index substitution* θ*, the following equation holds:* $(\mathsf{if}\,C\,\mathsf{then}\,\tau_1\,\mathsf{else}\,\tau_2)\theta = \mathsf{if}\,C\theta\,\mathsf{then}\,\tau_1\theta\,\mathsf{else}\,\tau_2\theta$*.*

Lemma 5.36 (Creating binary sums of quantified types)**.** *Let* $\tau_1 = [a < I_1]\cdot A$ *and* $\tau_2 = [a < I_2]\cdot B$ *be types with* $(\!|\tau_1|\!) = (\!|\tau_2|\!)$*. Then we can define types* ρ_1 *and* ρ_2 *such that* $\phi;\emptyset \vdash \tau_1 \equiv \rho_1$*,* $\phi;\emptyset \vdash \tau_2 \equiv \rho_2$*, and* $\rho_1 \uplus \rho_2$ *is defined.*

Proof. Define $C := \mathsf{if}\,a < I_1\,\mathsf{then}\,A\,\mathsf{else}\,B\{a - I_1/a\}$, and:

$$\begin{aligned}\rho_1 &:= [a < I_1]\cdot C\\ \rho_2 &:= [a < I_2]\cdot C\{a + I_1/a\}\\ \rho_3 &:= [a < I_1 + I_2]\cdot C\end{aligned}$$

The syntactic restriction on the sum $\rho_1 \uplus \rho_2 = \rho_3$ holds trivially. Note that A and C are not equal, but they are equivalent under the assumption $a < I_1$ (with Lemma 5.34). □

We can 'construct' a sum of $\mathsf{Nat}[I_1]$ and $\mathsf{Nat}[I_2]$ in a trivial way if we assume that the index terms are equivalent:

Lemma 5.37 (Creating binary sums of Nat types)**.** *Let* $\tau_i = \mathsf{Nat}[I_i]$ *and* $\phi;\Phi \vDash I_1 = I_2$*. Then we can define types* ρ_1*,* ρ_2 *such that* $\phi;\Phi \vdash \tau_i \equiv \rho_i$*, and* $\rho_1 \uplus \rho_2$ *is defined.*

Proof. Choose $\rho_1 = \rho_2 = \rho_3 = \mathsf{Nat}[I_1]$. Then clearly $\rho_1 \uplus \rho_2 = \rho_3$ and $\phi;\Phi \vdash \tau_i \equiv \rho_i$. □

Creating bounded sums

When creating bounded sums, and in the parametric joining lemma, we often need to 'decompose' a sum $c = b + \sum_{c<a} J$ into the 'index' a and the 'offset' $b < J\{a/c\}$. This can be done using a primitive recursive function, assuming that a is bounded:

---

sense. We only need to extend $\mathcal{L}^{\ell}_{idx}$ with a primitive recursive function *findSlot*. For completeness of natural functions, though, we need to extend $\mathcal{L}^{\ell}_{idx}$, which we will formalise in Section 5.5.8.

Definition 5.38 (Sum decomposition)**.** Let $I : \mathsf{Nat}$ and $J : \mathsf{Nat} \to \mathsf{Nat}$ be a function. We define the higher-order function $\mathit{findSlot}\, I\, J : \mathsf{Nat} \to \mathsf{Nat} \times \mathsf{Nat}$:

$$\begin{array}{ll} \mathit{findSlot}\,(1+I)\,J\,x := (0,x) & \text{if } x < J(0) \\ \mathit{findSlot}\,(1+I)\,J\,x := (1+a,b) & \text{ow. and } (a,b) = \mathit{findSlot}\,I\,(\lambda n.\, J(1+n))\,(x - J(0)) \end{array}$$

Note that the function $\mathit{findSlot}\, I\, J$ is only partially defined.

Lemma 5.39 (Correctness of sum decomposition)**.** *Let f^{-1} denote $\mathit{findSlot}\, I\, J$. Then the following propositions hold:*

1. $\forall a\, b.\, a < I \wedge b < J(a) \wedge f^{-1}\left(b + \sum_{d<a} J(d)\right) = (a', b') \Longrightarrow a = a' \wedge b = b'$
2. $\forall a\, b\, c.\, c < \sum_{d<I} J(d) \wedge f^{-1}(c) = (a,b) \Longrightarrow c = \left(b + \sum_{d<a} J(d)\right) \wedge a < I \wedge b < J(a)$

Proof. Both propositions can be proved by induction on I. Note that the assumptions always imply that $0 < I$, and thus f^{-1} is defined. □

We extend our language of index terms with the two operators $\pi_i(\mathit{findSlot}_a\, I\, J\, c)$ (with $i = 1, 2$). This notation makes explicit that c is a free variable of the operator and a is the free variable of J. For example, we could implement the operators using the following defining equations:

$$\pi_1(\mathit{findSlot}_a\, I\, J\, c) =$$
$$\begin{cases} \mathsf{if}\ c < J\{0/a\}\ \mathsf{then}\ 0\ \mathsf{else}\ 1 + \pi_1(\mathit{findSlot}_a\,(I \mathbin{\dot-} 1)\,(J\{1+a/a\})\,(c \mathbin{\dot-} J\{0/a\})) & \text{if } I > 0 \\ \bot & \text{if } I = 0 \end{cases}$$
$$\pi_2(\mathit{findSlot}_a\, I\, J\, c) = c \mathbin{\dot-} \textstyle\sum_{b < \pi_1(\mathit{findSlot}_a\, I\, J\, c)} J$$

Now, we use this operator to create bounded sums of quantified types.

Lemma 5.40 (Creating bounded sums of quantified types)**.** *Let $\sigma = [b < J] \cdot A$, where a is free in J. Let I be another index term (closed in ϕ). Then we can define a type σ', such that $a < I \vdash \sigma' \equiv \sigma$ and $\sum_{a<I} \sigma'$ is defined.*

Proof. Let $f^{-1} := \mathit{findSlot}\, I\, J$; then we define:

$$\begin{aligned} A' &:= A\{\pi_1(f^{-1}(c))/a, \pi_2(f^{-1}(c))/b\} \\ \sigma' &:= [b < J] \cdot A'\{b + \sum_{d<a} J\{d/a\}/c\} \\ \sum_{a<I} \sigma' &= [c < \sum_{a<I} J] \cdot A' \end{aligned}$$

We have $a < I \vdash \sigma' \equiv \sigma$, which follows from Lemma 5.39 (1). □

5.5.4 Joining lemmas

In the App and Ifz cases of the proof of converse substitutions, the inductive hypotheses will yield two typings of a closed value v with the same skeleton. The joining lemma says that we can 'join' the typings; the new type is the binary sum of the two types.

One key lemma for the binary joining lemma is a case distinction lemma:

Lemma 5.41 (Case distinction typing lemma)**.** *Let C be a constraint. Let $\Phi_i; \Gamma_i \vdash_{M_i} t : \tau_i \mathbin{@} s$ be two typings ($i = 1, 2$). Assume that the* PCF *structures of τ_i and $\Gamma_i(x)$ (for all variables x in the domain of Γ_1 and Γ_2) are equal. Then we can construct a typing for:*

$$\mathsf{if}\, C\, \mathsf{then}\, \Phi_1\, \mathsf{else}\, \Phi_2; \mathsf{if}\, C\, \mathsf{then}\, \Gamma_1\, \mathsf{else}\, \Gamma_2 \vdash_{\mathsf{if}\, C\, \mathsf{then}\, M_1\, \mathsf{else}\, M_2} t : \mathsf{if}\, C\, \mathsf{then}\, \tau_1\, \mathsf{else}\, \tau_2 \mathbin{@} s$$

The same holds for subtyping judgements.

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

We can refine this lemma to a form that is directly useful for the joining lemma:

Corollary 5.42 (Refined case distinction typing lemma)**.** *Let $a, \phi; a < I_1, \Phi; \Gamma_1 \vdash_{M_1} t : \rho \mathbin{@} s$ and $a, \phi; a < I_2, \Phi; \Gamma_2 \vdash_{M_2} t : \rho\{a + I_1/a\} \mathbin{@} s$. Then:*

$$a, \phi; a < I_1 + I_2, \Phi; \mathsf{if}\, a < I_1\, \mathsf{then}\, \Gamma_1\, \mathsf{else}\, \Gamma_2\{a - I_1/a\} \vdash_{\mathsf{if}\, a<I_1\, \mathsf{then}\, M_1\, \mathsf{else}\, M_2\{a-I_1/a\}} t : \rho \mathbin{@} s$$

Lemma 5.43 (Joining)**.** *Let v be a closed value. Given two typings $\phi; \Phi; \emptyset \vdash_{M_i} v : \tau_i \mathbin{@} s$ with the same skeleton ($i = 1, 2$), we can define types $\tau = \tau'_1 \uplus \tau'_2$ with $\phi; \Phi \vdash \tau'_i \equiv \tau_i$, and derive a typing $\phi; \Phi; \emptyset \vdash_{M_1+M_2} v : \tau \mathbin{@} s$.*

Proof. Case distinction on v.

- Case $v = \underline{n}$. Then $\tau_i = \mathsf{Nat}[I_i]$ with $\phi; \Phi; \emptyset \vDash I_1 = I_2$. Let $\tau = \mathsf{Nat}[I_1] \uplus \mathsf{Nat}[I_1]$. We can type $\phi; \Phi; \emptyset \vdash_0 \underline{n} : \tau$.
- Case $v = \lambda x.\, t$. We invert both typings ($i = 1, 2$):

$$\frac{a, \phi; a < I_i, \Phi; x : \sigma_i \vdash_{K_i} t : \tau}{\phi; \Phi; \emptyset \vdash_{M_i := I_i + \sum_{a<I_i} K_i} \lambda x.\, t : \rho_i = [a < I_i] \cdot (\sigma_i \multimap \tau_i)}$$

  The goal follows from Corollary 5.42 and the rule Lam.

- Case $v = \mu f x.\, t$. The two typings yield two recursion forests described by I_i containing K_i trees and consisting of H_i nodes each. We define a new recursion forest I^* of size $K_1 + K_2$ by:

$$I^* := \mathsf{if}\, b < H_1\, \mathsf{then}\, I_1\, \mathsf{else}\, I_2\{b - H_1/b\}$$

  Obviously, the size of the new recursion tree is $H^* = \triangle_b^{K_1+K_2} I^* = \triangle_b^{K_1} I_1 + \triangle_b^{K_2} I_2 = H_1 + H_2$. We apply the rule Fix; the typing and subtyping goals follow by case-distinction on $b < H_1$ using Corollary 5.42, as in the λ case. □

Parametric joining is a generalisation of joining, were we assume L typings and build a bounded sum:

Lemma 5.44 (Parametric joining)**.** *Let* $c, \phi; c < L, \Phi; \emptyset \vdash_M v : \rho$. *Then there exists a* ρ' *with* $c, \phi; c < L, \Phi \vdash \rho \equiv \rho'$ *and* $\phi; \Phi; \emptyset \vdash_{\sum_{c<L} M} v : \sum_{c<L} \rho'$ *(with the same skeleton).*

The proof of the above lemma can be found in Appendix A.1.1. The fixpoint case is complicated, because we also need to join recursion forests. The corresponding (parametric) joining proofs for the call-by-push-value variant of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ in Chapter 7 will be much simpler because we do not need to join recursion forests there.

5.5.5 Converse substitution

Converse substitution is the converse of the substitution lemma (Lemma 5.8). In general, converse substitution states that if $t\{v/x\}$ has type τ, then we can type $\vdash v : \sigma$ and $x : \sigma \vdash t : \tau$. However, this does not hold for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$: Consider again the counter-example from the beginning of this section:

$$(x\,\underline{0} + x\,(\lambda y.\,\underline{0})\,\underline{0})\{(\lambda z.\,z)/x\}$$

Here, the identity function would have to be typed with a types of the shapes $\mathsf{Nat} \to \mathsf{Nat}$ and $(\mathsf{Nat} \to \mathsf{Nat}) \to (\mathsf{Nat} \to \mathsf{Nat})$. However, we can only join two typings if the typings that have the same skeletons and compatible types. Therefore, we must assume that v only needs to be typed with *one* PCF skeleton, which is formalised below.

Lemma 5.45 (Converse substitution)**.** *Let* v *be a closed value. Assume the simple* PCF *typings* $x : A_x, (\!|\Gamma|\!) \vdash t : (\!|\rho|\!) \mathbin{@} s_1$ *and* $\emptyset \vdash v : A_x \mathbin{@} s_2$ *for a closed value* v. *Furthermore, assume the* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ *typing* $\phi; \Phi; \Gamma \vdash_M t\{v/x\} : \rho \mathbin{@} s'$, *where* $s' = \mathit{subst}(x; t; s_1; s_2)$, *as defined in Definition 5.26. Then there exist index terms* N_1 *and* N_2, *and a type* σ, *such that:*

$$\phi; \Phi; x : \sigma, \Gamma \vdash_{N_1} t : \rho \mathbin{@} s_1 \qquad \phi; \Phi; \emptyset \vdash_{N_2} v : \sigma \mathbin{@} s_2 \qquad \phi; \Phi \vDash N_1 + N_2 \equiv M \qquad (\!|\sigma|\!) = A_x$$

Proof (sketch). By size-induction on t. In every case of the induction, we can assume without loss of generality that x is a free variable of t, and thus $\Gamma(x)$ is defined. Otherwise, by assumption we can type $t\{v/x\} = t$, and we type v with a trivial type (Lemma 5.32). We now make a case distinction on t.

- Case $t = \underline{n}$. Contradicts the assumption that x is a free variable of t.
- Case $t = x$, $t\{v/x\} = v$, and $s_1 = \mathsf{Var}$, $s' = s_2$. Then we can type $\phi; \Phi; x : \rho \vdash_0 x : \rho \mathbin{@} s_1$ and $\phi; \Phi; \emptyset \vdash_M v : \rho \mathbin{@} s_2$.
- Case $t = t_1\,t_2$. By inversion on the $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ typing, we have:

$$\begin{aligned}
&\phi; \Phi; \Delta_1 \vdash_{K_1} t_1\{v/x\} : [a < 1] \cdot (\sigma \multimap \tau) \mathbin{@} \mathit{subst}(x; t_1; s_{11}; s_2)\\
&\phi; \Phi; \Delta_2 \vdash_{K_2} t_2\{v/x\} : \sigma\{0/a\} \mathbin{@} \mathit{subst}(x; t_2; s_{12}; s_2)\\
&\quad \phi; \Phi \vDash K_1 + K_2 \equiv M\\
&\quad \phi; \Phi \vdash \Delta_1 \uplus \Delta_2 \equiv \Gamma
\end{aligned}$$

with $s_1 = \mathsf{App}\,\llparenthesis\sigma\rrparenthesis\, s_{11}\, s_{12}$, and $\rho = \tau\{0/a\}$. The inductive hypotheses yield typings for t_1 and t_2, and two typings for v:

$$\begin{array}{lll} \phi;\Phi;x:\sigma_1,\Delta_1 \vdash_{N_{11}} t_1 : [a<1]\cdot(\sigma \multimap \tau) \mathbin{@} s_{11} & \phi;\Phi;\emptyset \vdash_{N_{12}} v : \sigma_1 \mathbin{@} s_2 & \phi;\Phi \vDash N_{11}+N_{12} \equiv K_1 \\ \phi;\Phi;x:\sigma_2,\Delta_2 \vdash_{N_{21}} t_2 : \sigma\{0/a\} \mathbin{@} s_{12} & \phi;\Phi;\emptyset \vdash_{N_{22}} v : \sigma_2 \mathbin{@} s_2 & \phi;\Phi \vDash N_{21}+N_{22} \equiv K_2 \end{array}$$

We can join the two value typings using Lemma 5.43 since they have the same skeleton s_2.[10] Applying the joining lemma yields types σ'_i equivalent to σ_i (for $i = 1, 2$) and a typing $\phi;\Phi;\emptyset \vdash_{N_{12}+N_{22}} v : \sigma'_1 \uplus \sigma'_2 \mathbin{@} s_2$. Now, we can type $\phi;\Phi;x : \sigma'_1 \uplus \sigma'_2, \Delta_1 \uplus \Delta_2 \vdash_{N_{11}+N_{21}} t_1\, t_2 : \tau\{0/a\} \mathbin{@} \mathsf{App}\,\llparenthesis\sigma\rrparenthesis\, s_{11}\, s_{12}$.

- Case $t = \lambda y.\, t'$. We have:

  $$a,\phi;a<I,\Phi;y:\sigma,\Delta \vdash_K t\{v/x\} : \tau \qquad \phi;\Phi \vdash [a<I]\cdot(\sigma \multimap \tau) \equiv \rho$$

  $$\phi;\Phi \vDash I + \textstyle\sum_{a<I} K \equiv M \qquad \phi;\Phi \vdash \textstyle\sum_{a<I} \Delta \equiv \Gamma$$

  We apply the inductive hypothesis on the typing of $t\{v/x\}$ and get a type σ_x such that $a,\phi;a<I,\Phi;x:\sigma_x,y:\sigma,\Delta \vdash_{N_1} t : \tau$ and $a,\phi;a<I,\Phi;\emptyset \vdash_{N_2} v : \sigma_x$ with $a,\phi;a<I,\Phi \vDash N_1 + N_2 = K$. We apply parametric joining (Lemma 5.44) on this typing, and get:

  $$\phi;\Phi;\emptyset \vdash_{\sum_{a<I} N_2} v : \textstyle\sum_{a<I} \sigma' \qquad a,\phi;a<I,\Phi \vdash \sigma' \equiv \sigma_x$$

  Thus, we can type $\phi;\Phi;x : \sum_{a<I}\sigma', \sum_{a<I}\Delta \vdash_{I+\sum_{N_1}} \lambda y.\, t : [a<I]\cdot(\sigma \multimap \tau)$, and we have $\phi;\Phi \vDash (I + \sum_{a<I} N_1) + (\sum_{a<I} N_2) \equiv M$.

- The other cases are similar. □

5.5.6 Subject expansion

Lemma 5.46 (Subject expansion of $\mathsf{d\ell PCF_v}$)**.** *Let* $(t;s) \succ_i (t';s')$. *Assume a* PCF *typing* $\emptyset \vdash t : \llparenthesis\rho\rrparenthesis \mathbin{@} s$, *and a* $\mathsf{d\ell PCF_v}$ *typing* $\phi;\Phi;\emptyset \vdash_M t' : \rho \mathbin{@} s'$. *Then we can type* $\phi;\Phi;\emptyset \vdash_{i+M} t : \rho \mathbin{@} s$.

Proof (sketch). Induction on the small-step semantics. The only non-trivial cases are the two β-substitution cases (where the weight increases by one). The proof of these cases can be found in Appendix A.1 (see Lemmas A.7 and A.8). □

5.5.7 Completeness for programs

Completeness for PCF programs follows directly from subject expansion. This theorem states that all terminating PCF programs can be typed with the type $\mathsf{Nat}[n]$, where n is the result. The weight of the typing is exactly the number of β substitution steps.

Corollary 5.47 (Subject expansion, multiple steps)**.** *Let* $(t;s) \succ^*_k (t';s')$, *where* k *is the number of* β*-substitutions in the execution. Assume a* PCF *typing* $\emptyset \vdash t : \llparenthesis\rho\rrparenthesis$, *and a* $\mathsf{d\ell PCF_v}$ *typing* $\phi;\Phi;\emptyset \vdash_M t' : \rho \mathbin{@} s'$. *Then* $\phi;\Phi;\emptyset \vdash_{k+M} t : \rho \mathbin{@} s$.

---

[10] This is the crucial point why we needed to introduce skeletons!

Theorem 5.48 (Relative completeness of $\mathsf{d}\ell\mathsf{PCF_v}$ for programs)**.** *Let* $\emptyset \vdash t : \mathsf{Nat}$ *be a* PCF *program, and assume* $t \Downarrow_k \underline{n}$*. Then we can type* $\emptyset; \emptyset; \emptyset \vdash_k t : \mathsf{Nat}[n]$*.*

Proof. By assumption, we have $\emptyset \vdash t : \mathsf{Nat}$. Since t is terminating (say, after k β-substitutions), it terminates to a constant, which can be typed in $\mathsf{d}\ell\mathsf{PCF_v}$: $\emptyset; \emptyset; \emptyset \vdash_0 \underline{n} : \mathsf{Nat}[n]$. By the above corollary, we have $\emptyset; \emptyset; \emptyset \vdash_k t : \mathsf{Nat}[n]$. ☐

5.5.8 Completeness for natural functions

For *total natural functions* $f : \mathsf{Nat} \to \mathsf{Nat}$, it is shown in [12] that we can give a typing that abstracts over the value. This typing will have an index variable a free, and the type is $[c < 1] \cdot (\mathsf{Nat}[a] \multimap \mathsf{Nat}[f(a)])$. This means that this typing can be later *instantiated* by substituting an index term for c, as we did in our $\mathsf{d}\ell\mathsf{T}$ examples in Section 4.6.

In this subsection, we will prove the *uniformisation lemma*, as outlined in [12]. We will also discuss some applications of this lemma.

The uniformisation lemma states that if we can type a typing for all valuations of an index variable a, then we can construct a typing where a is free. This means, that we *uniformise* (in the terminology of [12]) *infinitely many typings* into one typing. Note that this is conceptually different from *joining*: Joining of infinitely many typings would result in a typing with infinite weight.

We first define uniformisation of index terms and types. We assume that $\mathcal{L}^{\ell}_{idx}$ has an operator $\mathit{unif}_c(\{I_n\}_n)$ that takes an *enumeration* of index terms and returns a new index term with c as a free variable, such that the following equation holds for all valuations ν:

$$\forall i : \mathsf{Nat}.\, [\![\mathit{unif}_c(\{I_n\}_n)]\!](c := i, \nu) = [\![I_i]\!](\nu)$$

This operator can be lifted to constraints, $\mathit{unif}_c(\{C_n\}_n)$, and constraint lists, $\mathit{unif}_c(\{\Phi_n\}_n)$. Similarly, we can 'uniformise' enumerations of types, subtypings, and ultimately typings.

Lemma 5.49 (Uniformisation of semantical constraints)**.** *If for all* n*,* $\phi; \Phi_n \vDash P_n$*, then* $c, \phi; \mathit{unif}_c(\{\Phi_n\}_n) \vDash \mathit{unif}_c(\{P_n\}_n)$*.*

Definition 5.50 (Uniformisation of types)**.** Let $\{\tau_n\}_n$ be an enumeration of modal types with the same PCF shape $\hat{A}$ (i.e. $(\!|\tau_n|\!) = \hat{A}$ for all n). Also, let $\{A_n\}_n$ be a similar enumeration of linear types. We define $\mathit{unif}_c(\hat{A}, \{\tau_n\}_n)$ and $\mathit{unif}_c(\hat{A}, \{A_n\}_n)$ by mutual induction on $\hat{A}$:

$$\begin{aligned} \mathit{unif}_c(\mathsf{Nat}, \{\mathsf{Nat}[I_n]\}_n) &:= \mathsf{Nat}[\mathit{unif}_c(\{I_n\}_n\,,)] \\ \mathit{unif}_c(\hat{A}, \{[a < I_n] \cdot A_n\}_n) &:= [a < \mathit{unif}_c(\{I_n\}_n)] \cdot (\mathit{unif}_c(\hat{A}, \{A_n\}_n)) \\ \mathit{unif}_c(\hat{A} \to \hat{B}, \{A_n \multimap B_n\}_n) &:= \mathit{unif}_c(\hat{A}, \{A_n\}_n) \multimap \mathit{unif}_c(\hat{B}, \{B_n\}_n) \end{aligned}$$

We write $\mathit{unif}_c(\{\tau_n\}_n)$ or simply $\mathit{unif}_c(\tau_n)$ if we assume that all types in the enumeration $\{\tau_n\}_n$ have the same shape.

Lemma 5.51. *For all enumerations of types* $\{\sigma_n\}_n$*,* $\phi; \emptyset \vdash \mathit{unif}_c(\{\sigma_n\}_n)\{k/c\} \equiv \sigma_k$ *holds for every constant* k*.*

Note that the symbol *unif* is overloaded for enumerations of modal and linear types, as well for index variables and constraints.

Lemma 5.52 (Uniformisation of subtypings)**.** *Let* $\phi; \Phi_n; \sigma_n \equiv \tau_n$ *for all* n *be subtypings. Then we can derive a subtyping* $c, \phi; \Phi \vdash \mathit{unif}_c(\sigma) \equiv \mathit{unif}_c(\tau)$*.*

Proof. Follows from Lemma A.4 and Lemma 5.51. □

Lemma 5.53 (Uniformisation of modal sums)**.** *Let* $\sigma_n \uplus \tau_n = \rho_n$ *for all* n*. Then* $\mathit{unif}_c(\sigma_n) \uplus \mathit{unif}_c(\tau_n) = \mathit{unif}_c(\rho_n)$*.*

Lemma 5.54 (Uniformisation of bounded sums)**.** *Let* $\sum_{a<I_n} \sigma_n \equiv \tau_n$ *for all* n*. Then* $\sum_{a<\mathit{unif}_c(I_n)} \mathit{unif}_c(\sigma_n) \equiv \mathit{unif}_c(\tau_n)$ *(with* $a \neq c$*).*

Lemma 5.55 (Uniformisation of typings)**.** *Let* $\phi; \Phi_n; \Gamma_n \vdash_{M_n} t : \rho_n$ *for all* n *be typings with the same skeleton. Then we can derive a typing for the following judgement:*

$$c, \phi; \mathit{unif}_c(\Phi_n); \mathit{unif}_c(\Gamma_n) \vdash_{\mathit{unif}_c(M_n)} t : \mathit{unif}_c(\rho)$$

Proof (sketch). We do induction on t. In every case, we apply the inversion rules *under* the quantifier.

- Case $t = x$. We have $\forall n.\, \phi; \Phi_n; \Gamma_n \vdash_M x : \rho_n$. By inverting the typing statement *under the quantifier* ($\forall n$), we get:

  $$\forall n.\, \phi; \Phi_n \vdash \Gamma_n(x) \equiv \rho_n$$

  The goal follows by uniformisation of subtypings (Lemma 5.52).

- Case $t = t_1\, t_2$. We invert the enumeration of typings *under the quantifier*:

  $$\frac{\begin{array}{c}\forall n.\, \phi; \Phi_n; \Delta_{1,n} \vdash_{K_{1,n}} t_1 : [a < 1] \cdot (\sigma_n \multimap \tau_n) \\ \forall n.\, \phi; \Phi_n; \Delta_{2,n} \vdash_{K_{2,n}} t_2 : \sigma_n\{0/a\} \qquad \forall n.\, \phi; \Phi_n \vdash \Delta_{1,n} \uplus \Delta_{2,n} \equiv \Gamma_n \\ \forall n.\, \phi; \Phi_n \vdash \tau_n\{0/a\} \equiv \rho_i \qquad \forall n.\, \phi; \Phi_n \vDash M_n \equiv K_{1,n} + K_{2,n} \\ \text{(for enumerations } \{K_{1,n}\}_n,\ \{K_{2,n}\}_n,\ \{\sigma_n\}_n,\ \{\tau_n\}_n, \text{ etc.)} \\ \text{all } \sigma_n \text{ and } \tau_n \text{ have the same } \mathsf{PCF} \text{ shape}\end{array}}{\forall n.\, \phi; \Phi_n; \Gamma_n \vdash_{M_n} t_1\, t_2 : \rho_n}$$

  Then we apply the inductive hypotheses on the new typing enumerations of t_1 and t_2:

  $$\begin{array}{l} c, \phi; \mathit{unif}_c(\Phi_n); \mathit{unif}_c(\Delta_{1,n}) \vdash_{\mathit{unif}_c(K_{1,n})} t_1 : \mathit{unif}_c([a < 1] \cdot (\sigma_n \multimap \tau_n)) \\ \qquad\qquad\qquad\qquad\qquad\qquad = [a < 1] \cdot (\mathit{unif}_c(\sigma_n) \multimap \mathit{unif}_c(\tau_n)) \\ c, \phi; \mathit{unif}_c(\Phi_n); \mathit{unif}_c(\Delta_{2,n}) \vdash_{\mathit{unif}_c(K_{2,n})} t_1 : \mathit{unif}_c(\sigma_n\{0/a\}) \end{array}$$

  Finally, we apply rule App and Lemmas 5.52 and 5.53.

- Case $t = \lambda x.\, t$. We invert the typings:

  $$\frac{\begin{array}{cc}\forall n.\, a, \phi; a < I_n, \Phi_n; x : \sigma_n, \Delta_n \vdash_{K_n} t : \tau_n & \forall n.\, \phi; \Phi_n \vdash \sum_{a<I_n} \Delta_n \equiv \Gamma_n \\ \forall n.\, \phi; \Phi_n \vdash [a < I_n] \cdot (\sigma_n \multimap \tau_n) \equiv \rho_n & \forall n.\, \phi; \Phi_n \vDash I_n + \sum_{a<I_n} K_n \equiv M_n\end{array}}{\forall n.\, \phi; \Phi_n; \Gamma_n \vdash_{M_n} \lambda x.\, t : \rho_n}$$

  We apply the inductive hypothesis on the enumeration of typings of t:

  $$\begin{aligned} c, a, \phi; a < \mathit{unif}_c(\{I_n\}_n), \mathit{unif}_c(\{\Phi_n\}_n); x : \mathit{unif}_c(\{\sigma_n\}_n), \mathit{unif}_c(\{\Delta_n\}_n) \\ \vdash_{\mathit{unif}_c(\{K_n\}_n)} t : \mathit{unif}_c(\{\tau_n\}_n) \end{aligned}$$

  Using Lemmas 5.52 and 5.54, we can show:

  $$c, \phi; \mathit{unif}_c(\{\Phi_n\}_n) \vdash_{\sum_{a<\mathit{unif}_c(\{I_n\}_n)}} \mathit{unif}_c(\{\Delta_n\}_n) \equiv \mathit{unif}_c(\{\Gamma_n\}_n)$$

  The goal follows from the rule App; the remaining semantical obligations are easy.

- All other cases are similar. □

Theorem 5.56 (Completeness for natural functions)**.** *Let* $t : \mathsf{Nat} \to \mathsf{Nat}$ *be a total* PCF *function (not necessarily a* λ*-abstraction) such that for all* i, $t(\underline{i}) \Downarrow_{g(i)} \underline{f(i)}$. *Then we can type this function in* dℓPCF$_\mathsf{v}$ *as follows:*

$$a; \emptyset; \emptyset \vdash_{g(a)} t : [c < 1] \cdot (\mathsf{Nat}[a] \multimap \mathsf{Nat}[f(a)])$$

Proof. First we using completeness for programs (since $t(\underline{i})$ is a closed program for every i):

$$\forall i.\, \emptyset; \emptyset; \emptyset \vdash_{g(i)} t(\underline{i}) : \mathsf{Nat}[f(i)]$$

Now, we invert each typing. Since the constant $\underline{i}$ has weight 0, t must have the full weight $g(i)$. From this, we conclude:

$$\forall i.\, \emptyset; \emptyset; \emptyset \vdash_{g(i)} t : [c < 1] \cdot (\mathsf{Nat}[i] \multimap \mathsf{Nat}[f(i)])$$

The goal follows by uniformising this enumeration of typings (using Lemma 5.55). □

Notes on the proof of uniformisation of typings In the proof of Lemma 5.55, we convert an enumeration of dℓPCF$_\mathsf{v}$ typings into one typing. Because all typing derivations have the same skeleton, they only differ in the index terms. Essentially, what we do in the proof is that we 'overlay' the typings. A node of the new typing tree, where c is a free variable, corresponds to the node at the same position in the c^{th} typing tree. In other words, a *case distinction* over c is built into all index terms and constraints of every node in the new typing tree.

We can derive similar theorems as Theorem 5.56, for example for types $\mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat})$. However, due to the restriction that we can only unify *countable* sets of typings, this approach does not work for higher-order functions.

Note that the uniformisation theorem is not useful in practice, since it is morally impossible to generate countable infinite executions and convert them into typings. In Chapter 8 we will show how to construct typings without need to execute terms. This even allows us to type diverging terms.

Applications of completeness for natural numbers Note that in order to *use* the typing of a PCF function $\emptyset \vdash v : \mathsf{Nat} \to \mathsf{Nat}$, as provided by the proof of Lemma 5.55, we first have to substitute a for an index term K that corresponds to the value of argument of the function. This typing after substitution can only be used once, due to the bound $[c < 1]$.

Consider the case that v needs to be applied more than once, say K times. For example, for $a < K$, we want to apply v to the argument $\underline{p(a)}$, and $v(\underline{p(a)})$ yields the result $\underline{f(p(a))}$ in $g(p(a))$ steps. We can substitute an index term that is equivalent to $g(a)$ for a in the typing generated by Theorem 5.56, and we also add the constraint $a < K$:

$$a; a < K; \emptyset \vdash_{g(p(a))} t : [c < 1] \cdot (\mathsf{Nat}[p(a)] \multimap \mathsf{Nat}[f(p(a))])$$

Now, since we have assumed that v is a value, we can apply Lemma 5.44 on the above typing, which yields:

$$\begin{aligned}&\emptyset; \emptyset; \emptyset \vdash_{\sum_{a<K} g(p(a))} t : \\ &\qquad \sum_{a<K} ([c < 1] \cdot (\mathsf{Nat}[p(a)] \multimap \mathsf{Nat}[f(a)])) \equiv [a < K] \cdot (\mathsf{Nat}[p(a)] \multimap \mathsf{Nat}[f(p(a))])\end{aligned}$$

This typing can now be applied K times, and the weight already accounts for these K applications.

5.6 Embedding of dℓT in dℓPCF$_\mathsf{v}$

It is very easy to embed System T inside PCF. We can implement the iteration operator using unbounded recursion:

$$\mathsf{iter}\, t_1\, t_2 := \mu f x.\, \mathsf{ifz}\, x\, \mathsf{then}\, t_2\, \mathsf{else}\, t_1\, (f\, (\mathsf{Pred}(x)))$$

In this section, we show that it is also possible to show that the rule ITER is admissible in dℓPCF$_\mathsf{v}$. That is, we can derive the following rule:[11]

$$\frac{\begin{array}{c} a, b, \phi; b < K, a < I, \Phi; \Delta_1 \vdash_{M_1} t_1 : [\_ < 1] \cdot (\sigma \multimap \sigma\{1 + a/a\}) \\ b, \phi; b < K, \Phi; \Delta_2 \vdash_{M_2} t_2 : \sigma\{0/a\} \\ \Gamma := \sum_{b<K}((\sum_{b<I} \Delta_1\{I \dot- 1 \dot- a/a\}) \uplus \Delta_2) \qquad M := K + \sum_{b<K}(I + (\sum_{a<I} M_1) + M_2) \end{array}}{\phi; \Phi; \Gamma \vdash_M \mathsf{iter}\, t_1\, t_2 : [b < K] \cdot (\mathsf{Nat}[I] \multimap \sigma\{I/a\})}$$

This is a useful rule, since it immediately allows us to lift the typings of the addition and multiplication functions from Section 4.6, as well as other primitive recursive functions. This way, we do not have to reason about recursion forests.

Lemma 5.57. *The rule* ITER *is admissible in* dℓPCF$_\mathsf{v}$.

---

[11] Without loss of generality, we assume that the variable c in the bound $[c < 1]$ does not occur in σ. (If it does, it can simply be substituted with 0.) We thus write $[\_ < 1]$.

$$\begin{array}{ccc}
\mathsf{Nat}[I\{0/b\}] \multimap \sigma\{I/a, 0/c\}\{0/b\} & & \mathsf{Nat}[I\{0/b\}] \multimap \sigma\{I/a, 0/c\}\{K-1/b\} \\
\mathsf{Nat}[I\{0/b\}-1] \multimap \sigma\{I-1/a, 0/c\}\{0/b\} & & \mathsf{Nat}[I\{0/b\}-1] \multimap \sigma\{I-1/a, 0/c\}\{K-1/b\} \\
\cdots & \cdots & \cdots \\
\mathsf{Nat}[1] \multimap \sigma\{1/a, 1/b, 0/c\} & & \mathsf{Nat}[1] \multimap \sigma\{1/a, K-1/b, 0/c\} \\
\mathsf{Nat}[0] \multimap \sigma\{0/a, 0/b, 0/c\} & & \mathsf{Nat}[0] \multimap \sigma\{0/a, K-1/b, 0/c\}
\end{array}$$

Figure 5.3: The type B (depicted as a forest) in the embedding of dℓT in dℓPCF$_\mathsf{v}$

Proof. We have to type a fixpoint. For the parameter K of Fix, we just choose the index term K from the premise. This means, the new recursion forest (described by I^* as defined below) consists of K trees. Each of these trees is linear and has length $1 + I$ (for $b < K$) each. This means, the cardinality of the recursion forest is $H := \triangle_b^K I^* = K + \sum_{b<K} I$.

Note that both I and I^* have the variable b free, albeit with different meanings. In I, $b < K$ denotes the number of the *instance* of the iteration (which corresponds to the b^{th} recursion tree in I^*). In I^*, $b < H$ denotes the number of a node in the recursion forest.

Formally, we can define I^* using the 'function' $f^{-1} := \mathit{findSlot}_b\, K\,(1 + I)$ and the following equation:

$$I^* := (\mathsf{if}\; a < I\{b'/b\} \;\mathsf{then}\; 1 \;\mathsf{else}\; 0)\{\pi_1(f^{-1}(b))/b', \pi_2(f^{-1}(b))/a\}$$

This means that, given the index $b < H$ in the forest, we first compute the number $b' < K$ of the tree and the offset $a < I$ in this tree. The node has a child if and only if this offset is less than I (with the new index variable b' bound on b).

A visual presentation of the type B arranged in the shape of the recursion forest is depicted in Figure 5.3. Note that the 'resulting types', i.e. $\mathsf{Nat}[I] \multimap \sigma\{I/a\}$ occur at the roots of the forest. Formally, we can define B as follows:

$$B := (\mathsf{Nat}[a] \multimap \sigma)\theta \qquad\qquad \theta := \{\pi_1(f^{-1}(b))/b, I \mathbin{\dot{-}} \pi_2(f^{-1}(b))/a\}$$

We choose A such that the subtyping between B and A is trivial: Since a non-leaf node in I^* has exactly one child, we can define $A := B\{1 + b/b\}$.

Finally, we define J – the weight of the b^{th} node in the forest – using a similar case analysis. If the node is a leaf, the weight is equal to M_2 (i.e. the weight of t_2), otherwise M_1 with the corresponding $b < K$ and $a < I$. The context Δ is defined similarly:

$$J := (\mathsf{if}\; a \equiv 0 \;\mathsf{then}\; M_2 \;\mathsf{else}\; M_1)\theta \qquad\qquad \Delta := (\mathsf{if}\; a \equiv 0 \;\mathsf{then}\; \Delta_2 \;\mathsf{else}\; \Delta_1)\theta$$

We have to type the body of the fixpoint:

$$\frac{b, \phi; b < H, \Phi; x : \mathsf{Nat}[a]\,\theta, f : [a < I^*] \cdot B\{1 + b/b\}, \Delta \vdash_J \mathsf{ifz}\, x \,\mathsf{then}\, t_2 \,\mathsf{else}\, t_1\,(f\,(\mathsf{Pred}(x))) : \sigma\theta}{b, \phi; b < H, \Phi; f : [a < I^*] \cdot B\{1 + b/b\}, \Delta \vdash_J \lambda x.\, \mathsf{ifz}\, x \,\mathsf{then}\, t_2 \,\mathsf{else}\, t_1\,(f\,(\mathsf{Pred}(x))) : [\_ < 1] \cdot B}$$

We have to type the two cases corresponding to the branches of the case analysis on x:

- Case $0 \gtrsim a\theta$. This means that we are at the $(b\theta)^{th}$ leaf node in the forest. After simplification, we thus have to type the following judgement:

$$b, \phi; a\theta = 0, b < H; \Delta_2\theta \vdash_{M_2\theta} t_2 : \sigma\theta \equiv \sigma\{0/a\}$$

This case follows by applying the substitution θ to the original typing of t_2.

- Case $a\theta > 0$. This means that we are at a non-leaf node in the forest and thus $0 < a\theta \leq I\theta$ and $I^* = 1$. We can again simplify the typing judgement:

  $$b, \phi; a\theta > 0, b < H; \Delta_1\theta \vdash_{M_1\theta} t_1 : \sigma\theta \equiv \sigma$$

  This case also follows by applying the substitution θ to the original typing of t_1.

The subsumption obligations $\phi; \Phi \vDash \sum_{b<H} J \equiv M$ and $\phi; \Phi \vdash [a < K] \cdot B\{\triangle_b^a I/b\} \equiv [b < K] \cdot (\mathsf{Nat}[I] \multimap \sigma\{I/a\})$ hold by construction. The final subtyping obligation is on the context also holds by construction:

$$\phi; \Phi \vdash \sum_{b<H} \Delta \equiv \sum_{b<K} ((\sum_{b<I} \Delta_1\{a \dot{-} 1 \dot{-} I/a\}) \uplus \Delta_2) \qquad \square$$

The above proof clarifies why the context Δ_1 has to be 'reversed'. The following alternative rule is also admissible, where we 'swap' σ instead.

$$\textsc{iter2}\quad \frac{\begin{array}{c} a, b, \phi; b < K, a < I, \Phi; \Delta_1 \vdash_{M_1} t_1 : [\_ < 1] \cdot (\sigma\{1 + a/a\} \multimap \sigma) \\ b, \phi; b < K, \Phi; \Delta_2 \vdash_{M_2} t_2 : \sigma\{I/a\} \\ \Gamma := \sum_{b<K} ((\sum_{a<I} \Delta_1) \uplus \Delta_2) \qquad M := K + \sum_{b<K} (I + (\sum_{a<I} M_1) + M_2) \end{array}}{\phi; \Phi; \Gamma \vdash_M \mathsf{iter}\, t_1\, t_2 : [b < K] \cdot (\mathsf{Nat}[I] \multimap \sigma\{0/a\})}$$

Chapter 6

Summary of $\mathsf{d}\ell\mathsf{PCF_n}$

The call-by-name version of $\mathsf{d}\ell\mathsf{PCF}$, $\mathsf{d}\ell\mathsf{PCF_n}$, was initially published in [11]. In this chapter, we only review and explain this system. We will not prove soundness nor completeness, but we will derive these results from the same results of $\mathsf{d}\ell\mathsf{PCF_{pv}}$ in the next chapter.

The proofs in [11] are cumbersome, since the authors introduce a closure-based stack machine; typings have to be lifted to configurations of this machine. The main reason why closure-based semantics have to be used is that the *cost* of a CBN execution (i.e. the number of *variable lookups*) cannot be defined using small-step operational semantics. We have discussed in Section 3.3 why this metric is the correct metric for $\mathsf{d}\ell\mathsf{PCF_n}$.

6.1 Syntax of $\mathsf{d}\ell\mathsf{PCF_n}$ types

As in the call-by-value version of $\mathsf{d}\ell\mathsf{PCF}$, there are two syntactic categories of types.

$$\begin{aligned} \text{Basic types:}\quad & \sigma, \tau, \rho ::= \mathsf{Nat}[I] \mid A \multimap \sigma \\ \text{Modal types:}\quad & A, B ::= [a < I] \cdot \sigma \\ \text{Contexts:}\quad & \Gamma, \Delta ::= \emptyset \mid x : A, \Gamma \end{aligned}$$

For arrow types, the *argument* is always quantified. This means that we bound how often (if at all) the argument has to be (re)evaluated.

6.2 (Bounded) sums

The definition of modal sums is simpler as in $\mathsf{d}\ell\mathsf{T}$ and $\mathsf{d}\ell\mathsf{PCF_v}$, since Nat types are not modal types:

Definition 6.1 (Binary and bounded sums)**.**

$$\frac{A_1 = [a < I_1] \cdot \sigma \qquad A_2 = [a < I_2] \cdot \sigma\{a + I_1/a\}}{A_1 \uplus A_2 = [a < I_1 + I_2] \cdot \sigma} \qquad \frac{A = [c < J] \cdot \sigma\{c + \sum_{d<a} J\{d/a\}/b\}}{\sum_{a<I} A = [c < \sum_{a<I} J] \cdot \sigma}$$

Modal sums can be constructed in exactly the same way as we have shown for $\mathsf{d\ell PCF_v}$ in Section 5.5.3.

6.3 Typing rules

The typing rules, as published in [11], are shown in Figure 6.1. As in [11], for variety, the rules do not include an explicit subsumption rule. Instead, there are subtyping judgements in all premises where needed; the subsumption rule is thus still admissible. Below, we will explain the different definition of *weights*, and we will also explain the typing rules.

Explanation of weights

Weights in $\mathsf{d\ell PCF_n}$ are an upper bound on how often variables are looked up. However, this bound only holds if the initial program is closed, since variable lookups in the contexts are not counted. We always increment the weight in the application rule, where we account for the potential number of variable lookups of the argument by the function. For example, the term $(\lambda x.\, \underline{0})\, \underline{1}$ has weight 0, because the variable x is never used. Here, the function $\lambda x.\, \underline{0}$ has type $([a < 0] \cdot \mathsf{Nat}[1]) \multimap \mathsf{Nat}[0]$, which means that the argument is not needed, and thus will be never executed. We could as well apply this function to a diverging term.

Example typing

The following typing, which *prima facie* looks nonsensical, is valid in $\mathsf{d\ell PCF_n}$ (with an addition operator):

$$\emptyset; \emptyset; \emptyset \vdash_2 (\lambda x.\, x + x) : ([a < 2] \cdot \mathsf{Nat}[a]) \multimap \mathsf{Nat}[0 + 1]$$

In the above function, the variable x can be used twice, but with different values each. The weight of the above function is 2, because when evaluating the function, the variable x will be used twice.

Although it is not possible to define a closed term of $\mathsf{d\ell PCF_n}$ type $[a < 2] \cdot \mathsf{Nat}[a]$, this would be possible in an impure extension of $\mathsf{d\ell PCF_n}$. We discussed a similar issue in the previous chapter.

Explanation of the typing rules

Variables As noted above, variable access is bounded. In order to use a variable x, the bound on it has to be shown to be positive.

Lambda In order to type a λ expression, we bound how often the argument may be evaluated, using an index term I. We simply add $[a < I] \cdot \sigma$ to the context and type $t : \tau$. Unlike the call-by-value version of $\mathsf{d\ell PCF}$, the weight of $\lambda x.\, t$ is equal to the weight of t.

Application The argument has to be typed I times, because t potentially needs the argument I-times. The weight of the application is equal to the weight of t_1 plus the sum over weights of t_2 plus I (to account for the I lookups of the argument t_2 by the function to which t_1 evaluates).

Fixpoint The index term I represents a recursion tree (not a forest, as in $\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$). This means, that I is a bound on the number of times x may be called in the b^{th} node in the tree. In other words, the variable x is used I times, and thus is assumed to have type $[a < I] \cdot \sigma$. σ represents the types of the children of node b, and τ is the type of node b. The type of the fixpoint is equal to the type of the root node. To the weight $\sum_{b<H} J$, we add $H \dot{-} 1$ – one for every edge of the tree – to account for the lookups of the variable x.

The rules Const, Succ, Pred, and Ifz are exactly as in $\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$.

6.4 Soundness and completeness

Theorem 6.2 (Soundness of $\mathsf{d}\ell\mathsf{PCF}_\mathsf{n}$ programs)**.** *Theorem 4.5 holds for* $\mathsf{d}\ell\mathsf{PCF}_\mathsf{n}$*: Let t be a closed program (i.e. a* PCF *term with simple type* Nat*). Then we can show:*

- *Let $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{k} t : \mathsf{Nat}[I]$ be a* $\mathsf{d}\ell\mathsf{PCF}_\mathsf{n}$ *typing. Then there is a number $k' \leq k$ such that $t \Downarrow_{k'} \underline{n}$. In other words, t evaluates to $\underline{n}$, and needs at most k variable lookups in the big-step closure semantics. Furthermore, $\vDash n \sqsubseteq I$. In particular, if $\vDash I \equiv m$, then $m = n$.*

- *Let $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{K} t : \mathsf{Nat}[I]$ be a precise typing and $t \Downarrow_k \underline{n}$. Then $\vDash K \equiv k$ and $\vDash I \equiv n$.*

Theorem 6.3 ($\mathsf{d}\ell\mathsf{PCF}_\mathsf{n}$ completeness for programs)**.** *Let $\langle t; \emptyset \rangle \Downarrow_k \langle \underline{n}; \xi \rangle$. Then $\emptyset; \emptyset; \emptyset \vdash_k t : \mathsf{Nat}[n]$.*

We will prove these theorems as corollaries in the next chapter.

$$\frac{\phi;\Phi \vDash I \sqsubseteq J}{\phi;\Phi \vdash \mathsf{Nat}[I] \sqsubseteq \mathsf{Nat}[J]} \qquad \frac{\phi;\Phi \vdash A_2 \sqsubseteq A_1 \qquad \phi;\Phi \vdash \sigma_1 \sqsubseteq \sigma_2}{\phi;\Phi \vdash A_1 \multimap \sigma_1 \sqsubseteq A_2 \multimap \sigma_2}$$

$$\frac{\phi;\Phi \vDash J \leq I \qquad \phi; a < J, \Phi \vdash \sigma \sqsubseteq \tau}{\phi;\Phi \vdash [a < I] \cdot \sigma \sqsubseteq [a < J] \cdot \tau} \qquad \frac{\begin{array}{c}\phi;\Phi \vdash \sigma \sqsubseteq \tau\\ \phi;\Phi \vdash \tau \sqsubseteq \sigma\end{array}}{\phi;\Phi \vdash \sigma \equiv \tau} \qquad \frac{\begin{array}{c}\phi;\Phi \vdash A \sqsubseteq B\\ \phi;\Phi \vdash B \sqsubseteq A\end{array}}{\phi;\Phi \vdash A \equiv B}$$

Const

$$\frac{\phi;\Phi \vdash \mathsf{Nat}[n] \sqsubseteq \rho}{\phi;\Phi;\Gamma \vdash_M \underline{n} : \rho}$$

Succ

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma \vdash_M t : \mathsf{Nat}[J]\\ \phi;\Phi \vdash \mathsf{Nat}[1 + J] \sqsubseteq \rho\end{array}}{\phi;\Phi;\Gamma \vdash_M \mathsf{Succ}(t) : \rho}$$

Pred

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma \vdash_M t : \mathsf{Nat}[J]\\ \phi;\Phi \vdash \mathsf{Nat}[J \mathbin{\dot{-}} 1] \sqsubseteq \rho\end{array}}{\phi;\Phi;\Gamma \vdash_M \mathsf{Pred}(t) : \rho}$$

Var

$$\frac{\phi;\Phi \vDash 1 \leq I \qquad \phi;\Phi \vdash \sigma\{0/a\} \sqsubseteq \rho}{\phi;\Phi; x : [a < I] \cdot \sigma, \Gamma \vdash_M x : \rho}$$

Lam

$$\frac{\phi;\Phi; x : A, \Gamma \vdash_M t : \tau \qquad \phi;\Phi \vdash (A \multimap \tau) \sqsubseteq \rho}{\phi;\Phi;\Gamma \vdash_M \lambda x.\, t : \rho}$$

App

$$\frac{\begin{array}{c}\phi;\Phi;\Delta_1 \vdash_{K_1} t_1 : ([a < I] \cdot \sigma) \multimap \tau \qquad a, \phi; a < I, \Phi; \Delta_2 \vdash_{K_2} t_2 : \sigma\\ \phi;\Phi \vdash \Gamma \sqsubseteq \Delta_1 \uplus \displaystyle\sum_{a<I} \Delta_2 \qquad \phi;\Phi \vDash K_1 + I + \displaystyle\sum_{a<I} K_2 \leq M\end{array}}{\phi;\Phi;\Gamma \vdash_M t_1\, t_2 : \tau}$$

Fix

$$\frac{\begin{array}{c}b, \phi; b < H, \Phi; x : [a < I] \cdot \sigma, \Delta \vdash_J t : \tau\\ a, b, \phi; a < I, b < H, \Phi \vdash \tau\{1 + b + \left(\mathop{\triangle}\limits_{c}^{a} I\{1 + b + c/b\}\right)/b\} \sqsubseteq \sigma \qquad \phi;\Phi \vdash \Gamma \sqsubseteq \displaystyle\sum_{b<H} \Delta\\ \phi;\Phi; \vDash H \mathbin{\dot{-}} 1 + \displaystyle\sum_{b<H} J \leq M \qquad \phi;\Phi \vdash \tau\{0/b\} \sqsubseteq \rho \qquad \phi;\Phi \vDash H \equiv \mathop{\triangle}\limits_{b}^{1} I\end{array}}{\phi;\Phi;\Gamma \vdash_M \mu x.\, t : \rho}$$

Ifz

$$\frac{\begin{array}{c}\phi;\Phi;\Delta_1 \vdash_{K_1} t_1 : \mathsf{Nat}[J] \qquad \phi; 0 \gtrsim J, \Phi; \Delta_2 \vdash_{K_2} t_2 : \rho \qquad \phi; 0 < J, \Phi; \Delta_2 \vdash_{K_2} t_3 : \rho\\ \phi;\Phi \vdash \Gamma \sqsubseteq \Delta_1 \uplus \Delta_2 \qquad \phi;\Phi \vDash K_1 + K_2 \leq M\end{array}}{\phi;\Phi;\Gamma \vdash_M \mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3 : \rho}$$

Figure 6.1: Subtyping and typing rules of dℓPCF$_{\mathsf{n}}$.

Chapter 7

Call-by-push-value $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$

In this section, we introduce a (novel) variant of $\mathsf{d}\ell\mathsf{PCF}$ that targets the call-by-push value variant of PCF. We will see that $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ and $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ typings can be translated to $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ typings. Moreover, we can derive soundness and completeness proofs for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ and $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ from the same properties for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$. In this way, $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ subsumes the other variants. Interestingly, the proofs of these theorems are *simpler* in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$.

7.1 $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ types

As in the simple type system for CBPV (see Section 2.3.2), types are divided into value types and computation types.

$$\begin{aligned} \text{Value types:} \quad & A ::= [a < I] \cdot \underline{B} \mid \mathsf{Nat}[I] \\ \text{Computation types:} \quad & \underline{B} ::= \mathsf{F}\, A \mid A \multimap \underline{B} \\ \text{Contexts:} \quad & \Gamma, \Delta ::= \emptyset \mid x : A, \Gamma \end{aligned}$$

In the syntax of the simple CBPV types, the lift from computation types to value types is called U. Here, we refine U with a bound $[a < I]$.

Again, we can erase the annotations and compute the *shape* of a $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ (value/computation) type, which is a simple CBPV (value/computation) type.

Definition 7.1 (Annotation erasure)**.** By mutual recursion on value/computation types:

$$\begin{aligned} (\!| \mathsf{Nat}[I] \,|\!) &:= \mathsf{Nat} & \qquad (\!| \mathsf{F}\, A |\!) &:= \mathsf{F}\, (\!| A |\!) \\ (\!| [a < I] \cdot \underline{B} |\!) &:= \mathsf{U}\, (\!| \underline{B} |\!) & \qquad (\!| A \multimap \underline{B} |\!) &:= (\!| A |\!) \to (\!| \underline{B} |\!) \end{aligned}$$

In $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$, we bound the number of times thunks can be forced. For example, values of the type $[a < 2] \cdot (\mathsf{F}\, \mathsf{Nat}[1])$ are thunked computations that can be forced twice, and each forcing yields a computation that will terminate as $\mathsf{return}\, \underline{1}$ (or diverge).

Note that the syntax of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$'s modal types $\tau ::= [a < I] \cdot A \mid \mathsf{Nat}[I]$ is similar to syntax of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ value types. The only difference between $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ linear types and dℓPCF$_{\mathsf{pv}}$ computation types is that the latter have the lifting F from value types. Therefore, we can define modal sums over value types (i.e. $A_1 \uplus A_2$ and $\sum_{a<I}$) in the same way we did for modal types in dℓT and dℓPCF$_{\mathsf{v}}$, see Section 4.3. We can also 'construct' them in the same way, which will be needed in the completeness proof.

7.2 Typing Rules

We have two typing judgements, $\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_K t : \underline{B}$ for computations, and $\phi; \Phi; \Gamma \vdash^{\mathsf{v}}_K t : A$ for values. The typing and subtyping rules are depicted in Figure 7.1. For readability, we use explicit subsumption rules.

Interestingly, many of the rules are similar to the corresponding rules either in dℓPCF$_{\mathsf{v}}$ or in dℓPCF$_{\mathsf{n}}$. This is summarised in the table below:

both	dℓPCF$_{\mathsf{v}}$	dℓPCF$_{\mathsf{n}}$	new
CONST, IFZ	VAR, APP,	LAM, FIX	RETURN, BIND, THUNK, FORCE, SUCC, PRED

Thunk The thunk rule is similar to the LAM of dℓPCF$_{\mathsf{v}}$. In the dℓPCF$_{\mathsf{v}}$ rule, the weight already accounts for the cost of all applications of the function. In THUNK, the weight already accounts for the cost of all its potential forcings.

Force As in the dℓPCF$_{\mathsf{v}}$ APP rule, the cost for the forcing was already accounted for in THUNK. Therefore, the weight is not increased.

Fixpoint The fixpoint rule is identical to the fixpoint rule of dℓPCF$_{\mathsf{n}}$. We also add $H \dot{-} 1$ to the weight, since x has to be forced at every recursive self-application.

The other rules are self-explanatory.

7.3 Call-by-name translation

Recall from Section 2.3.3 that the function $\cdot^{\mathsf{n}}$ translates PCF terms to CBPV computations. Below, we will translate dℓPCF$_{\mathsf{n}}$ typing to dℓPCF$_{\mathsf{pv}}$ typings. The translation preserves the weight.

dℓPCF$_{\mathsf{n}}$ *modal types* ($A ::= [a < I] \cdot \sigma$) are translated to d$\ell$PCF$_{\mathsf{pv}}$ value types. *Basic types* ($\sigma ::= \mathsf{Nat}[I] \mid A \multimap \sigma$) are translated to computation types.

Definition 7.2 (Translation of dℓPCF$_{\mathsf{n}}$ types)**.**

$$\begin{aligned}([a < I] \cdot \sigma)^{\mathsf{n}} &:= [a < I] \cdot \sigma^{\mathsf{n}} \\ (\mathsf{Nat}[I])^{\mathsf{n}} &:= \mathsf{F}\,\mathsf{Nat}[I] \\ (A \multimap \sigma)^{\mathsf{n}} &:= A^{\mathsf{n}} \multimap \sigma^{\mathsf{n}}\end{aligned}$$

dℓPCF$_{\mathsf{n}}$ contexts (consisting of modal types) are pointwisely lifted to dℓPCF$_{\mathsf{pv}}$ contexts.

$$\frac{\phi;\Phi\vDash I\sqsubseteq J}{\phi;\Phi\vdash\mathsf{Nat}[I]\sqsubseteq\mathsf{Nat}[J]}\qquad\frac{\phi;\Phi\vdash A_1\sqsubseteq A_2}{\phi;\Phi\vdash\mathsf{F}\,A_1\sqsubseteq\mathsf{F}\,A_2}\qquad\frac{\phi;\Phi\vdash A_2\sqsubseteq A_1\qquad\phi;\Phi\vdash\underline{B}_1\sqsubseteq\underline{B}_2}{\phi;\Phi\vdash A_1\multimap\underline{B}_1\sqsubseteq A_2\multimap\underline{B}_2}$$

$$\frac{\phi;\Phi\vDash J\leq I\qquad\phi;a<J,\Phi\vdash\underline{B}_1\sqsubseteq\underline{B}_2}{\phi;\Phi\vdash[a<I]\cdot\underline{B}_1\sqsubseteq[a<J]\cdot\underline{B}_2}\qquad\frac{\begin{array}{c}\phi;\Phi\vdash A_1\sqsubseteq A_2\\\phi;\Phi\vdash A_2\sqsubseteq A_1\end{array}}{\phi;\Phi\vdash A_1\equiv A_2}\qquad\frac{\begin{array}{c}\phi;\Phi\vdash\underline{B}_1\sqsubseteq\underline{B}_2\\\phi;\Phi\vdash\underline{B}_2\sqsubseteq\underline{B}_1\end{array}}{\phi;\Phi\vdash\underline{B}_1\equiv\underline{B}_2}$$

SubV

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma'\vdash^{\mathsf{v}}_{K_1}v:A_1\qquad\phi;\Phi\vdash A_1\sqsubseteq A_2\\\phi;\Phi\vdash\Gamma\sqsubseteq\Gamma'\qquad\phi;\Phi\vDash K_1\leq K_2\end{array}}{\phi;\Phi;\Gamma\vdash^{\mathsf{v}}_{K_2}v:A_2}$$

SubC

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma'\vdash^{\mathsf{c}}_{K_1}t:\underline{B}_1\qquad\phi;\Phi\vdash\underline{B}_1\sqsubseteq\underline{B}_2\\\phi;\Phi\vdash\Gamma\sqsubseteq\Gamma'\qquad\phi;\Phi\vDash K_1\leq K_2\end{array}}{\phi;\Phi;\Gamma\vdash^{\mathsf{c}}_{K_2}t:\underline{B}_2}$$

Const

$$\phi;\Phi;\Gamma\vdash^{\mathsf{v}}_{0}\underline{n}:\mathsf{Nat}[n]$$

Var

$$\phi;\Phi;\Gamma\vdash^{\mathsf{v}}_{0}x:\Gamma(x)$$

Lam

$$\frac{\phi;\Phi;x:A,\Gamma\vdash^{\mathsf{c}}_{M}t:\underline{B}}{\phi;\Phi;\Gamma\vdash^{\mathsf{c}}_{M}\lambda x.\,t:A\multimap\underline{B}}$$

App

$$\frac{\begin{array}{c}\phi;\Phi;\Delta_1\vdash^{\mathsf{c}}_{K_1}t:A\multimap\underline{B}\\\phi;\Phi;\Delta_2\vdash^{\mathsf{v}}_{K_2}v:A\end{array}}{\phi;\Phi;\Delta_1\uplus\Delta_2\vdash^{\mathsf{c}}_{K_1+K_2}t\,v:\underline{B}}$$

Fix

$$\frac{\begin{array}{c}b,\phi;b<H,\Phi;x:[a<I]\cdot\underline{B}_1,\Delta\vdash^{\mathsf{c}}_{J}t:\underline{B}_2\\a,b,\phi;a<I,b<H,\Phi\vdash\underline{B}_2\{1+b+\left(\overset{a}{\underset{c}{\bigtriangleup}}I\{1+b+c/b\}\right)/b\}\sqsubseteq\underline{B}_1\qquad\phi;\Phi\vDash H\equiv\overset{1}{\underset{b}{\bigtriangleup}}I\end{array}}{\phi;\Phi;\sum_{b<H}\Delta\vdash^{\mathsf{c}}_{H\mathbin{\dot{-}}1+\sum_{b<H}J}\mu x.\,t:\underline{B}_2\{0/b\}}$$

Succ

$$\frac{\begin{array}{c}\phi;\Phi;\Delta_1\vdash^{\mathsf{v}}_{K_1}v:\mathsf{Nat}[J]\\\phi;\Phi;x:\mathsf{Nat}[1+J],\Delta_2\vdash^{\mathsf{c}}_{K_2}t:\underline{B}\end{array}}{\phi;\Phi;\Delta_1\uplus\Delta_2\vdash^{\mathsf{c}}_{K_1+K_2}\mathsf{calc}\,x\leftarrow\mathsf{Succ}(v)\,\mathsf{in}\,t:\underline{B}}$$

Pred

$$\frac{\begin{array}{c}\phi;\Phi;\Delta_1\vdash^{\mathsf{v}}_{K_1}v:\mathsf{Nat}[J]\\\phi;\Phi;x:\mathsf{Nat}[J\mathbin{\dot{-}}1],\Delta_2\vdash^{\mathsf{c}}_{K_2}t:\underline{B}\end{array}}{\phi;\Phi;\Delta_1\uplus\Delta_2\vdash^{\mathsf{c}}_{K_1+K_2}\mathsf{calc}\,x\leftarrow\mathsf{Pred}(v)\,\mathsf{in}\,t:\underline{B}}$$

Return

$$\frac{\phi;\Phi;\Gamma\vdash^{\mathsf{v}}_{K}v:A}{\phi;\Phi;\Gamma\vdash^{\mathsf{c}}_{K}\mathsf{return}\,v:\mathsf{F}\,A}$$

Bind

$$\frac{\phi;\Phi;\Delta_1\vdash^{\mathsf{c}}_{K_1}t_1:\mathsf{F}\,A\qquad\phi;\Phi;x:A,\Delta_2\vdash^{\mathsf{c}}_{K_2}t_2:\underline{B}}{\phi;\Phi;\Delta_1\uplus\Delta_2\vdash^{\mathsf{c}}_{K_1+K_2}\mathsf{bind}\,x\leftarrow t_1\,\mathsf{in}\,t_2:\underline{B}}$$

Thunk

$$\frac{a,\phi;a<I,\Phi;\Delta\vdash^{\mathsf{c}}_{K}t:\underline{B}}{\phi;\Phi;\sum_{a<I}\Delta\vdash^{\mathsf{v}}_{I+\sum_{a<I}K}\mathsf{thunk}\,t:[a<I]\cdot\underline{B}}$$

Force

$$\frac{\phi;\Phi;\Gamma\vdash^{\mathsf{v}}_{K}v:[a<1]\cdot\underline{B}}{\phi;\Phi;\Gamma\vdash^{\mathsf{c}}_{K}\mathsf{force}\,v:\underline{B}\{0/a\}}$$

Figure 7.1: Subtyping and typing rules of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$

For example, the type $([a < 1] \cdot \mathsf{Nat}[0]) \multimap \mathsf{Nat}[0]$ is translated to $([a < 1] \cdot \mathsf{F}\,\mathsf{Nat}[0]) \multimap \mathsf{F}\,\mathsf{Nat}[0]$. That is, the argument is a thunk which, after being thunked, will evaluate to $\mathsf{return}\,\underline{0}$ (or diverge).

Lemma 7.3 (Translation of subtypings)**.** *We can translate* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ *subtypings to* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ *subtypings:*

- *If* $\phi; \Phi \vdash A \sqsubseteq B$*, then* $\phi; \Phi \vdash A^{\mathsf{n}} \sqsubseteq B^{\mathsf{n}}$*.*
- *If* $\phi; \Phi \vdash \sigma \sqsubseteq \tau$*, then* $\phi; \Phi \vdash \sigma^{\mathsf{n}} \sqsubseteq \tau^{\mathsf{n}}$*.*

Proof. By mutual induction over the subtypings of (modal/basic) types. □

Lemma 7.4 (Translation and index substitution)**.** *Let θ be an index substitution. Then* $(\sigma\theta)^{\mathsf{n}} = (\sigma^{\mathsf{n}})\theta$*. The same holds for modal types.*

Lemma 7.5 (Translation of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ typings)**.** *Every* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ *typing* $\phi; \Phi; \Gamma \vdash_M t : \rho$ *can be translated into a* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ *typing* $\phi; \Phi; \Gamma^{\mathsf{n}} \vdash^{\mathsf{c}}_M t^{\mathsf{n}} : \rho^{\mathsf{n}}$*.*

Proof. By induction on the $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ typing. Subtypings are taken care of by Lemma 7.3.

- Case Var: $t = x$, $\Gamma(x) = [a < I] \cdot \sigma$, $\sigma\{0/a\} = \rho$, and $\phi; \Phi \vDash 1 \leq I$:

$$\dfrac{\dfrac{}{\phi; \Phi; \Gamma^{\mathsf{n}} \vdash^{\mathsf{v}}_M x : \Gamma^{\mathsf{n}}(x) = [a < I] \cdot \sigma^{\mathsf{n}}} \qquad \phi; \Phi \vDash 1 \leq I}{\phi; \Phi; \Gamma^{\mathsf{n}} \vdash^{\mathsf{c}}_M \mathsf{force}\, x : \sigma^{\mathsf{n}}\{0/a\} = (\sigma^{\mathsf{n}})\{0/a\}}$$

- Case Const. We have $t = \underline{k}$ and $\rho = \mathsf{Nat}[k]$. We can show $\phi; \Phi; \Gamma^{\mathsf{n}} \vdash^{\mathsf{c}}_M \mathsf{return}\,\underline{k} : \mathsf{F}\,\mathsf{Nat}[k]$ with the $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ rules Return and Const.
- Case Lam. The inductive hypothesis yields $\phi; \Phi; x : A^{\mathsf{n}}, \Delta^{\mathsf{n}} \vdash^{\mathsf{c}}_M t^{\mathsf{n}} : \sigma^{\mathsf{n}}$. The goal follows from the $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ rule Lam.
- Case Fix. As above. Note that the λ and fixpoint rules of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ have exactly the same shape and weights as their $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ counterparts.
- Case App. Using the inductive hypotheses, we can type:

$$\dfrac{\phi; \Phi; \Delta_1^{\mathsf{n}} \vdash^{\mathsf{c}}_{K_1} t_1^{\mathsf{n}} : ([a < I] \cdot \sigma^{\mathsf{n}}) \multimap \tau^{\mathsf{n}} \qquad \dfrac{a, \phi; a < I, \Phi; \Delta_2^{\mathsf{n}} \vdash^{\mathsf{c}}_{K_2} t_2^{\mathsf{n}} : \sigma^{\mathsf{n}}}{\phi; \Phi; \sum_{a<I} \Delta_2^{\mathsf{n}} \vdash^{\mathsf{v}}_{I + \sum_{a<I} K_2} \mathsf{thunk}\, t_2 : [a < I] \cdot \sigma^{\mathsf{n}}}}{\phi; \Phi; \Delta_1 \uplus \sum_{a<I} \Delta_2^{\mathsf{n}} \vdash^{\mathsf{c}}_{I + K_1 + \sum_{a<I} K_2} t_1^{\mathsf{n}}\,(\mathsf{thunk}\, t_2^{\mathsf{n}}) : \tau^{\mathsf{n}}}$$

- Case Ifz. Using the inductive hypotheses, we can type:

$$\dfrac{\phi; \Phi; \Delta_1^{\mathsf{n}} \vdash^{\mathsf{c}}_{K_1} t_1^{\mathsf{n}} : \mathsf{F}\,\mathsf{Nat}[I] \qquad \dfrac{\dfrac{}{\phi; \Phi; z : \mathsf{Nat}[I] \vdash^{\mathsf{v}}_0 z : \mathsf{Nat}[I]} \qquad \phi; 0 \gtrsim I \text{ resp. } 0 < I, \Phi; \Delta_2^{\mathsf{n}} \vdash_{K_2} t_{2,3}^{\mathsf{n}} : \rho^{\mathsf{n}}}{\phi; \Phi; z : \mathsf{Nat}[I]\,, \Delta_2^{\mathsf{n}} \vdash^{\mathsf{c}}_{K_2} \mathsf{ifz}\, z\, \mathsf{then}\, t_2^{\mathsf{n}}\, \mathsf{else}\, t_3^{\mathsf{n}} : \rho^{\mathsf{n}}}}{\phi; \Phi; \Delta_1^{\mathsf{n}} \uplus \Delta_2^{\mathsf{n}} \vdash^{\mathsf{c}}_{K_1 + K_2} \mathsf{bind}\, z \leftarrow t_1^{\mathsf{n}}\, \mathsf{in}\, \mathsf{ifz}\, z\, \mathsf{then}\, t_2^{\mathsf{n}}\, \mathsf{else}\, t_3^{\mathsf{n}} : \rho^{\mathsf{n}}}$$

- Case Succ We derive the following typing:

$$\dfrac{\phi;\Phi;\Gamma^{\mathsf{n}} \vdash^{\mathsf{c}}_{K} t^{\mathsf{n}} : \mathsf{F}\,\mathsf{Nat}[I] \qquad \dfrac{\dfrac{}{\phi;\Phi;x:\mathsf{Nat}[I] \vdash^{\mathsf{v}}_{0} x : \mathsf{Nat}[I]} \qquad \dfrac{\dfrac{}{\phi;\Phi;y:\mathsf{Nat}[1+I] \vdash^{\mathsf{v}}_{0} y : \mathsf{Nat}[1+I]}}{\phi;\Phi;y:\mathsf{Nat}[1+I] \vdash^{\mathsf{c}}_{0} \mathsf{return}\,y : \mathsf{F}\,\mathsf{Nat}[1+I]}}{\phi;\Phi;x:\mathsf{Nat}[I] \vdash^{\mathsf{c}}_{0} \mathsf{calc}\,y \leftarrow \mathsf{Succ}(x)\,\mathsf{in}\,\mathsf{return}\,y : \mathsf{F}\,\mathsf{Nat}[1+I]}}{\phi;\Phi;\Gamma^{\mathsf{n}} \vdash^{\mathsf{c}}_{K} \mathsf{bind}\,x \leftarrow t^{\mathsf{n}}\,\mathsf{in}\,\mathsf{calc}\,y \leftarrow \mathsf{Succ}(x)\,\mathsf{in}\,\mathsf{return}\,y : \mathsf{F}\,\mathsf{Nat}[1+I]}$$

- Case Pred. As the above case. ☐

7.4 Call-by-value translation

We show how to translate $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ typings into $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ typings.

We translate $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ *modal types* ($\sigma ::= [a < I] \mid \mathsf{Nat}[I]$) to $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ values types, and *linear types* ($A ::= \sigma \multimap \tau$) to computation types.

Definition 7.6 (Translation of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ types)**.**

$$\begin{aligned}([a < I] \cdot \sigma)^{\mathsf{v}} &:= [a < I] \cdot \sigma^{\mathsf{v}}\\ \mathsf{Nat}[I]^{\mathsf{v}} &:= \mathsf{Nat}[I]\\ (\sigma \multimap \tau)^{\mathsf{v}} &:= \sigma^{\mathsf{v}} \multimap \mathsf{F}\,\tau^{\mathsf{v}}\end{aligned}$$

$\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ contexts (consisting of modal types) are pointwisely lifted to $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ contexts (consisting of value types).

For example, the type $[a < 1] \cdot (\mathsf{Nat}[0] \multimap \mathsf{Nat}[0])$ is translated to $[a < 1] \cdot (\mathsf{Nat}[0] \multimap \mathsf{F}\,\mathsf{Nat}[0])$.

Lemma 7.7 (Translation of subtypings)**.** *We can translate* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ *subtypings to* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ *subtypings:*

- *If* $\phi;\Phi \vdash A \sqsubseteq B$*, then* $\phi;\Phi \vdash A^{\mathsf{v}} \sqsubseteq B^{\mathsf{v}}$.
- *If* $\phi;\Phi \vdash \sigma \sqsubseteq \tau$*, then* $\phi;\Phi \vdash \sigma^{\mathsf{v}} \sqsubseteq \tau^{\mathsf{v}}$.

Proof. By mutual induction over the (modal/linear) subtyping judgements. ☐

Lemma 7.8 (Translation and index substitution)**.** *Let* θ *be an index substitution. Then* $(\sigma\theta)^{\mathsf{v}} = (\sigma^{\mathsf{v}})\theta$*. The same holds for linear types.*

The following admissible typing rule will be important in the conversion of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ to $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ typings below, and it will also be helpful in the soundness proof of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$. Note that the rule looks similar to the the $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ rule Fix, but H is added to the weight.

Lemma 7.9 (Admissible rule for $\mathsf{thunk}\,\mu x.\,t$)**.** *The following rule is admissible:*

$$\textsc{ThunkFix}\quad \frac{\begin{array}{c} b, \phi; b < H, \Phi; x : [a < I] \cdot \underline{B}_1, \Delta \vdash^{\mathsf{c}}_{J} t : \underline{B}_2 \\ a, b, \phi; a < I, b < H, \Phi \vdash \underline{B}_2\{1 + b + \left(\underset{c}{\overset{a}{\triangle}} I\{1 + b + c/b\}\right)/b\} \sqsubseteq \underline{B}_1 \qquad \phi; \Phi \vdash \Gamma \sqsubseteq \sum_{b<H} \Delta \\ \phi; \Phi \vDash H + \sum_{b<H} J \leq M \qquad \phi; \Phi \vdash [a < K] \cdot \underline{B}_2\{\underset{b}{\overset{a}{\triangle}} I/b\} \sqsubseteq A \qquad \phi; \Phi \vDash H \equiv \underset{b}{\overset{K}{\triangle}} I \end{array}}{\Phi; \Gamma \vdash^{\mathsf{v}}_{M} \mathsf{thunk}\,\mu x.\,t : A}$$

Proof. We will first split the typing of t into K typings; then using the fixpoint rule, we will build K typings for $\mu x.\,t$, which are finally thunked.

We define the following index terms:

$$M := \underset{b}{\overset{c}{\triangle}} I \qquad\qquad N := \underset{b}{\overset{1}{\triangle}} I\{M + b/b\}$$

Note that M and N both have the index variable c free. M stands for the sum of the sizes of the first c trees in the forest; N denotes the size of the c^{th} tree. We can prove the following (in)equations:

$$\phi; \Phi \vDash H \equiv \underset{b}{\overset{K}{\triangle}} \equiv \sum_{c<K} N \qquad\qquad b, c, \phi; c < K, b < N, \Phi \vDash M + b < H$$

Now we substitute $M+b$ for b in the typing of t, and we weaken using the above inequation:

$$\begin{aligned} &b, c, \phi; c < K, b < N, \Phi; x : [a < I\{M + b/b\}] \cdot \underline{B}_1\{M + b/b\}, \Delta\{M + b/b\} \\ &\qquad\qquad\qquad\qquad \vdash^{\mathsf{c}}_{J\{M+b/b\}} t : \underline{B}_2\{M + b/b\} \end{aligned} \tag{7.1}$$

We also substitute $M + b$ for b in the subtyping between $\underline{B}_2$ and $\underline{B}_1$:

$$\begin{aligned} &a, b, c, \phi; a < I\{M + b/b\}, c < K, b < N \vdash \\ &\qquad \underline{B}_2\{1 + b + \left(\underset{d}{\overset{a}{\triangle}} I\{1 + b + d/b\}\right)/b\}\{M + b/b\} \sqsubseteq \underline{B}_1\{M + b/b\} \end{aligned} \tag{7.2}$$

Note that we can rewrite the above substitution for $\underline{B}_2$:

$$\begin{aligned} &\underline{B}_2\{1 + b + \left(\underset{d}{\overset{a}{\triangle}} I\{1 + b + d/b\}\right)/b\}\{M + b/b\} \\ = {}&\underline{B}_2\{M + 1 + b + \left(\underset{d}{\overset{a}{\triangle}} I\{M + 1 + b + d/b\}\right)/b\} \\ = {}&\underline{B}_2\{M + b/b\}\{1 + b + \left(\underset{d}{\overset{a}{\triangle}} I\{M + b/b\}\{1 + b + d/b\}\right)/b\} \end{aligned}$$

Using (7.1) and (7.2), we apply the rule Fix (with $I := I\{M + b/b\}$ and $H := N = \triangle_b^1 I\{M + b/b\}$, $\underline{B}_i := \underline{B}_i\{M + b/b\}$, and $J := J\{M + b/b\}$):

$$c, \phi; c < K, \Phi; \sum_{b<N} \Delta\{M + b/b\} \vdash^{\mathsf{c}}_{N \dot{-} 1 + \sum_{b<N} J\{M+b/b\}} \mu x.\, t : \underline{B}_2\{M + 0/b\}$$

Now we apply the rule Thunk:

$$\phi; \Phi; \sum_{c<K} \sum_{b<N} \Delta\{M + b/b\} \vdash^{\mathsf{c}}_{K + \sum_{c<K}(N \dot{-} 1 + \sum_{b<N} J\{M+b/b\})}$$
$$\mathsf{thunk}\, \mu x.\, t : [c < K] \cdot \underline{B}_2\{M/b\} = [a < K] \cdot \underline{B}_2\{\mathop{\triangle}\limits_{b}^{a} I/b\} \sqsubseteq A$$

It can be shown that $\phi; \Phi \vdash \sum_{b<K} \Delta \equiv \sum_{c<K} \sum_{b<N} \Delta\{M + b/b\}$. Finally, we show that the weight is correct:

$$\begin{aligned}
\phi; \Phi \vDash H + \sum_{b<H} J &\equiv K + (H \dot{-} K) + \sum_{c<K} \sum_{b<N} J\{M + b/b\} \\
&\equiv K + (\sum_{c<K} N \dot{-} 1) + \sum_{c<K} \sum_{b<N} J\{M + b/b\} \\
&\equiv K + \sum_{c<K} (N \dot{-} 1 + \sum_{b<N} J\{M + b/b\})
\end{aligned}$$ □

We will later also show that this rule is invertible. For the call-by-value translation, we need one more lemma:

Lemma 7.10 (Inversion of Thunk)**.** *Let* $\phi; \Phi; \Gamma \vdash^{\mathsf{v}}_{M} \mathsf{thunk}\, t : [a < 1] \cdot \underline{B}$. *Then there exists an* M' *such that* $\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_{M'} t : \underline{B}\{0/a\}$ *and* $\phi; \Phi \vDash 1 + M' \le M$.

Proof. By inverting the typing, we get:

$$\phi; \Phi \vdash [a < I] \cdot \underline{B}' \sqsubseteq [a < 1] \cdot \underline{B} \qquad a, \phi; a < I, \Phi; \Delta \vdash^{\mathsf{c}}_{K} t : \underline{B}'$$

$$\phi; \Phi \vDash I + \sum_{a<I} K \le M \qquad \phi; \Phi \vdash \Gamma \sqsubseteq \sum_{a<1} \Delta$$

We choose $M' := K\{0/a\}$. Inversion of the subtyping yields $\phi; \Phi \vdash 1 \le I$ and $a, \phi; a < 1, \Phi \vdash \underline{B}' \sqsubseteq \underline{B}$. By substituting 0 for a in this subtyping and the typing of t, we get:

$$\phi; \Phi; \Gamma \sqsubseteq \Delta\{0/a\} \vdash_{K\{0/a\}} t : \underline{B}'\{0/a\} \sqsubseteq \underline{B}\{0/a\}$$ □

Lemma 7.11 (Translation of $\mathsf{d\ell PCF_v}$ typings)**.** *We can translate* $\mathsf{d\ell PCF_v}$ *typings of values* v *and any terms* t.

- *Every* $\mathsf{d\ell PCF_v}$ *value typing* $\phi; \Phi; \Gamma \vdash_K v : \rho$ *can be translated into a* $\mathsf{d\ell PCF_{pv}}$ *value typing* $\phi; \Phi; \Gamma^{\mathsf{v}} \vdash^{\mathsf{v}}_{K} v^{\mathsf{val}} : \rho^{\mathsf{v}}$.

- *Every* $\mathsf{d\ell PCF_{v}}$ *term typing* $\phi; \Phi; \Gamma \vdash_K t : \rho$ *can be translated into a* $\mathsf{d\ell PCF_{pv}}$ *computation typing* $\phi; \Phi; \Gamma^{\mathsf{v}} \vdash^{\mathsf{c}}_K t^{\mathsf{v}} : \mathsf{F}\,\rho^{\mathsf{v}}$.

Proof. By mutual induction on the $\mathsf{d\ell PCF_{v}}$ typings (with subtypings taken care of by Lemma 7.7). We first consider the value cases.

- Case Const: The goal follows from the corresponding $\mathsf{d\ell PCF_{pv}}$ rule.
- Case Lam. We apply the rules Lam and Thunk to the inductive hypothesis:

$$\dfrac{\dfrac{a, \phi; a < I, \Phi; x : \sigma^{\mathsf{v}}, \Delta^{\mathsf{v}} \vdash^{\mathsf{c}}_J t^{\mathsf{v}} : \mathsf{F}\,\tau^{\mathsf{v}}}{a, \phi; a < I, \Phi; \Delta^{\mathsf{v}} \vdash^{\mathsf{c}}_J \lambda x.\, t^{\mathsf{v}} : \sigma^{\mathsf{v}} \multimap \mathsf{F}\,\tau^{\mathsf{v}}}}{\phi; \Phi; \sum_{a<I} \Delta^{\mathsf{v}} \vdash^{\mathsf{v}}_{I + \sum_{a<I} J} \mathsf{thunk}\,\lambda x.\, t^{\mathsf{v}} : [a < I] \cdot (\sigma^{\mathsf{v}} \multimap \mathsf{F}\,\tau^{\mathsf{v}})}$$

- Case Fix. We have the following:

  $$b, \phi; b < H, \Phi; f : [a < I] \cdot A^{\mathsf{v}}, \Delta^{\mathsf{v}} \vdash^{\mathsf{v}}_J \mathsf{thunk}\,\lambda x.\, t^{\mathsf{v}} : [a < 1] \cdot B^{\mathsf{v}}$$

  $$a, b, \phi; a < I, b < H, \Phi \vdash B^{\mathsf{v}}\{0/a, 1 + b + \left(\overset{a}{\underset{d}{\triangle}} I\{1 + b + d/b\}\right)/b\} \sqsubseteq A^{\mathsf{v}}$$

  with $\phi; \Phi \vDash H \equiv \triangle_b^K I$, $\rho = [a < K] \cdot B\{0/a, \triangle_b^a I/b\}$, $\Gamma = \sum_{b<K} \Delta$, and the total weight is $\sum_{b<H} J$. First we invert the thunk typing (using Lemma 7.10) and get:

  $$b, \phi; b < H, \Phi; f : [a < I] \cdot A^{\mathsf{v}}, \Delta^{\mathsf{v}} \vdash^{\mathsf{c}}_{J'} \lambda x.\, t^{\mathsf{v}} : B^{\mathsf{v}}\{0/a\}$$

  with an index term J' such that $b, \phi; b < H, \Phi \vDash 1 + J' \leq J$.

  Now, we apply the admissible rule ThunkFix (Lemma 7.9), and we get:

  $$\phi; \Phi; \sum_{b<H} \Delta^{\mathsf{v}} \vdash^{\mathsf{v}}_{H + \sum_{b<H} J'} \mathsf{thunk}\,\mu f.\, \lambda x.\, t^{\mathsf{v}} : [a < K] \cdot B^{\mathsf{v}}\{0/a, \overset{a}{\underset{b}{\triangle}} I/b\}$$

  We are done now, since $\phi; \Phi \vDash H + \sum_{b<H} J' \equiv \sum_{b<H}(1 + J') \leq \sum_{b<H} J$.

Now we consider the computation cases:

- Case $t = v$ (special case where t is a value). We use the inductive hypothesis and the rule Return:

$$\dfrac{\phi; \Phi; \Gamma^{\mathsf{v}} \vdash^{\mathsf{v}}_K v^{\mathsf{val}} : \rho^{\mathsf{v}}}{\phi; \Phi; \Gamma^{\mathsf{v}} \vdash^{\mathsf{c}}_K \mathsf{return}\, v^{\mathsf{val}} : \mathsf{F}\,\rho^{\mathsf{v}}}$$

- Case App. Using the inductive hypotheses, we can type:

$$\dfrac{\begin{array}{c}\phi; \Phi; \Delta_1^{\mathsf{v}} \vdash^{\mathsf{c}}_{K_1} t_1^{\mathsf{v}} : \mathsf{F}\,([a < 1] \cdot (\sigma^{\mathsf{v}} \multimap \mathsf{F}\,\tau^{\mathsf{v}})) \\ \phi; \Phi; \Delta_2^{\mathsf{v}} \vdash^{\mathsf{c}}_{K_2} t_2^{\mathsf{v}} : \mathsf{F}\,\sigma^{\mathsf{v}}\{0/a\}\end{array} \quad \dfrac{\dfrac{}{\phi; \Phi; x : \cdots \vdash^{\mathsf{c}}_0 \mathsf{force}\,x : (\sigma^{\mathsf{v}} \multimap \mathsf{F}\,\tau^{\mathsf{v}})\{0/a\}} \quad \dfrac{}{\phi; \Phi; y : \cdots \vdash^{\mathsf{v}}_0 y : \sigma^{\mathsf{v}}\{0/a\}}}{\phi; \Phi; x : [a < 1] \cdot (\sigma^{\mathsf{v}} \multimap \mathsf{F}\,\tau^{\mathsf{v}}), y : \sigma^{\mathsf{v}}\{0/a\} \vdash^{\mathsf{c}}_0 (\mathsf{force}\,x)\, y : \mathsf{F}\,\tau^{\mathsf{v}}\{0/a\}}}{\phi; \Phi; \Delta_1^{\mathsf{v}} \uplus \Delta_2^{\mathsf{v}} \vdash^{\mathsf{c}}_{K_1 + K_2} \mathsf{bind}\, x \leftarrow t_1^{\mathsf{v}},\, y \leftarrow t_2^{\mathsf{v}}\, \mathsf{in}\, (\mathsf{force}\,x)\, y : \mathsf{F}\,\tau^{\mathsf{v}}\{0/a\}}$$

- Cases Ifz, Succ, and Pred: As in Lemma 7.5.
- Case Var: The goal follows from the $\mathsf{d\ell PCF_{pv}}$ rules Return and Var. □

7.5 Soundness of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$

The proof of soundness of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ is very similar to that of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$. We will prove subject reduction; the weight decreases after every forcing step (i.e. $\mathsf{force}\,\mathsf{thunk}\,t \succ_1 t$). First we show that we can split value typings.

Lemma 7.12 (Splitting)**.** *Let* $\phi; \Phi; \emptyset \vdash^{\mathsf{v}}_{M} v : A_1 \uplus A_2$ *for a closed value* v*. Then* $\phi; \Phi; \emptyset \vdash^{\mathsf{v}}_{N_1} v : A_1$ *and* $\phi; \Phi; \emptyset \vdash^{\mathsf{v}}_{N_2} v : A_2$ *for index terms* N_1 *and* N_2 *with* $\phi; \Phi \vDash N_1 + N_2 \leq M$*.*

Proof. By case analysis on the value typing.

- Case $v = \underline{k}$. Then $A_i = \mathsf{Nat}[I_i]$ for some I with $\phi; \Phi \vDash k = I_1 = I_2$. Then we can type $\underline{k}$ twice as $\mathsf{Nat}[I_1]$, and $\mathsf{Nat}[I_1] \uplus \mathsf{Nat}[I_1] = \mathsf{Nat}[I_1]$.

- Case $v = \mathsf{thunk}\,t$; the typing has the following shape:

$$\frac{\begin{array}{c} a, \phi; a < I, \Phi; \emptyset \vdash^{\mathsf{c}}_{K} t : \underline{B} \\ \phi; \Phi \vdash [a < I] \cdot \underline{B} \sqsubseteq A_1 \uplus A_2 \qquad \phi; \Phi \vDash I + \sum_{a<I} K \leq M \end{array}}{\phi; \Phi; \emptyset \vdash^{\mathsf{v}}_{M} \mathsf{thunk}\,t : A_1 \uplus A_2}$$

By definition of $\uplus$, A_1 and A_2 must have the following shape:

$$\begin{aligned} A_1 &= [a < I_1] \cdot \underline{B}' \\ A_2 &= [a < I_2] \cdot \underline{B}'\{I_1 + a/a\} \\ A_1 \uplus A_2 &= [a < I_1 + I_2] \cdot \underline{B}' \end{aligned}$$

Because $\phi; \Phi \vdash [a < I] \cdot \underline{B} \sqsubseteq A_1 \uplus A_2$, we have $\phi; \Phi \vDash I_1 + I_2 \leq I$ and $a, \phi; a < I_1 + I_2 \vdash \underline{B} \sqsubseteq B'$. Now we define $N_1 := I_1 + \sum_{a<I_1} K$ and $N_2 := I_2 + \sum_{a<I_2} K\{I_1 + a/a\}$. Then it obviously holds that $\phi; \Phi \vDash N_1 + N_2 \leq M$. Because $a < I_1$ implies $a < I$, we can derive the first typing using subsumption:

$$\frac{a, \phi; a < I_1, \Phi; \emptyset \vdash^{\mathsf{c}}_{K} t : \underline{B} \qquad \phi; \Phi \vdash [a < I_1] \cdot \underline{B} \sqsubseteq [a < I_1] \cdot \underline{B}' = A_1}{\phi; \Phi \vdash^{\mathsf{v}}_{N_1} \mathsf{thunk}\,t : A_1}$$

The second typing is derived by substituting $I_1 + a$ for a:

$$\frac{\begin{array}{c} a, \phi; a < I_2, \Phi; \emptyset \vdash^{\mathsf{c}}_{K\{I_1+a/a\}} t : \underline{B}\{I_1 + a/a\} \\ \phi; \Phi \vdash [a < I_2] \cdot \underline{B}\{I_1 + a/a\} \sqsubseteq [a < I_2] \cdot \underline{B}'\{I_1 + a/a\} = A_2 \end{array}}{\phi; \Phi \vdash^{\mathsf{v}}_{N_2} \mathsf{thunk}\,t : A_2}$$

□

Note that it already pays off here that we use CBPV here; in the corresponding $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ proof, we also need to consider the fixpoint case, which requires splitting recursion forests into two forests. Here, the only non-trivial case is thunk, which is similar to the λ case in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$.

Next, we also need to split value typings of bounded modal sums:

Lemma 7.13 (Parametric splitting)**.** *Let* $\phi; \Phi; \emptyset \vdash^{\mathsf{v}}_{M} v : \sum_{c<J} A$*. Then exists an index term* N *(with* c *as a free variable) such that* $c, \phi; c < J, \Phi; \emptyset \vdash^{\mathsf{v}}_{N} v : A$ *and* $\phi; \Phi \vDash \sum_{c<J} N \leq M$*.*

Proof. Again, by case analysis on v; the only interesting case is $v = \mathsf{thunk}\, t$. Inverting the value typing of $\mathsf{thunk}\, t : \sum_{c<J} A$ yields:

$$a, \phi; a < I, \Phi; \emptyset \vdash^{\mathsf{c}}_{K} t : \underline{B} \qquad \phi; \Phi \vdash [a < I] \cdot \underline{B} \sqsubseteq \textstyle\sum_{c<J} A \qquad \phi; \Phi \vDash I + \textstyle\sum_{a<I} K \leq M$$

By definition of bounded sum, we have:

$$\begin{aligned} A &= [b < L] \cdot \underline{B}'\{b + \textstyle\sum_{d<c} L\{d/c\}/a\} \\ \textstyle\sum_{c<J} A &= [a < \textstyle\sum_{c<J} L] \cdot \underline{B}' \end{aligned}$$

Note that c is free in A; L is the size of the c^{th} 'component' of the sum.

Now, we split M into J parts: $N := L + \sum_{b<L} K\{b + \sum_{d<c} L\{d/c\}/a\}$, and we show:

$$\begin{aligned} \phi; \Phi \vDash \sum_{c<J} N &\leq \sum_{c<J} L + \sum_{c<J} \sum_{b<L} K\{b + \sum_{d<c} L\{d/c\}/a\} \\ &\leq I + \sum_{a<\sum_{c<J} L} K \leq I + \sum_{a<I} K \leq M \end{aligned}$$

Finally, we apply the substituting $\theta := \{b + \sum_{b<c} L\{d/c\}/a\}$ to the typing of t:

$$\frac{a, c, \phi; a < L, c < J, \Phi; \emptyset \vdash^{\mathsf{c}}_{K\theta} t : \underline{B}\theta \qquad c, \phi; c < J, \Phi \vdash [a < L] \cdot \underline{B}\theta \sqsubseteq [a < L] \cdot \underline{B}'\theta = A}{c, \phi; c < J, \Phi \vdash^{\mathsf{v}}_{N} \mathsf{thunk}\, t : A}$$

□

Using (parametric) splitting, it is easy to show that substitution of a value preserves typings. However, note that we have two substitution lemmas: one for value typings and one for computation typings. In both cases, we substitute a value for a variable – either in a value term or in a computation term.

Lemma 7.14 (Substitution)**.** *Let* $\phi; \Phi; \emptyset \vdash^{\mathsf{v}}_{M_2} v : A_x$*. Then:*

- *If* $\phi; \Phi; x : A_x, \Gamma \vdash^{\mathsf{c}}_{M_1} t : \underline{B}$*, then* $\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_{M_1+M_2} t\{v/x\} : \underline{B}$*.*
- *If* $\phi; \Phi; x : A_x, \Gamma \vdash^{\mathsf{v}}_{M_1} u : A$*, then* $\phi; \Phi; \Gamma \vdash^{\mathsf{v}}_{M_1+M_2} u\{v/x\} : A$*.*

Proof. By mutual induction on the typing of t or u.

- Case $t\, u$ (where u is a value). We have:

$$\phi; \Phi; x : A_1, \Delta_1 \vdash^{\mathsf{c}}_{K_1} t : A \multimap \underline{B} \qquad \phi; \Phi; x : A_2, \Delta_2 \vdash^{\mathsf{v}}_{K_2} u : A$$

$$\phi; \Phi \vDash K_1 + K_2 \leq M_1 \qquad \phi; \Phi \vdash x : A_x, \Gamma \sqsubseteq (x : A_1, \Delta_1) \uplus (x : A_2, \Delta_2)$$

Using the splitting lemma (Lemma 7.12), we obtain two typings of v:

$$\phi;\Phi;\emptyset \vdash^{\mathsf{v}}_{M_{21}} v : A_1 \qquad \phi;\Phi;\emptyset \vdash^{\mathsf{v}}_{M_{22}} v : A_2 \qquad \phi;\Phi \vDash M_{21}+M_{22} \le M_2$$

Using the inductive hypotheses, we type:

$$\frac{\begin{array}{cc}\phi;\Phi;\Delta'_1 \vdash^{\mathsf{c}}_{K_1+M_{21}} t\{v/x\} : A \multimap \underline{B} & \phi;\Phi;\Delta'_2 \vdash^{\mathsf{v}}_{K_2+M_{22}} u\{v/x\} : A \\ \phi;\Phi \vdash \Gamma \sqsubseteq \Delta'_1 \uplus \Delta'_2 & \phi;\Phi \vDash (K_1+M_{21})+(K_2+M_{22}) \le M_1+M_2\end{array}}{\phi;\Phi;\Gamma \vdash_{M_1+M_2} (t\,u)\{v/x\} : \underline{B}}$$

- Case $u = \mathsf{thunk}\,t$. We have:

  $$a,\phi; a<I,\Phi; x:A,\Delta \vdash^{\mathsf{c}}_{K} t : \underline{B} \qquad \phi;\Phi \vdash x : A_x, \Gamma \sqsubseteq \textstyle\sum_{a<I}(x:A,\Delta)$$

  $$\phi;\Phi \vDash I + \textstyle\sum_{a<I} K \le M_1 \qquad \phi;\Phi \vdash [a<I]\cdot\underline{B} \sqsubseteq A$$

  Using parametric splitting (Lemma 7.13), we get:

  $$a,\phi; a<I,\Phi;\emptyset \vdash^{\mathsf{v}}_{M'_2} v : A \qquad \phi;\Phi \vDash \textstyle\sum_{a<I} M'_2 \le M_2$$

  We arrive at the goal by using the inductive hypothesis and Thunk.

- Case $t = \mathsf{bind}\,y \leftarrow t_1 \,\mathsf{in}\, t_2$, where $x \neq y$. We have:

  $$\phi;\Phi; x:A_1,\Delta_1 \vdash^{\mathsf{c}}_{K_1} t_1 : \mathsf{F}\,A \qquad \phi;\Phi; y:A, x:A_2,\Delta_2 \vdash^{\mathsf{c}}_{K_2} t_2 : \underline{B}$$

  As in the application case, we use splitting and the inductive hypotheses to derive:

  $$\frac{\phi;\Phi;\Delta_1 \vdash^{\mathsf{c}}_{K_1+M_{21}} t_1\{v/x\} : \mathsf{F}\,A \qquad \phi;\Phi; y:A,\Delta_2 \vdash^{\mathsf{c}}_{K_2+M_{22}} t_2\{v/x\} : \underline{B}}{\phi;\Phi;\Gamma \vdash_{M_1+M_2} \mathsf{bind}\,y \leftarrow t_1\{v/x\} \,\mathsf{in}\, t_2\{v/x\} : \underline{B}}$$

- The cases constant, variable and ifz are similar to the corresponding cases in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ (see Lemma 5.8). The remaining cases (Succ, Pred, force, return, and μ) are also similar. □

Now we can prove subject reduction. In the following, we show the interesting cases as lemmas. First, we show that a force step reduces the weight by one.

Lemma 7.15 (Subject reduction, case thunk)**.** *Let* $\phi;\Phi;\emptyset \vdash^{\mathsf{c}}_{M} \mathsf{force}\,\mathsf{thunk}\,t : \underline{B}$. *Then* $\phi;\Phi;\emptyset \vdash^{\mathsf{c}}_{M'} t : \underline{B}$ *with* $\phi;\Phi \vDash 1+M' \le M$.

Proof. By inversion of the force typing, we get: $\phi;\Phi;\emptyset \vdash^{\mathsf{c}}_{M} \mathsf{thunk}\,t : [a<1]\cdot\underline{B}'$ such that $\phi;\Phi \vdash \underline{B}'\{0/a\} \equiv \underline{B}$. Now, using Lemma 7.10, we get $\phi;\Phi;\emptyset \vdash^{\mathsf{c}}_{M'} t : \underline{B}'\{0/a\}$ with $\phi;\Phi \vDash 1+M' \le M$. □

Lemma 7.16 (Subject reduction, case λ-application)**.** *Let* $\phi;\Phi;\emptyset \vdash^{\mathsf{c}}_{M} (\lambda x.\,t)\,v : \underline{B}$. *Then* $\phi;\Phi;\emptyset \vdash^{\mathsf{c}}_{M} t\{v/x\} : \underline{B}$.

Proof. By inversion, we get:

$$\phi;\Phi;x:A\vdash^{\mathsf{c}}_{M_1} t:\underline{B} \qquad\qquad \phi;\Phi;\emptyset\vdash^{\mathsf{v}}_{M_2} v:A \qquad\qquad \phi;\Phi\vDash M_1+M_2\leq M$$

The goal follows from substitution (Lemma 7.14). □

The cases for the steps of terms like $\mathsf{ifz}\,\underline{0}\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2$ and $\mathsf{calc}\,x\leftarrow\mathsf{Succ}(\underline{n})\,\mathsf{in}\,t$ can be shown similarly. In fact, these cases are similar to their counterparts in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$. The only non-trivial case is the fixpoint case:

Lemma 7.17 (Subject reduction, case fixpoint unrolling)**.** *Let* $\phi;\Phi;\emptyset\vdash^{\mathsf{c}}_{M}\mu x.\,t:\underline{B}$*. Then* $\phi;\Phi;\emptyset\vdash^{\mathsf{c}}_{M} t\{\mathsf{thunk}\,\mu x.\,t/x\}:\underline{B}$*.*

Proof. We first invert the typing of $\mu x.\,t$:

$$b,\phi;b<H,\Phi;x:[a<I]\cdot\underline{B}_1\vdash^{\mathsf{c}}_{J} t:\underline{B}_2 \tag{7.3}$$

$$a,b,\phi;a<I,b<H,\Phi\vdash\underline{B}_2\{1+b+\left(\mathop{\triangle}\limits_{c}^{a} I\{1+b+c/b\}\right)/b\}\sqsubseteq\underline{B}_1 \tag{7.4}$$

$$\phi;\Phi\vDash H\equiv\mathop{\triangle}\limits_{b}^{1} I \qquad\qquad \phi;\Phi\vdash\underline{B}_2\{0/b\}\sqsubseteq\underline{B} \qquad\qquad \phi;\Phi\vDash(H\mathbin{\dot{-}}1)+\textstyle\sum_{b<H}J\leq M$$

Now we substitute 0 for b in (7.3):

$$\phi;\Phi;x:[a<I\{0/b\}]\cdot\underline{B}_1\{0/b\}\vdash^{\mathsf{c}}_{J\{0/b\}} t:\underline{B}_2\{0/b\}$$

Remember that I describes the recursion tree. This means that $I\{0/b\}$ is the number of children of the root – this is the number how often x is forced (and usually recursively called) at the first application of $t\{\mathsf{thunk}\,\mu x.\,t/x\}$. The type $[a<I]\cdot\underline{B}_1\{0/b\}$ is the type that x is expected to have at the root. This means that we are done (using Lemma 7.14), if we can type:

$$\phi;\Phi;\emptyset\vdash_{M^*}\mathsf{thunk}\,\mu x.\,t:[a<I\{0/b\}]\cdot\underline{B}_1\{0/b\} \tag{7.5}$$

with an M^* such that $\phi;\Phi\vDash J\{0/b\}+M^*\leq M$.

We define the following index terms and types:

$$K^*:=I\{0/b\} \qquad I^*:=I\{1+b/b\} \qquad H^*:=\triangle_b^{K^*} I^*=H\mathbin{\dot{-}}1 \qquad \underline{B}^*_{1,2}:=\underline{B}_{1,2}\{1+b/b\}$$

$$J^*:=J\{1+b/b\} \qquad\qquad M^*:=H^*+\textstyle\sum_{b<H^*}J^*$$

By substituting $1+b$ for b in (7.3) and (7.4), we get:

$$b,\phi;b<H^*,\Phi;x:[a<I^*]\cdot\underline{B}^*_1\vdash^{\mathsf{c}}_{J^*} t:\underline{B}^*_2$$

$$a,b,\phi;a<I^*,b<H^*,\Phi\vdash\underline{B}_2\{1+b+\left(\mathop{\triangle}\limits_{c}^{a} I\{1+b+c/b\}\right)/b\}\{1+b/b\}\sqsubseteq\underline{B}^*_1$$

The above substitution of $\underline{B}_2$ can be rewritten:

$$\underline{B}_2\{1+b+\left(\mathop{\triangle}\limits_{c}^{a} I\{1+b+c/b\}\right)/b\}\{1+b/b\}=\underline{B}^*_2\{1+b+\left(\mathop{\triangle}\limits_{c}^{a} I^*\{1+b+c/b\}\right)/b\}$$

Now, using ThunkFix (Lemma 7.9), the only obligation left to show (7.5) is:

$$a, \phi; a < I\{0/b\}, \Phi \vdash \underline{B}_2^*\{\overset{a}{\underset{b}{\triangle}} I^*/b\} = \underline{B}_2\{1+0+\overset{a}{\underset{c}{\triangle}} I\{1+0+c/b\} \sqsubseteq \underline{B}_1\{0/b\}$$

This subtyping follows from (7.4) by substituting 0 for b.

Finally, we have to show that the overall weight is correct:

$$\begin{aligned}\phi; \Phi \vDash J\{0/b\} + M^* = H^* + J\{0/b\} + \sum_{b<H^*} J^* &\equiv (H \mathbin{\dot{-}} 1) + J\{0/b\} + \sum_{b<H-1} J\{1+b/b\} \\ &\equiv (H \mathbin{\dot{-}} 1) + \sum_{b<H} J \le M \quad \square\end{aligned}$$

All together, we are ready to prove subject reduction.

Theorem 7.18 (Subject reduction of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$)**.** *Let $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{M} t : \underline{B}$, and let $t \succ_i t'$ be a step. Then there exists an M' such that $\phi; \Phi; \emptyset \vdash_{M'} t' : \rho$ and $\phi; \Phi \vDash i + M' \le M$.*

Proof. By induction on the small step. The context reduction cases are trivial. The interesting head reduction cases are the Lemmas 7.15 to 7.17. The other head reduction cases are trivial. ☐

We prove soundness of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ as we did for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$. Therefore, we first define a size function on computations.

Definition 7.19 (Size of a computation term)**.**

$$\begin{aligned} |\lambda x.\, t| &:= 1 + |t| & |\mathsf{bind}\, x \leftarrow t_1 \,\mathsf{in}\, t_2| &:= 1 + |t_1| + |t_2| \\ |\mu x.\, t| &:= 1 + |t| & |\mathsf{calc}\, x \leftarrow \mathsf{Succ}(v) \,\mathsf{in}\, t| &:= 1 + |t| \\ |t\, v| &:= 1 + |t| & |\mathsf{calc}\, x \leftarrow \mathsf{Pred}(v) \,\mathsf{in}\, t| &:= 1 + |t| \\ |\mathsf{force}\, v| := |\mathsf{return}\, v| &:= 1 & |\mathsf{ifz}\, v \,\mathsf{then}\, t_1 \,\mathsf{else}\, t_2| &:= 1 + |t_1| + |t_2| \end{aligned}$$

Note that the size of a computation does not change if we substitute a value for a variable. This is crucial in the β-substitution steps (i.e. $(\lambda x.\, t)\, v \succ_0 t\{v/x\}$ and $\mu x.\, t \succ_0 t\{\mathsf{thunk}\, \mu x.\, t/x\}$), which (in contrast to $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$) do not decrement the weight.

Corollary 7.20 (Soundness of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$)**.** *Let $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{k} t : \underline{B}$. Then there exists a terminal computation T and a number k', such that $t \Downarrow_{k'} T$ and $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{k-k'} T : \underline{B}$.*

Proof. We prove the lemma by well-founded induction on the lexicographical order of k and the size of t. If t is a terminal computation (that is, $t = \mathsf{return}\, v$ or $t = \lambda x.\, t'$), we are done. Otherwise, let $t \succ_i t'$ be the first step of t. Using Theorem 7.18, we get a k' such that $\emptyset; \emptyset \vDash k' + i \le k$ and $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{k'} t' : \tau$. Now, we do a case distinction on the cost i of the step. If $i = 1$ (i.e. the step was a forcing step), we can apply the inductive hypothesis on t' since $k' - i < k$. Otherwise ($i = 0$), we know that the size of t' is smaller than the size of t, so we can also apply the inductive hypothesis on t'. ☐

Corollary 7.21 (Soundness of $\mathsf{d\ell PCF_{pv}}$ programs)**.** *Let* $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_k t : \mathsf{F}\,\mathsf{Nat}[n]$*. Then there is a* $k' \leq k$ *such that* $t \Downarrow_{k'} \mathsf{return}\,\underline{n}$*.*

Proof. By the above theorem, t evaluates to a term t' of type $\mathsf{F}\,\mathsf{Nat}[n]$ in $k' \leq k$ steps. Because this term is a *closed terminal computation*, it must be equal to $\mathsf{return}\,\underline{n}$. ☐

We can of course derive the same soundness corollaries for $\mathsf{d\ell PCF_{pv}}$ precise typings, as for $\mathsf{d\ell PCF_{v}}$ (see Section 5.4):

Theorem 7.22 (Precise subject reduction of $\mathsf{d\ell PCF_{pv}}$)**.** *Let* $\phi; \Phi; \emptyset \vdash_M t : \rho$ *be a precise typing, and let* $t \succ_i t'$ *be a step. Then there exists an index term* M' *such that* $\phi; \Phi; \emptyset \vdash_{M'} t' : \rho$ *and* $\phi; \Phi \vDash i + M' \equiv M$*.*

Corollary 7.23 (Precise soundness of $\mathsf{d\ell PCF_{pv}}$)**.** *Let* $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_K t : \underline{B}$ *be a precise typing and* $t \Downarrow_k T$*. Then* $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{K \dot{-} k} T : \underline{B}$ *is a precise typing.*

Corollary 7.24 (Precise soundness of $\mathsf{d\ell PCF_{pv}}$)**.** *Let* $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_K t : \underline{B}$ *be a precise typing and* $t \Downarrow_k T$*, and let* $\underline{B}$ *be disposable. Then* $\vDash K \equiv k$ *and* $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_0 T : \underline{B}$*.*

Corollary 7.25 (Precise soundness of $\mathsf{d\ell PCF_{pv}}$ for programs)**.** *Theorem 4.5 (2) holds for* $\mathsf{d\ell PCF_{pv}}$*: Let* $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_K t : \mathsf{F}\,\mathsf{Nat}[I]$ *be a precise typing and* $t \Downarrow_k \mathsf{return}\,\underline{n}$*. Then* $\vDash K \equiv k$ *and* $\vDash I \equiv n$*.*

7.5.1 Deriving soundness of $\mathsf{d\ell PCF_{n}}$ and $\mathsf{d\ell PCF_{v}}$

We already have everything we need to derive soundness of $\mathsf{d\ell PCF_{n}}$ and $\mathsf{d\ell PCF_{v}}$ from soundness of $\mathsf{d\ell PCF_{pv}}$. We have already proved soundness of $\mathsf{d\ell PCF_{v}}$, but it is possible to derive the result from $\mathsf{d\ell PCF_{pv}}$.

Corollary 7.26 (Soundness of $\mathsf{d\ell PCF_{v}}$ programs)**.** *Let* $\emptyset; \emptyset; \emptyset \vdash_k t : \mathsf{Nat}[n]$*. Then there is a* $k' \leq k$ *such that* $t \Downarrow_{k'} \underline{n}$ *– i.e.* t *does* k' β*-substitution steps in the* CBV *semantics.*

Proof. First, we translate the $\mathsf{d\ell PCF_{v}}$ typing to a $\mathsf{d\ell PCF_{pv}}$ typing using Lemma 7.11:

$$\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_k t : \mathsf{F}\,\mathsf{Nat}[n]$$

By soundness of $\mathsf{d\ell PCF_{pv}}$ (Corollary 7.21), we have that $t^{\mathsf{v}} \Downarrow_{k'} \mathsf{return}\,\underline{n}$ – i.e. t^{v} needs $k' \leq k$ forcing steps in the CBPV semantics. Using Lemma 2.23, we have that $t \Downarrow_{k'} \underline{n}$. ☐

Corollary 7.27 (Soundness of $\mathsf{d\ell PCF_{n}}$ programs)**.** *Let* $\emptyset; \emptyset; \emptyset \vdash_k t : \mathsf{Nat}[n]$*. Then there is a* $k' \leq k$ *such that* $t \Downarrow_{k'} \underline{n}$ *– i.e., in the environment semantics,* t *evaluates to* $\underline{n}$ *after* k' *variable lookups.*

Proof. First, we translate the $\mathsf{d\ell PCF_{n}}$ typing to a $\mathsf{d\ell PCF_{pv}}$ typing using Lemma 7.5:

$$\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_k t : \mathsf{F}\,\mathsf{Nat}[n]$$

By soundness of $\mathsf{d\ell PCF_{pv}}$ (Corollary 7.21), we have that $t^{\mathsf{n}} \Downarrow_{k'} \mathsf{return}\,\underline{n}$ – i.e. t^{n} needs $k' \leq k$ forcing steps in the CBPV semantics. This big-step execution can be translated into an environment big-step execution $\langle t^{\mathsf{n}}; \emptyset \rangle \Downarrow_{k'} tc$ with $\mathit{unf}(tc) = \mathsf{return}\,\underline{n}$. Using Lemma 2.16, we have that $\langle t; \emptyset \rangle \Downarrow_{k'} tc_{\mathsf{CBN}}$ and $tc^{\mathsf{n}}_{\mathsf{CBN}} = tc$, and thus $\mathit{unf}(tc_{\mathsf{CBN}}) = \underline{n}$. Thus, by definition, $t \Downarrow_{k'} \underline{n}$. ☐

7.6 Completeness of $\mathsf{d\ell PCF_{pv}}$

The $\mathsf{d\ell PCF_{pv}}$ completeness proof also follows the same structure as in the call-by-value case in Section 5.5. The proofs of the joining lemmas are simpler since we do not need to join recursion trees.

7.6.1 Preliminaries

We first define *precise typings* with skeletons; we write $\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_{K} t : A @ cs$ (and $\phi; \Phi; \Gamma \vdash^{\mathsf{v}}_{K} v : A @ vs$) for precise computation (or value) typings that have the skeleton cs (or vs), respectively. Recall that a typing is precise if only bi-directional subtyping ($\equiv$) is allowed and the weight may not be increased.[1] Additionally, only *disposable types* (e.g. $\mathsf{Nat}[I]$ and $[a < 0] \cdot \underline{B}$) are allowed in contexts of closed programs.

Binary and bounded sums of value typings can be constructed in exactly the same way as in the call-by-value case, since the syntax of $\mathsf{d\ell PCF_{v}}$ modal types is similar to the syntax of $\mathsf{d\ell PCF_{pv}}$ value types.

As in Section 5.5.1, we have to define *skeletons* for simple CBPV typings.

Definition 7.28 (CBPV skeletons)**.** Computation and value skeletons are labelled trees, where each node is labelled with the name of the corresponding CBPV simple typing rule. For the rule App, we additionally store the CBPV value type A.

$$\begin{aligned} \text{Value skel.:} \quad & vs ::= \mathsf{Var} \mid \mathsf{Const} \mid \mathsf{Thunk}\, cs \\ \text{Comp. skel.:} \quad & cs ::= \mathsf{Lam}\, cs \mid \mathsf{Fix}\, cs \mid \mathsf{App}\, A\, cs\, vs \mid \mathsf{Ifz}\, vs\, cs_1\, cs_2 \mid \mathsf{Return}\, vs \\ & \quad \mid \mathsf{Bind}\, cs_1\, cs_2 \mid \mathsf{Force}\, vs \mid \mathsf{Succ}\, vs\, cs \mid \mathsf{Pred}\, vs\, cs \end{aligned}$$

A simple CBPV typing can be assigned a skeleton by ignoring the contexts. Again, it can be shown that two simple CBPV typing derivations for $\Gamma \vdash^{\mathsf{c}} t : \underline{B}$ are equal if and only if their skeletons are equal. It is also possible to define subject reduction on CBPV computation skeletons: $(t; cs) \succ_i (t'; cs')$. This is completely analogous to the CBV case in Section 5.5.1.

7.6.2 Converse substitution

In order to prove converse substitution, we first prove the joining lemmas. We use the same technique as for $\mathsf{d\ell PCF_{v}}$.

Lemma 7.29 (Case distinction typing lemma)**.** *Let C be a constraint. Let $\Phi_i; \Gamma_i \vdash^{\mathsf{c}}_{M_i} t : \underline{B}_i @ cs$ be two computation typings ($i = 1, 2$). Assume that the* CBPV *structures of $\underline{B}_i$ and $\Gamma_i(x)$ (for all variables x in the domain of Γ_1 and Γ_2) are equal. Then we can construct a typing for:*

$$\mathsf{if}\, C\, \mathsf{then}\, \Phi_1\, \mathsf{else}\, \Phi_2; \mathsf{if}\, C\, \mathsf{then}\, \Gamma_1\, \mathsf{else}\, \Gamma_2 \vdash_{\mathsf{if}\, C\, \mathsf{then}\, M_1\, \mathsf{else}\, M_2} t : \mathsf{if}\, C\, \mathsf{then}\, \underline{B}_1\, \mathsf{else}\, \underline{B}_2 @ cs$$

The same holds for subtyping and value typings.

---

[1] In particular, it is not allowed to weaken a finite weight to the undefined/infinite weight $\bot$.

Again, this lemma is refined to make it useful for the binary joining lemma:

Corollary 7.30 (Refined case distinction typing lemma). *Let* $a, \phi; a < I_1, \Phi; \Gamma_1 \vdash^{\mathsf{c}}_{M_1} t : \underline{B} \mathbin{@} cs$ *and* $a, \phi; a < I_2, \Phi; \Gamma_2 \vdash^{\mathsf{c}}_{M_2} t : \underline{B}\{a + I_1/a\} \mathbin{@} cs$*. Then:*

$$a, \phi; a < I_1 + I_2, \Phi; \mathsf{if}\ a < I_1\ \mathsf{then}\ \Gamma_1\ \mathsf{else}\ \Gamma_2\{a - I_1/a\} \vdash^{\mathsf{c}}_{\mathsf{if}\ a<I_1\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2\{a-I_1/a\}} t : \underline{B} \mathbin{@} cs$$

Lemma 7.31 (Joining). *Let* v *be a closed value. Given two value typings* $\phi; \Phi; \emptyset \vdash^{\mathsf{v}}_{M_i} v : A_i \mathbin{@} vs$ *with the same skeleton* ($i = 1, 2$)*, we can derive value types* $A = A'_1 \uplus A'_2$ *with* $\phi; \Phi \vdash A'_i \equiv A_i$*, and derive a typing* $\phi; \Phi; \emptyset \vdash_{M_1+M_2} v : A \mathbin{@} vs$*.*

Proof (sketch). If $v = \underline{n}$, then $A = \mathsf{Nat}[I_i]$ with $\phi; \Phi; \emptyset \vDash I_1 = I_2$. Let $A := \mathsf{Nat}[I_1] = \mathsf{Nat}[I_1] \uplus \mathsf{Nat}[I_1]$. We can again type $\phi; \Phi; \emptyset \vdash^{\mathsf{v}}_0 \underline{n} : A$.

Let us now examine the interesting case, $v = \mathsf{thunk}\, t$. We invert both typings ($i = 1, 2$):

$$\frac{a, \phi; a < I_i, \Phi; \emptyset \vdash^{\mathsf{c}}_{K_i} t : \underline{B}}{\phi; \Phi; \emptyset \vdash^{\mathsf{v}}_{M_i := I_i + \sum_{a<I_i} K_i} \mathsf{thunk}\, t : A_i = [a < I_i] \cdot \underline{B}_i}$$

The goal follows from Corollary 7.30 and the rule Thunk. ☐

Lemma 7.32 (Parametric joining). *Let* $c, \phi; c < L, \Phi; \emptyset \vdash^{\mathsf{c}}_M v : A \mathbin{@} vs$*. Then there exists an* A' *with* $\phi; \Phi \vdash A' \equiv A$ *and* $\phi; \Phi; \emptyset \vdash^{\mathsf{v}}_{\sum_{a<I} M} v : \sum_{c<L} A' \mathbin{@} vs$*.*

Proof (sketch). The value v could be a constant (in which case the proof is trivial) or $v = \mathsf{thunk}\, t$. By inverting the typing of $\mathsf{thunk}\, t$, we get:

$$a, c, \phi; a < I, c < L, \Phi; \emptyset \vdash^{\mathsf{c}}_K t : \underline{B} \qquad A = [a < I] \cdot \underline{B} \qquad M = I + \textstyle\sum_{c<L} K$$

Exactly as in the Lam case of the proof of Lemma 5.44 (see Appendix A.1.1), we construct the sum over A, using the function $\mathit{findSlot}_c\, L\, I$. ☐

Lemma 7.33 (Converse substitution). *Let* v *be a closed* CBPV *value. Assume simple* CBPV *typings* $x : \hat{A}_x, (\!|\Gamma|\!) \vdash^{\mathsf{c}} t : (\!|\underline{B}|\!) \mathbin{@} s_1$ *and* $\emptyset \vdash^{\mathsf{v}} v : \hat{A}_x \mathbin{@} s_2$ *for a closed value* v*. Furthermore, assume a* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ *typing* $\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_M t\{v/x\} : \underline{B} \mathbin{@} s'$*, where* $s' = \mathit{subst}(x; t; s_1; s_2)$*. Then there exist index terms* N_1 *and* N_2*, and a value type* A*, such that:*

$$\phi; \Phi; x : A, \Gamma \vdash^{\mathsf{c}}_{N_1} t : \underline{B} \mathbin{@} s_1 \qquad \phi; \Phi; \emptyset \vdash^{\mathsf{v}}_{N_2} v : A \mathbin{@} s_2 \qquad \phi; \Phi \vDash N_1 + N_2 \equiv M \qquad (\!|A|\!) = \hat{A}_x$$

The same holds for values u *instead of computations* t*.*

Proof (sketch). This is proved in the same way as Lemma 5.45. ☐

7.6.3 Subject expansion

Lemma 7.34 (Subject expansion of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$)**.** *Let* $(t; cs) \succ_i (t'; cs')$*. Assume a* CBPV *typing* $\emptyset \vdash^{\mathsf{c}} t : \llparenthesis \underline{B} \rrparenthesis \mathbin{@} sc$*, and a* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ *typing* $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{M} t' : \underline{B} \mathbin{@} sc'$*. Then we can type* $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{i+M} t : \underline{B} \mathbin{@} sc$*.*

Proof. Induction on the small-step semantics. The only non-trivial head reduction steps are the following:

- $(\lambda x.\, t)\, v \succ_0 t\{v/x\}$: Lemma 7.35.
- $\mu x.\, t \succ_0 t\{\mathsf{thunk}\, \mu x.\, t/x\}$: Lemma 7.37.
- $\mathsf{force}\, \mathsf{thunk}\, t \succ_1 t$: By applying the rules Thunk and Force (which increases the weight by one). □

Lemma 7.35 (Subject expansion (application case))**.** *Let* $\emptyset \vdash^{\mathsf{c}} (\lambda x.\, t)\, v : \llparenthesis \underline{B} \rrparenthesis \mathbin{@} cs$ *be a simple* CBPV *typing and let* cs' *be the successor skeleton of this typing. Assume the* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ *typing* $\phi; \Phi; \emptyset \vdash_{M} t\{v/x\} : \underline{B} \mathbin{@} cs'$*. Then we can type* $\phi; \Phi; \emptyset \vdash_{M} (\lambda x.\, t)\, v : \underline{B} \mathbin{@} cs$*.*

Proof. By inverting the simple CBPV typing, we get:

$$\emptyset \vdash^{\mathsf{c}} \lambda x.\, t : \hat{A} \to \llparenthesis \underline{B} \rrparenthesis \mathbin{@} s_1 \qquad\qquad \emptyset \vdash^{\mathsf{v}} v : \hat{A} \mathbin{@} s_2$$

We have $cs = \mathsf{App}\, \hat{A}\, s_1\, s_2$, and thus $cs' = \mathit{subst}(x; t; s_1; s_2)$. Using converse substitution (Lemma 7.33) on the typing of $t\{v/x\}$, we get:

$$\phi; \Phi; x : A \vdash^{\mathsf{c}}_{N_1} t : \underline{B} \qquad \phi; \Phi; \emptyset \vdash^{\mathsf{v}}_{N_2} v : A \qquad \phi; \Phi \vDash N_1 + N_2 = M \qquad \llparenthesis A \rrparenthesis = \hat{A}$$

Thus, we can type $(\lambda x.\, t)\, v$ using the rules App and Lam. □

For the fixpoint unrolling case of subject expansion, we have to prove that the admissible typing rule ThunkFix in Lemma 7.9 is invertible.

Lemma 7.36 (Inversion of ThunkFix)**.** *Assume a* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ *typing* $\phi; \Phi; \Gamma \vdash^{\mathsf{v}}_{M} \mathsf{thunk}\, \mu x.\, t :$ $A \mathbin{@} s$*. Then there exist types* $\underline{B}_1, \underline{B}_2$*, and index terms* I*,* K*,* H*, such that we can derive:*

$$b, \phi; b < H, \Phi; x : [a < I] \cdot \underline{B}_1, \Delta \vdash^{\mathsf{c}}_{J} t : \underline{B}_2 \mathbin{@} s'$$

$$a, b, \phi; a < I, b < H, \Phi \vdash \underline{B}_2\{1 + b + \left(\overset{a}{\underset{d}{\triangle}} I\{1 + b + d/b\} \right) /b\} \sqsubseteq \underline{B}_1$$

$$\phi; \Phi \vdash [a < K] \cdot \underline{B}_2\{\overset{a}{\underset{b}{\triangle}} I/b\} \sqsubseteq A \qquad \phi; \Phi \vdash \Gamma \sqsubseteq \textstyle\sum_{b<H} \Delta \qquad \phi; \Phi \vDash H + \textstyle\sum_{b<H} J \leq M \qquad \phi; \Phi \vDash H \equiv \triangle_b^K I$$

with $s = \mathsf{Thunk}\, (\mathsf{Fix}\, s')$*. (Furthermore, if the* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ *typing was precise, we have* $\equiv$ *instead of* $\sqsubseteq$ *or* $\leq$*)*

Proof. We first invert the typing of thunk:

$$c, \phi; c < K, \Phi; \Delta' \vdash^{\mathsf{c}}_{M'} \mu x.\, t : \underline{B} \qquad\qquad \phi; \Phi \vdash [c < K] \cdot \underline{B} \sqsubseteq A$$

$$\phi; \Phi \vDash K + \sum_{c<K} M' \leq M \qquad\qquad \phi; \Phi \vdash \Gamma \sqsubseteq \sum_{c<K} \Delta'$$

The K is already the K we need to provide in the lemma. We further invert the typing of $\mu x.\, t$:

$$b, c, \phi; b < H^*, c < K, \Phi; x : [a < I^*] \cdot \underline{B}_1^*, \Delta'' \vdash^{\mathsf{c}}_{J^*} t : \underline{B}_2^*$$

$$a, b, c, \phi; a < I^*, b < H^*, c < K, \Phi \vdash \underline{B}_2^*\{1 + b + \left(\overset{a}{\underset{d}{\triangle}} I^*\{1 + b + d/b\} \right) /b\} \sqsubseteq \underline{B}_1^*$$

$$c, \phi; c < K, \Phi \vdash \underline{B}_2^*\{0/b\} \sqsubseteq \underline{B} \qquad\qquad c, \phi; c < K, \Phi \vDash H^* \equiv \overset{1}{\underset{b}{\triangle}} I^*$$

$$c, \phi; c < K, \Phi \vDash (H^* \dot{-} 1) + \textstyle\sum_{b<H^*} J^* \leq M' \qquad\qquad c, \phi; c < K, \Phi \vdash \Delta' \sqsubseteq \textstyle\sum_{b<H^*} \Delta''$$

Note that I^* has c as a free variable; for $c < K$, it describes a recursion tree of size H^*. Using the function $g^{-1} := \mathit{findSlot}_c\, K\, H$, we define the two inverting substitutions:

$$\theta := \{b + \sum_{d<c} H\{d/c\}/b\} \qquad\qquad \theta^* := \{\pi_1(g^{-1}(b))/c, \pi_2(g^{-1}(b))/b\}$$

The joined forest $I := I^*\theta^*$ consists of K trees and has size $H := \sum_{c<K} H^*$. Similarly, choose $A := A^*\theta^*$, $\underline{B}_1 := \underline{B}_1^*\theta^*$, $\underline{B}_2 := \underline{B}_2^*\theta^*$, $\Delta := \Delta''\theta^*$, and $J := J^*\theta^*$.

By applying the substitution θ^*, we can type:

$$b, \phi; b < H, \Phi; x : [a < I] \cdot \underline{B}_1, \Delta \vdash^{\mathsf{c}}_{J} t : \underline{B}_2$$

$$a, b, \phi; a < I, b < H, \Phi \vdash \underline{B}_2\{1 + b + \left(\overset{a}{\underset{d}{\triangle}} I\{1 + b + d/b\} \right) /b\} \sqsubseteq \underline{B}_1$$

There are only three remaining subtypings and inequations to show, which are all very similar:

$$\begin{aligned}
\phi; \Phi \vdash [a < K] \cdot \underline{B}_2\{\overset{a}{\underset{b}{\triangle}} I/b\} &\equiv [a < K] \cdot \underline{B}_2^*\theta^*\{\sum_{c<a} H/b\} \\
&\equiv [a < K] \cdot \underline{B}_2^*\{\pi_1(g^{-1}(\sum_{c<a} H))/c, \pi_2(g^{-1}(\sum_{c<a} H))/b\} \\
&\equiv [a < K] \cdot \underline{B}_2^*\{a/c, 0/b\} \sqsubseteq [a < K] \cdot \underline{B}\{a/c\} = [c < K] \cdot \underline{B} \sqsubseteq A
\end{aligned}$$

$$\phi; \Phi \vdash \Gamma \sqsubseteq \sum_{c<K} \Delta' \sqsubseteq \sum_{c<K} \sum_{b<H^*} \Delta'' \sqsubseteq \sum_{c<K} \sum_{b<H^*} \Delta''\theta^*\theta \equiv \sum_{c<\sum_{c<K} H^*} \Delta''\theta^* = \sum_{b<H} \Delta$$

$$\phi; \Phi \vDash H + \sum_{b<H} J = K + \sum_{b<K} H^* \dot{-} K + \sum_{c<K} \sum_{b<H^*} J^* = K + \sum_{c<K} (H^* \dot{-} 1 + \sum_{b<H^*} J^*) \leq M \quad \square$$

With the above lemma, proving the fixpoint subject expansion is similar to Lemma A.9 in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$.

Lemma 7.37 (Subject expansion (fixpoint case))**.** *Let* $\emptyset \vdash \mu x.\, t : (\!|\underline{B}|\!) \mathbin{@} \mathsf{Fix}\, s$ *be a simple* CBPV *typing and let* $s' = \mathit{subst}(x; t; s; \mathsf{Thunk}\, (\mathsf{Fix}\, s))$ *be the successor skeleton of this typing. Assume a* $\mathsf{d\ell PCF_{pv}}$ *typing* $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{M} t\{\mathsf{thunk}\, \mu x.\, t/x\} : \underline{B} \mathbin{@} s'$*. Then we can type* $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{M} \mu x.\, t : \underline{B} \mathbin{@} s$*.*

Proof. By converse substitution (Lemma 7.33), we get a value typing for $\mathsf{thunk}\, \mu x.\, t$:

$$\phi; \Phi; x : A_\mu \vdash^{\mathsf{c}}_{N_1} t : \underline{B} \mathbin{@} s \qquad \phi; \Phi; \emptyset \vdash^{\mathsf{v}}_{N_2} \mathsf{thunk}\, \mu x.\, t : A_\mu \mathbin{@} \mathsf{Fix}\, s \qquad \phi; \Phi \vDash M = N_1 + N_2$$

By applying Lemma 7.36 on the (precise) typing of $\mathsf{thunk}\, \mu x.\, t$, we get:

$$b, \phi; b < H, \Phi; x : [a < I] \cdot \underline{B}_1, \Delta \vdash^{\mathsf{c}}_{J} t : \underline{B}_2 \mathbin{@} s$$

$$a, b, \phi; a < I, b < H, \Phi \vdash \underline{B}_2\{1 + b + \left(\overset{a}{\underset{d}{\triangle}} I\{1 + b + d/b\}\right)/b\} \equiv \underline{B}_1$$

$$\phi; \Phi \vdash [a < K] \cdot \underline{B}_2\{\overset{a}{\underset{b}{\triangle}} I/b\} \equiv A_\mu \qquad \phi; \Phi \vdash \Gamma \equiv \sum_{b<H} \Delta \qquad \phi; \Phi \vDash H + \sum_{b<H} J \equiv M \qquad \phi; \Phi \vDash H \equiv \overset{K}{\underset{b}{\triangle}} I$$

In order to type the computation $\mu x.\, t$ with type $\underline{B}$, we define a new recursion forest with the cardinality $H^* := 1 + H$ by introducing a new root node with the K roots of I as children. Similarly as in the proof of Lemma A.9, we define:

$$I^* := \mathsf{ifz}\, b\, \mathsf{then}\, K\, \mathsf{else}\, I\{1 + b/b\} \qquad J^* := \mathsf{ifz}\, b\, \mathsf{then}\, N_1\, \mathsf{else}\, J\{1 + b/b\}$$

$$\underline{B}_1^* := \mathsf{ifz}\, b\, \mathsf{then}\, \underline{B}_2\{\overset{a}{\underset{b}{\triangle}} I/b\}\, \mathsf{else}\, \underline{B}_1\{1 + b/b\} \qquad \underline{B}_2^* := \mathsf{ifz}\, b\, \mathsf{then}\, \underline{B}\, \mathsf{else}\, \underline{B}_2\{1 + b/b\}$$

We apply rule Fix with these parameters. The typing and the subtyping follow by case distinction over b. The case $b = 0$ in the typing of $b, \phi; b < H^*, \Phi; x : [a < I^*] \cdot \underline{B}_1^* \vdash_{J^*} t : \underline{B}_2^*$ follows from the typing of t that we got by converse substitution. The cases where $b > 0$ follow by substituting $1 + b$ for b in the (sub)typings. ☐

7.6.4 Completeness for programs

As a corollary of subject expansion, we can show that all computations that terminate in $\mathsf{return}\, \underline{n}$ have type $\mathsf{F}\, \mathsf{Nat}[n]$.

Corollary 7.38 (Subject expansion, multiple steps)**.** *Let* $(t; s) \succ^*_k (t'; s')$*, where* k *is the number of forcing steps in this execution. Assume a* CBPV *typing* $\emptyset \vdash^{\mathsf{c}} t : (\!|\underline{B}|\!)$*, and a* $\mathsf{d\ell PCF_{pv}}$ *typing* $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{M} t' : \rho \mathbin{@} s'$*. Then* $\phi; \Phi; \emptyset \vdash_{k+M} t : \underline{B} \mathbin{@} s$*.*

Theorem 7.39 (Completeness for programs)**.** *All terminating* CBPV *programs (i.e. those terminating computations of simple type* $\mathsf{F}\, \mathsf{Nat}$*) can be typed with the type* $\mathsf{F}\, \mathsf{Nat}[n]$*, where* n *is the result. The weight of the typing is exactly the number of forcing steps.*

Proof. By assumption, we have $\emptyset \vdash^{\mathsf{c}} t : \mathsf{F}\, \mathsf{Nat}$. Since t is terminating (say, after k forcing steps), it terminates to $\mathsf{return}\, \underline{n}$ for some n, which can be typed in $\mathsf{d\ell PCF_{pv}}$: $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{0} \mathsf{return}\, \underline{n} : \mathsf{F}\, \mathsf{Nat}[n]$. By the above corollary, we have $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{k} t : \mathsf{F}\, \mathsf{Nat}[n]$. ☐

7.6.5 Deriving completeness for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ and $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$

In Sections 7.3 and 7.4, we have shown how to translate $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ and $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ typings to $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ typings. The shape of the translations can, of course, be computed from the shape of the original PCF typing. It is not difficult to show that if a $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ typing has the *right shape*, it can be translated back to a $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ or $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ typing. This way, we can derive completeness for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ and $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$.

Our first step to convert (properly shaped) $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ typings back to $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ typings is to define what *properly shaped* means. Informally, it means that the $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ has the skeleton of a typing generated by Lemma 7.5. We formalise this using a translation function $\cdot^{\mathsf{n}}$ from PCF skeletons to CBPV skeletons.

Definition 7.40 (CBN skeleton translation)**.**

$$\begin{aligned}
\mathsf{Var}^{\mathsf{n}} &:= \mathsf{Force}\,\mathsf{Var} \\
\mathsf{Const}^{\mathsf{n}} &:= \mathsf{Const} \\
(\mathsf{Lam}\,s)^{\mathsf{n}} &:= \mathsf{Lam}\,s^{\mathsf{n}} \\
(\mathsf{Fix}\,s)^{\mathsf{n}} &:= \mathsf{Fix}\,s^{\mathsf{n}} \\
(\mathsf{App}\,A\,s_1\,s_2)^{\mathsf{n}} &:= \mathsf{App}\,A^{\mathsf{n}}\,s_1^{\mathsf{n}}\,(\mathsf{Thunk}\,s_2^{\mathsf{n}}) \\
(\mathsf{Ifz}\,s_1\,s_2\,s_3)^{\mathsf{n}} &:= \mathsf{Bind}\,s_1^{\mathsf{n}}\,(\mathsf{Ifz}\,\mathsf{Var}\,s_2^{\mathsf{n}}\,s_3^{\mathsf{n}})
\end{aligned}$$

Recall that the function A^{n} maps (simple) PCF types to CBPV types, as defined in Definition 2.10.

Now we can prove the backtranslation theorem. Informally, it holds because the translation in Lemma 7.5 is invertible.

Lemma 7.41 (Backtranslation to $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$)**.** *Let* $\phi;\Phi;\Gamma^{\mathsf{n}} \vdash^{\mathsf{c}}_{M} t^{\mathsf{n}} : \rho^{\mathsf{n}} \mathbin{@} s^{\mathsf{n}}$ *be a* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ *typing. Then we can type* $\phi;\Phi;\Gamma \vdash_{M} t : \rho @ s$ *in* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$. *(Moreover, if the* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ *typing is precise, so is the generated* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ *typing.)*

Proof (sketch). By induction on the skeleton s and inversion of the $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ typing.

- Case $t = x$. By inverting $\phi;\Phi;\Gamma^{\mathsf{n}} \vdash^{\mathsf{c}}_{M} \mathsf{force}\,x : \rho^{\mathsf{n}}$, we get: $\phi;\Phi \vdash \Gamma^{\mathsf{n}}(x) \sqsubseteq [a<1]\cdot\sigma$ with $\phi;\Phi \vdash \sigma\{0/a\} \sqsubseteq \rho^{\mathsf{n}}$. Thus, there must be a $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ type σ' such that $\sigma'^{\mathsf{n}} = \sigma$ and $\phi;\Phi \vdash \sigma'\{0/a\} \sqsubseteq \rho$. From that, it follows that $\phi;\Phi;\Gamma \vdash_{M} x : \rho$.

- Case $t = \underline{n}$. We have $\rho^{\mathsf{n}} = \mathsf{Nat}[n'] = \rho$ with $\phi;\Phi \vDash n \sqsubseteq n'$, and thus also $\phi;\Phi;\Gamma \vdash_{M} \underline{n} : \rho$.

- Case $t = t_1\,t_2$. We invert the $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ typing of $t^{\mathsf{n}} = t_1^{\mathsf{n}}\,(\mathsf{thunk}\,t_2^{\mathsf{n}})$, which has the skeleton $s = \mathsf{App}\,\hat{A}\,s_1\,s_2$, where $\hat{A}$ is a PCF type, and thus $s^{\mathsf{n}} = \mathsf{App}\,\hat{A}^{\mathsf{n}}\,s_1^{\mathsf{n}}\,(\mathsf{Thunk}\,s_2^{\mathsf{n}})$.

$$\phi;\Phi;\Delta_1 \vdash^{\mathsf{c}}_{K_1} t_1^{\mathsf{n}} : A \multimap \rho^{\mathsf{n}} \mathbin{@} s_1^{\mathsf{n}} \qquad \phi;\Phi;\Gamma_2 \vdash^{\mathsf{v}}_{K_2} \mathsf{thunk}\,t_2^{\mathsf{n}} : A \mathbin{@} \mathsf{Thunk}\,s_2^{\mathsf{n}}$$

$$(\!|A|\!) = \hat{A}^{\mathsf{n}} \qquad \phi;\Phi \vDash K_1 + K_2 \leq M \qquad \phi;\Phi \vdash \Gamma^{\mathsf{n}} \sqsubseteq \Delta_1 \uplus \Gamma_2$$

We further invert the typing of $\mathsf{thunk}\, t_2^{\mathsf{n}}$:

$$a, \phi; a < I, \Phi; \Delta_2 \vdash^{\mathsf{c}}_{J} t_2^{\mathsf{n}} : \underline{B} \mathbin{@} s_2^{\mathsf{n}} \qquad \phi; \Phi \vdash [a < I] \cdot \underline{B} \sqsubseteq A$$

$$\phi; \Phi \vDash I + \sum_{a<I} J \leq K_2 \qquad \phi; \Phi \vdash \Gamma_2 \sqsubseteq \sum_{a<I} \Delta_2$$

We can define a dℓPCF$_{\mathsf{n}}$ (*basic*) type σ such that $\sigma^{\mathsf{n}} = A$.

Since $\phi; \Phi \vdash \Gamma^{\mathsf{n}} \sqsubseteq \Delta_1 \uplus \Gamma_2$, we can define d$\ell$PCF$_{\mathsf{n}}$ contexts Δ_1' and Γ_2' such that $\Delta_1 = {\Delta_1'}^{\mathsf{n}}$, $\Gamma_2 = {\Gamma_2'}^{\mathsf{n}}$, and $\phi; \Phi \vdash \Gamma \sqsubseteq \Delta_1' \uplus \Gamma_2'$. Similarly, we can define a Δ_2' such that $\Delta_2 = {\Delta_2'}^{\mathsf{n}}$ and $\phi; \Phi \vdash \Gamma_2' \sqsubseteq \sum_{a<I} \Delta_2'$.

Now we can apply the inductive hypotheses on the typings of t_1 and t_2, which yields:

$$\frac{\phi; \Phi; \Delta_1' \vdash_{K_1} t_1 : \sigma \multimap \rho \mathbin{@} s_1 \qquad a, \phi; a < I, \Phi; \Delta_2' \vdash_J t_2 : \sigma \mathbin{@} s_2}{\phi; \Phi; \Gamma \sqsubseteq \Delta_1' \uplus \sum_{a<I} \Delta_2' \vdash_{K_1 + I + \sum_{a<I} J \leq M} t_1\, t_2 : \rho \mathbin{@} \mathsf{App}\, \hat{A}\, s_1\, s_2}$$

- Case $t = \lambda x.\, t'$. We have $\phi; \Phi; x : A, \Gamma \vdash^{\mathsf{c}}_{M} t^{\mathsf{n}} : \underline{B}$ with $\phi; \Phi \vdash A \multimap \underline{B} \sqsubseteq \rho^{\mathsf{n}}$. Therefore, we can define types A' and σ such that ${A'}^{\mathsf{n}} = A$, $\sigma^{\mathsf{n}} = \underline{B}$ and $\phi; \Phi \vdash A' \multimap \sigma \sqsubseteq \rho$. By the inductive hypothesis, we have $\phi; \Phi; x : A' \vdash_M t : \sigma$, and therefore $\phi; \Phi; \Gamma \vdash_M \lambda x.\, t : A' \multimap \sigma \sqsubseteq \rho$.

- Case $t = \mu x.\, t'$. As above.

- Case $t = \mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3$.

  We invert the typing of $t^{\mathsf{n}} = \mathsf{bind}\, z \leftarrow t_1\, \mathsf{in}\, \mathsf{ifz}\, z\, \mathsf{then}\, t_2^{\mathsf{n}}\, \mathsf{else}\, t_3^{\mathsf{n}}$:

$$\frac{\phi; \Phi; \Delta_1 \vdash^{\mathsf{c}}_{K_1} t_1^{\mathsf{n}} : \mathsf{F}\,\mathsf{Nat}[I] \mathbin{@} s_1^{\mathsf{n}} \qquad \dfrac{\dfrac{\phi; \Phi \vdash \mathsf{Nat}[I] \sqsubseteq \mathsf{Nat}[I']}{\phi; \Phi; z : \mathsf{Nat}[I] \vdash^{\mathsf{v}}_{K_3} z : \mathsf{Nat}[I'] \mathbin{@} \mathsf{Var}} \qquad \begin{array}{c} \phi; 0 \gtrsim I', \Phi; \Delta_2 \vdash_{K_2} t_2^{\mathsf{n}} : \rho^{\mathsf{n}} \mathbin{@} s_2 \\ \phi; 0 < I', \Phi; \Delta_2 \vdash_{K_2} t_3^{\mathsf{n}} : \rho^{\mathsf{n}} \mathbin{@} s_3 \end{array}}{\phi; \Phi; z : \mathsf{Nat}[I], \Delta_2 \vdash^{\mathsf{c}}_{K_2} \mathsf{ifz}\, z\, \mathsf{then}\, t_2^{\mathsf{n}}\, \mathsf{else}\, t_3^{\mathsf{n}} : \rho^{\mathsf{n}} \mathbin{@} \mathsf{Ifz}\, \mathsf{Var}\, s_2^{\mathsf{n}}\, s_3^{\mathsf{n}}}}{\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_{M} \mathsf{bind}\, z \leftarrow t_1^{\mathsf{n}}\, \mathsf{in}\, \mathsf{ifz}\, z\, \mathsf{then}\, t_2^{\mathsf{n}}\, \mathsf{else}\, t_3^{\mathsf{n}} : \rho^{\mathsf{n}} \mathbin{@} s^{\mathsf{n}}}$$

  (With $\phi; \Phi \vdash \Gamma^{\mathsf{n}} \sqsubseteq \Delta_1 \uplus \Delta_2$ and $\phi; \Phi \vDash K_1 + K_2 + K_3 \leq M$). Note that by inversion of the typing of the temporary variable z, we get a I' such that $\phi; \Phi \vDash I \sqsubseteq I'$. Moreover, if the d$\ell$PCF$_{\mathsf{pv}}$ typing is precise, we have $\phi; \Phi \vDash K_3 \equiv 0$. Using the same technique as in the above cases, we can define dℓPCF$_{\mathsf{n}}$ contexts Δ_i' such that ${\Delta_i'}^{\mathsf{n}} = \Delta_i$ and $\phi; \Phi \vdash \Gamma \sqsubseteq \Delta_1' \uplus \Delta_2'$. Using the inductive hypotheses, we can now type:

$$\frac{\phi; \Phi; \Delta_1' \vdash_{K_1} t_1 : \mathsf{Nat}[I'] \mathbin{@} s_1 \qquad \phi; 0 \gtrsim I', \Phi; \Delta_2' \vdash_{K_2} t_2 : \rho \mathbin{@} s_2 \qquad \phi; 0 < I', \Phi; \Delta_2' \vdash_{K_2} t_3 : \rho \mathbin{@} s_3}{\phi; \Phi; \Gamma \vdash_M \mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3 : \rho \mathbin{@} \mathsf{Ifz}\, s_1\, s_2\, s_3}$$

- Cases $t = \mathsf{calc}\, x \leftarrow \mathsf{Succ}(v)\, \mathsf{in}\, t'$ and $t = \mathsf{calc}\, x \leftarrow \mathsf{Pred}(v)\, \mathsf{in}\, t'$: As above. □

We can now prove completeness for dℓPCF$_{\mathsf{n}}$ programs (Theorem 6.3).

Proof. We assume a closure execution $t \Downarrow_k \underline{n}$ and need to show $\emptyset; \emptyset; \emptyset \vdash_k t : \mathsf{Nat}[n]$. Using Lemma 2.17, we can translate the execution to the CBPV closure semantics: $\langle t^{\mathsf{n}}; \emptyset \rangle \Downarrow_k tc$ with $\mathit{unf}(tc) = \mathsf{return}\, \underline{n}$. It can be shown that this closure execution can be converted to a normal CBPV execution $t^{\mathsf{n}} \Downarrow_k \mathsf{return}\, \underline{n}$. Thus, using $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ completeness (Theorem 7.39), we have $\emptyset; \emptyset; \emptyset \vdash_k^{\mathsf{c}} t^{\mathsf{n}} : \mathsf{F}\,\mathsf{Nat}[n]$. Since this is a typing of a translated term, this typing must have the skeleton s^{n} for some s. Using Lemma 7.41, we can translate the $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ typing to a $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ typing, and we get $\emptyset; \emptyset; \emptyset \vdash_k t : \mathsf{Nat}[n]$, as required. ☐

We can do the same for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ (although we have already proved completeness of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ in Section 5.5). We define two translations: for skeletons vs of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ value typings, and skeletons ts of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ term typings.

Definition 7.42 (CBV skeleton translation)**.**

$$\begin{aligned}
\mathsf{Const}^{\mathsf{val}} &:= \mathsf{Const}\\
(\mathsf{Lam}\, s)^{\mathsf{val}} &:= \mathsf{Thunk}\,(\mathsf{Lam}\, s^{\mathsf{v}})\\
(\mathsf{Fix}\,(\mathsf{Lam}\, s))^{\mathsf{val}} &:= \mathsf{Thunk}\,(\mathsf{Fix}\,(\mathsf{Lam}\, s^{\mathsf{v}}))\\
vs^{\mathsf{v}} &:= \mathsf{Return}\; vs^{\mathsf{val}}\\
\mathsf{Var}^{\mathsf{v}} &:= \mathsf{Return}\,\mathsf{Var}\\
(\mathsf{App}\,\tau\, s_1\, s_2)^{\mathsf{v}} &:= \mathsf{Bind}\, s_1^{\mathsf{v}}\,(\mathsf{Bind}\, s_2^{\mathsf{v}}\,(\mathsf{App}\,\tau^{\mathsf{v}}\,(\mathsf{Force}\,\mathsf{Var})\,\mathsf{Var}))\\
(\mathsf{Ifz}\, s_1\, s_2\, s_3)^{\mathsf{v}} &:= \mathsf{Bind}\,(s_1^{\mathsf{v}})\,(\mathsf{Ifz}\,\mathsf{Var}\, s_2^{\mathsf{v}}\; s_3^{\mathsf{v}})
\end{aligned}$$

The function τ^{n} that maps (simple) PCF types to CBPV types is defined in Definition 2.20.

Lemma 7.43 (Backtranslation to $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$)**.** *Let* $\phi; \Phi; \Gamma^{\mathsf{v}} \vdash_M^{\mathsf{v}} v^{\mathsf{v}} : \tau^{\mathsf{val}} @ s^{\mathsf{val}}$ *be a* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ *value typing. Then we can type* $\phi; \Phi; \Gamma \vdash_M v : \tau @ s$*. Also, if* $\phi; \Phi; \Gamma^{\mathsf{v}} \vdash_M^{\mathsf{c}} t^{\mathsf{v}} : \mathsf{F}\,\tau^{\mathsf{v}} @ s^{\mathsf{v}}$*, then* $\phi; \Phi; \Gamma \vdash_M t : \tau @ s$*. (Moreover, if the* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ *typing is precise, so is the generated* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ *typing.)*

Proof (sketch). Similarly to Lemma 7.41. For inverting the typing of $(\mu f x.\, t)^{\mathsf{val}} = \mathsf{thunk}\, \mu f.\, \lambda x.\, t^{\mathsf{v}}$, we use Lemma 7.36. ☐

7.7 Conjunctives and disjunctives

We have not considered products (conjunctives) and sums (disjunctives) of CBPV and $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ yet. In fact, there are two variants of products – multiplicative and additive products – which behave like the products in CBV and CBN, respectively. Extending $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ with conjunctives and disjunctives is straightforward.

Multiplicative conjunctive (and unit) Values of the type $A_1 \otimes A_2$ are similar to pairs in the CBV version of PCF: They consist of two components, $(v_1; v_2)$, and we can access both components using pattern-matching.

$$\begin{aligned} A &::= \cdots \mid 1 \mid A_1 \otimes A_2 \\ v &::= \cdots \mid () \mid (v_1; v_2) \\ t &::= \cdots \mid \mathsf{let}\,(x; y) := v \,\mathsf{in}\, t \end{aligned}$$

(In CBPV, we write $\times$ instead of $\otimes$.)

Since we introduced new value $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ types, we also have to extend the definition of (binary and bounded) modal sums. For the unit type, we simply define $1 \uplus 1 := 1$ and $\sum_{a<I} 1 := 1$. Modal sums over multiplicative conjunctives are built component-wise:

$$\frac{A_1 \uplus A_1' = A_1'' \qquad A_2 \uplus A_2' = A_2''}{(A_1 \otimes A_2) \uplus (A_1' \otimes A_2') = A_1'' \otimes A_2''} \qquad \frac{\sum_{a<I} A_1 = A_1' \qquad \sum_{a<I} A_2 = A_2'}{\sum_{a<I}(A_1 \otimes A_2) = A_1' \otimes A_2'}$$

Since the conjunctive $\otimes$ is *multiplicative* we have to build the modal sum over two contexts. In other words, the resources are distributed among the two components of the tuple.

$$\textsc{Unit}\quad \phi; \Phi; \emptyset \vdash^{\mathsf{v}} () : 1 \qquad \textsc{MProd}\quad \frac{\phi; \Phi; \Delta_1 \vdash^{\mathsf{v}}_{K_1} v_1 : A_1 \qquad \phi; \Phi; \Delta_2 \vdash^{\mathsf{v}}_{K_2} v_2 : A_2}{\phi; \Phi; \Delta_1 \uplus \Delta_2 \vdash^{\mathsf{v}}_{K_1+K_2} (v_1; v_2) : A_1 \otimes A_2}$$

$$\textsc{LetPair}\quad \frac{\begin{array}{c}\phi; \Phi; \Delta_1 \vdash^{\mathsf{v}}_{K_1} v : A_1 \otimes A_2 \\ \phi; \Phi; x : A_1, y : A_2, \Delta_2 \vdash^{\mathsf{c}}_{K_2} t : \underline{B}\end{array}}{\phi; \Phi; \Delta_1 \uplus \Delta_2 \vdash^{\mathsf{c}}_{K_1+K_2} \mathsf{let}\,(x; y) := v \,\mathsf{in}\, t : \underline{B}}$$

For the soundness and completeness proofs, we have to modify the splitting and joining lemmas, respectively. Instead of doing a case analysis over the value, we now have to induct over the value. The inductive case for $\otimes$ follows trivially from the inductive hypotheses. We also have to prove the subject reduction/expansion cases for the new head reduction step: $\mathsf{let}\,(x; y) := (v_1, v_2) \,\mathsf{in}\, t \succ_0 t\{v_1/x, v_2/y\}$, which is trivial.

We can type projection functions, e.g:

$$\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_0 \mathit{fst} := \lambda z.\, \mathsf{let}\,(x; y) := z \,\mathsf{in}\,\mathsf{return}\, x : A_1 \otimes A_2 \multimap \mathsf{F}\, A_1$$

Note that this typing is not *precise* (unless A_2 is disposable), which means that this function may throw away resources.

Additive conjunctive The additive conjunctive (&) is a 'tuple' over computations.

$$\begin{aligned} \underline{B} &::= \cdots \mid \underline{B}_1 \,\&\, \underline{B}_2 \\ t &::= \cdots \mid \langle t_1; t_2 \rangle \mid \pi_1(t) \mid \pi_2(t) \\ T &::= \cdots \mid \langle t_1; t_2 \rangle \end{aligned}$$

(We also write & in CBPV.) Note that $\langle t_1; t_2\rangle$ is a terminal computation. Using the projections, we can decide to execute one (but not both) of the components. Thus, the small-step rules are $\pi_i\,\langle t_1; t_2\rangle \succ_0 t_i$ (for $i = 1, 2$). Of course, we can also thunk an additive conjunctive if we want to apply more than one projection. For example, values of type $[a < I] \cdot (\underline{B}_1 \mathbin{\&} \underline{B}_2)$ could be forced I times, and we could apply I projections (either $\pi_1(\cdot)$ or $\pi_2(\cdot)$) in total.

Since, only one of the computations can be executed, both computations are typed with the same context and are assigned the same weight. For this reason, we say that & is additive.

$$\textsc{AProd}\quad \frac{\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_M t_1 : \underline{B}_1 \qquad \phi; \Phi; \Gamma \vdash^{\mathsf{c}}_M t_2 : \underline{B}_2}{\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_M \langle t_1; t_2\rangle : \underline{B}_1 \mathbin{\&} \underline{B}_2} \qquad \textsc{Proj}\quad \frac{\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_M t : \underline{B}_1 \mathbin{\&} \underline{B}_2}{\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_M \pi_i(t) : \underline{B}_i}$$

Additive sum Additive sums are similar to ordinary sums: We introduce new values $\mathsf{inl}(v)$ and $\mathsf{inr}(v)$, and a case distinction operator. The typing rule of the case distinction operator is similar to the ifz rule – only one the branches is executed (depending on the value), and the typings thus have the same context and weight.

$$\begin{aligned} A &::= \cdots \mid A_1 \oplus A_2 \\ v &::= \cdots \mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \\ t &:= \mathsf{case}\, v\, [\mathsf{inl}(x) \Rightarrow t_1 \mid \mathsf{inr}(y) \Rightarrow t_2] \end{aligned}$$

(In CBPV, we write + instead of $\oplus$.) We again have to extend the definition of modal sums and introduce three new typing rules.

$$\frac{A_1 \uplus A_1' = A_1'' \qquad A_2 \uplus A_2' = A_2''}{(A_1 \oplus A_2) \uplus (A_1' \oplus A_2') = A_1'' \oplus A_2''} \qquad \frac{\sum_{a<I} A_1 = A_1' \qquad \sum_{a<I} A_2 = A_2'}{\sum_{a<I} (A_1 \oplus A_2) = A_1' \oplus A_2'}$$

$$\textsc{Inl}\quad \frac{\phi; \Phi; \Gamma \vdash^{\mathsf{v}}_M v : A_1}{\phi; \Phi; \Gamma \vdash^{\mathsf{v}}_M \mathsf{inl}(v) : A_1 \oplus A_2} \qquad \textsc{Inr}\quad \frac{\phi; \Phi; \Gamma \vdash^{\mathsf{v}}_M v : A_2}{\phi; \Phi; \Gamma \vdash^{\mathsf{v}}_M \mathsf{inr}(v) : A_1 \oplus A_2}$$

$$\textsc{CaseSum}\quad \frac{\phi; \Phi; \Delta_1 \vdash^{\mathsf{v}}_{K_1} v : A_1 \oplus A_2 \qquad \phi; \Phi; x : A_1, \Delta_2 \vdash^{\mathsf{c}}_{K_2} t_1 : \underline{B} \qquad \phi; \Phi; y : A_2, \Delta_2 \vdash^{\mathsf{c}}_{K_2} t_2 : \underline{B}}{\phi; \Phi; \Delta_1 \uplus \Delta_2 \vdash^{\mathsf{c}}_{K_1+K_2} \mathsf{case}\, v\, [\mathsf{inl}(x) \Rightarrow t_1 \mid \mathsf{inr}(y) \Rightarrow t_2] : \underline{B}}$$

Disposable types and precise typings The new value types $A_1 \otimes A_2$ and $A_1 \oplus A_2$ are considered *disposable* (as in Section 5.4), if and only if A_1 and A_2 are disposable; units are always disposable. This means that a precise value typing with type $1 \otimes (1 \oplus \mathsf{Nat}[3])$ must have weight 0.

Chapter 8

Compositionality and polymorphism

In this chapter, we will first answer the question whether $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ is *compositionally complete*. We have already demonstrated in Section 5.5.8 that we can construct $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ typings for total (CBV) functions of simple type $\mathsf{Nat} \to \mathsf{Nat}$ in a way that allows us to instantiate and apply this typing for any argument of simple type Nat. However, the *relative completeness* theorems are of a *semantic* nature, since they assume (enumerations of) executions and 'convert' them into a typing. For example, the actual recursion trees of fixpoints will be encoded in the 'generated' typing. This is a fundamental problem in practice, since we want to type programs without executing them. Moreover, we may even want to type programs that are known to diverge. Ideally, a *syntactic* typing annotation algorithm would work by structural recursion on a simple typing. Also, this approach does not work for higher-order functions. In particular, is it possible to type the higher-order function $\lambda x.\, x\, \underline{0}$ in a way such that we can *reuse* this typing for every possible application with an argument of simple type $\mathsf{Nat} \to \mathsf{Nat}$?

One problem is that when typing a function in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$, we have to know how often the function can be applied later (and the refinements of the arguments). Similarly, in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$, we have to know how often a parameter is used (and the type refinements of each use). However, this number depends on the *context* in which the function is used. This number may even depend on the value of the argument itself, as in $\lambda x.\, \mathsf{ifz}\, x\, \mathsf{then}\, \underline{0}\, \mathsf{else}\, \mathsf{Succ}(x)$: If the argument evaluates to $\underline{0}$, it is only needed once; if it is positive, the argument needs to be evaluated again.

Perhaps surprisingly, compositionality can actually be attained in an extension of $\mathsf{d}\ell\mathsf{PCF}$, which was first shown in [13]. To this end, we need to parametrise over the negative annotations of types. For example, to annotate a typing of a function with the simple type $\mathsf{U}\,(\mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat})$, we need to parametrise over:

1. the number of times the function can be forced and applied, and
2. for each application, the refinement of the argument.

Thus, we introduce two *refinement variables* $i/0$ and $j/1$, where the numbers denote their arities. The $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ type assigned to this function is $[a < i()] \cdot (\mathsf{Nat}[j(a)] \multimap \mathsf{F}\,\mathsf{Nat}[K(a)]$, where $K(a)$ stands for a 'concrete' index term, that may be defined using mutually recursive equations. When forcing and applying this function with a value of type $\mathsf{Nat}[L]$, we substitute $i() := 1$ and $j(a) := L$. In the next section of this chapter, we will summarise a type inference algorithm for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ that is based on this idea.

Besides compositionality, many functional programming languages also feature *polymorphism.* For example, the function $\mathit{fst} : A_1 \otimes A_2 \multimap \mathsf{F}\,A_1$ can be typed in the same way regardless of the value types A_1 and A_2. Thus, we can assign a polymorphic type to this function: $\forall \alpha_1\, \alpha_2.\, \alpha_1 \otimes \alpha_2 \multimap \mathsf{F}\,\alpha_1$. Another well-known application of polymorphism is that we can encode inductive data types using *Church encoding.* It was already observed in [19] that bounded exponentials together with polymorphism can be used to encode *bounded* data types. For example, it is possible to define a type $\mathsf{Nat}_{\leq I}$ of (encodings of) natural numbers bounded by I. As another interesting application of bounded Church encoding, we will define a type $\mathsf{List}_{a<I}A$ of lists with no more than I elements, where a is free in A.

In the last section of this chapter, we will show that compositionality and polymorphism play well together: It is, in fact, possible to achieve polymorphic *and* compositionally reusable typings.

8.1 Compositionality

In this section, we describe a syntax-directed type inference algorithm for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ that is based on an algorithm for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ in [13].[1] Instead of defining all cases of the algorithm formally, we will give illustrative abstract and concrete examples and explain the general cases.

Extension of the language of index terms ($\mathcal{L}^{\ell}_{idx}$) We first extend $\mathcal{L}^{\ell}_{idx}$ with *function variables* and variables for *mutually recursively defined* index terms:

$$\begin{aligned} \text{Index terms:} \quad & I ::= \cdots \mid j(I_1, \ldots, I_n) \mid K(I_1, \ldots, I_n) \\ \text{Signatures:} \quad & \Sigma ::= \emptyset \mid j/n, \Sigma \\ \text{Equations:} \quad & \mathcal{E} ::= \emptyset \mid (K(a, ..., c) := I), \mathcal{E} \end{aligned}$$

Every function variable has an *arity*, which is declared in the signature Σ. Using an *equational program* $\mathcal{E}$, we can define index terms K by mutual recursion.

To extend the semantics of $\mathcal{L}^{\ell}_{idx}$, context valuations ν now also map function variables

---

[1] The authors of [13] have implemented the algorithm in `OCaml` (however, for the call-by-name version of $\mathsf{d}\ell\mathsf{PCF}$), but the code is not publicly available (any more). The article also has a mistake in the fixpoint case, which we correct here. In contrast to the paper, we track the refinement variables in a separate *signature context* (Σ), and we also allow concrete index terms to appear at positive positions. These (purely cosmetic) changes make the generated typings much easier to read.

$$\frac{\phi; \Sigma; \mathcal{E} \vdash I : \text{well-formed}}{pa^{+}(\phi; \Sigma; \mathcal{E}; \mathsf{Nat}[I])} \qquad \frac{h/\,|\phi| \in \Sigma}{pa^{-}(\phi; \Sigma; \mathcal{E}; \mathsf{Nat}[h(\phi)])} \qquad \frac{pa^{+}(a, \phi; \Sigma; \mathcal{E}; \underline{B}) \qquad h/\,|\phi| \in \Sigma}{pa^{+}(\phi; \Sigma; \mathcal{E}; [a < h(\phi)] \cdot \underline{B})}$$

$$\frac{\phi; \Sigma; \mathcal{E} \vdash I : \text{well-formed} \qquad pa^{-}(a, \phi; \Sigma; \mathcal{E}; \underline{B})}{pa^{-}(\phi; \Sigma; \mathcal{E}; [a < I] \cdot \underline{B})} \qquad \frac{pa^{\mp}(\phi; \Sigma; \mathcal{E}; A) \qquad pa^{\pm}(\phi; \Sigma; \mathcal{E}; \underline{B})}{pa^{\pm}(\phi; \Sigma; \mathcal{E}; A \multimap \underline{B})}$$

$$\frac{pa^{\pm}(\phi; \Sigma; \mathcal{E}; A_1) \qquad pa^{\pm}(\phi; \Sigma; \mathcal{E}; A_2)}{pa^{\pm}(\phi; \Sigma; \mathcal{E}; A_1 \otimes / \oplus A_2)} \qquad \frac{pa^{\pm}(\phi; \Sigma; \mathcal{E}; \underline{B}_1) \qquad pa^{\pm}(\phi; \Sigma; \mathcal{E}; \underline{B}_2)}{pa^{\pm}(\phi; \Sigma; \mathcal{E}; \underline{B}_1 \& \underline{B}_2)}$$

Figure 8.1: Definition of $pa^{\pm}(\phi; \Sigma; \mathcal{E}; A)$ and $pa^{\pm}(\phi; \Sigma; \mathcal{E}; \underline{B})$

to meta-level functions:[2] $[\![i(I_1, \ldots, I_n)]\!](\nu) = \nu(i)([\![I_1]\!], \ldots, [\![I_n]\!])$. The semantics of mutually recursive equations is defined by the least fixpoint that satisfies all the equations in $\mathcal{E}$.Computation/value typing judgements now have the following shape:

$$\phi; \Sigma; \mathcal{E}; \Phi; \Gamma \vdash^{\mathsf{v}}_{M} v : A \qquad\qquad \phi; \Sigma; \mathcal{E}; \Phi; \Gamma \vdash^{\mathsf{c}}_{M} t : \underline{B}$$

The type system is extended in a trivial way, since at no typing rule, anything is added to Σ or $\mathcal{E}$. We always assume assume that all index terms are closed (well-formed) under ϕ, Σ, and $\mathcal{E}$. Moreover, we often leave out the equational program $\mathcal{E}$ if it is empty or if we specify the equations separately.

We can substitute (abstracted) index terms for refinement index variables. A function substitution on types is written $A[j(a, b) := \cdots]$. For example:

$$\begin{aligned}&([b < j(a)] \cdot (\mathsf{Nat}[f(a, b)] \multimap \mathsf{F}\,\mathsf{Nat}[g(a, b)]))[f(a, b) := a + b, g(a, b) := a \dot{-} b] = \\ &([b < j(a)] \cdot (\mathsf{Nat}[a + b] \multimap \mathsf{F}\,\mathsf{Nat}[a \dot{-} b]))\end{aligned}$$

This definition can of course also be lifted to subtypings and typings. Similarly, symbols from $\mathcal{E}$ can be eliminated from a typing if their definitions are not recursive.

Positively and negatively annotated types As already hinted above, the algorithm computes annotations of the *positive positions* that depend on the refinements of the *negative positions* in a typing. The polarity of an annotation is defined in the standard way:

- The refinement I in $\mathsf{Nat}[I]$ is in a positive position;
- the refinement I in $[a < I] \cdot \underline{B}$ is in a negative position;
- all positive/negative positions of $\underline{B}$ are positive/negative in $A \multimap \underline{B}$;

---

[2] Note that valuations of *ordinary* index variables are not $\perp$. However, a function variable $i/0$ could be assigned the valuation $i() := \perp$.

- all positive/negative positions of A are negative/positive in $A \multimap \underline{B}$;
- positive/negative positions of types in contexts Γ are considered negative/positive positions of the typing.

Thus, for a simple typing of the judgement $x : \mathsf{U}\,\mathsf{F}\,\mathsf{Nat} \vdash^{\mathsf{c}} t : \underline{B}$, the algorithm will compute an index term that describes exactly how often the variable x is forced.

A type is *positively annotated*, if all refinements at negative positions are applications of function variables to the index variables ϕ and the other ordinary variables that are bound in the type. Dually, a type is annotated negatively if all refinements at the positive positions are the same kind of index terms. Formally, we define the predicates $pa^{\pm}(\phi; \Sigma; \mathcal{E}; A)$ and $pa^{\pm}(\phi; \Sigma; \mathcal{E}; \underline{B})$ by mutual induction, as in Figure 8.1. In addition to these rules, we assume that each function variable in Σ is used *exactly once* at the negative positions. However, they may be used at different positive parts of the type. Furthermore, we define:

- A value/computation type τ is a *$pa^{\pm}$-annotation* of a simple value/computation type $\hat{\tau}$ (in $\phi; \Sigma; \mathcal{E}$), if $pa^{\pm}(\phi; \Sigma; \mathcal{E}; \tau)$ and the types have the same shape.
- A context Γ is a *pa^{-}-annotation* of a simple context $\hat{\Gamma}$ (in $\phi; \Sigma; \mathcal{E}$), if for all variables x in Γ, $pa^{-}(\phi; \Sigma; \mathcal{E}; \Gamma(x))$ and $\Gamma(x)$ and $\hat{\Gamma}(x)$ have the same shape. Again, we assume that all variables in Σ are used exactly once at the positive positions in Γ (which are negative positions in the typing).

Formal specification of the algorithm The type inference algorithm takes as input a (simple) CBPV (value/computation) typing $\hat{\Gamma} \vdash^{\mathsf{c}} v : \hat{A}$ (or $\hat{\Gamma} \vdash^{\mathsf{c}} t : \underline{\hat{B}}$). It produces as output:

- a list of mutually recursive equations $\mathcal{E}$;
- a pa^{-}-annotation Γ of $\hat{\Gamma}$ (closed in $\phi; \Sigma; \mathcal{E}$). In particular, if $\hat{\Gamma}(x) = \mathsf{U}\,\underline{\hat{B}}$, then $\Gamma(x) = [a < I] \cdot \underline{B}$, where I is a concrete index term that denotes (exactly) how often the term forces the variable x;
- a pa^{+}-annotation A of $\hat{A}$ (or $\underline{B}$ of $\underline{\hat{B}}$) in $\phi; \Sigma; \mathcal{E}$;
- an index term M that is closed in $\phi; \Sigma; \mathcal{E}$;
- a precise $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ typing $\phi; \Sigma; \mathcal{E}; \emptyset; \Gamma \vdash^{\mathsf{v}}_{M} v : A$ (or $\phi; \Sigma; \mathcal{E}; \emptyset; \Gamma \vdash^{\mathsf{c}}_{M} t : \underline{B}$).

Similar to the algorithm in [13], our algorithm even computes a typing for diverging programs. However, in their work, the (quasi) typing is deemed ‘invalid’ for diverging programs, since their version of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{v}}$ does not support diverging index terms. They thus compute a list of ‘side conditions’, which are constraints of the form $\phi; \Phi \vDash I \downarrow$, that state that all generated index terms terminate. Proving these side conditions is out of scope of the type inference algorithm, since it is equivalent to showing that the program terminates.

8.1.1 Examples

Instead of defining the typing annotation algorithm formally, we will discuss here a series of illustrative examples. We cover all cases of the algorithm and also discuss abstract examples.

Example 1: Forcing and application

As our first example, we consider the following CBPV function:

$$\dfrac{\dfrac{\dfrac{}{(1)\ x : \mathsf{U}\,(\mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}) \vdash^{\mathsf{v}} x : \mathsf{U}\,(\mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat})}}{(3)\ x : \mathsf{U}\,(\mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}) \vdash^{\mathsf{c}} \mathsf{force}\,x : \mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}} \qquad \dfrac{}{(2)\ y : \mathsf{Nat} \vdash^{\mathsf{v}} y : \mathsf{Nat}}}{\dfrac{(4)\ x : \mathsf{U}\,(\mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}), y : \mathsf{Nat} \vdash^{\mathsf{c}} (\mathsf{force}\,x)\,y : \mathsf{F}\,\mathsf{Nat}}{(5)\ \emptyset \vdash^{\mathsf{c}} t_1 := \lambda x\,y.\,(\mathsf{force}\,x)\,y : \mathsf{U}\,(\mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}}}$$

Since the annotation algorithm works by structural recursion on the simple typing, we begin annotating the leaf nodes in the typing derivation. So let us begin annotating (1). The type on right hand side of the typing should be pa^{+}-annotated, and the type of x in the context should be pa^{-}-annotated. Thus, the translated typing of (1) should have the following shape:

$$\emptyset; i/0, l/1, k/1; \emptyset; x : [c < I] \cdot (\mathsf{Nat}[L(c)] \multimap \mathsf{F}\,\mathsf{Nat}[k(c)]) \vdash^{\mathsf{v}}_{0} x : [c < i()] \cdot (\mathsf{Nat}[l(c)] \multimap \mathsf{F}\,\mathsf{Nat}[K(c)])$$

Here, $i/0, l/1, k/1$ are fresh function variables and I, L, K are placeholders for index terms that we have to define. Since the (strict variant of the) rule VAR requires that both types are equivalent, we have to unify the types. In the variable case, the unification is always trivial. In this example, we simply define $I := i()$, $L(c) := l(c)$, and $K(c) := k(c)$:

$$\emptyset; i/0, l/1, k/1; \emptyset; x : [c < i()] \cdot (\mathsf{Nat}[l(c)] \multimap \mathsf{F}\,\mathsf{Nat}[k(c)]) \vdash^{\mathsf{v}}_{0} x : [c < i()] \cdot (\mathsf{Nat}[l(c)] \multimap \mathsf{F}\,\mathsf{Nat}[k(c)])$$

The typing (2) is translated in the same way: $\emptyset; j/0; \emptyset; y : \mathsf{Nat}[j()] \vdash^{\mathsf{v}}_{0} y : \mathsf{Nat}[j()]$ for a fresh variable $j/0$. When translating the forcing (3), we substitute 1 for $i()$ on the left side of the $\vdash$. On the right side, we remove the bound and substitute $l(c) := l()$ for a new variable $l/0$. We still have to parametrise the typing over $k/1$, since the result of the application is not known yet:

$$\emptyset; k/1, l/0; \emptyset; x : [c < 1] \cdot (\mathsf{Nat}[l()] \multimap \mathsf{F}\,\mathsf{Nat}[k(c)]) \vdash^{\mathsf{c}}_{0} \mathsf{force}\,x : \mathsf{Nat}[l()] \multimap \mathsf{F}\,\mathsf{Nat}[k(0)]$$

To translate the typing of the application (4), we have to unify $\mathsf{Nat}[l()]$ with $\mathsf{Nat}[j()]$. Therefore, we substitute $l() := j()$. As the last step, we apply LAM twice:

$$\emptyset; j/0, k/1; \emptyset; \emptyset \vdash^{\mathsf{c}}_{0} t_1 : [c < 1] \cdot (\mathsf{Nat}[j()] \multimap \mathsf{F}\,\mathsf{Nat}[k(c)]) \multimap \mathsf{Nat}[j()] \multimap \mathsf{F}\,\mathsf{Nat}[k(0)]$$

Now, let us type the "thunked successor function" ($\mathsf{thunk}\,s$) as an argument to t_1. We first type $\mathsf{thunk}\,s$ parametrically. Thus, we again parametrise over the number of times $\mathsf{thunk}\,s$ can be forced. In the application to t_1, this bound will of course be instantiated to 1. Let us first type s:

$$\emptyset; j'/0; \emptyset; \emptyset \vdash^{\mathsf{c}}_{0} s := \lambda x.\,\mathsf{calc}\,y \leftarrow \mathsf{Succ}(x)\,\mathsf{in}\,\mathsf{return}\,y : \mathsf{Nat}\left[j'()\right] \multimap \mathsf{F}\,\mathsf{Nat}\left[1 + j'()\right]$$

To thunk this function, we introduce an *ordinary* index variable c, a function variable $i'/0$, and the constraint $c < i'()$. We also increment the arity of j', since c is now a parameter of j'. In other words, $\mathsf{thunk}\, s$ can be forced i' times and afterwards applied with a value of type $\mathsf{Nat}[j'(c)]$, for $c < i'()$.

$$\frac{c; i'/0, j'/1; c < i'(); \emptyset \vdash^{\mathsf{c}}_{0} s : \mathsf{Nat}[j'(c)] \multimap \mathsf{F}\,\mathsf{Nat}[1 + j'(c)]}{\emptyset; i'/0, j'/1; \emptyset; \emptyset \vdash^{\mathsf{c}}_{i'() + \sum_{c<i'()} 0} \mathsf{thunk}\, s : [c < i'()] \cdot (\mathsf{Nat}[j'(c)] \multimap \mathsf{F}\,\mathsf{Nat}[1 + j'(c)])}$$

To type the application $t_1\,(\mathsf{thunk}\, s)$, we have to equalise $[c < 1] \cdot (\mathsf{Nat}[j()] \multimap \mathsf{F}\,\mathsf{Nat}[k(c)]) \equiv [c < i'()] \cdot (\mathsf{Nat}[j'(c)] \multimap \mathsf{F}\,\mathsf{Nat}[1 + j'(c)])$. For this, we need to apply the substitutions $i'() := 1$, $j'(c) := j()$ and $k(c) := 1 + j'(c) = 1 + j()$. Finally, we get:

$$\emptyset; j/0; \emptyset; \emptyset \vdash^{\mathsf{c}}_{0+1} t_1\,(\mathsf{thunk}\, s) : \mathsf{Nat}[j()] \multimap \mathsf{Nat}[1 + j()]$$

Note that only remaining function variable is $j/0$, which stands for the value of the second curried argument of t_1.

Example 2: Multiplicative product and binary modal sum

We explain now how binary modal sums are handled. We translate the following abstract typing of a multiplicative product, for arbitrary values v_1 and v_2 with a free variable x:

$$\frac{(1)\ x : \mathsf{U}\,(\mathsf{Nat} \multimap \mathsf{F}\,\mathsf{Nat}) \vdash v_1 : \hat{A}_1 \qquad (2)\ x : \mathsf{U}\,(\mathsf{Nat} \multimap \mathsf{F}\,\mathsf{Nat}) \vdash v_2 : \hat{A}_2}{x : \mathsf{U}\,(\mathsf{Nat} \multimap \mathsf{F}\,\mathsf{Nat}) \vdash (v_1; v_2) : \hat{A}_1 \otimes \hat{A}_2}$$

Recursively applying the typing annotation algorithm on (1) and (2) yields two pairs index terms (I_1, I_2, and K_1, K_2) that denote how often (exactly) x is used by v_1 and v_2, and the arguments of each of the applications after the forcings. Note that we assume that the algorithm generates the same fresh variables a and $j/(1 + |\phi|)$ for the annotation of the type of the variable x.

$$\phi; j/(1 + |\phi|), \Sigma; x : [a < I_1(\phi)] \cdot (\mathsf{Nat}[K_1(a, \phi)] \multimap \mathsf{F}\,\mathsf{Nat}[j(a, \phi)]) \vdash^{\mathsf{v}}_{M_1} v_1 : A_1$$
$$\phi; j/(1 + |\phi|), \Sigma; x : [a < I_2(\phi)] \cdot (\mathsf{Nat}[K_2(a, \phi)] \multimap \mathsf{F}\,\mathsf{Nat}[j(a, \phi)]) \vdash^{\mathsf{v}}_{M_2} v_2 : A_2$$

In order to build the modal sum over the two contexts, we first need to 'shift' all occurrences of a in the second typing by $I_1(\phi)$. For this, we first substitute all refinement functions that have a as argument. In this case, we only need to substitute $j(a, \phi) := j(a + I_1(\phi), \phi)$. Let ρ denote this function substitution, which we apply to the second typing:

$$\ldots; x : [a < I_2(\phi)\rho] \cdot (\mathsf{Nat}[K_2(a, \phi)\rho] \multimap \mathsf{F}\,\mathsf{Nat}[j(a + I_1(\phi), \phi)]) \vdash^{\mathsf{v}}_{M_2\rho} v_2 : A_2\rho$$

In the next step, we have to compute the modal sum. We define types for x that are equivalent to the above types but are in the right shape so that the binary modal sum is defined:

$$\underline{B} := \mathsf{Nat}[\mathsf{if}\ a < I_1(\phi)\ \mathsf{then}\ K_1(a, \phi)\ \mathsf{else}\ K_2(a, \phi)\rho\{a - I_1(\phi)/a\}] \multimap \mathsf{F}\,\mathsf{Nat}[j(a, \phi)]$$

$$\cdots \vdash [a < I_1(\phi)] \cdot \underline{B} \equiv [a < I_1(\phi)] \cdot (\mathsf{Nat}[K_1(a,\phi)] \multimap \mathsf{F}\,\mathsf{Nat}[j(a,\phi)])$$
$$\cdots \vdash [a < I_2\rho(\phi)] \cdot \underline{B}\{a + I_1(\phi)/a\} \equiv [a < I_2\rho(\phi)] \cdot (\mathsf{Nat}[K_2(a,\phi)\rho] \multimap \mathsf{F}\,\mathsf{Nat}[j(a + I_1(\phi),\phi)])$$

We now apply subsumption in the contexts of the two above typings and derive the desired typing:

$$\frac{\ldots; x : [a < I_1(\phi)] \cdot \underline{B} \vdash (M_1)v_1 : A_1 \qquad \ldots; x : [a < I_2\rho(\phi)] \cdot \underline{B}\{a + I_1(\phi)/a\} \vdash^{\mathsf{v}}_{M_2\rho} v_2 : A_2\rho}{\phi; j/(1 + |\phi|), \Sigma; x : [a < I_1(\phi) + I_2\rho(\phi)] \cdot \underline{B} \vdash^{\mathsf{v}}_{M_1 + M_2\rho} (v_1; v_2) : A_1 \otimes A_2\rho}$$

In the general case where the type of x contains more function variables, we also have to substitute all of them such that the parameter a is shifted by $I_1(\phi)$. Moreover, if there is more than one variable, we have to construct the modal sums over the types in the same way.

Example 3: Forcing and applying twice

In this example, we type $t_3 := \mathsf{thunk}\,\lambda x\,y.\,\mathsf{bind}\,z \leftarrow (\mathsf{force}\,x)\,y\,\mathsf{in}\,(\mathsf{force}\,x)\,z$.[3] We will also apply this function, which shows that the algorithm may also compute recursive equations even for non-recursive functions. The function is simply typed as follows:

$$\cfrac{\cfrac{(1)\ y : \mathsf{Nat}, x : \mathsf{U}\,(\mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}) \vdash^{\mathsf{c}} (\mathsf{force}\,x)\,y : \mathsf{F}\,\mathsf{Nat} \qquad (2)\ z : \mathsf{Nat}, x : \mathsf{U}\,(\mathsf{Nat} \to \mathsf{Nat}) \vdash^{\mathsf{c}} (\mathsf{force}\,x)\,z : \mathsf{F}\,\mathsf{Nat}}{(3)\ y : \mathsf{Nat}, x : \mathsf{U}\,(\mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}) \vdash^{\mathsf{c}} \mathsf{bind}\,z \leftarrow (\mathsf{force}\,x)\,y\,\mathsf{in}\,(\mathsf{force}\,x)\,z : \mathsf{F}\,\mathsf{Nat}}}{\emptyset \vdash^{\mathsf{c}} t_3 : \mathsf{U}\,(\mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}) \to (\mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat})}$$

We first annotate the typings of (1) and (2) as in Example 1. We can assume that the algorithm generates the same annotating variables for x.

$$\emptyset; j/0, k/1;\ \emptyset; x : [a < 1] \cdot (\mathsf{Nat}[j()] \multimap \mathsf{F}\,\mathsf{Nat}[k(a)]),\ y : \mathsf{Nat}[j()] \vdash^{\mathsf{c}}_0 (\mathsf{force}\,x)\,y : \mathsf{F}\,\mathsf{Nat}[k(0)]$$
$$\emptyset; j'/0, k/1; \emptyset; x : [a < 1] \cdot (\mathsf{Nat}[j'()] \multimap \mathsf{F}\,\mathsf{Nat}[k(a)]), z : \mathsf{Nat}[j'()] \vdash^{\mathsf{c}}_0 (\mathsf{force}\,x)\,z : \mathsf{F}\,\mathsf{Nat}[k(0)]$$

We now have to join the two types for x in the contexts. We already know that the new bound will be $1 + 1$, since both parts of the program force x exactly once. As in the previous example, we have to substitute $k(a) := k(1 + a)$ and define $\underline{B} := \mathsf{Nat}[\mathsf{if}\ a < 1\ \mathsf{then}\ j()\ \mathsf{else}\ j'()\{a - 1/a\}] \multimap \mathsf{F}\,\mathsf{Nat}[k(a)]$.

$$\emptyset; k/1, j/0, j'/0; \emptyset; x : [a < 1] \cdot \underline{B}, \qquad\qquad y : \mathsf{Nat}[j()] \vdash^{\mathsf{c}}_0 (\mathsf{force}\,x)\,y : \mathsf{F}\,\mathsf{Nat}[k(0)]$$
$$\emptyset; k/1, j/0, j'/0; \emptyset; x : [a < 1] \cdot \underline{B}\{1 + a/a\}, z : \mathsf{Nat}[j'()] \vdash^{\mathsf{c}}_0 (\mathsf{force}\,x)\,z : \mathsf{F}\,\mathsf{Nat}[k(1)]$$

To apply Bind, we need to unify $\mathsf{Nat}[k(0)] \equiv \mathsf{Nat}[j'()]$. Therefore, we substitute $j'() := k(0)$ in the above two typings, and we get:

$$\emptyset; j/0, k/1; \emptyset; \emptyset \vdash^{\mathsf{c}}_0 t_3 : [a < 2] \cdot (\mathsf{Nat}[\mathsf{if}\ a < 1\ \mathsf{then}\ j()\ \mathsf{else}\ k(0)] \multimap \mathsf{F}\,\mathsf{Nat}[k(a)]) \multimap (\mathsf{Nat}[j()] \multimap \mathsf{F}\,\mathsf{Nat}[k(1)])$$

Note that the refinement of the input of the argument depends on its first output.

Now we apply this function to $\mathsf{thunk}\,s$, which we have already typed in Example 1. After renaming, the typing is: $\emptyset; i'/0, l/1; \emptyset; \emptyset \vdash^{\mathsf{c}}_{i'()} \mathsf{thunk}\,s : [c < i'()] \cdot (\mathsf{Nat}[l(c)] \multimap$

---

[3] Note that this function is not (after thunking) equal to the CBV translation of $\lambda x\,y.\,x\,(x\,y)$, which was chosen as an example in [13]. Although both thunked functions are *observationally* equivalent, they have different $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ refinements.

$\mathsf{F}\,\mathsf{Nat}[1 + l(c)])$. We have to substitute $i'() := 2$, $l(a) := \mathsf{if}\, a < 1 \,\mathsf{then}\, j() \,\mathsf{else}\, k(0)$, and $k(a) := 1 + l(a)$. However, this substitution is circular, so we end up with a recursive definition for $l/1$, which can be easily solved (by unfolding) to a non-recursive index term:

$$\begin{aligned} i'() &:= 2 \\ k(a) &:= 1 + l(a) \\ l(a) &:= \mathsf{if}\, a < 1 \,\mathsf{then}\, j() \,\mathsf{else}\, k(0) = \mathsf{if}\, a < 1 \,\mathsf{then}\, j() \,\mathsf{else}\, 1 + l(0) = \mathsf{if}\, a < 1 \,\mathsf{then}\, j() \,\mathsf{else}\, 1 + j() \\ k(1) &:= 1 + l(1) = 2 + j() \end{aligned}$$

Thus, after simplification, the generated typing has the following judgement:

$$\emptyset; j/0; \emptyset; \emptyset \vdash^{\mathsf{c}}_{2} t_3\,(\mathsf{thunk}\, s) : \mathsf{Nat}[j()] \multimap \mathsf{F}\,\mathsf{Nat}[2 + j()]$$

Example 4: Case distinction

The rule Ifz is a combination of a multiplicative and an additive part. The second and third typing use the same context and weight, since only one of the branches is executed. We first annotate the value typing and the two computation typings of t_1 and t_2. We can assume that the generated signatures are identical in the three typings.

$$\phi; \Sigma; \emptyset; \Delta_1 \vdash^{\mathsf{v}}_{K_1} v : \mathsf{Nat}[J] \qquad \phi; \Sigma; \emptyset; \Delta_2 \vdash^{\mathsf{c}}_{K_2} t_1 : \underline{B}_1 \qquad \phi; \Sigma; \emptyset; \Delta_3 \vdash^{\mathsf{c}}_{K_3} t_2 : \underline{B}_2$$

We define a new context $\Delta_{23} := \mathsf{if}\, J \equiv 0 \,\mathsf{then}\, \Delta_2 \,\mathsf{else}\, \Delta_3$, $\underline{B} := \mathsf{if}\, J \equiv 0 \,\mathsf{then}\, \underline{B}_1 \,\mathsf{else}\, \underline{B}_2$, and $K_{23} := \mathsf{if}\, J \equiv 0 \,\mathsf{then}\, K_2 \,\mathsf{else}\, K_3$.[4] Here, we use a case-distinction operator on types with the same shape, as in Definition 5.33 in Section 5.5.3. By adding the constraints $J \equiv 0$ and $J > 0$ to the second and third typing, we can substitute Δ_1 and Δ_2 with Δ_{23}, respectively. Finally, we build the binary modal sum over Δ_1 and Δ_{23} as in Example 2 and apply Ifz.

Example 5: Thunks and bounded sums

We will now annotate the following abstract thunked computation and explain how bounded sums are built: $x : \mathsf{U}\,(\mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}) \vdash^{\mathsf{v}} \mathsf{thunk}\, t : \mathsf{U}\,(\mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat})$. We first run the algorithm on t and introduce a fresh refinement variable $i/0$ and the bound $a < i()$:

$$a; i/0, k/2, l/1; a < i(); x : A(a) \vdash^{\mathsf{c}}_{M(a)} t : \mathsf{Nat}[l(a)] \multimap \mathsf{F}\,\mathsf{Nat}[N(a)]$$

with $A(a) := [b < J(a)] \cdot (\mathsf{Nat}[K(a,b)] \multimap \mathsf{F}\,\mathsf{Nat}[k(a,b)])$, where $K(\cdot,\cdot)$, $N(\cdot)$, and $M(\cdot)$ stand for concrete index terms (that may refer to $k/2$ and $l/1$). To create the bounded sum, we substitute $k(a,b) := \hat{k}(b + \sum_{d<a} J(d))$ for a fresh refinement index variable $\hat{k}/1$. (In the following, we abbreviate this substitution to ρ).

$$\begin{aligned} a; i/0, k/2, \hat{k}/1, l/1; a < i(); x : [b < J(a)] \cdot (\mathsf{Nat}[K(a,b)]\,\rho \multimap \mathsf{F}\,\mathsf{Nat}[\hat{k}(b + \textstyle\sum_{d<a} J(d))]) \\ \vdash^{\mathsf{c}}_{M(a)\rho} t : \mathsf{Nat}[l(a)] \multimap \mathsf{F}\,\mathsf{Nat}[N(a)]\,\rho \end{aligned}$$

---

[4] Remember that we can use $\vDash J \equiv 0$ instead of $\vDash 0 \gtrsim J$, since the generated typings are precise.

Then, we use the 'function' $f^{-1}(c) := \mathit{findSlot}_{\underline{a}}\,(i())\,(J(a))\,c$ to construct the modal sum $\sum_{a<i()}[b < J(a)] \cdot \underline{B}\{b + \sum_{d<a} J(d)/c\} = [c < \sum_{a<i()} J(a)] \cdot \underline{B}$ with the following equivalence:

$$\begin{aligned} \theta^{-1} &:= \{\pi_1(f^{-1}(c))/a, \pi_2(f^{-1}(c))/b\} \\ \underline{B} &:= \mathsf{Nat}[K(a,b)]\,\rho\theta^{-1} \multimap \mathsf{F}\,\mathsf{Nat}[\hat{k}] \equiv (\mathsf{Nat}[K(a,b)]\,\rho \multimap \mathsf{F}\,\mathsf{Nat}[\hat{k}(b + \textstyle\sum_{d<a} J(d))])\theta^{-1} \\ \ldots; a < i() &\vdash [b < J(a)] \cdot (\mathsf{Nat}[K(a,b)]\,\rho \multimap \mathsf{F}\,\mathsf{Nat}[\hat{k}(b + \sum_{d<a} J(d))]) \equiv [b < J(a)] \cdot \underline{B}\{b + \sum_{d<a} J(d)/c\} \end{aligned}$$

We apply the above equivalence to the typing of t and we can finally apply Thunk:

$$\frac{a; i/0, \hat{k}/1, l/1; a < i(); x : [b < J(a)] \cdot \underline{B}\{b + \sum_{d<a} J(d)/c\} \vdash^{\mathsf{c}}_{M(a)} t : \mathsf{Nat}[l(a)] \multimap \mathsf{F}\,\mathsf{Nat}[N(a)]\,\rho}{\emptyset; i/0, \hat{k}/1, l/1; \emptyset; x : [c < \sum_{a<i()} J(a)] \cdot \underline{B} \vdash^{\mathsf{v}}_{i() + \sum_{a<i()} M(a)} \mathsf{thunk}\, t : [a < i()] \cdot (\mathsf{Nat}[l(a)] \multimap \mathsf{F}\,\mathsf{Nat}[N(a)])}$$

Example 6: Recursion

We now discuss the fixpoint case.[5] The basic idea of annotating this typing is that annotating the body yields a description of the recursion tree:

- After annotating the body, we introduce a fresh ordinary index variable b and increment the arity of function variables.
- This yields an index term $I(b)$ which denotes how often x is forced at each point in the recursion tree.
- $H := \triangle_b^1\, I(b)$ denotes the size of the recursion tree.
- We already know the weight $J(b)$ of each node, so we can define the total weight of the fixpoint as $M := H \dot{-} 1 + \sum_{b<H} J(b)$.
- After annotating the typing of the body, we have to define dℓPCF$_{\mathsf{pv}}$ types $\underline{B}_1, \underline{B}_2$ that satisfy the equivalence in Fix. For this, we have to introduce mutually recursive equations, which (optionally) can be simplified manually.
- Finally, we apply Fix. The final type is equivalent to $\underline{B}_2\{0/b\}$.

As an example, we consider the following primitive recursive function, which always returns the constant $\underline{0}$. In the following simple typing, we abbreviate $\underline{\hat{B}} := \mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}$.

$$\frac{x : \mathsf{U}\,\underline{\hat{B}} \vdash^{\mathsf{c}} t_6 := \lambda y.\, \mathsf{ifz}\, y\, \mathsf{then\ return}\, \underline{0}\, \mathsf{else\ calc}\, y' \leftarrow \mathsf{Pred}(y)\, \mathsf{in}\, (\mathsf{force}\, x)\, y' : \underline{\hat{B}}}{\emptyset \vdash^{\mathsf{c}} \mu x.\, t_6 : \underline{\hat{B}}}$$

---

[5] This case is broken in [13]. In particular, *Lemma 4.6* is wrong. It postulates an algorithm that generates an equational program that equalises $\tau\{I/a\} \equiv \sigma$, where one of the types τ and σ is positively annotated and the other is negatively annotated. However, such an equivalence does not make sense if a itself is a free variable of I, which happens in the fixpoint case. The same bug can be observed in the (non-public) `OCaml` code, where non-wellformed equations are defined.

Typing the body t_6 is routine. We introduce variables $k/2$ and $g/1$. $g(b)$ stands for the input (on the right side of the $\vdash$) of the b-th node in the forest. $k(a,b)$ stands for the a-th result of applying x at the b-th node in the tree.

$$b; g/1, k/2; b < H; x : [a < I(b)] \cdot \underline{B}_1(a,b) \vdash^{\mathsf{c}}_{J(b)} t_6 : \underline{B}_2(b)$$

$$\begin{array}{rlrl} \underline{B}_1(a,b) & := \mathsf{Nat}[G(a,b)] \multimap \mathsf{F}\,\mathsf{Nat}[k(a,b)] & \qquad \underline{B}_2(b) & := \mathsf{Nat}[g(b)] \multimap \mathsf{F}\,\mathsf{Nat}[K(b)] \\ H & := \triangle^1_b\, I(b) & M & := (H \mathbin{\dot{-}} 1) + \sum_{b<H} J(b) \end{array}$$

$$\begin{array}{rll} J(b) & := 0 & \text{(weight at the } b^{th} \text{ node in the recursion tree)} \\ I(b) & := \mathsf{if}\; g(b) \equiv 0 \;\mathsf{then}\; 0 \;\mathsf{else}\; 1 & \text{(number of recursive calls (= no. of children) at/of node } b\text{)} \\ G(a,b) & := \mathsf{if}\; g(b) \equiv 0 \;\mathsf{then}\; \bot \;\mathsf{else}\; g(b) \mathbin{\dot{-}} 1 & \text{(input of } x \text{ at the } b^{th} \text{ node in the recursion tree)} \\ K(b) & := \mathsf{if}\; g(b) \equiv 0 \;\mathsf{then}\; 0 \;\mathsf{else}\; k(0,b) & \text{(output of the body at the } b^{th} \text{ node)} \end{array}$$

Note that $G(a,b)$ is not defined in case $g(b) \equiv 0$. Intuitively, this is because x is not called at the leaf of the recursion tree. The crucial step in the fixpoint typing is that we have to ensure that the following subtyping holds:

$$\begin{array}{l} a, b; g/1, h/2; a < I(b), b < H \vdash \underline{B}_2(\mathsf{child}_b(a)) \equiv \underline{B}_1(a,b) \\ \qquad\qquad\qquad\qquad\qquad\qquad \Longleftrightarrow \\ \cdots \vdash \mathsf{Nat}[g(\mathsf{child}_b(a))] \multimap \mathsf{F}\,\mathsf{Nat}[K(\mathsf{child}_b(a))] \equiv \mathsf{Nat}[G(a,b)] \multimap \mathsf{F}\,\mathsf{Nat}[k(a,b)] \end{array}$$

where $\mathsf{child}_b(a) := 1 + b + (\triangle^a_c\, I\{1 + b + c/b\})$, which is an encoding for the number of the node that is the a^{th} child of a node b in the recursion tree. (In this example, we have $\mathsf{child}_b(a) = b + 1$, since the recursion tree is linear.)

To solve the equivalence, we first substitute $k(a,b) := K(\mathsf{child}_b(a))$. However, $g(0)$ is not specified by the above equivalence, since $\mathsf{child}_b(a) > 0$. In fact, $g(0)$ denotes the 'external' input of the fixpoint (at the root of the recursion tree), so we have to generalise over this value by introducing a fresh function variable $d/0$. For $b > 0$, we can define $g(b)$ using an 'inverse' of the function $\mathsf{child}_b(a)$. Let $\pi_1(\mathsf{parent}(b))$, for $b > 0$, denote the number of the parent node of b in the recursion tree and let $\pi_2(\mathsf{parent}(b))$ be the child number. In other words, we have $\cdots \vDash b \equiv \mathsf{child}_{\pi_1(\mathsf{parent}(b))}(\pi_2(\mathsf{parent}(b)))$. In our example, we simply have $\mathsf{parent}(b) = (b \mathbin{\dot{-}} 1, 0)$ for $b > 0$, since the recursion tree is linear. Now, we can substitute $g/1$:

$$g(b) := \mathsf{if}\; b \equiv 0 \;\mathsf{then}\; d() \;\mathsf{else}\; \mathsf{let}\; (b,a) := \mathsf{parent}(b) \;\mathsf{in}\; G(a,b)$$

However, note that we have just introduced *mutually recursive* equations for the index terms K and G:

$$\begin{array}{rlrl} K(b) & = \mathsf{if}\; g(b) \equiv 0 \;\mathsf{then}\; 0 \;\mathsf{else}\; k(0,b) & \qquad g(b) & = \mathsf{if}\; b \equiv 0 \;\mathsf{then}\; d() \;\mathsf{else}\; G(0, b \mathbin{\dot{-}} 1) \\ G(a,b) & = \mathsf{if}\; g(b) \equiv 0 \;\mathsf{then}\; \bot \;\mathsf{else}\; g(b) \mathbin{\dot{-}} 1 & k(a,b) & = K(b+1) \end{array}$$

Since the annotation algorithm cannot solve recurrence equations, it has to output the equations for K and G (possibly after substituting g and k). The algorithm is done afterwards, since the subtypings hold by definition. The final type is $\underline{B}_2(0)$, which is by definition equivalent to $\mathsf{Nat}[d()] \multimap \mathsf{F}\,\mathsf{Nat}[K(0)]$.

We can solve the recurrences and simplify the weight M:

$$\begin{aligned} g(b) &= \mathsf{if}\ b \equiv 0\ \mathsf{then}\ d()\ \mathsf{else}\ \mathsf{if}\ g(b-1) \equiv 0\ \mathsf{then}\ \bot\ \mathsf{else}\ g(b-1) - 1 = d() \mathbin{\dot-} b \quad \text{(by induct. if } b \le d()\text{)} \\ K(b) &= \mathsf{if}\ g(b) \equiv 0\ \mathsf{then}\ 0\ \mathsf{else}\ K(b+1) = \mathsf{if}\ d() - b \equiv 0\ \mathsf{then}\ 0\ \mathsf{else}\ K(b+1) = 0 \\ I(b) &= \mathsf{if}\ b \le d()\ \mathsf{then}\ 0\ \mathsf{else}\ 1 \qquad H = \triangle_b^1\, I(b) = 1 + d() \qquad M = (H \mathbin{\dot-} 1) + \textstyle\sum_{b<H} J(b) = d() \end{aligned}$$

Therefore, the final (simplified) typing has following judgement: $\emptyset; d/0; \emptyset \vdash^{\mathsf{c}}_{d()} \mu x.\, t_6 : \mathsf{Nat}[d()] \multimap \mathsf{F}\,\mathsf{Nat}[0]$. By the soundness theorem of $\mathsf{d\ell PCF_{pv}}$, this means that $(\mu x.\, t_6)\, \underline{n} \Downarrow_n \underline{0}$ for all constants n.

Example 7: Call-by-value iteration

We have demonstrated in the previous example how to annotate arbitrary fixpoints. However, the generated weight uses an explicit encoding of the recursion tree. This can make reasoning over the index terms complicated in general. However, we can derive admissible typing rules for restricted forms of recursion. In Section 5.6, we have already shown that higher-order iteration can be embedded in $\mathsf{d\ell PCF_v}$. Of course, we can also implement System T like iteration in CBPV:

$$\mathsf{iter}\, t_1\, t_2 := \mu f.\, \lambda x.\, \mathsf{ifz}\ x\ \mathsf{then}\ t_2\ \mathsf{else}\ \mathsf{calc}\ x' \leftarrow \mathsf{Pred}(x)\ \mathsf{in}\ \mathsf{bind}\ y \leftarrow \mathsf{force}\, f\, x'\ \mathsf{in}\ (\mathsf{force}\,(\mathsf{thunk}\, t_1))\, y$$

Note that we use $\mathsf{force}\,(\mathsf{thunk}\, t_1)$ to increment the cost by one for each application of t_1. Since we do not have to thunk the fixpoint computation, the admissible typing rule is simpler as in $\mathsf{d\ell T}$:

$$\frac{a, \phi; \Sigma; a < I, \Phi; \Delta_1 \vdash^{\mathsf{c}}_{M_1} t_1 : \hat{A}\{1 + a/a\} \multimap \mathsf{F}\,\hat{A} \qquad \phi; \Sigma; \Phi; \Delta_2 \vdash^{\mathsf{c}}_{M_2} t_2 : \mathsf{F}\,\hat{A}\{i()/a\}}{\phi; \Sigma; \Phi; (\sum_{a<I} \Delta_1) \uplus \Delta_2 \vdash^{\mathsf{c}}_{I + \left(\sum_{a<I} M_1\right) + M_2} \mathsf{iter}\, t_1\, t_2 : \mathsf{Nat}[I] \multimap \mathsf{F}\,\hat{A}\{0/a\}}$$

We can extend the annotation algorithm with a special case for $\mathsf{iter}\, t_1\, t_2$:

- We recursively annotate the typing of t_2 as usual.
- Then we annotate the typing of t_1 and add a function variable $i/0$ (for I). We set the constraint to $a < i()$, where a is a fresh ordinary index variable.
- This computes a negative annotation for the simple type $\hat{A}$ and positive annotation for $\mathsf{F}\,\hat{A}$.
- We define a $\mathsf{d\ell PCF_{pv}}$ type A and compute substitutions such that the first type is equivalent to $A\{1 + a/a\} \multimap \mathsf{F}\,A$ and the second type is equivalent to $\mathsf{F}\,A\{i()/a\}$.
- Finally, we build the bounded and binary sum as in the previous cases and apply the above admissible typing rule.

As an example, we annotate the following typing:

$$\emptyset \vdash^{\mathsf{c}} \lambda x.\, t_7 := \mathsf{iter}\,(\mathsf{force}\, x)\,(\mathsf{force}\, x\, 1) : \mathsf{U}\,(\mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat}) \to (\mathsf{Nat} \to \mathsf{F}\,\mathsf{Nat})$$

Recursively applying the annotation algorithm to the bodies of the iteration is routine:

$$a; i/0; j/1, k/2; a < i(); x : [b < 1] \cdot (\mathsf{Nat}[j(a)] \multimap \mathsf{F}\,\mathsf{Nat}[k(a,b)]) \vdash^{\mathsf{c}}_{0} \mathsf{force}\, x : \mathsf{Nat}[j(a)] \multimap \mathsf{F}\,\mathsf{Nat}[k(a,0)]$$
$$\emptyset; k'/1; \emptyset; x : [b < 1] \cdot (\mathsf{Nat}[1] \multimap \mathsf{F}\,\mathsf{Nat}\big[k'(b)\big]) \vdash^{\mathsf{c}}_{0} \mathsf{force}\, x\, \underline{1} : \mathsf{F}\,\mathsf{Nat}\big[k'(0)\big]$$

Now we define a type A that satisfies the following two equalities:

$$a; i/0, j/1, k/2; a < i() \vdash A\{1 + a/a\} \multimap \mathsf{F}\, A \equiv \mathsf{Nat}[j(a)] \multimap \mathsf{F}\,\mathsf{Nat}[k(a,0)]$$
$$\emptyset; k'/1; \emptyset \vdash \mathsf{F}\, A\{i()/a\} \equiv \mathsf{F}\,\mathsf{Nat}[k'(0)]$$

The (positively annotated) type A is defined by case analysis on a: If $a < i()$, it is equivalent to the result type of t_1 and otherwise to the type of t_2.

$$A := \mathsf{if}\ a < i()\ \mathsf{then}\ \mathsf{Nat}[k(a,0)]\ \mathsf{else}\ \mathsf{Nat}[k'(0)] = \mathsf{Nat}\big[\mathsf{if}\ a < i()\ \mathsf{then}\ k(a,0)\ \mathsf{else}\ k'(0)\big]$$

To satisfy the left equality, we also need to substitute $j/1$:

$$A\{1 + a/a\} = \mathsf{Nat}\big[\mathsf{if}\ 1 + a < i()\ \mathsf{then}\ k(1 + a, 0)\ \mathsf{else}\ k'(0)\big] \equiv \mathsf{Nat}[j(a)]$$
$$\Longrightarrow j(a) := J(a) := \mathsf{if}\ 1 + a < i()\ \mathsf{then}\ k(1 + a, 0)\ \mathsf{else}\ k'(0)$$

We therefore apply subsumption and the substitution of $j/1$ on the two typings:

$$a; i/0, k'/1, k/2; a < i(); x : [b < 1] \cdot (\mathsf{Nat}[J(a)] \multimap \mathsf{F}\,\mathsf{Nat}[k(a,b)]) \vdash^{\mathsf{c}}_{0} \mathsf{force}\, x : A\{1 + a/a\} \multimap \mathsf{F}\, A$$
$$\emptyset; i/0, k'/1, k/2; \emptyset; x : [b < 1] \cdot (\mathsf{Nat}[1] \multimap \mathsf{F}\,\mathsf{Nat}[k'(b)]) \vdash^{\mathsf{c}}_{0} \mathsf{force}\, x\, \underline{1} : \mathsf{F}\, A\{i()/a\}$$

We now build the bounded modal sum over the context of the above two typings. We have to build a bounded sum over the first context (as in Example 5) and build a binary sum of this sum and the second context. At the end, only the refinement variables $\hat{k}/1$ and $i/0$ remain.

$$\frac{\emptyset; i/0, \hat{k}/1; \emptyset; x : [c < i() + 1] \cdot (\mathsf{Nat}[J'(c)] \multimap \mathsf{F}\,\mathsf{Nat}[\hat{k}(c)]) \vdash^{\mathsf{c}}_{i()} t_7 : \mathsf{Nat}[i()] \multimap \mathsf{F}\,\mathsf{Nat}[\hat{k}(0)]}{\emptyset; i/0, \hat{k}/1; \emptyset; \emptyset \vdash^{\mathsf{c}}_{i()} \lambda x.\, t_7 : [c < i() + 1] \cdot (\mathsf{Nat}[J'(c)] \multimap \mathsf{F}\,\mathsf{Nat}[\hat{k}(c)]) \multimap \mathsf{Nat}[i()] \multimap \mathsf{F}\,\mathsf{Nat}[\hat{k}(0)]}$$

$$\begin{aligned} J'(c) &:= \mathsf{if}\ c < i()\ \mathsf{then}\ (J(c)[k(a,b) := \hat{k}(a), k'(b) := \hat{k}(b + i())])\ \mathsf{else}\ 1 \\ &= \mathsf{if}\ c < i()\ \mathsf{then}\ (\mathsf{if}\ 1 + c < i()\ \mathsf{then}\ \hat{k}(c+1)\ \mathsf{else}\ \hat{k}(0 + i()))\ \mathsf{else}\ 1 \\ &= \mathsf{if}\ c < i()\ \mathsf{then}\ \hat{k}(c+1)\ \mathsf{else}\ 1 \quad \text{(simplification for this specific example)} \end{aligned}$$

Now, let us type $(\lambda x.\, t_7)\,(\mathsf{thunk}\, s)$, where $\mathsf{thunk}\, s$ is typed as in the first example:

$$\emptyset; i'/0, l/1; \emptyset; \emptyset \vdash^{\mathsf{v}}_{i'()} \mathsf{thunk}\, s : [c < i'()] \cdot (\mathsf{Nat}[l(c)] \multimap \mathsf{F}\,\mathsf{Nat}[1 + l(c)])$$

We have to substitute $i'() := i() + 1$ (since the successor function is applied $i() + 1$ times) and $l(c) := J'(c)$, and $\hat{k}(c) := 1 + l(c)$. The only remaining refinement index variable is $i/0$. However, note that the substitutions are circular, so J' is a recursively defined function. We can manually solve the recurrence and compute $\hat{k}(0)$, which is the final result.

$$l(c) := J'(c) \qquad\qquad \hat{k}(c) := 1 + l(c)$$
$$J'(c) := \mathsf{if}\ c < i()\ \mathsf{then}\ \hat{k}(c+1)\ \mathsf{else}\ 1 = \mathsf{if}\ c < i()\ \mathsf{then}\ 1 + J'(c+1)\ \mathsf{else}\ 1 = 1 + (i() \dot{-} c)$$
$$\hat{k}(0) = 1 + l(0) = 1 + J'(0) = 2 + i()$$

We thus get the final typing $\emptyset; i/0; \emptyset; \emptyset \vdash^{\mathsf{c}}_{i()+i()+1} (\lambda x.\, t_7)\,(\mathsf{thunk}\, s) : \mathsf{Nat}[i()] \multimap \mathsf{Nat}[2 + i()]$.

Example 8: Non-termination

In our generalisation of dℓPCF, we can type non-terminating computations. A diverging program must have weight $\bot$ (which can be thought as *infinite* resource usage) and can have any type $\mathsf{Nat}[K]$, in particular $\mathsf{Nat}[\bot]$ (where $\bot$ can be thought as *undefined* or *unknown*). For example, we can type a diverging program that forces a variable in every iteration, which is simply typed as follows:

$$\frac{x : \mathsf{U\,F\,Nat}, y : \mathsf{U\,F\,Nat} \vdash^{\mathsf{c}} t_8 := \mathsf{bind}\ \_ \leftarrow \mathsf{force}\, y \,\mathsf{in}\, \mathsf{force}\, x : \mathsf{F\,Nat}}{y : \mathsf{U\,F\,Nat} \vdash^{\mathsf{c}} \mu x.\, t_8 : \mathsf{F\,Nat}}$$

Let K be an arbitrary index term (e.g. any constant or $\bot$). We can then assign the type $\mathsf{F\,Nat}[K]$ to the fixpoint computation:

$$\frac{b; l/1; b < H; x : [a < 1] \cdot (\mathsf{F\,Nat}[K]), y : [a < 1] \cdot (\mathsf{F\,Nat}[l(a)]) \vdash^{\mathsf{c}}_{0} t_8 : \mathsf{F\,Nat}[K]}{\emptyset; l/1; \emptyset; y : [a < \bot] \cdot (\mathsf{F\,Nat}[l(a)]) \vdash^{\mathsf{c}}_{M} \mu x.\, t_8 : \mathsf{F\,Nat}[K]}$$

where $H := \triangle^{1}_{b}\, 1 \equiv \bot$ and $M := (H \dot{-} 1) + \sum_{b<H} 0 \equiv \bot$.

8.2 Polymorphism

We extend our type system with type variables (e.g. α) and abstraction over type variables. Every type variable α has an arity, which is the number of index terms arguments. For example, if α has the arity 2, written as $\alpha/2$, then $\alpha(I_1, I_2)$ is a well-formed value type. The signature context Σ now assigns arities to type variables. Finally, we introduce quantification over value types at the level of computation types: $\forall \alpha/n$.[6]

$$\begin{aligned}
\text{Value types:}\quad & A ::= \cdots \mid \alpha(I_1, \ldots, I_n) \\
\text{Computation types:}\quad & \underline{B} ::= \cdots \mid \forall \alpha/n.\, \underline{B} \\
\text{Signature context:}\quad & \Sigma ::= \emptyset \mid \alpha/n, \Sigma \\
\text{Computations:}\quad & t ::= \cdots \mid \Lambda.\, t \mid t\langle\rangle \\
\text{Terminal comp.:}\quad & T ::= \cdots \mid \Lambda.\, t
\end{aligned}$$

Type variables α/n are placeholders for value types that are abstracted over n index variables. For example, we may apply the instantiation $\alpha(a, b) := \mathsf{Nat}[a + b]$ to $\mathsf{F}\,\alpha(I_1, I_2)$, which yields $\mathsf{F\,Nat}[I_1 + I_2]$.

Into the syntax of computations, we introduce type abstraction and instantiation operators, Λ and $\langle\rangle$. These operators are uninteresting from the perspective of the operational semantics; they just denote the places at which types are abstracted and instantiated. We

---

[6] Other combinations are also possible, e.g. $\forall \beta/n.\, A$ as a value type abstracted over a computation type (as mentioned in [27]). However, we will only need one kind of quantification in our examples.

add the following rules for Λ and $\langle\rangle$:[7]

$$\frac{\alpha/n, \Sigma; \Phi \vDash \alpha(I_i) \equiv \alpha(J_n) \text{ for } i = 1, \ldots, n}{\alpha/n, \Sigma; \Phi \vdash \alpha(I_1, \ldots, I_n) \sqsubseteq \alpha(J_1, \ldots, J_n)} \qquad \frac{\alpha/n, \Sigma; \phi; \Phi \vdash \underline{B}_1 \sqsubseteq \underline{B}_2}{\Sigma; \phi; \Phi \vdash \forall\alpha/n.\, \underline{B}_1 \sqsubseteq \forall\alpha/n.\, \underline{B}_2}$$

$$\frac{\alpha/n, \Sigma; \phi; \Phi; \Gamma \vdash^{\mathsf{c}}_{M} t : \underline{B}}{\Sigma; \phi; \Phi; \Gamma \vdash^{\mathsf{c}}_{M} \Lambda.\, t : \forall\alpha/n.\, \underline{B}} \qquad \frac{\Sigma; \phi; \Phi; \Gamma \vdash^{\mathsf{c}}_{M} t : \forall\alpha/n.\, \underline{B}}{\Sigma; \phi; \Phi; \Gamma \vdash^{\mathsf{c}}_{M} t\langle\rangle : \underline{B}[\alpha(a_1, \ldots, a_n) := A]} \qquad \frac{}{(\Lambda.\, t)\langle\rangle \succ_0 t}$$

8.2.1 Church encoding

Church encoding is a scheme to encode recursive (inductive) data types using polymorphism. For example, the type of natural numbers can be encoded as $\forall\alpha.\, \alpha \to (\alpha \to \alpha) \to \alpha$. The number n is encoded in System F as $\Lambda.\, \lambda x.\, \lambda f.\, f^n(x)$, which means that the function f is applied n times to x, similar to the iteration operator of System T.

Bounded numbers In [19], it was noted that it is possible in BLL to exploit polymorphism and bounded exponentials to define a type $\mathsf{Nat}_{\leq I}$ that encodes natural numbers less then or equal to I. Unsurprisingly this is also possible in our polymorphic extension of $\mathsf{d\ell PCF_{pv}}$:

$$\mathsf{Nat}_{\leq I} := \forall\alpha/1.\, \alpha(0) \multimap [a < I] \cdot (\alpha(a) \multimap \mathsf{F}\, \alpha(1 + a)) \multimap \mathsf{F}\, \alpha(I)$$

This (computation) type expresses that the 'successor' function can be applied at most I times, where the index variable a stands for the number of the current iteration. Note that the type $\mathsf{Nat}_{\leq I}$ is similar to the (polymorphic version of the) type of iteration in System T (see rule ITER in Figure 4.2).

It is easy to convert a Church-encoded 'number' of type $\mathsf{Nat}_{\leq I}$ into a computation of type $\mathsf{F}\,\mathsf{Nat}[I]$. For this, we instantiate $\alpha(a) := \mathsf{Nat}[a]$, and we apply this to $\underline{0}$ and the 'native' thunked successor function:

$$\dfrac{\dfrac{\cdots \vdash^{\mathsf{c}}_{M} t : \mathsf{Nat}_{\leq I}}{\cdots \vdash^{\mathsf{c}}_{M} t\langle\rangle : \cdots} \qquad \dfrac{}{\cdots \vdash^{\mathsf{v}}_{0} \underline{0} : \mathsf{Nat}[0]} \qquad \dfrac{}{\cdots \vdash^{\mathsf{v}}_{I} \mathsf{thunk}\, s : [a < I] \cdot (\mathsf{Nat}[a] \multimap \mathsf{F}\,\mathsf{Nat}[1 + a])}}{\cdots \vdash^{\mathsf{c}}_{M+I} t\langle\rangle\, \underline{0}\, (\mathsf{thunk}\, s) : \mathsf{F}\,\mathsf{Nat}[I]}$$

The successor function on Church-encoded numbers can be implemented and typed as follows, where we parametrise over the natural index variable a that stands for the bound of the input.

$$\dfrac{\begin{array}{r} \cdots, f : [b < a] \cdot (\alpha(b) \multimap \mathsf{F}\,\alpha(1 + b)) \\ \vdash^{\mathsf{c}}_{0} (\mathsf{force}\, n)\langle\rangle\, x\, f : \mathsf{F}\,\alpha(a) \end{array} \qquad \begin{array}{r} \cdots, y : \alpha(a), f : [b < 1] \cdot (\alpha(a + b) \multimap \mathsf{F}\,\alpha(a + 1 + b)) \\ \vdash^{\mathsf{c}}_{0} \mathsf{force}\, f\, y : \mathsf{F}\,\alpha(1 + a) \end{array}}{\begin{array}{r} \alpha/1; a; \emptyset; n : [c < 1] \cdot \mathsf{Nat}_{\leq a}, x : \alpha(0), f : [b < 1 + a] \cdot (\alpha(b) \multimap \mathsf{F}\,\alpha(1 + b)) \\ \vdash^{\mathsf{c}}_{0} \mathsf{bind}\, x \leftarrow (\mathsf{force}\, n)\langle\rangle\, x\, f\, \mathsf{in}\, \mathsf{force}\, f\, y : \mathsf{F}\,\alpha(1 + a) \end{array}}$$

$$\vdots$$

$$\emptyset; a; \emptyset; \emptyset \vdash^{\mathsf{c}}_{0} \lambda n.\, \Lambda.\, \lambda x\, f.\, \mathsf{bind}\, x \leftarrow (\mathsf{force}\, n)\langle\rangle\, x\, f\, \mathsf{in}\, \mathsf{force}\, f\, y : ([c < 1] \cdot \mathsf{Nat}_{\leq a}) \multimap \mathsf{Nat}_{\leq 1+a}$$

---

[7] In BLL, the index variables in an instantiation $\alpha(a_1, \ldots, a_n) := A$ may only be used at the positive positions in A. Thus, BLL has the weaker premise $\vDash I \leq J$ for $\vdash \alpha(I) \sqsubseteq \alpha(J)$.

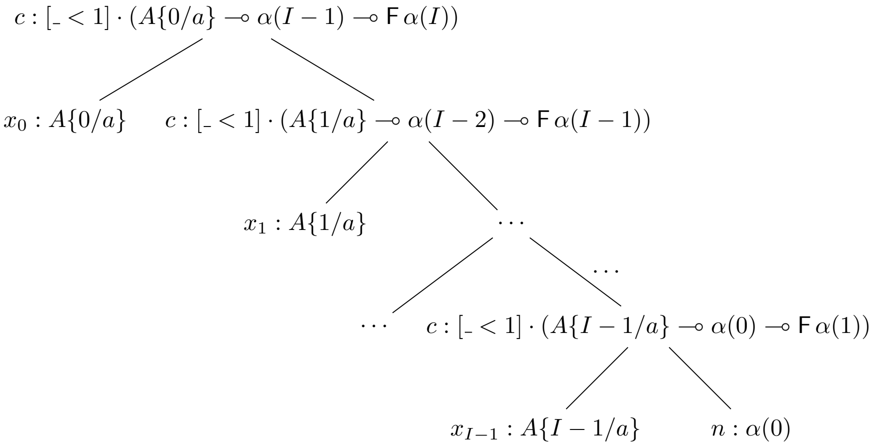

Figure 8.2: Visualisation of the type $\mathsf{List}_{a<I}A$ as a recursion tree of the *right fold* operation

Observe that in the typing, the argument forces the new successor function f I-times, and f is forced one more time afterwards. Moreover, note that the typing has weight 0, since the computation just 'consumes' resources from its input and does not allocate new resources.

Bounded lists We can define the type $\mathsf{List}_{a<I}A$ of lists with at most I elements. Here, a may be a free index variable of A that ranges from 0 to $I \dot{-} 1$.

$$\mathsf{List}_{a<I} := \forall\alpha/1.\, [a < I] \cdot (A\{I \dot{-} a \dot{-} 1/a\} \multimap \alpha(a) \multimap \mathsf{F}\,\alpha(1+a)) \multimap \alpha(0) \multimap \mathsf{F}\,\alpha(I)$$

The type corresponds to the type of the *right fold* operation, which replaces the *cons* constructor with a function f and the *nil* constructor with a value n, which is visualised in Figure 8.2. Note that the type of the function argument implies that the function c can be applied (at most) I times. The 0^{th} application has the head of the list as first argument and the result of the fold of the tail of the list. For example, the list $[0; 1; 2]$ can be encoded as the following computation of type $\mathsf{List}_{a<3}\mathsf{Nat}[a]$:

$$\Lambda.\, \lambda c\, n.\, \mathsf{bind}\, x_1 \leftarrow (\mathsf{force}\, c)\, \underline{2}\, n\, \mathsf{in}\, \mathsf{bind}\, x_2 \leftarrow (\mathsf{force}\, c)\, \underline{1}\, x_1\, \mathsf{in}\, (\mathsf{force}\, c)\, \underline{0}\, x_1$$

Note that although this function is observationally equivalent to the (unthunked) translation of the CBV function $\Lambda.\, \lambda c\, n.\, c\, \underline{0}\, (c\, \underline{1}\, (c\, \underline{2}\, n))$, these computations do not have equivalent types:

$$\begin{aligned} \mathsf{List}_{a<3}\mathsf{Nat}[a] &= \forall\alpha/1.\, [a < 3] \cdot (\mathsf{Nat}[2 \dot{-} a] \multimap \alpha(a) \multimap \mathsf{F}\,\alpha(1+a)) \multimap \alpha(0) \multimap \mathsf{F}\,\alpha(3) \\ &\not\equiv \forall\alpha/1.\, [a < 3] \cdot (\mathsf{Nat}[a] \multimap \alpha(2 \dot{-} a) \multimap \mathsf{F}\,\alpha(3 \dot{-} a)) \multimap \alpha(0) \multimap \mathsf{F}\,\alpha(3) \end{aligned}$$

Observe that 'order' the refinements in the second arrow type is reversed (i.e. $a - 2$ is substituted for a). The reason for this is that the modal sum operator contracts the types of variables in syntactic order, not in the execution order. In the first computation, the left-most use of the variable c corresponds to the application with 2, and 0 in the second computation.

Of course, we can also type the constructors. For example, the typing derivation for the constructor *cons* is similar to the derivation for the successor function.

$$\begin{aligned}
&\emptyset;\emptyset;\emptyset;\emptyset \vdash^{\mathsf{c}}_{0} \mathit{nil} := \Lambda.\, \lambda c\, n.\, \mathsf{return}\, n : \forall\alpha/1.\, \mathsf{List}_{a<0}\alpha(a)\\
&\emptyset;\phi;\emptyset;\emptyset \vdash^{\mathsf{c}}_{0} \mathit{cons} := \Lambda.\, \lambda x\, xs.\, \Lambda.\, \lambda c\, n.\, \mathsf{bind}\, y \leftarrow (\mathsf{force}\, xs)\langle\rangle\, c\, n\, \mathsf{in}\, (\mathsf{force}\, c)\, x\, y :\\
&\qquad\quad \forall\alpha/1.\, \alpha(0) \multimap [\_ < 1] \cdot \mathsf{List}_{a<I}\alpha(1+a) \multimap \mathsf{List}_{a<1+I}\alpha(a)\\
&\emptyset;\phi;\emptyset;\emptyset \vdash^{\mathsf{c}}_{1} \mathit{cons}' := \Lambda.\, \lambda x.\, \lambda xs.\, \mathsf{return}\, \mathsf{thunk}\, (\mathit{cons}\langle\rangle\, x\, xs) :\\
&\qquad\quad \forall\alpha/1.\, \alpha(0) \multimap [\_ < 1] \cdot \mathsf{List}_{a<I}\alpha(1+a) \multimap \mathsf{F}\, ([\_ < 1] \cdot \mathsf{List}_{a<1+I}\alpha(a))\\
&\emptyset;\phi;\emptyset;\emptyset \vdash^{\mathsf{c}}_{2I_1} \mathit{app} := \Lambda.\, \lambda xs\, ys.\, (\mathsf{force}\, xs)\langle\rangle\, ys\, (\mathsf{thunk}\, \mathit{cons}') :\\
&\qquad\quad \forall\alpha/1.\, [\_ < 1] \cdot \mathsf{List}_{a<I_1}\alpha(a) \multimap [\_ < 1] \cdot \mathsf{List}_{b<I_2}\alpha(b+I_1) \multimap \mathsf{F}\, [\_ < 1] \cdot \mathsf{List}_{b<I_1+I_2}\alpha(b)
\end{aligned}$$

The typing of *nil* is similar to the typing of the encoding of the constant 0:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha/1;\emptyset;\emptyset;n:\alpha(0),c:[a<0]\cdot\cdots \vdash^{\mathsf{v}}_{0} n:\alpha(0)}}{\alpha/1;\emptyset;\emptyset;n:\alpha(0),c:[a<0]\cdot\cdots \vdash^{\mathsf{c}}_{0} \mathsf{return}\, n : \mathsf{F}\,\alpha(0)}}{\alpha/1;\emptyset;\emptyset;\emptyset \vdash^{\mathsf{c}}_{0} \lambda n\, c.\, \mathsf{return}\, n : \mathsf{List}_{a<0}\alpha(a)}}{\emptyset;\emptyset;\emptyset;\emptyset \vdash^{\mathsf{c}}_{0} \Lambda.\, \lambda n\, c.\, \mathsf{return}\, n : \forall\alpha/1.\, \mathsf{List}_{a<0}\alpha(a)}$$

The typings of the other functions are shown in Figure 8.3. The typing of *cons* is similar to the typing of the successor function. The function *cons′* is an auxiliary function needed in *app* that thunks the resulting list. The function *app* iterates over the elements of the first list *xs* and adds them to a new, growing list. Visually, we replace n by *ys* and c by *cons′* in Figure 8.2. We have to instantiate the type variable of *cons′* to a type that represents exactly these intermediate lists. Note that the result of *app* is a thunked list.

8.3 Compositionality and polymorphism

As a last remark for this part of the thesis, note that we can combine both features of this chapter. This means that we can define a variant of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ that both supports compositional typings, polymorphism, and in which simple typings can be embedded (using $\bot$ as refinements).

First, the environment Σ has to track the arity of function variables and type variables. However, we have to be careful not to allow non-precise typings. In particular, there is no way to precisely type the function $\mathit{fst} : \forall\alpha_1\,\alpha_2.\, \alpha_1 \otimes \alpha_2 \multimap \alpha_1$, since α_2 could stand for a non-disposable type. Therefore, we have to thunk the arguments. The annotation algorithm works when every type in the context is either a disposable type (i.e. $\mathsf{Nat}[I]$ or 1) or a thunked type.

$$\dfrac{\emptyset;\alpha_1/1,\alpha_2/1;\emptyset;x : ([a<1]\cdot \mathsf{F}\,\alpha_1(a)) \otimes ([a<0]\cdot \mathsf{F}\,\alpha_2(a)) \vdash^{\mathsf{c}}_{0} \mathsf{let}\,(y;z) := x\, \mathsf{in}\, \mathsf{force}\, y : \mathsf{F}\,\alpha_1(0)}{\emptyset;\emptyset;\emptyset;\emptyset \vdash^{\mathsf{c}}_{0} \Lambda.\,\Lambda.\,\lambda x.\, \mathsf{let}\,(y;z) := x\, \mathsf{in}\, \mathsf{force}\, y : \forall\alpha_1/1\,\alpha_2/1.\, ([a<1]\cdot \mathsf{F}\,\alpha_1(a)) \otimes ([a<0]\cdot \mathsf{F}\,\alpha_2(a)) \multimap \mathsf{F}\,\alpha_1(0)}$$

Alternatively, we can assume as an invariant that all type variables stand for disposable types. Then, in the type instantiation rule, we have to check that the type is disposable.

$$
\dfrac{
\begin{array}{c}
\dfrac{\text{(by instantiating } xs \text{ with } \alpha(a) := \alpha'(a))}{\cdots \vdash_0 (\mathsf{force}\, xs)\langle\rangle : [a < I] \cdot (\alpha(I \dot{-} 1 \dot{-} a) \multimap \alpha'(a) \multimap \mathsf{F}\, \alpha'(1+a)) \multimap \alpha'(0) \multimap \mathsf{F}\, \alpha'(I)} \\
\vdots \\
\cdots, c : [a < I] \cdot (\alpha(I \dot{-} a) \multimap \alpha'(a) \multimap \mathsf{F}\, \alpha'(1+a)) \vdash_0^{\mathsf{c}} (\mathsf{force}\, xs)\langle\rangle\, c\, n : \mathsf{F}\, \alpha'(I)
\end{array}
\quad
\begin{array}{c}
\overline{\cdots \vdash_0^{\mathsf{c}} \mathsf{force}\, c : \alpha(0) \multimap \alpha'(I) \multimap \mathsf{F}\, \alpha'(1+I)} \\
\vdots \\
\cdots; c : [a<1] \cdot (\alpha(a + \cancel{I \dot{-} I}) \multimap \alpha'(a+I) \multimap \mathsf{F}\,\alpha'(1+a+I)), \\
x : \alpha(0), y : \alpha'(I) \vdash_0^{\mathsf{c}} (\mathsf{force}\, c)\, x\, y : \mathsf{F}\,\alpha'(1+I)
\end{array}
}{
\begin{array}{c}
\alpha/1, \alpha'/1; \phi; \emptyset; x : \alpha(0), xs : [\_ < 1] \cdot \mathsf{List}_{a<I}\alpha(1+a), c : [a < 1 + I_1] \cdot (\alpha(I + \cancel{1 \dot{-} 1} \dot{-} a) \multimap \alpha'(a) \multimap \mathsf{F}\,\alpha'(1+a)), n : \alpha'(0) \vdash_0^{\mathsf{c}} \\
\mathsf{bind}\, y \leftarrow (\mathsf{force}\, xs)\langle\rangle\, c\, n\, \mathsf{in}\, (\mathsf{force}\, c)\, x\, y : \mathsf{F}\,\alpha'(1+I)
\end{array}
}
$$

$$\vdots$$

$$\emptyset; \phi; \emptyset; \emptyset \vdash_0^{\mathsf{c}} cons := \Lambda.\, \lambda x\, xs.\, \Lambda.\, \lambda c\, n.\, \mathsf{bind}\, y \leftarrow (\mathsf{force}\, xs)\langle\rangle\, c\, n\, \mathsf{in}\, (\mathsf{force}\, c)\, x\, y : \forall\alpha/1.\, \alpha(0) \multimap [\_ < 1] \cdot \mathsf{List}_{a<I}\alpha(1+a) \multimap \mathsf{List}_{a<1+I}\alpha(a)$$

$$
\dfrac{
\dfrac{\text{by instantiating } xs \text{ with } \alpha(a) := [\_ < 1] \cdot \mathsf{List}_{b<a+I_2}\alpha(b + I_1 \dot{-} a)}{\begin{array}{c}\cdots \vdash_0^{\mathsf{c}} (\mathsf{force}\, xs)\langle\rangle : [a < I_1] \cdot \underline{B}_{cons'}(a) \multimap [\_ < 1] \cdot \mathsf{List}_{b<I_2}\alpha(b + I_1) \\ \multimap \mathsf{F}\, ([a < 1] \cdot \mathsf{List}_{b<I_1+I_2}\alpha(b + \cancel{I_1 \dot{-} I_1}))\end{array}}
\quad
\dfrac{\dfrac{\text{by instantiating } cons' \text{ with } \alpha(b) := \alpha(b + I_1 \dot{-} 1 \dot{-} a)}{\cdots; a < I_1; \cdots \vdash_1^{\mathsf{v}} cons' : \underline{B}_{cons'}(a)}}{\cdots \vdash^{\mathsf{v}}_{I_1 + \sum_{a<I_1} 1} \mathsf{thunk}\, cons' : [a < I_1] \cdot \underline{B}_{cons'}(a)}
}{
\alpha/1; \phi; \emptyset; xs : [\_ < 1] \cdot \mathsf{List}_{a<I_1}\alpha(a), ys : [\_ < 1] \cdot \mathsf{List}_{b<I_2}\alpha(b + I_1) \vdash_{2I_1} (\mathsf{force}\, xs)\langle\rangle\, ys\, (\mathsf{thunk}\, cons') : \mathsf{F}\, [\_ < 1] \cdot \mathsf{List}_{b<I_1+I_2}\alpha(b)
}
$$

$$\vdots$$

$$\emptyset; \phi; \emptyset; \emptyset \vdash_{2I_1}^{\mathsf{c}} app := \Lambda.\, \lambda xs\, ys.\, (\mathsf{force}\, xs)\langle\rangle\, ys\, (\mathsf{thunk}\, cons') : \forall\alpha/1.\, [\_ < 1] \cdot \mathsf{List}_{a<I_1}\alpha(a) \multimap [\_ < 1] \cdot \mathsf{List}_{b<I_2}\alpha(b + I_1) \multimap \mathsf{F}\, [\_ < 1] \cdot \mathsf{List}_{b<I_1+I_2}\alpha(b)$$

With: $\underline{B}_{cons'}(a) := \alpha(I_1 \dot{-} a \dot{-} 1) \multimap [\_ < 1] \cdot \mathsf{List}_{b<a+I_2}\alpha(b + I_1 \dot{-} a) \multimap \mathsf{F}\, [\_ < 1] \cdot \mathsf{List}_{b<1+a+I_2}\alpha(b + I_1 \dot{-} a \dot{-} 1)$

Figure 8.3: Example typings of polymorphic list operations *cons* and *app*

Part II

Effect Systems

Chapter 9

Introduction

In the first part of this thesis, we discussed dℓPCF – a family of sound and relatively complete coeffect-based type systems for complexity analysis.

In dℓPCF, the *weight* of a typing is a static upper bound on the number of *resource allocations*. A resource always has to be used whenever a term is to be (re)evaluated. In dℓPCF$_\mathsf{n}$, this happens at variable lookups (since variables denote suspended computations). We thus bound the number of variables uses; 'resources' are allocated at applications. In dℓPCF$_\mathsf{v}$, a suspended computation is forced upon a function application; we bound the number of applications and allocate resources at λ-abstractions and recursive functions. In dℓPCF$_\mathsf{pv}$, the same happens whenever a thunked computation is forced. Thus, we bound how often a thunked computation may be forced, and we allocate at thunk.

In any of these systems, the weight of a closed term is an upper bound for the cost of its execution, since only these resources may be used that the term allocated itself. The remaining resources are reserved for *potential* uses of the term. For example, in dℓPCF$_\mathsf{n}$, the weight of a typing of an abstraction $\lambda x.\, t$ is just the weight of t – the potential cost of executing the body is already included in the weight of $\lambda x.\, t$. For all dℓPCF systems, we can thus state the following (informal) equation for typings of closed terms:

$$\text{weight} = \text{actual execution cost} + \text{potential cost}$$

If we only know the weight of an arbitrary typing, we cannot determine the actual execution cost. For example, consider the following dℓPCF$_\mathsf{pv}$ typing judgements. Both functions have weight 1, but the second computation needs one forcing step to reduce to λ.

$$\begin{aligned} &\emptyset;\emptyset;\emptyset \vdash^{\mathsf{c}}_{1} \lambda y.\, (\mathsf{force}\,(\mathsf{thunk}\,s))\,\underline{0} && : \mathsf{Nat}[\bot] \multimap \mathsf{F}\,\mathsf{Nat}[1] \\ &\emptyset;\emptyset;\emptyset \vdash^{\mathsf{c}}_{1} \mathsf{bind}\,x \leftarrow (\mathsf{force}\,(\mathsf{thunk}\,s))\,\underline{0}\,\mathsf{in}\,\lambda y.\,\mathsf{return}\,x && : \mathsf{Nat}[\bot] \multimap \mathsf{F}\,\mathsf{Nat}[1] \end{aligned}$$

One further problem of dℓPCF is that types reveal information about the implementation of a function. In other words, *full abstraction* does not hold for dℓPCF: There are *observationally equivalent* computations with non-equivalent (precise) types, for example:

$$\begin{aligned} &\emptyset; k/1; \emptyset; \emptyset \vdash^{\mathsf{c}}_{0} t_1 := \lambda x.\, \mathsf{bind}\,y \leftarrow (\mathsf{force}\,x)\,\underline{0}\,\mathsf{in}\,(\mathsf{force}\,x)\,y : \\ &\qquad [a < 2] \cdot (\mathsf{Nat}[\mathsf{if}\,a = 0\,\mathsf{then}\,0\,\mathsf{else}\,k(0)] \multimap \mathsf{Nat}[k(a)]) \multimap \mathsf{Nat}[k(1)] \end{aligned}$$

$$\emptyset; k/1; \emptyset; \emptyset \vdash^{\mathsf{c}}_{0} t_2 := \lambda x.\, \mathsf{bind}\, x' \leftarrow \mathsf{return}\, x \,\mathsf{in}\, \mathsf{bind}\, y \leftarrow (\mathsf{force}\, x)\, \underline{0} \,\mathsf{in}\, (\mathsf{force}\, x')\, y :$$
$$[a < 2] \cdot (\mathsf{Nat}[\mathsf{if}\, a = 0 \,\mathsf{then}\, k(1) \,\mathsf{else}\, 0] \multimap \mathsf{Nat}[k(a)]) \multimap \mathsf{Nat}[k(0)]$$

Although the type inference algorithm from Chapter 8 will of course compute equivalent annotations for t_1 ($\mathsf{thunk}\, s$) and t_2 ($\mathsf{thunk}\, s$), the computed recursive equations are different. Consequently, replacing a program with an equivalent (or more efficient) program requires the user to simplify the equations again.

The effect-based approach presented in this part of the thesis will solve both problems, while attaining compositionality at the same time. In $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{n}}$ and $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$, the weight of a typing of $\lambda x.\, t$ is just the weight of the typing of t. However, the *cost* of $\lambda x.\, t$ is zero, since it is already a value. Thus, effect systems assign the empty effect (i.e. 0) to abstractions. Like simple types (and $[\_ < \bot]$ types in $\mathsf{d}\ell\mathsf{PCF}$), $\mathsf{d}f\mathsf{PCF}$ types are non-linear: We can discard or use a variable arbitrarily often.

Arrow types in $\mathsf{d}f\mathsf{PCF}$ have the shape $\forall \vec{h}.\, \sigma \xrightarrow{K} \tau$, where $\vec{h}$ is a vector of index variables and K is an index term (that may have the variables $\vec{h}$ free). The index variables $\vec{h}$ may be used to 'characterise' the argument. For example, the function $\lambda x.\, \mathsf{Succ}(x)$ may be assigned the type $\forall i.\, \mathsf{Nat}[i] \xrightarrow{0} \mathsf{Nat}[1 + i]$. Thus, if we can type an argument t with type $\mathsf{Nat}[I]$ (where I is some index term), then $(\lambda x.\, \mathsf{Succ}(x))\, t$ has type $\mathsf{Nat}[1 + I]$. We can also type higher-order functions:

$$\lambda x.\, x\, \underline{0} + x\, \underline{1} : \forall h_1\, h_2.\, (\forall i.\, \mathsf{Nat}[i] \xrightarrow{h_1(i)} \mathsf{Nat}[h_2(i)]) \xrightarrow{h_1(0)+h_1(1)} \mathsf{Nat}[h_2(0) + h_2(1)]$$

The higher-order index terms of $\mathsf{d}f\mathsf{PCF}$ ($\mathcal{L}^{f}_{idx}$) are typed, but, to avoid confusion, we use the word *sort*. In the above example, the variables h_1 and h_2 have the sort $\mathsf{Nat} \to \mathsf{Nat}$. In general, we can have higher-order index terms – in contrast to $\mathsf{d}\ell\mathsf{PCF}$, where index terms evaluate to natural numbers (or diverge).

The above scheme can be generalised. We will introduce the notion of *effect-parametricity* in the next chapter.

It is already evident that, if we aim for (relative) completeness, our new higher-order language of index terms has to be at least as expressive as the language that we want to type. Consider a function $\lambda x.\, t : \forall i.\, \mathsf{Nat}[i] \xrightarrow{\cdot} \mathsf{Nat}[\cdot]$. At the right dot, we need to write an index term that is equivalent to the λ-abstraction. This means that in order to annotate a Turing-complete language like PCF with annotations for complexity, we also need a Turing-complete higher-order language of index terms.

In the next chapter, we first consider the Turing-incomplete language System T. We will introduce $\mathsf{d}f\mathsf{T}$, and prove soundness and completeness of $\mathsf{d}f\mathsf{T}$. Although proving soundness is almost trivial, the completeness proof needs some effort. There, we will provide a procedure that takes as input a System T typing and computes an *effect-parametric annotation* of this typing in $\mathsf{d}\ell\mathsf{T}$. Since this generated typing is precise, the generated refinements will terminate if and only if the term terminates. We will demonstrate this procedure on several examples, including the Ackermann function. In Chapter 11, we consider the call-by-push-value variant of PCF.

Chapter 10

An effect system for System T: $\mathsf{d}f\mathsf{T}$

As in the first part of this thesis, our first effect-based type system, $\mathsf{d}f\mathsf{T}$, targets System T. We first define a new language of index terms, $\mathcal{L}^{f}_{idx}$, based on the call-by-name version of PCF. We will use the same language in the next chapter, where we generalise the results to CBPV. We will prove *soundness* and *compositional completeness.*

10.1 Index terms ($\mathcal{L}^{f}_{idx}$) and constraints

As already discussed in the introduction of this part, the language of index terms must be at least as computationally expressive as the target language. We define the index term language $\mathcal{L}^{f}_{idx}$ based on CBN with n-ary tuples and projections (which can of course also be defined as syntactic sugar using binary tuples). We already include a fixpoint operator ($\mu x.\, I$) here, which is not needed for $\mathsf{d}f\mathsf{T}$.

$$
\begin{array}{rll}
\text{Index terms:} & I, J, \ldots ::= n \mid a \mid \lambda x.\, I \mid \mu x.\, I \mid \mathsf{iter}\, I_1\, I_2 \mid I_1\, I_2 \mid \mathsf{ifz}\, I_1\, \mathsf{then}\, I_2\, \mathsf{else}\, I_3 & \\
& \quad \mid \mathsf{Succ}(I) \mid \mathsf{Pred}(I) \mid \langle I_1; \ldots; I_n \rangle \mid \pi_i(I) & \\
& \quad \mid I + J \mid I \dot{-} J \mid \sum_{a<I} J \mid I \cdot J & \\
\text{Constraints:} & C ::= I \sqsubseteq J \mid I \equiv J \mid I \leq J \mid I \gtrsim J \mid I \downarrow & \text{Sorts:} \quad S ::= \mathsf{Nat} \mid S \to S \\
\text{Constr. list:} & \Phi := \emptyset \mid C, \Phi & \text{Sort contexts:} \quad \phi := \emptyset \mid a : S, \phi
\end{array}
$$

Here, n again stands for natural numbers (which we do not underline in $\mathcal{L}^{f}_{idx}$), and a, b, c and $g, h, i, j, \ldots$ are index variables. The arithmetic operations can be regarded as syntactic sugar. Furthermore, we often use the following syntactic sugar for abstractions and fixpoints:

$$
\lambda\langle i_1, \ldots, i_n\rangle.\, t := \lambda i.\, t\{\pi_1(i)/i_1, \ldots, \pi_n(i)/i_n\} \qquad \mu\langle i_1, \ldots, i_n\rangle.\, t := \mu i.\, t\{\pi_1(i)/i_1, \ldots, \pi_n(i)/i_n\}
$$

We use standard call-by-name semantics for $\mathcal{L}^{f}_{idx}$ and a simple type system very similar to CBN, as discussed in Section 2.2.2. To avoid confusion, we use the word *sort* for $\mathcal{L}^{f}_{idx}$ types. The symbol ϕ is used for sorting contexts.

The syntax and semantics of *constraints* is similar to those of $\mathcal{L}^{\ell}_{idx}$. The only difference is that the constraint $<$ is not included (since it is not needed). Again, we use $\gtrsim$ in the case distinction rule (in particular for non-precise typings) only.

$$\frac{\exists n : \mathsf{Nat}.\, I \Downarrow n}{\vDash I \downarrow} \qquad \frac{\forall n : \mathsf{Nat}.\, J \Downarrow n \Rightarrow I \Downarrow n}{\vDash I \sqsubseteq J} \qquad \frac{\vDash I \sqsubseteq J \qquad \vDash J \sqsubseteq I}{\vDash I \equiv J}$$

$$\frac{\forall n : \mathsf{Nat}.\, J \Downarrow n \Rightarrow \exists m : \mathsf{Nat}.\, I \Downarrow m \wedge m \le n}{\vDash I \le J} \quad \frac{\forall n : \mathsf{Nat}.\, J \Downarrow n \Rightarrow \exists m : \mathsf{Nat}.\, I \Downarrow m \wedge m \ge n}{\vDash I \gtrsim J}$$

Note that constraints are not part of the syntax of $\mathcal{L}^{f}_{idx}$ terms (unlike in $\mathcal{L}^{\ell}_{idx}$). Moreover, if index terms contain fixpoints, constraints are undecidable in general. Assertions are defined exactly as in $\mathcal{L}^{\ell}_{idx}$, where $val(\phi)$ is a substitution that replaces index variables a with closed index terms of sort $\phi(a)$ (that do not have to terminate, in contrast to $\mathcal{L}^{\ell}_{idx}$):

$$\frac{}{\vDash \emptyset} \qquad \frac{\vDash C \qquad \vDash \Phi}{\vDash C, \Phi} \qquad \frac{\forall \nu \in val(\phi).\ \vDash \Phi\nu \Rightarrow \vDash C\nu}{\phi; \Phi \vDash C}$$

We sometimes use set-like notation for tuples of index terms and index variables. A tuple $\vec{K} = \langle K_1; \ldots; K_n \rangle$ is essentially a list of index terms. We write $\emptyset := \langle\rangle$. $K', \vec{K}$ adds the index term K' to the tuple/list $\vec{K}$. For example, if $\vec{K_1} = \langle 0; 1 \rangle$ and $\vec{K_2} = \langle 2 \rangle$, then $\vec{K_1}, \vec{K_2} = \langle 0; 1; 2 \rangle$. Similarly, we can add tuples/lists of index variables $\vec{h}$ to a list of (implicitly sorted) index variables, given that the types of the index variables $\vec{h}$ is clear from the context: $\vec{h}, \phi := h_1, \ldots, h_n, \phi$. Furthermore, abusing notation, we sometimes apply an index term to a list or tuple of index terms or variables. For example, $M(\phi)$ should be read as $M \langle h_1; \ldots; h_n \rangle$ where $\phi = h_1 : S_1, \ldots, h_n : S_n$. Also abusing notation, we sometimes write $f(i)$ for $f \langle i \rangle$ and $f()$ for $f \langle\rangle$.

10.2 Typing rules

The types of $\mathsf{d}f\mathsf{T}$ are defined inductively according to the following grammar:

$$\begin{aligned} \text{Types:} \quad & \sigma, \tau ::= \mathsf{Nat}[I] \,\big|\, \forall \vec{h}.\, \sigma \xrightarrow{I} \tau \\ \text{Contexts:} \quad & \Gamma ::= \emptyset \,\big|\, x : \sigma, \Gamma \end{aligned}$$

In arrows we may introduce a list $\vec{h}$ of (implicitly sorted) higher-order index variables.

The two subtyping rules and the typing rules of $\mathsf{d}f\mathsf{T}$ are depicted in Figure 10.1. Two of the most outstanding differences from $\mathsf{d}\ell\mathsf{T}$ are that we do not need sums of typing contexts (since $\mathsf{d}f\mathsf{T}$ is not a linear type system) and that λ-abstractions and iterations have zero cost (because abstractions are values). For readability of the rules, we use an explicit subsumption rule.

Iteration rule The iteration rule deserves an explanation. We assume that τ may have the variables $\vec{h}, \phi$ free. G is an index term that 'updates' the list of index terms $\vec{h}$. For example, if $t_1 = \lambda x.\, \mathsf{Succ}(t)$, then $\vec{h}$ consists only of a single index variable i, and G

$$\frac{\phi;\Phi \vDash I_1 \sqsubseteq I_2}{\phi;\Phi \vdash \mathsf{Nat}[I_1] \sqsubseteq \mathsf{Nat}[I_2]} \qquad \frac{\vec{h},\phi;\Phi \vDash I_1 \le I_2 \qquad \vec{h},\phi;\Phi \vdash \sigma_2 \sqsubseteq \sigma_1 \qquad \vec{h},\phi;\Phi \vdash \tau_1 \sqsubseteq \tau_2}{\phi;\Phi \vdash \forall \vec{h}.\,\sigma_1 \xrightarrow{I_1} \tau_1 \sqsubseteq \forall \vec{h}.\,\sigma_2 \xrightarrow{I_2} \tau_2}$$

$$\textsc{Sub}\quad \frac{\phi;\Phi;\Gamma \vdash_{K_1} t : \sigma \qquad \phi;\Phi \vdash \sigma \sqsubseteq \tau \qquad \phi;\Phi \vDash K_1 \le K_2}{\phi;\Phi;\Gamma \vdash_{K_2} t : \tau}$$

$$\textsc{Const}\quad \phi;\Phi;\emptyset \vdash_0 \underline{n} : \mathsf{Nat}[n] \qquad \textsc{Var}\quad \phi;\Phi;x:\sigma \vdash_0 x : \sigma \qquad \textsc{Lam}\quad \frac{\vec{h},\phi;\Phi;x:\sigma,\Gamma \vdash_K t : \tau}{\phi;\Phi;\Gamma \vdash_0 \lambda x.\,t : \forall \vec{h}.\,\sigma \xrightarrow{K} \tau}$$

$$\textsc{Succ}\quad \frac{\phi;\Phi;\Gamma \vdash_M t : \mathsf{Nat}[K]}{\phi;\Phi;\Gamma \vdash_M \mathsf{Succ}(t) : \mathsf{Nat}[1+K]} \qquad \textsc{Pred}\quad \frac{\phi;\Phi;\Gamma \vdash_M t : \mathsf{Nat}[K]}{\phi;\Phi;\Gamma \vdash_M \mathsf{Pred}(t) : \mathsf{Nat}[K \dot{-} 1]}$$

$$\textsc{App}\quad \frac{\begin{array}{c}\phi;\Phi;\Gamma \vdash_{K_1} t_1 : \forall \vec{h}.\,\sigma \xrightarrow{K_3} \tau \\ \phi;\Phi;\Gamma \vdash_{K_2} t_2 : \sigma\{\vec{I}/\vec{h}\}\end{array}}{\phi;\Phi;\Gamma \vdash_{1+K_1+K_2+K_3\{\vec{I}/\vec{h}\}} t_1\, t_2 : \tau\{\vec{I}/\vec{h}\}} \qquad \textsc{Ifz}\quad \frac{\begin{array}{c}\phi;\Phi;\Gamma \vdash_{K_1} t_1 : \mathsf{Nat}[J] \\ \phi;0 \gtrsim J,\Phi;\Gamma \vdash_{K_2} t_2 : \tau \\ \phi;1 \le J,\Phi;\Gamma \vdash_{K_2} t_3 : \tau\end{array}}{\Phi;\Gamma \vdash_{K_1+K_2} \mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3 : \tau}$$

$$\textsc{Iter}\quad \frac{\phi;\Phi;\Gamma \vdash_{M_1} t_1 : \forall \vec{h}.\,\tau \xrightarrow{K} \tau(\vec{h} := G(\vec{h})) \qquad \phi;\Phi;\Gamma \vdash_{M_2} t_2 : \tau(\vec{h} := F)}{\phi;\Phi;\Gamma \vdash_0 \mathsf{iter}\, t_1\, t_2 : \forall i : \mathsf{Nat}.\, \mathsf{Nat}[i] \xrightarrow{i\cdot(2+M_1)+M_2+\sum_{a<i} K(\vec{h}:=\mathsf{iter}\, G\, F\, a)} \tau(\vec{h} := \mathsf{iter}\, G\, F\, i)}$$

Figure 10.1: Subtyping and typing rules of $\mathsf{d}f\mathsf{T}$

increments this variable: $G = \lambda\langle i\rangle.\,\langle 1+i\rangle$. The index term F is the 'base'; for example, if $t_2 = 0$, then $F = \langle \underline{0}\rangle$.

The notation $\tau(\vec{h} := F)$, where F is a tuple or list of closed index terms, means that we instantiate the index variables $\vec{h}$ component-wise:

$$\tau(\vec{h} := F) := \tau\{\pi_1(F)/h_1, \ldots, \pi_n(F)/h_n\}$$

Now, the type of $\mathsf{iter}\, t_1\, t_2$ is $\forall i : \mathsf{Nat}.\, \mathsf{Nat}[i] \xrightarrow{\cdots} \tau(\vec{h} := \mathsf{iter}\, G\, F\, i)$. This means, we apply the 'step function' G i-times to the 'base' F, where i is the argument given to $\mathsf{iter}\, t_1\, t_2$. Note that $\mathsf{iter}\, t_1\, t_2$ is a value, so the cost is 0. The cost annotation over the arrow is:

$$i \cdot (2 + M_1) + M_2 + \sum_{a<i} K(\vec{h} := \mathsf{iter}\, G\, F\, a)$$

We have to pay two steps for each iteration: First for the 'unrollings' ($\mathsf{iter}\, t_1\, t_2\, \underline{1+n} \succ_1 t_1\, (\mathsf{iter}\, t_1\, t_2\, \underline{n})$), and the second for the applications. To account for the cost of executing t_1 i-times, we also add $i \cdot M_1$. Finally, we add $\sum_{a<i} K(\mathsf{iter}\, g\, f\, a)$ for the effects of all applications (after t_1 evaluates to a value in each iteration).

10.3 Soundness

As in dℓPCF, we prove soundness using *subject reduction.* We prove that the cost decreases after every β-substitution or iter unfolding step. Thus, the cost of a typing is an upper bound on the actual execution cost (provided that Φ is empty or tautological).

We can show that values always have cost 0. This is useful if we have a typing that uses the subsumption rule.

Lemma 10.1 (Retyping values)**.** *Let v be a value and $\phi; \Phi; \Gamma \vdash_M v : \tau$. Then we can type $\phi; \Phi; \Gamma \vdash_0 v : \tau$. Furthermore, if the typing is precise (i.e. subsumption is only used with $\equiv$ instead of $\sqsubseteq$ and $\leq$), then $\phi; \Phi \vDash M \equiv 0$.*

Proof (sketch). If there are no uses of the subsumption rule in the typing derivation before CONST, LAM, or ITER, then M is already 0. Otherwise, we just have to modify or remove these subsumption rules such that they do not increase the cost. ☐

Lemma 10.2 (Substitution)**.** *Let $\phi; \Phi; x : \sigma, \Gamma \vdash_M t : \tau$ and $\phi; \Phi; \emptyset \vdash_0 v : \sigma$. Then we can type $\phi; \Phi; \Gamma \vdash_M t\{v/x\} : \tau$.*

Proof. By induction on the typing of t. ☐

We also need a substitution lemma for index terms.

Lemma 10.3 (Index term substitution)**.** *Let $\vec{h}, \phi; \Phi; \Gamma \vdash_K^{\mathsf{c}} t : \tau$ and let ν be a valuation for the (implicitly sorted) index variables $\vec{h}$. Then $\phi; \Phi\nu; \Gamma\nu \vdash_{K\nu}^{\mathsf{c}} t : \tau\nu$.*

Proof. By induction on the typing. ☐

Proving subject reduction is routine now.

Theorem 10.4 (Subject reduction of dfT)**.** *Let $\phi; \Phi; \emptyset \vdash_M t : \rho$, and let $t \succ_i t'$ be a step. Then there exists an index term M' such that $\phi; \Phi; \emptyset \vdash_{M'} t' : \rho$ and $\phi; \Phi \vDash M' + i \leq M$.*

Proof (sketch). By induction on the step. We consider the head reduction rules; the context rules are trivial.

- Case $\mathsf{iter}\, t_1\, t_2\, \underline{1+n} \succ_1 t_1\, (\mathsf{iter}\, t_1\, t_2\, \underline{n})$. We first invert the typing of the application and the constant:

$$\begin{aligned}
&\phi; \Phi; \emptyset \vdash_{K_1} \mathsf{iter}\, t_1\, t_2 : \forall i.\, \mathsf{Nat}[i] \xrightarrow{K_3} \rho' \qquad (10.1)\\
&\phi; \Phi; \emptyset \vdash_{K_2} \underline{1+n} : \mathsf{Nat}[1+n]\\
&\quad \phi; \Phi \vDash 1 + K_1 + K_2 + K_3\{1+n/i\} \leq M\\
&\quad \phi; \Phi \vdash \rho'\{1+n/i\} \sqsubseteq \rho
\end{aligned}$$

Now we invert the typing of $\mathsf{iter}\, t_1\, t_2$:

$$\phi; \Phi; \emptyset \vdash_{M_1} t_1 : \forall \vec{h}.\, \tau \xrightarrow{K} \tau(\vec{h} := g(\vec{h})) \tag{10.2}$$

$$\phi; \Phi; \emptyset \vdash_{M_2} t_2 : \tau(\vec{h} := f) \tag{10.3}$$

$$\begin{aligned}\phi; \Phi \vdash \forall i : \mathsf{Nat}.\, \mathsf{Nat}[i] &\xrightarrow{i\cdot(2+M_1)+M_2+\sum_{a<i} K(\vec{h} := \mathsf{iter}\, g\, f\, a)} \tau(\vec{h} := \mathsf{iter}\, g\, f\, i) \sqsubseteq \\ \forall i : \mathsf{Nat}.\, \mathsf{Nat}[i] &\xrightarrow{K_3} \rho'\end{aligned}$$

Now it is easy to type the successor term: First, we type

$$\phi; \Phi; \emptyset \vdash_{1+K_1+K_2+K_3\{n/i\}} \mathsf{iter}\, t_1\, t_2\, \underline{n} : \tau(\vec{h} := \mathsf{iter}\, g\, f\, n)$$

using (10.1) and the rules App and Const. Then we use (10.2) and App to show the required typing:

$$\begin{aligned}\phi; \Phi; \emptyset \vdash_{1+K_1+K_2+K_3\{n/i\}+1+K(\vec{h} := \mathsf{iter}\, g\, f\, n)} t_1\, (\mathsf{iter}\, t_1\, t_2\, \underline{n}) :& \\ \tau(\vec{h} := \mathsf{iter}\, g\, f\, (n+1)) \sqsubseteq \rho'\{1 + n/i\} \sqsubseteq \rho&\end{aligned}$$

Finally, note that the cost of the above typing can be shown to be less than M.

- Case $\mathsf{iter}\, t_1\, t_2\, \underline{0} \succ_1 t_2$. Similar to the above, we invert the typing of the application. Equation (10.3) gives us the required typing of t_2.
- Case $(\lambda x.\, t)\, v \succ_1 t\{v/x\}$: By inversion, we have:

$$\phi; \Phi; \emptyset \vdash_{M_1} \lambda x.\, t_1 : \forall \vec{h}.\, \sigma \xrightarrow{K} \tau \qquad \phi; \Phi; \emptyset \vdash_{M_2} v : \sigma(\vec{h} := \vec{I})$$

$$\phi; \Phi \vDash 1 + M_1 + M_2 + K(\vec{h} := \vec{I}) \leq M \qquad \phi; \Phi \vdash \tau(\vec{h} := \vec{I}) \sqsubseteq \rho.$$

  By inverting the typing of $\lambda x.\, t$, we have $\vec{h}, \phi; \Phi; x : \sigma \vdash_K t : \tau$. With index term substitution (Lemma 10.3), we get $\phi; \Phi; x : \sigma(\vec{h} := \vec{I}) \vdash_{K(\vec{h} := \vec{I})} t : \tau(\vec{h} := \vec{I})$. With Lemmas 10.1 and 10.2, we finally can type $\phi; \Phi; \emptyset \vdash_{K(\vec{h} := \vec{I})} t\{v/x\} : \tau(\vec{h} := \vec{I}) \sqsubseteq \rho$.
- Cases $\mathsf{Succ}(\underline{n}) \succ_0 \underline{1+n}$ and $\mathsf{Pred}(\underline{n}) \succ_0 \underline{1+n}$: trivial.
- The cases $\mathsf{ifz}\, \underline{n}\, \mathsf{then}\, t_1\, \mathsf{else}\, t_2 \succ_0 t_{1,2}$ follow by inversion of the typing. □

Corollary 10.5 (Subject reduction, multiple steps)**.** *Let* $\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_M t : \underline{B}$ *and* $t \Downarrow_k t'$*. Then* $\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_{M-k} t' : \underline{B}$*.*

Corollary 10.6 (Soundness of $\mathsf{d}f\mathsf{T}$)**.** *Let* $\emptyset; \emptyset; \emptyset \vdash_k t : \tau$*. Then there exists a number* $k' \leq k$ *and a value* v*, such that* $t \Downarrow_{k'} v$ *and* $\emptyset; \emptyset; \emptyset \vdash_0 v : \tau$*.*

Proof (sketch). As in Corollary 7.20, we can prove the existence of k', v, and $\emptyset; \emptyset; \emptyset \vdash_{k-k'} v : \tau$. Then, using Lemma 10.1, we can change the cost of this typing to zero. □

10.4 Effect parametricity

As in Section 8.1, we will present an algorithm that takes as input a simple typing and annotates it. The main idea is similar to $\mathsf{d}\ell\mathsf{PCF}$. Instead of parametrising *typings* over the arguments of functions, however, we now use quantifiers on the type-level to parametrise over the refinements of arguments. In this section, we first define *effect-parametric* types. Informally, a type τ is (effect-) parametric if the index terms at negative positions of τ are fully parametrised using quantifiers. For example, the following types are parametric:

- $\forall i.\,\mathsf{Nat}[i] \xrightarrow{K_1(i)} \mathsf{Nat}[K_2(i)]$, where K_1 and K_2 are index terms of sort $\mathsf{Nat} \to \mathsf{Nat}$;
- $\forall h_1\, h_2 : \mathsf{Nat} \to \mathsf{Nat}.\,(\forall i.\,\mathsf{Nat}[i] \xrightarrow{h_1(i)} \mathsf{Nat}[h_2(i)]) \xrightarrow{K_1\langle h_1; h_2\rangle} \mathsf{Nat}[K_2(h_2)]$, where $K_1 : (\mathsf{Nat} \to \mathsf{Nat}) \times (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat}$ and $K_2 : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat}$ are higher-order index terms.

The following type is *not* parametric, because there is no uniform way of applying a term of this type:

$$\forall i : \mathsf{Nat}.\,\mathsf{Nat}[\mathsf{ifz}\, i\, \mathsf{then}\, 0\, \mathsf{else}\, 1] \overset{0}{\to} \mathsf{Nat}[i]$$

Note that this type is not inhabited since PCF is deterministic; it is not possible that a function maps an argument (e.g. $\underline{1}$) to more than one result. There are other non-inhabited types, like $\forall i.\,\mathsf{Nat}[i] \overset{0}{\to} \mathsf{Nat}[i+i]$ (if we do not extend PCF with primitive addition or multiplication), but this is not a concern for effect-parametricity.

Recall that in the types of $\mathsf{d}f\mathsf{T}$, there are two kinds of refinements: The index terms in $\mathsf{Nat}[\cdot]$ and $\overset{\cdot}{\to}$. In an effect-parametric type, there is always exactly one concrete Nat-refinement (which is located at the right-most Nat in the type), but there may be many concrete arrow-refinements.

At negative positions, the index terms are always quantified. For this, we split the sorting context ϕ into two contexts ϕ_1, ϕ_2: ϕ_1 only contains index variables that are used for refinements of arrows and ϕ_2 only for Nat-refinements. The (unique) Nat-refinement may depend on variables from ϕ_2 but not from ϕ_1 (since there is no way to observe costs within the language).

We define effect-parametricity using two predicates $pa^-(\tau; \vec{h}_1; h_2)$ and $pa^+(\tau; \phi_1; \phi_2; \vec{I}_1; I_2)$. Informally, the first predicate means that the type describes the behaviour of an argument, and the concrete behaviour is parametrised by the quantified index variables $\vec{h}_1$ and h_2. For example, the type of x in the context $x : \mathsf{Nat}[i]$ is parametrised by the index variable i. The second predicate means that there are 'concrete' annotations at positive positions in τ that depend on ϕ_1 and ϕ_2. The index terms in $\vec{I}_1$ appear above arrows at positive positions, and I_2 is the rightmost Nat annotation. For example, the type $\forall i.\,\mathsf{Nat}[i] \xrightarrow{I_1(i)} \mathsf{Nat}[I_2(i)]$ is parametric, where I_1 and I_2 are concrete index terms of sort $\mathsf{Nat} \to \mathsf{Nat}$.

Definition 10.7 (Effect-parametricity)**.** We define using mutual induction:

$$\frac{}{pa^{+}(\mathsf{Nat}[K(\phi_2)]\,;\phi_1;\phi_2;\emptyset;K)} \qquad \frac{pa^{-}(\sigma;\vec{h}_1;h_2) \qquad pa^{+}(\tau;\vec{h}_1,\phi_1;h_2,\phi_2;\vec{K}_1;K_2)}{pa^{+}(\forall \vec{h}_1\, h_2.\, \sigma \xrightarrow{I(\vec{h}_1,h_2,\phi_1,\phi_2)} \tau;\phi_1;\phi_2;I,\vec{K}_1;K_2)}$$

$$\frac{}{pa^{-}(\mathsf{Nat}[i]\,;\emptyset;i)} \qquad \frac{pa^{-}(\sigma;\vec{k}_1;k_2) \qquad pa^{+}(\tau;\vec{k}_1;\langle k_2\rangle\,;\vec{h}_1;h_2)}{pa^{-}(\forall \vec{k}_1\, k_2.\, \sigma \xrightarrow{k(\vec{k}_1\, k_2)} \tau;k,\vec{h}_1;h_2)}$$

In the arrow rules, $\vec{h}_1$ and h_2 must be fresh index variables.

We say that a type τ is *parametrised* (over index variables $\vec{h}_1, h_2$) if $pa^{-}(\tau;\vec{h}_1;h_2)$.

A type τ is *parametric* (over lists of index variables ϕ_1, ϕ_2) if there exists *concrete index terms* $\vec{K}_1$ and K_2 such that $pa^{+}(\tau;\phi_1;\phi_2;\vec{K}_1;K_2)$.

In the first rule of the definition of pa^{+}, we state that the $\mathsf{d}f\mathsf{T}$ type $\mathsf{Nat}[K(\phi_2)]$ is parametric in ϕ_1, ϕ_2. K is the only concrete Nat-refinement term, which has the Nat-refinement variables ϕ_2 (but not ϕ_1) as arguments.

At negative positions, $\mathsf{Nat}[i]$ is parametrised by an index variable i.

In the second rule of pa^{+}, we want to show that a type of shape $\forall \vec{h}_1\, h_2.\, \sigma \xrightarrow{\cdots} \tau$ is parametric in ϕ_1, ϕ_2. For this, we use pa^{-} to parametrise σ over the index variables $\vec{h}_1$ and h_2. After this, we use the definition of pa^{+} on τ, but we add the index variables $\vec{h}_1$ and h_2 to ϕ_1 and ϕ_2, respectively. I is a new 'concrete' index term, which appears over the arrow, and is applied to the quantifiers and ϕ_1, ϕ_2. In addition to I, all concrete index terms of τ are also concrete index terms of the arrow type.

In the arrow rule of pa^{-}, we want to parametrise over the effect annotations $\vec{h}_1, h_2$ of an arrow type $\forall \vec{k}_1\, k_2.\, \sigma \xrightarrow{k(\vec{k}_1,k_2)} \tau$. The type σ should of course be parametrised over the quantified variables $\vec{k}_1, k_2$, which is formalised using the first premise. Furthermore, the effect annotations of τ must be exactly the index variables $\vec{h}_1, h_2$. This is expressed using the second premise. Finally, the variable k is added to the arrow annotation variables $\vec{k}_1$.

Examples The formal definition of effect-parametricity is maybe best understood with a couple of examples. In Figure 10.2 derivations of pa^{+} for two complex types are shown. The shape of the first type is $\mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat}$, and the shape of the second type is $((\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat}) \to \mathsf{Nat}$.

Properties of effect-parametric types The definition of pa^{+} entails that the index terms in effect-parametric types τ are closed. The index terms occurring in τ over arrows are applications of 'concrete index terms' to the index variables ϕ_1, ϕ_2, and the additional index variables bound by quantifiers.

Also note that if a type is effect-parametric (in some set of index variables), the concrete index terms I_1 and $\vec{I}_2$ can be read off the type. Similarly, given a simple System T type A and index variables ϕ_1, ϕ_2, it is easy to compute a $\mathsf{d}f\mathsf{T}$ type τ with $(\!|\tau|\!) = A$ that

$$\dfrac{\dfrac{}{pa^-(\mathsf{Nat}[i]\,;\emptyset;i)} \qquad \dfrac{\dfrac{\dfrac{}{pa^-(\mathsf{Nat}[j]\,;\emptyset;j)} \qquad \dfrac{}{pa^+(\mathsf{Nat}[h_2(j)]\,;\emptyset;\langle j\rangle\,;\emptyset;h_2)}}{pa^-(\forall j.\,\mathsf{Nat}[j] \xrightarrow{h_1(j)} \mathsf{Nat}[h_2(j)]\,;\langle h_1\rangle\,;h_2)} \qquad \dfrac{}{pa^+(\mathsf{Nat}[K_2(h_2,i,\phi_2)]\,;h_1,\phi_1;h_2,i,\phi_2;K_2;\emptyset)}}{pa^+(\forall h_1\,h_2.\,(\forall j.\,\mathsf{Nat}[j] \xrightarrow{h_1(j)} \mathsf{Nat}[h_2(j)]) \xrightarrow{K_{12}(h_1,h_2,i,\phi_1,\phi_2)} \mathsf{Nat}[K_2(h_2,i,\phi_1)]\,;\phi_1;i,\phi_2;\langle K_{12}\rangle\,;K_2)}}{pa^+(\forall i.\,\mathsf{Nat}[i] \xrightarrow{K_{11}(i,\phi_1,\phi_2)} \forall h_1\,h_2.\,(\forall j.\,\mathsf{Nat}[j] \xrightarrow{h_1(j)} \mathsf{Nat}[h_2(j)]) \xrightarrow{K_{12}(h_1,h_2,i,\phi_1,\phi_2)} \mathsf{Nat}[K_2(h_2,i,\phi_2)]\,;\phi_1;\phi_2;\langle K_{11};K_{12}\rangle\,;K_2)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{pa^-(\mathsf{Nat}[i]\,;\emptyset;i)} \qquad \dfrac{}{pa^+(\mathsf{Nat}[h_2(i)]\,;\emptyset;\langle i\rangle\,;\emptyset;h_2)}}{pa^-(\forall i.\,\mathsf{Nat}[i] \xrightarrow{h_1(i)} \mathsf{Nat}[h_2(i)]\,;\langle h_1\rangle\,;h_2)} \qquad \dfrac{}{pa^+(\mathsf{Nat}[k_2(h_2)]\,;\langle h_1\rangle\,;\langle h_2\rangle\,;\emptyset;k_2)}}{pa^-(\forall h_1\,h_2.\,(\forall i.\,\mathsf{Nat}[i] \xrightarrow{h_1(i)} \mathsf{Nat}[h_2(i)]) \xrightarrow{k_1(h_1,h_2)} \mathsf{Nat}[k_2(h_2)]\,;\langle k_1\rangle\,;k_2)} \qquad \dfrac{}{pa^+(\mathsf{Nat}[K_2(k_2,\phi_2)]\,;k_1,\phi_1;k_2,\phi_2;\emptyset;K_2)}}{pa^+(\forall k_1\,k_2.\,(\forall h_1\,h_2.\,(\forall i.\,\mathsf{Nat}[i] \xrightarrow{h_1(i)} \mathsf{Nat}[h_2(i)]) \xrightarrow{k_1(h_1,h_2)} \mathsf{Nat}[k_2(h_2)]) \xrightarrow{K_1(k_1,k_2,\phi_1,\phi_2)} \mathsf{Nat}[K_2(k_2,\phi_2)]\,;\phi_1;\phi_2;\langle K_1\rangle\,;K_2)}$$

Figure 10.2: Examples of parametric types

is parametrised in some index variables $\vec{h}_1$ and h_2, i.e. $pa^-(\tau; \vec{h}_1; h_2)$. The number of quantified index variables is always determined by the structure of the simple type.

Note that $pa^-(\tau; \vec{h}_1; h_2)$ is essentially equivalent to $pa^+(\tau; \emptyset; \emptyset; \vec{h}_1; h_2)$ – the only difference is that we need an abstraction in $pa^+(\mathsf{Nat}[(\lambda\langle\rangle.\, i)\langle\rangle]\,; \emptyset; \emptyset; \emptyset; i)$. We use two symbols to emphasise the different roles of the annotations in negative and positive positions of types.

10.5 Parametric Completeness

Given a simple System T typing, we can annotate it and thus compute a $\mathsf{d}f\mathsf{T}$ typing with the same structure. Before we prove our main theorem, we first formally define (effect-) parametric annotations of a simple typings.

Definition 10.8 (Parametrised type annotation)**.** Let A be a simple type and τ be a $\mathsf{d}f\mathsf{T}$ type. We say that τ is a *parametrised annotation* of A over $\vec{h}_1, h_2$, if:

- $(\!|\tau|\!) = A$,
- $pa^-(\tau; \vec{h}_1; h_2)$.

The types in the typing contexts are parametrised over the index variables ϕ_1, ϕ_2.

Definition 10.9 (Parametrised context annotation)**.** Let $\phi = \phi_1, \phi_2$ be index variables. Let $\hat{\Gamma}$ be a simple context and Γ be a $\mathsf{d}f\mathsf{T}$ context. We say that Γ is an *(effect-)parametrised annotation* of a simple context $\hat{\Gamma}$ (in ϕ_1, ϕ_2) if:

- For each x in the domain of Γ, there is a distinct index variable h_2^x of ϕ_2;
- ϕ_1 can be partitioned into lists of index variables $\vec{h}_1^x$ for the variables x;
- for every x in Γ, $\Gamma(x)$ is a parametrised annotation of $\hat{\Gamma}(x)$ in $\vec{h}_1^x, h_2^x$, as in the above definition.

The $\mathsf{d}f\mathsf{T}$ type derived from a simple typing is *parametric* in the sort contexts ϕ_1, ϕ_2:

Definition 10.10 (Parametric annotations of types)**.** Let $\phi = \phi_1, \phi_2$ be index variables. Let A be a simple type and τ be a $\mathsf{d}f\mathsf{T}$ type. We say that τ is an *(effect-)parametric annotation* of A (in $\phi_1; \phi_2$), if:

- $(\!|\tau|\!) = A$,
- $pa^+(\tau; \phi_1; \phi_2; \vec{I}_1; I_2)$ for some index terms $\vec{I}_1$, I_2.

Now, we can state and prove our main theorem.

Theorem 10.11 (Annotating typings)**.** *Let $\hat{\Gamma} \vdash t : A$ be a simple System T typing. Let Γ be any effect-parametrised annotation of $\hat{\Gamma}$ (in $\phi = \phi_1, \phi_2$). Then we can compute an effect-parametric annotation ρ of A (in ϕ_1, ϕ_2), and a closed index term M, together with a typing $\phi; \emptyset; \Gamma \vdash_{M(\phi_1, \phi_2)} t : \rho$. Moreover, this typing is precise. (Such a typing is called an (effect-) parametric annotation of a simple typing.)*

Proof. By induction on the simple typing.

- Case $\underline{n}$; $A = \mathsf{Nat}$. Using Const, we can type

  $$\phi; \emptyset; \Gamma \vdash_{(\lambda\_.\, 0)\,(\phi_1, \phi_2)} \underline{n} : \mathsf{Nat}[(\lambda\_.\, n)\,(\phi_1, \phi_2)]$$

  Note that we need the seemingly useless abstractions in order to bring the typing into the required form.

- Case x; $A = \hat{\Gamma}(x)$. Note that by the assumption on Γ, we know that $\Gamma(x)$ is *parametrised* over index variables $\vec{h}_1(x)$ and $h_2(x)$ that are part of ϕ_1 and ϕ_2, respectively. We can convert $\Gamma(x)$ into a type that is *parametric* over ϕ_1, ϕ_2 but which is otherwise equivalent to $\Gamma(x)$. For this, we only have to introduce abstractions. Then, we can use Var to type x with this new type.

  For example, if $\Gamma = x : \mathsf{Nat}[i_1]\,, y : \mathsf{Nat}[i_2]$ and $\phi = i_1, i_2$, we type x with type $\mathsf{Nat}[(\lambda\langle i_1; i_2\rangle.\, i_1)\,\langle i_1; i_2\rangle]$.

- Case $\mathsf{Succ}(t)$. By the inductive hypothesis, we have:

  $$\phi; \emptyset; \Gamma \vdash_{K(\phi_1, \phi_2)} t : \mathsf{Nat}[I(\phi_1, \phi_2)]$$

  Using Succ, we can type:

  $$\phi; \emptyset; \Gamma \vdash_{K(\phi_1, \phi_2)} \mathsf{Succ}(t) : \mathsf{Nat}[(\lambda(\phi_1, \phi_2).\, 1 + I(\phi_1, \phi_2))\,(\phi_1, \phi_2)]$$

- Case $\mathsf{Pred}(t)$: as above.

- Case $\lambda x.\, t$. We have $A = A_1 \to A_2$ and $x : A_1, \hat{\Gamma} \vdash t : A_2$ for some simple types A_1 and A_2. Let $\vec{h}_1$ and h_2 be fresh index variables and let σ be a type with $(\!|\sigma|\!) = A_1$ such that $pa^{-}(\sigma; \vec{h}_1; h_2)$. Now, observe that $x : \sigma, \Gamma$ is a parametrised annotation of the PCF context $x : A_1, \hat{\Gamma}$ (over the index variables $\vec{h}_1, \phi_1$ and h_2, ϕ_2). Therefore, we can apply the inductive hypothesis on the PCF typing $x : A_1, \hat{\Gamma} \vdash t : A_2$, which yields a type τ that is an effect-parametric annotation of A_2, and a $\mathsf{d}f\mathsf{T}$ typing $\vec{h}_1, \phi_1, h_2, \phi_2; \emptyset; x : \sigma \vdash_{K(\vec{h}_1, \phi_1, h_2, \phi_2)} t : \tau$. Note that the type $\rho := \forall \vec{h}_1\, h_2.\, \sigma \xrightarrow{K(\vec{h}_1, \phi_1, h_2, \phi_2)} \tau$ is an effect-parametric annotation of the type $A = A_1 \to A_2$, as required. Using rule Lam, we can type $\phi; \emptyset; \Gamma \vdash_{(\lambda\_.\, 0)(\phi_1, \phi_2)} \lambda x.\, t : \rho$.

- Case $\mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3$; we have $\hat{\Gamma} \vdash t_1 : \mathsf{Nat}$ and $\hat{\Gamma} \vdash t_{2,3} : A$. We can apply the inductive hypothesis on the three typings:

  $$\begin{aligned} \phi; \emptyset; \Gamma &\vdash_{M_1(\phi_1, \phi_2)} t_1 : \mathsf{Nat}[J(\phi_1)] \\ \phi; J(\phi_1) = 0; \Gamma &\vdash_{M_2(\phi_1, \phi_2)} t_2 : \tau_2 \\ \phi; 1 \le J(\phi_1); \Gamma &\vdash_{M_3(\phi_1, \phi_2)} t_3 : \tau_3 \end{aligned}$$

  Note that the two $\mathsf{d}f\mathsf{T}$ types τ_2 and τ_3 have the same PCF shape (namely A), but they may have different annotations. We have to merge τ_2 and τ_3 to a new $\mathsf{d}f\mathsf{T}$

type ρ that is equivalent to either τ_2 or τ_3 under the constraints $J(\phi_1) = 0$ or $1 \leq J(\phi_1)$, respectively. This type $\rho := \mathsf{ifz}\, J(\phi_1)\, \mathsf{then}\, \tau_2\, \mathsf{else}\, \tau_3$ can be defined as in Definition 5.33. Now, we can apply the rule IFZ, and together with subsumption, we can derive the typing:

$$\phi; \emptyset; \Gamma \vdash_{(\lambda(\phi_1,\phi_2).\, M_1(\phi_1,\phi_2) + \mathsf{ifz}\, J(\phi_1)\, \mathsf{then}\, M_2(\phi_1,\phi_2)\, \mathsf{else}\, M_3(\phi_1,\phi_2))\, (\phi_1,\phi_2)} \mathsf{ifz}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, t_3 : \rho$$

- Case $t_1\, t_2$ with $\hat{\Gamma} \vdash t_1 : B \to A$ and $\hat{\Gamma} \vdash t_2 : B$. The inductive hypotheses yield two typings:

  $$\begin{aligned} &\phi; \emptyset; \Gamma \vdash_{K_1(\phi_1,\phi_2)} t_1 : \forall \vec{h}_1\, h_2.\, \sigma \xrightarrow{K_3(\vec{h}_1, h_2, \phi_1, \phi_2)} \tau \\ &\phi; \emptyset; \Gamma \vdash_{K_2(\phi_1,\phi_2)} t_2 : \sigma' \end{aligned}$$

  We know that σ' has the same shape as σ. We also know that $pa^-(\sigma; \vec{h}_1; h_2)$ and $pa^+(\sigma'; \phi_1; \phi_2; \vec{I}_1; I_2)$ for some concrete index terms $\vec{I}_1$ and I_2. We proceed by 'unifying' the negative type σ with the positive type σ'. For this, we define a new list of index terms I^* such that σ' is equivalent to $\sigma(\vec{h}_1, h_2 := I^*)$. I^* roughly is $\vec{I}_1, I_2$, but with the index variables ϕ_1, ϕ_2 free. Note that the index terms $\vec{I}_1 = I_{11}, \ldots, I_{1n}$ have different numbers of arguments, which are denoted with dots below:

  $$\begin{aligned} I^* &:= \big\langle \lambda(\cdots).\, I_{11}(\cdots, \vec{h}_1, h_2, \phi_1, \phi_2); \ldots; \lambda(\cdots).\, I_{1n}(\cdots, \vec{h}_1, h_2, \phi_1, \phi_2) \\ &\qquad \lambda(\cdots).\, I_2(\cdots, h_2, \phi_2);\big\rangle \\ K_3^* &:= \lambda(\vec{h}_1, h_2).\, K_3(\vec{h}_1, h_2, \phi_1, \phi_2) \end{aligned}$$

  Using APP, we can now type:

  $$\phi; \emptyset; \Gamma \vdash_{(\lambda(\phi_1,\phi_2).\, 1 + K_1(\phi_1,\phi_2) + K_2(\phi_1,\phi_2) + K_3^*(I^*))(\phi_1,\phi_2)} t_1\, t_2 : \tau(\vec{h}_1, h_2 := I^*)$$

  However, we are not done yet, since the type $\tau(\vec{h}_1, h_2 := I^*)$ is not parametric. The reason is that the index terms are not applications of ϕ_1, ϕ_2 plus the additional bound variables. This can be fixed by introducing these abstractions again.

  For example, let $\phi = j : \mathsf{Nat}$ and let the type of t_1 be $\forall i.\, \mathsf{Nat}[i] \xrightarrow{K_1\langle i; j\rangle} \mathsf{Nat}[K_2\, \langle i; j\rangle]$, and let $\sigma' := \mathsf{Nat}[(\lambda\langle j\rangle.\, j+1)\, \langle j\rangle]$ be the type of t_2. Then we can type $t_1\, t_2 : \mathsf{Nat}[(\lambda\langle j\rangle.\, K_2\, \langle j_1 + 1; j\rangle)\, \langle j\rangle]$ with cost $(\lambda\langle j\rangle.\, 1 + K_1\, \langle j\rangle + K_2\, \langle j\rangle + K_3\, \langle j_1 + 1; j\rangle)\, \langle j\rangle$.

- Case $\mathsf{iter}\, t_1\, t_2$. We have $\hat{\Gamma} \vdash t_1 : B \to B$, $\hat{\Gamma} \vdash t_2 : B$, and $A = \mathsf{Nat} \to B$. By the inductive hypotheses, we can annotate the typings of t_1 and t_2. The first inductive hypothesis yields the following effect-parametric typing:

  $$\phi; \emptyset; \Gamma \vdash_{M_1(\phi_1,\phi_2)} t_1 : \forall \vec{h}_1\, h_2.\, \sigma \xrightarrow{K(\vec{h}_1, h_2, \phi_1, \phi_2)} \tau$$

  We have $pa^-(\sigma; \vec{h}_1; h_2)$ and $pa^+(\tau; \vec{h}_1, \phi_1; h_2, \phi_2; \vec{G}_1; G_2)$ for free index variables $\vec{h}_1, h_2$ and closed index terms $\vec{G}_1, G_2$.

Before we can apply the rule Iter, we first have to do some 'binder bureaucracy' since the annotated type of t_1 (although it is effect-parametric) is not in the right shape to apply the iteration rule. We first have to bring the type of t_1 into the shape $\forall h^*.\, \tau \xrightarrow{K^*} \tau(h^* := G^*(h^*))$, with index terms as defined below.

First, we merge the index variables into a new index variable list: $h^* := \vec{h}_1, h_2$. Now, we construct the index term G^* that takes the tuple h^* as argument, and has the index variables ϕ free. Note that the index terms in $\vec{G}_1 = \langle G_{11}; \ldots; G_{1n}\rangle$ have different additional parameters apart from $\vec{h}_1$ and h_2. Also, G_2 has, in addition to h_2, ϕ_2, several (Nat-refinement) index variables as argument. We define the index term G^* that takes the tuple of index variables h^* as argument and returns a tuple of index terms: As in the application case, we write dots for the additional variables.

$$
\begin{aligned}
G^* &:= \lambda(\vec{h}_1, h_2).\\
&\qquad \big\langle \lambda(\cdots).\, G_{11}(\cdots, \vec{h}_1, h_2, \phi_1, \phi_2); \ldots; \lambda(\cdots).\, G_{1n}(\cdots, \vec{h}_1, h_2, \phi_1, \phi_2)\big\rangle\\
&\qquad\ \lambda(\cdots).\, G_2(\cdots, h_2, \phi_2);\\
K^* &:= \lambda(\vec{h}_1, h_2).\, K(\vec{h}_1, h_2, \phi_1, \phi_2)
\end{aligned}
$$

For example, if $\tau = \forall k.\, \mathsf{Nat}[k] \xrightarrow{G_{11}(k,h_{11},h_{12},h_2,\phi_1,\phi_2)} \forall j.\, \mathsf{Nat}[j] \xrightarrow{G_{12}(j,k,h_{11},h_{12},h_1,\phi_1,\phi_2)} \mathsf{Nat}[G_2(j, k, h_2, \phi_2)]$, then:

$$
\begin{aligned}
G^* = \lambda\langle h_{11}; h_{12}; h_2\rangle.&\\
&\big\langle \lambda\langle j; k\rangle.\, G_2(j, k, h_2, \phi_2);\\
&\ \ \lambda\langle k\rangle.\, G_{11}(k, h_{11}, h_{12}, h_1, \phi_1, \phi_2); \lambda\langle j; k\rangle.\, G_{12}(j, k, h_{11}, h_{12}, h_2, \phi_1, \phi_2)\big\rangle
\end{aligned}
$$

We can now write the typing of t_1 as follows:

$$\phi; \emptyset; \Gamma \vdash_{M_1} t_1 : \forall \vec{h}_1\, h_2.\, \sigma \xrightarrow{K(\vec{h}_1,h_2,\phi_1,\phi_2)} \sigma(\vec{h}_1, h_2 := G^*(\vec{h}_1, h_2))$$

Note that in the index terms of the right type, the index variables ϕ_1, ϕ_2 are free; we will bind these index variables again after applying the iteration rule.

The second inductive hypothesis yields an effect-parametric typing for t_2 with a type σ' that has the same shape as σ and τ. Similarly to the above, we can extract index terms $\vec{F}_1$ and F_2, and we define an index term F^* such that σ' can be obtained from σ by substituting F^* for $\vec{h}_1, h_2$.

Using rule Iter, we can now type:

$$\phi; \emptyset; \Gamma \vdash_{(\lambda\_.\, 0)\,(\phi_1,\phi_2)} \mathsf{iter}\, t_1\, t_2 : \forall i.\, \mathsf{Nat}[i] \xrightarrow{M^*(i,\phi_1,\phi_2)} \sigma(\vec{h}_1, h_2 := \mathsf{iter}\, G^*\, F^*\, i)$$

with $M^* := \lambda(i, \phi_1, \phi_2).\ i \cdot (2 + M_1(\phi_1, \phi_2)) + M_2(\phi_1, \phi_2) + \sum_{a<i} K^*(\mathsf{iter}\, G^*\, F^*\, a)$. We are done after we abstract over ϕ_1, ϕ_2 in all arrow refinement terms (at the positive positions), and ϕ_2 in the rightmost Nat-refinement term.

In the above example, the final type would be:

$$\begin{aligned}\forall i.\, \mathsf{Nat}[i] &\xrightarrow{M^*(i,\phi_1,\phi_2)} \\ &\sigma\big\{(\lambda(k,\phi_1,\phi_2).\, \pi_1(\mathsf{iter}\, I^*\, F^*\, i)\,(k,\phi_1,\phi_2))/h_{11}, \\ &\quad(\lambda(j,k,\phi_1,\phi_2).\, \pi_2(\mathsf{iter}\, I^*\, F^*\, i)\,(j,k,\phi_1,\phi_2))/h_{12}, \\ &\quad(\lambda(j,k,\phi_2).\, \pi_3(\mathsf{iter}\, I^*\, F^*\, i)\,(j,k,\phi_2))/h_2\big\}\end{aligned}$$

□

Remarks As in the $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ annotation algorithm in Section 8.1, the generated typing is unconstrained. We do not exploit the fact that System T terms terminate. In the next chapter, we extend the algorithm to CBPV, and since the generated typings are precise, the generated index terms terminate if and only if the input term terminates.

Consequently, in the ifz case, even if it is certain that only one branch is reachable, we also have to annotate the impossible branch. However, we introduce a case distinction in the index terms. For example, the annotated type for $\mathsf{ifz}\,\underline{1}\,\mathsf{then}\,\underline{2}\,\mathsf{else}\,\underline{3}$ is $\mathsf{Nat}[\mathsf{ifz}\,1\,\mathsf{then}\,2\,\mathsf{else}\,3]$, which is equivalent to $\mathsf{Nat}[3]$. Similarly, to annotate an iteration, we use index term iteration for the refinement.

10.6 Annotation Examples

In this section, we apply the ‘algorithm’ in the proof of Theorem 10.11 to some arithmetic functions. We will be less strict regarding the invariant of the algorithm that all index terms have to be abstractions.

Addition

Recall the definition of addition and multiplication in System T, which we showed in Section 2.1.4:

$$\begin{aligned} s &:= \lambda x.\, \mathsf{Succ}(x) \\ \mathit{add} &:= \lambda x.\, \mathsf{iter}\, s\, x \end{aligned}$$

The annotated typing of the successor function s is easy:

$$\dfrac{\dfrac{\dfrac{}{i : \mathsf{Nat}; \emptyset; x : \mathsf{Nat}[i] \vdash_0 x : \mathsf{Nat}[i]}}{i : \mathsf{Nat}; \emptyset; x : \mathsf{Nat}[i] \vdash_0 \mathsf{Succ}(x) : \mathsf{Nat}[1+i]}}{\emptyset; \emptyset; \emptyset \vdash_0 \lambda x.\, \mathsf{Succ}(x) : \forall i : \mathsf{Nat}.\, \mathsf{Nat}[i] \xrightarrow{0} \mathsf{Nat}[1+i]}$$

Note that the definition of *add* begins with a λ-abstraction, where the parameter has type Nat. We first annotate the body of this λ-abstraction, where we set $\phi := i : \mathsf{Nat}$ and $\Gamma := x : \mathsf{Nat}[i]$. We introduce a fresh index variable k, and we define $\tau := \mathsf{Nat}[k]$ and the following index terms:

$$\begin{aligned}
G &:= \lambda\langle k;i\rangle.\, 1+k \\
G^{*} &:= \lambda k.\, G(k,i) = 1+k \\
F &:= i \\
K &:= \lambda\langle k;i\rangle.\, 0 \\
K^{*} &:= \lambda k.\, K\,\langle k;i\rangle \\
M_1 &:= M_2 := \lambda i.\, 0 \\
M &:= \lambda\langle j;i\rangle.\ j\cdot(2+M_1(i)) + M_2(i) + \sum_{a<j} K^{*}(\mathsf{iter}\, G\, F a) = 2\cdot j
\end{aligned}$$

The above typing of s can be rewritten as an effect-parametric typing with i as an additional index variable (although it is not needed):

$$i : \mathsf{Nat}; \emptyset; x : \mathsf{Nat}[i] \vdash_{M_1(i)} s : \forall k : \mathsf{Nat}.\, \tau \xrightarrow{K\langle k;i\rangle} \tau(k := g(k,i))$$

This typing needs to be slightly changed again, because the rule Iter requires the type on the right side of the arrow to be $\tau(k := g^{*}(k))$. Then, we can type:

$$\dfrac{\dfrac{\begin{array}{c} i : \mathsf{Nat}; \emptyset; x : \mathsf{Nat}[i] \vdash_{M_1(i)} s : \forall k : \mathsf{Nat}.\, \tau \xrightarrow{K\langle k;i\rangle} \tau(k := g^{*}(k)) \\ i : \mathsf{Nat}; \emptyset; x : \mathsf{Nat}[i] \vdash_{M_2(i)} x : \tau(k := f) = \mathsf{Nat}[i] \end{array}}{i : \mathsf{Nat}; \emptyset; x : \mathsf{Nat}[i] \vdash_{(\lambda i.\, 0)\, i} \mathsf{iter}\, s\, x : \forall j.\, \tau \xrightarrow{M\langle j;i\rangle} \tau(k := \mathsf{iter}\, F\, G\, j)}}{\emptyset; \emptyset; \emptyset \vdash_0 \mathit{add} : \forall i.\, \mathsf{Nat}[i] \xrightarrow{0} \forall j.\, \mathsf{Nat}[j] \xrightarrow{2\cdot j} \mathsf{Nat}[i+j]}$$

We can apply the functions to two constants m and n:

$$\emptyset; \emptyset; \emptyset \vdash_{2+2\cdot n} \mathit{add}\, \underline{m}\, \underline{n} : \mathsf{Nat}[m+n]$$

Multiplication

We can reuse the effect-parametric typing of *add* to type multiplication. Recall the definition:

$$\mathit{mult} := \lambda x.\, \mathsf{iter}\, (\mathit{add}\, x)\, \underline{0}$$

Again, we first have to type the body of the λ-abstraction, which is an iteration. This time, we use the index variable $i : \mathsf{Nat}$ and the variable $x : \mathsf{Nat}[i]$ for every iteration.

We first define $\tau := \mathsf{Nat}[k]$ (where k is a fresh index variable). Using the above typing, we can type $\mathit{add}\, x$:

$$\dfrac{\emptyset; \emptyset; \emptyset \vdash_0 \mathit{add} : \forall i.\, \mathsf{Nat}[i] \xrightarrow{0} \forall k.\, \mathsf{Nat}[k] \xrightarrow{2\cdot k} \mathsf{Nat}[i+k] \qquad \dfrac{}{i; \emptyset; x : \mathsf{Nat}[i] \vdash_0 x : \mathsf{Nat}[i]}}{i; \emptyset; x : \mathsf{Nat}[i] \vdash_{M_1(i)} \mathit{add}\, x : \forall k.\, \tau \xrightarrow{K\langle k;i\rangle} \tau(k := g(k,i))}$$

with the following index terms:

$$\begin{aligned}
G &:= \lambda\langle k; i\rangle.\, i + k \\
G^* &:= \lambda k.\, G\,\langle k; i\rangle = i + k \\
F &:= 0 \\
K &:= \lambda\langle k; i\rangle.\, 2k \\
K^* &:= \lambda k.\, K\,\langle k; i\rangle \\
M_1 &:= \lambda i.\, 1 \\
M_2 &:= \lambda i.\, 0
\end{aligned}$$

Now, we can derive:

$$\dfrac{\dfrac{\begin{array}{l} i; \emptyset; x : \mathsf{Nat}[i] \vdash_{M_1(i)} \mathit{add}\, x : \forall k.\, \tau \xrightarrow{K^*(k)} \tau(k := g^*(k)) \\ i; \emptyset; x : \mathsf{Nat}[i] \vdash_{M_2(i)} \underline{0} : \tau(k := f) \end{array}}{i; \emptyset; x : \mathsf{Nat}[i] \vdash_{(\lambda i.\, 0)\, i} \mathsf{iter}\, \mathit{add}\, \underline{0} : \forall j.\, \mathsf{Nat}[j] \xrightarrow{M\langle j; i\rangle} \tau(k := \mathsf{iter}\, g\, f\, j) = \mathsf{Nat}[i \cdot j]}}{\emptyset; \emptyset; \emptyset \vdash_0 \mathit{mult} : \forall i.\, \mathsf{Nat}[i] \xrightarrow{0} \forall j.\, \mathsf{Nat}[j] \xrightarrow{M\langle j; i\rangle} \mathsf{Nat}[i \cdot j]}$$

Where the index term M is defined as:

$$M := \lambda\langle j; i\rangle.\ j \cdot (2 + M_1(i)) + M_2(i) + \sum_{a<j} K^*(\mathsf{iter}\, g\, f a) = 3j + \sum_{a<j} (2ai) = ij^2 - ij + 3j$$

This means that the cost of $\mathit{mult}\, \underline{m}\, \underline{n}$ is $2 + mn^2 - mn + 3n$, which is the same cost that we derived in Section 4.6.

Ackermann function

The above examples were relatively simple because these functions are all primitive recursive. Now, we will give an annotated typing for the Ackermann function. Recall the definition:

$$\begin{aligned}
\mathit{ack} &:= \mathsf{iter}\, u\, s \\
u &:= \lambda x.\, \mathsf{iter}\, x\, (x\, \underline{1})
\end{aligned}$$

One obvious complication is that u itself is a λ-abstraction with an iteration as its body. We begin to type this inner iteration $\mathsf{iter}\, x\, (x\, \underline{1})$.[1] Here, the context is $\Gamma := x : \forall j.\, \mathsf{Nat}[j] \xrightarrow{h_1(j)} \mathsf{Nat}[h_2(j)]$, where h_1 and h_2 are index variables of sort $\mathsf{Nat} \to \mathsf{Nat}$.

$$\dfrac{\dfrac{\dfrac{}{h_1, h_2; \emptyset; \Gamma \vdash_0 x : \forall j.\, \mathsf{Nat}[j] \xrightarrow{h_1(j)} \mathsf{Nat}[h_2(j)]} \qquad \dfrac{}{h_1, h_2; \emptyset; \Gamma \vdash_{1+h_1(1)} x\, \underline{1} : \mathsf{Nat}[h_2(1)]}}{h_1, h_2; \emptyset; \Gamma \vdash_0 \mathsf{iter}\, x\, (x\, 1) : \forall j.\, \mathsf{Nat}[j] \xrightarrow{G_1(j, h_1, h_2)} \mathsf{Nat}[G_2(j, h_2)]}}{\emptyset; \emptyset; \emptyset \vdash_0 u : \forall h_1\, h_2.\, (\forall j.\, \mathsf{Nat}[j] \xrightarrow{h_1(j)} \mathsf{Nat}[h_2(j)]) \xrightarrow{0} (\forall j.\, \mathsf{Nat}[j] \xrightarrow{G_1(j, h_1, h_2)} \mathsf{Nat}[G_2(j, h_2)])}$$

---

[1] We have also typed this function in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$; see Example 7 in Section 8.1.1.

with the following higher-order index terms:

$$\begin{aligned}
G_1 &:= \lambda\langle j; h_1; h_2\rangle.\ j \cdot (2+0) + (1 + h_1(1)) + \sum_{b<j} h_1(\mathsf{iter}\, h_1\, 1\, (1+j)) \\
G_2 &:= \lambda\langle j; h_2\rangle.\ \mathsf{iter}\, h_2\, (h_2(1))\, j = \mathsf{iter}\, h_2\, 1\, (1+j)
\end{aligned}$$

Now, we define $\tau := \forall j.\, \mathsf{Nat}[j] \xrightarrow{h_1(j)} \mathsf{Nat}[h_2(j)]$, and type:

$$\frac{\emptyset;\emptyset;\emptyset \vdash_0 u : \forall h_1\, h_2.\, \tau \xrightarrow{0} \tau(h_1, h_2 := G^*\, \langle h_1; h_2\rangle) \qquad \emptyset;\emptyset;\emptyset \vdash_0 s : \tau(h^* := F)}{\emptyset;\emptyset;\emptyset \vdash_0 ack : \forall i.\, \mathsf{Nat}[i] \xrightarrow{K(i)} \tau(h_1, h_2 := \mathsf{iter}\, G^*\, F\, i)}$$

with the following index terms:

$$\begin{aligned}
G^* &:= \lambda\langle h_1; h_2\rangle.\ \langle G_1\, \langle j; h_1; h_2\rangle\, ; G_2\, \langle j; h_2\rangle\rangle \\
F &:= \langle \lambda k.\, 0; \lambda k.\, 1 + k\rangle \\
K &:= \lambda i.\, i \cdot (2+0) + 0 + \sum_{a<i} 0 = 2i
\end{aligned}$$

If we expand the final type, we get:

$$\emptyset;\emptyset;\emptyset \vdash_0 ack : \forall i.\, \mathsf{Nat}[i] \xrightarrow{2i} \forall j.\, \mathsf{Nat}[j] \xrightarrow{\pi_1(\mathsf{iter}\, G^*\, F\, i)} \mathsf{Nat}[(\pi_2(\mathsf{iter}\, G^*\, F\, i))(j)]$$

It can be shown that $\pi_2(\mathsf{iter}\, G^*\, F\, i\, j)$ is equal to the $ack\, i\, j$. We have also implemented these index terms in Haskell and compared the results with Table 2.1.

Chapter 11

An effect system for call-by-push-value PCF

In this chapter, we introduce $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$, an effect system for the call-by-push-value variant of PCF. Since the target language is Turing complete, the annotation algorithm will produce diverging annotations (only) for diverging timers. We use the same language of index terms, $\mathcal{L}^{f}_{idx}$, as in Section 10.1, where we have already included (but not yet used) the fixpoint operator $\mu x.\, I$.

11.1 Typing rules

The types of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ are defined using the following grammar. Since we target the call-by-push-value variant of PCF, there are *value types* A and computation types $\underline{B}$:

$$\begin{array}{rl}
\text{Value types:} & A ::= \mathsf{U}_I\,\underline{B} \mid \mathsf{Nat}[I] \\
\text{Computation types:} & \underline{B} ::= \mathsf{F}\,A \mid \forall \vec{h}.\, A \xrightarrow{I} \underline{B} \\
\text{Contexts:} & \Gamma, \Delta ::= \emptyset \mid x : A, \Gamma
\end{array}$$

The type constructor U from the simple type system CBPV is annotated with an index term that stands for the cost of forcing the thunk. Thunked computations can be forced arbitrarily often and, since CBPV is deterministic, it will always have the same cost. We also annotate arrows with a static upper bound on the cost of an application.

The typing rules are depicted in Figure 11.1. The subtyping rules (for value and computation types) can be obtained straightforwardly by extension from the subtyping rules of $\mathsf{d}\ell\mathsf{T}$. Note that value typings are not assigned a cost.

The premise of the rule Fix is $\phi; \Phi; x : \mathsf{U}_K\,\underline{B}, \Gamma \vdash^{\mathsf{c}}_{K} t : \underline{B}$, where K is also the cost of the fixpoint. These costs are always the same, since the cost of $\mu x.\, t$ is just the cost of $t\{\mathsf{thunk}\,\mu x.\, t/x\}$, which only depends on the context Γ. For example, we have $K = 0$ if t is a λ-abstraction. All other rules are not surprising, as they are easy refinements of the simple CBPV typing rules in Figure 2.5.

$$\frac{\phi;\Phi \vDash I \sqsubseteq J}{\phi;\Phi \vdash \mathsf{Nat}[I] \sqsubseteq \mathsf{Nat}[J]} \qquad \frac{\phi;\Phi \vdash A_1 \sqsubseteq A_2}{\phi;\Phi \vdash \mathsf{F}\,A_1 \sqsubseteq \mathsf{F}\,A_2} \qquad \frac{\phi;\Phi \vDash K_1 \leq K_2 \qquad \phi;\Phi \vdash \underline{B}_1 \sqsubseteq \underline{B}_2}{\phi;\Phi \vdash \mathsf{U}_{K_1}\,\underline{B}_1 \sqsubseteq \mathsf{U}_{K_2}\,\underline{B}_2}$$

$$\frac{\begin{array}{c}\vec{h},\phi;\Phi \vDash I_1 \leq I_2 \\ \vec{h},\phi;\Phi \vdash A_2 \sqsubseteq A_1 \qquad \vec{h},\phi;\Phi \vdash \underline{B}_1 \sqsubseteq \underline{B}_2\end{array}}{\phi;\Phi \vdash \forall\vec{h}.\,A_1 \xrightarrow{I_1} \underline{B}_1 \sqsubseteq \forall\vec{h}.\,A_2 \xrightarrow{I_2} \underline{B}_2} \qquad \frac{\begin{array}{c}\phi;\Phi \vdash A_1 \sqsubseteq A_2 \\ \phi;\Phi \vdash A_2 \sqsubseteq A_1\end{array}}{\phi;\Phi \vdash A_1 \equiv A_2} \qquad \frac{\begin{array}{c}\phi;\Phi \vdash \underline{B}_1 \sqsubseteq \underline{B}_2 \\ \phi;\Phi \vdash \underline{B}_2 \sqsubseteq \underline{B}_1\end{array}}{\phi;\Phi \vdash \underline{B}_1 \equiv \underline{B}_2}$$

SubV

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma' \vdash^{\mathsf{v}} v : A_1 \qquad \phi;\Phi \vdash A_1 \sqsubseteq A_2 \\ \phi;\Phi \vdash \Gamma \sqsubseteq \Gamma'\end{array}}{\phi;\Phi;\Gamma \vdash^{\mathsf{v}} v : A_2}$$

SubC

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma' \vdash^{\mathsf{c}}_{K_1} t : \underline{B}_1 \qquad \phi;\Phi \vdash \underline{B}_1 \sqsubseteq \underline{B}_2 \\ \phi;\Phi \vdash \Gamma \sqsubseteq \Gamma' \qquad \phi;\Phi \vDash K_1 \leq K_2\end{array}}{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{K_2} t : \underline{B}_2}$$

Const

$$\phi;\Phi;\emptyset \vdash^{\mathsf{v}} \underline{n} : \mathsf{Nat}[n]$$

Var

$$\phi;\Phi;x : A \vdash^{\mathsf{v}} x : A$$

Lam

$$\frac{\vec{h},\phi;\Phi;x : A,\Gamma \vdash^{\mathsf{c}}_{K} t : \underline{B}}{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{0} \lambda x.\,t : \forall\vec{h}.\,A \xrightarrow{K} \underline{B}}$$

Fix

$$\frac{\phi;\Phi;x : \mathsf{U}_K\,\underline{B},\Gamma \vdash^{\mathsf{c}}_{K} t : \underline{B}}{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{K} \mu x.\,t : \underline{B}}$$

App

$$\frac{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{K_1} t : \forall\vec{h}.\,A \xrightarrow{K_2} \underline{B} \qquad \phi;\Phi;\Gamma \vdash^{\mathsf{v}} v : A\{\vec{h} := \vec{I}\}}{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{K_1+K_2\{\vec{h}:=\vec{I}\}} t\,v : \underline{B}\{\vec{h} := \vec{I}\}}$$

Ifz

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma \vdash^{\mathsf{v}} v : \mathsf{Nat}[J] \\ \phi;0 \gtrsim J,\Phi;\Gamma \vdash^{\mathsf{c}}_{K} t_1 : \tau \\ \phi;1 \leq J,\Phi;\Gamma \vdash^{\mathsf{c}}_{K} t_2 : \tau\end{array}}{\phi;\Phi;\Gamma \vdash_{K} \mathsf{ifz}\,v\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2 : \tau}$$

Succ

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma \vdash^{\mathsf{v}} v : \mathsf{Nat}[J] \\ \phi;\Phi;x : \mathsf{Nat}[1+J],\Gamma \vdash^{\mathsf{c}}_{K} t : \underline{B}\end{array}}{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{K} \mathsf{calc}\,x \leftarrow \mathsf{Succ}(v)\,\mathsf{in}\,t : \underline{B}}$$

Pred

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma \vdash^{\mathsf{v}} v : \mathsf{Nat}[J] \\ \phi;\Phi;x : \mathsf{Nat}[J \mathbin{\dot{-}} 1],\Gamma \vdash^{\mathsf{c}}_{K} t : \underline{B}\end{array}}{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{K} \mathsf{calc}\,x \leftarrow \mathsf{Pred}(v)\,\mathsf{in}\,t : \underline{B}}$$

Return

$$\frac{\phi;\Phi;\Gamma \vdash^{\mathsf{v}} v : A}{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{0} \mathsf{return}\,v : \mathsf{F}\,A}$$

Bind

$$\frac{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{K_1} t_1 : \mathsf{F}\,A \qquad \phi;\Phi;x : A,\Gamma \vdash^{\mathsf{c}}_{K_2} t_2 : \underline{B}}{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{K_1+K_2} \mathsf{bind}\,x \leftarrow t_1\,\mathsf{in}\,t_2 : \underline{B}}$$

Thunk

$$\frac{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{K} t : \underline{B}}{\phi;\Phi;\Gamma \vdash^{\mathsf{v}} \mathsf{thunk}\,t : \mathsf{U}_K\,\underline{B}}$$

Force

$$\frac{\phi;\Phi;\Gamma \vdash^{\mathsf{v}} v : \mathsf{U}_K\,\underline{B}}{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{1+K} \mathsf{force}\,v : \underline{B}}$$

Figure 11.1: Typing rules of $\mathsf{d}f\mathsf{PCF}_{\mathsf{pv}}$

11.2 Soundness

The soundness proof of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ by now is pure routine; we proceed in the same way as in $\mathsf{d}\ell\mathsf{T}$.

Lemma 11.1 (Substitution)**.** *Let* $\phi; \Phi; x : A, \Gamma \vdash^{\mathsf{c}}_{M} t : \underline{B}$ *and* $\phi; \Phi; \emptyset \vdash^{\mathsf{v}} v : A$*. Then we can type* $\phi; \Phi; \Gamma \vdash_{M} t\{v/x\} : \underline{B}$*.*

Proof. We prove the lemma (and the analogous statement for value typings) by mutual induction on the typings. ☐

Lemma 11.2 (Index term substitution)**.** *Let* $\vec{h}, \phi; \Phi; \Gamma \vdash^{\mathsf{c}}_{K} t : \underline{B}$ *and let* ν *be a valuation for the (implicitly typed) index variables* $\vec{h}$*. Then* $\phi; \Phi\nu; \Gamma\nu \vdash^{\mathsf{c}}_{K\nu} t : \underline{B}\nu$*.*

Proof. By induction on the typing. ☐

Theorem 11.3 (Subject reduction of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$)**.** *Let* $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{M} t : \underline{B}$*, and let* $t \succ_i t'$ *be a step. Then there exists an index term* M' *such that* $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{M'} t' : \underline{B}$ *and* $\phi; \Phi \vDash M' + i \le M$*.*

Proof. By induction on the step. We consider the head reduction rules; reductions under contexts are trivial.

- Case $(\lambda x.\, t)\, v \succ_0 t\{v/x\}$: By inversion, we have:

  $$\begin{array}{ll} \phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{K_1} \lambda x.\, t_1 : \forall \vec{h}.\, A \xrightarrow{K_2} \underline{B}' \qquad & \phi; \Phi; \emptyset \vdash^{\mathsf{v}} v : A(\vec{h} := \vec{I}) \\[1em] \phi; \Phi \vDash 1 + K_1 + K_2(\vec{h} := \vec{I}) \le M \qquad & \phi; \Phi \vdash \underline{B}'(\vec{h} := \vec{I}) \sqsubseteq \underline{B}. \end{array}$$

  By inverting the typing of $\lambda x.\, t$, we have $\vec{h}, \phi; \Phi; x : A \vdash^{\mathsf{c}}_{K_2} t : \underline{B}'$. With index term substitution (similar to Lemma 10.3), we get $\phi; \Phi; x : A(\vec{h} := \vec{I}) \vdash^{\mathsf{c}}_{K_2(\vec{h} := \vec{I})} t : \underline{B}'(\vec{h} := \vec{I})$. With the substitution lemma (Lemma 11.1), we can type $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{K_2(\vec{h} := \vec{I})} t\{v/x\} : \underline{B}'(\vec{h} := \vec{I}) \sqsubseteq \underline{B}$.

- Case $\mu x.\, t \succ_0 t\{\mathsf{thunk}\, \mu x.\, t/x\}$. By inversion of the typing, we get:[1] $\emptyset; \emptyset; x : \mathsf{U}_M\, \underline{B} \vdash^{\mathsf{c}}_{M} t : \underline{B}$. To prove $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{M} t\{\mathsf{thunk}\, \mu x.\, t/x\} : \underline{B}$, we use substitution. With Thunk, it suffices to show $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{M} \mu x.\, t : \underline{B}$, which was our assumption.

- Case $\mathsf{force}\, (\mathsf{thunk}\, t) \succ_1 t$. By inverting the typing, we get: $\phi; \Phi; \emptyset \vdash^{\mathsf{v}} \mathsf{thunk}\, t : \mathsf{U}_{M'}\, \underline{B}$ and $\phi; \Phi \vDash 1 + M' \le M$. The goal follows by inverting the above typing of $\mathsf{thunk}\, t$.

- Case $\mathsf{bind}\, x \leftarrow \mathsf{return}\, v\, \mathsf{in}\, t \succ_0 t\{v/x\}$. By inversion, we get: $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{M_1} \mathsf{return}\, v : \mathsf{F}\, A$ and $\phi; \Phi; x : A \vdash^{\mathsf{c}}_{M_2} t : \underline{B}$. By inverting the typing of $\mathsf{return}\, v$, we get $\phi; \Phi; \emptyset \vdash^{\mathsf{v}} v : A$. By substitution, we can type $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{M_2} t\{v/x\} : \underline{B}$.

---

[1] If the typing used subsumption, it could also be that there is a weaker type in the context: $x : \mathsf{U}_{M'}\, B'$ with $\phi; \Phi \vDash M \le M'$ and $\phi; \Phi \vdash B \sqsubseteq B'$. However, we can arrive at the typing of t that follows this footnote after applying subsumption again.

- Cases $\mathsf{calc}\, x \leftarrow \mathsf{Succ}(\underline{n}) \,\mathsf{in}\, t \succ_0 t\{\underline{1+n}/x\}$ and $\mathsf{calc}\, x \leftarrow \mathsf{Pred}(\underline{n}) \,\mathsf{in}\, t \succ_0 t\{\underline{n \dot{-} 1}/x\}$: As the above case.

- The cases $\mathsf{ifz}\, \underline{n} \,\mathsf{then}\, t_1 \,\mathsf{else}\, t_2 \succ_0 t_{1,2}$ follow by inversion of the typing. □

Corollary 11.4 (Soundness of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$)**.** *Let $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{k} t : \underline{B}$. Then there exists a number $k' \le k$ and a terminal computation T, such that $t \Downarrow_{k'} T$ and $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{k-k'} T : \underline{B}$.*

Corollary 11.5 (Soundness of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$)**.** *Let $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{k} t : \mathsf{F}\, A$. Then there exists a number $k' \le k$ and a value v, such that $t \Downarrow_{k'} \mathsf{return}\, v$ and $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{v}} v : A$.*

It is important to note that we have to assume that the cost of the typing is a constant (or equivalently, a terminating index term). If the cost is diverging/undefined, we cannot derive from the typing whether the computation terminates or diverges.

If a typing of a program has a constant cost, we can show that the Nat-refinement index term is also terminating:

Corollary 11.6 (Soundness of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ programs)**.** *Let $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{k} t : \mathsf{F}\, \mathsf{Nat}[I]$. Then there exists a number $k' \le k$ and a constant n, such that $t \Downarrow_{k'} \mathsf{return}\, \underline{n}$ and $\phi; \Phi \vDash I = n$.*

Proof. With Corollary 11.5, we get a value v with $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{v}} v : \mathsf{Nat}[I]$. By inversion, we have $v = \underline{n}$ and $\emptyset; \emptyset \vDash I = n$. □

Moreover, if we assume a precise typing, it is easy to show that the cost of a typing is precisely the cost of executing the computation. This also implies that if a computation terminates, the cost of the precise typing also has to terminate.

Lemma 11.7 (Subject reduction for precise typings)**.** *Let $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{K} t : \underline{B}$ be a precise typing, and let $t \succ_i t'$ be a step. Then there exists an index term K' such that $\phi; \Phi; \emptyset \vdash^{\mathsf{c}}_{K'} t' : \underline{B}$ is a precise typing and $\phi; \Phi \vDash K' + i \equiv K$.*

Proof (sketch). As in Theorem 11.3. Replace $\le$ and $\sqsubseteq$ with $\equiv$. □

Lemma 11.8. *Let $\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_{K} T : \underline{B}$ be a precise typing of a terminal computation. Then $\phi; \Phi \vDash K \equiv 0$.*

Corollary 11.9 (Adequacy)**.** *Assume a precise typing $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{K} t : \underline{B}$. Also assume that $t \Downarrow_k T$. Then $\phi; \Phi \vDash K \equiv k$.*

Proof. By repeatedly applying Lemma 11.7 on the precise typing, we get a typing for the terminal computation T: $\emptyset; \emptyset; \emptyset \vdash_{K'} T : \underline{B}$ with $\phi; \Phi \vDash K' + k \equiv K$. (As in the proof of Corollary 7.20, this procedure is well-founded, since there are exactly k forcing steps and the size of t decreases after every other step.) This typing is also precise, since subject reduction preserves precision. By Lemma 11.8, we have $\phi; \Phi \vDash K' \equiv 0$ and hence $\phi; \Phi \vDash K \equiv k$. □

11.3 Semantic soundness

In the previous section, we have proved a strong version of subject reduction. It not only entails *type safety*, but we can also prove normalisation: If the cost of a computation typing is defined, Corollary 11.4 states that the computation terminates. Subject reduction also implies that the cost of a typing of a terminating program is also terminating, and so is its Nat refinement.

Soundness of type systems is usually shown using (unary) *logical relations*. In the fixpoint case, however, our reasoning would be circular. This problem is usually remedied by using *step indexing*. However, it is not known to us whether step indexing can also be used to show that well-typed terms terminate. Nevertheless, we can still prove *semantic soundness* – as a corollary of subject reduction.

Definition 11.10 (Semantic typing)**.** Let $val(\phi)$ denote the set of valuations of a sorting context ϕ. We define the sets of closed terms $\mathbb{V}[\![A]\!]$, $\mathbb{T}[\![\underline{B}]\!]$, $\mathbb{C}[\![\underline{B}]\!]_K$ by mutual induction on the shape of the types. As an invariant, we assume that the types are closed.

$$\begin{aligned}
\mathbb{V}[\![\mathsf{Nat}[I]]\!] &:= \{\underline{n} \mid I \Downarrow n \vee I \not\Downarrow\} \\
\mathbb{V}[\![\mathsf{U}_K\,\underline{B}]\!] &:= \{\mathsf{thunk}\,t \mid t \in \mathbb{C}[\![\underline{B}]\!]_K\} \\
\mathbb{T}[\![\mathsf{F}\,A]\!] &:= \{\mathsf{return}\,v \mid v \in \mathbb{V}[\![A]\!]\} \\
\mathbb{T}[\![\forall\vec{h}.\,A \xrightarrow{K} \underline{B}]\!] &:= \left\{\lambda x.\,t \mid \emptyset \vdash^{\mathsf{c}} \lambda x.\,t : (\!|A \to \underline{B}|\!) \wedge \forall\nu \in val(\vec{h}).\,\forall v \in \mathbb{V}[\![A\nu]\!].\,t\{v/x\} \in \mathbb{C}[\![\underline{B}\nu]\!]_{K\nu}\right\} \\
\mathbb{C}[\![\underline{B}]\!]_K &:= \{t \mid \emptyset \vdash^{\mathsf{c}} t : (\!|\underline{B}|\!) \wedge \forall k.\,K \Downarrow k \Rightarrow \exists T\,k'.\,t \Downarrow_{k'} T \wedge k' \leq k \wedge T \in \mathbb{T}[\![\underline{B}]\!]\} \\
\mathbb{G}[\![\Gamma]\!] &:= \{\gamma \mid \forall(x : A) \in \Gamma.\,\gamma(x) \in \mathbb{V}[\![A]\!]\}
\end{aligned}$$

$\mathbb{G}[\![\Gamma]\!]$ assigns a term substitution to a context. We now define semantic subtypings and typings and prove semantic soundness:

$$\begin{aligned}
\phi;\Phi \vDash^{\mathsf{v}} A_1 \sqsubseteq A_2 &:= (\!|A_1|\!) = (\!|A_2|\!) \wedge \forall\nu \in val(\phi).\,\vDash \Phi\nu \Rightarrow \mathbb{V}[\![A_1\nu]\!] \subseteq \mathbb{V}[\![A_2\nu]\!] \\
\phi;\Phi \vDash^{\mathsf{v}} \underline{B}_1 \sqsubseteq \underline{B}_2 &:= (\!|\underline{B}_1|\!) = (\!|\underline{B}_2|\!) \wedge \forall\nu \in val(\phi).\,\vDash \Phi\nu \Rightarrow \mathbb{T}[\![\underline{B}_1\nu]\!] \subseteq \mathbb{T}[\![\underline{B}_2\nu]\!] \\
\phi;\Phi;\Gamma \vDash^{\mathsf{v}} v : A &:= (\!|\Gamma|\!) \vdash^{\mathsf{v}} v : (\!|A|\!) \wedge \forall\nu \in val(\phi).\,\vDash \Phi\nu \Rightarrow \forall\gamma \in \mathbb{G}[\![\Gamma]\!].\,v\gamma \in \mathbb{V}[\![A\nu]\!] \\
\phi;\Phi;\Gamma \vDash^{\mathsf{c}}_K t : \underline{B} &:= (\!|\Gamma|\!) \vdash^{\mathsf{c}} t : (\!|\underline{B}|\!) \wedge \forall\nu \in val(\phi).\,\vDash \Phi\nu \Rightarrow \forall\gamma \in \mathbb{G}[\![\Gamma]\!].\,t\gamma \in \mathbb{C}[\![\underline{B}\nu]\!]_{K\nu}
\end{aligned}$$

Lemma 11.11 (Semantic soundness)**.** *We prove the following statements:*

1. *For a closed value v, $\emptyset;\emptyset;\emptyset \vdash^{\mathsf{v}} v : A$ implies $v \in \mathbb{V}[\![A]\!]$.*
2. *For a closed terminal computation T, $\emptyset;\emptyset;\emptyset \vdash^{\mathsf{c}}_0 T : \underline{B}$ implies $T \in \mathbb{T}[\![\underline{B}]\!]$.*
3. *For a closed computation t, $\emptyset;\emptyset;\emptyset \vdash^{\mathsf{c}}_K t : \underline{B}$ implies $t \in \mathbb{C}[\![\underline{B}]\!]_K$.*

Proof. We prove the first two statements by induction on their typings (or on the size of the terms); the last point is independent.

1. Case analysis on the value typing:
    - $v = \underline{n}$ and $A = \mathsf{Nat}[I]$ with $\emptyset;\emptyset \vDash n \sqsubseteq I$. Trivial, since (by definition) we either have that $I \Downarrow n$ or I diverges.

  - Case $v = \mathsf{thunk}\, t$, $A = \mathsf{U}_K\, \underline{B}$ and $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_K t : \underline{B}$: By the inductive hypothesis (point 2), we have $t \in \mathbb{T}[\![\underline{B}]\!]$. This implies the goal.

2. Case analysis on the typing of the terminal computation T:

   - Case $T = \mathsf{return}\, v$, $\underline{B} = \mathsf{F}\, A$, and $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{v}} v : A$. By the inductive hypothesis (point 1), we have $v \in \mathbb{V}[\![A]\!]$. This implies the goal.

   - Case $T = \lambda x.\, t$, $\underline{B} = \forall \vec{h}.\, A \xrightarrow{K} \underline{B}'$, and $\emptyset; \emptyset; x : A \vdash^{\mathsf{c}}_K t : A$. Let $\nu \in \mathit{val}(\vec{h})$ and $v \in \mathbb{V}[\![A\nu]\!]$. We have to show $t\{v/x\} \in \mathbb{C}[\![\underline{B}\nu]\!]_{K\nu}$, as in point 3: Assume that $K\nu \Downarrow k$. By Corollary 11.4, we have that $t\{v/x\} \Downarrow_{k'} T$ for $k' \leq k$ steps. By the inductive hypothesis (point 2), we have $T \in \mathbb{T}[\![\underline{B}]\!]$.

3. Let $K \Downarrow k$. By Corollary 11.4, we have that $t \Downarrow_{k'} T$ with $k' \leq k$. By point 2, we have $T \in \mathbb{T}[\![\underline{B}]\!]$. □

Theorem 11.12 (Semantic soundness for arbitrary typings)**.** *For an arbitrary term, $\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_K t : \underline{B}$ implies $\phi; \Phi; \Gamma \vDash^{\mathsf{c}}_K t : \underline{B}$.*

For an arbitrary value, $\phi; \Phi; \Gamma \vdash^{\mathsf{v}} v : \underline{B}$ implies $\phi; \Phi; \Gamma \vDash^{\mathsf{v}} v : \underline{B}$.

Proof. Both statements follow from Lemma 11.11, substitution (Lemma 11.1), and index term substitution (Lemma 11.2). □

Note that if we want to prove Theorem 11.12 directly by induction on the typings, we get stuck in the fixpoint case. It is impossible to prove compatibility of the fixpoint rule directly:

$$\frac{\phi; \Phi; x : \mathsf{U}_K\, \underline{B}, \Gamma \vDash^{\mathsf{c}}_K t : \underline{B}}{\phi; \Phi; \Gamma \vDash^{\mathsf{c}}_K \mu x.\, t : \underline{B}}$$

11.4 Parametric Completeness

The definitions of parametric and parametrised $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{pv}}$ (value and computation) types is similar to those in the previous chapter.

Definition 11.13 (Parametricity)**.** We define using mutual induction:

$$\frac{}{pa^{+}(\mathsf{Nat}[K(\phi_2)]\,;\phi_1;\phi_2;\emptyset;K)} \qquad \frac{pa^{-}(A;\vec{h}_1;h_2) \qquad pa^{+}(\underline{B};\vec{h}_1,\phi_1;h_2,\phi_2;\vec{K}_1;K_2)}{pa^{+}(\forall \vec{h}_1\,h_2.\,A \xrightarrow{I(\vec{h}_1,h_2,\phi_1,\phi_2)} \underline{B};\phi_1;\phi_2;I,\vec{K}_1;K_2)}$$

$$\frac{}{pa^{-}(\mathsf{Nat}[i]\,;\emptyset;i)} \qquad \frac{pa^{-}(A;\vec{k}_1;k_2) \qquad pa^{+}(\underline{B};\vec{k}_1;\langle k_2\rangle\,;\vec{h}_1;h_2)}{pa^{-}(\forall \vec{k}_1\,k_2.\,A \xrightarrow{k(\vec{k}_1\,k_2)} \underline{B};k,\vec{h}_1;h_2)}$$

$$\frac{pa^{+}(A;\phi_1;\phi_2;\vec{K}_1;K_2)}{pa^{+}(\mathsf{F}\,A;\phi_1;\phi_2;\vec{K}_1;K_2)} \qquad \frac{pa^{+}(\underline{B};\phi_1;\phi_2;\vec{K}_1;K_2)}{pa^{+}(\mathsf{U}_{I(\phi_1,\phi_2)}\,\underline{B};\phi_1;\phi_2;I,\vec{K}_1;K_2)}$$

$$\frac{pa^{-}(A;\vec{h}_1;h_2)}{pa^{-}(\mathsf{F}\,A;\vec{h}_1;h_2)} \qquad \frac{pa^{-}(\underline{B};\vec{h}_1;h_2)}{pa^{-}(\mathsf{U}_i\,\underline{B};i,\vec{h}_1;h_2)}$$

We define *parametrised* and *parametric* type annotations analogously to Definition 10.8 and Definition 10.10, respectively:

Definition 11.14 (Parametrised type annotation)**.** Let $\hat{A}$ be a simple type and τ be a $\mathsf{d}f\mathsf{PCF}_{\mathsf{pv}}$ (either a value or computation) type. We say that τ is an *parametrised annotation* of $\hat{A}$ over $\vec{h}_1, h_2$, if $(\!|\tau|\!) = \hat{A}$ and $pa^{-}(\tau;\vec{h}_1;h_2)$.

Definition 11.15 (Parametric annotations of types)**.** Let $\phi = \phi_1, \phi_2$ be index variables. Let $\hat{A}$ be a simple type and τ be a $\mathsf{d}f\mathsf{PCF}_{\mathsf{pv}}$ (either a value or computation) type. We say that τ is an *(effect-)parametric annotation* of $\hat{A}$ (in $\phi_1;\phi_2$), if $(\!|\tau|\!) = \hat{A}$ and $pa^{+}(\tau;\phi_1;\phi_2;\vec{I}_1;I_2)$ for some index terms $\vec{I}_1$, I_2.

In $\mathsf{d}f\mathsf{PCF}_{\mathsf{pv}}$, there are two kinds of variables. First, as in $\mathsf{d}f\mathsf{T}$, variables can be introduced in λ-abstractions and fixpoints. For this kind of variables, we make the types in the context *parametrised*, as in $\mathsf{d}f\mathsf{T}$. The second kind of variables is introduced in the rules Bind, Succ, and Pred. There, we add a value type to the context that represents the value of a previous computation. Therefore, the type for this kind of variables is already *parametric.* In the variable case of the annotation algorithm below, we have to make a case distinction over the kind of the variable.

For example, when we annotate the computation $\lambda x.\,\mathsf{calc}\,y \leftarrow \mathsf{Succ}(x)\,\mathsf{in}\,\lambda z.\,y$, the final typing of y has the following shape:

$$i,j;\ \underbrace{x : \mathsf{Nat}[i]\,,}_{\text{parametrised by } i}\ \underbrace{y : \mathsf{Nat}[1+i]\,,}_{\text{parametric in } i}\ \underbrace{z : \mathsf{Nat}[j]}_{\text{parametrised by } j}\ \vdash^{\mathsf{v}} y : \underbrace{\mathsf{Nat}[1+i]}_{\text{parametric in } i,j}$$

Definition 11.16 (Context annotation)**.** Let $\phi = \phi_1, \phi_2$ be sort contexts. Let $\hat{\Gamma}$ be a simple context and Γ be a $\mathsf{d}f\mathsf{PCF}_{\mathsf{pv}}$ context. We say that Γ is an *annotation* of $\hat{\Gamma}$ (in ϕ_1, ϕ_2) if:

- Every variable of Γ is labelled either as ‘parametrised’ or as ‘parametric’;

- for each parametrised x, there is a distinct index variable $h_2(x)$ of ϕ_2;
- ϕ_1 can be partitioned into lists of index variables $\vec{h}_1(x)$ for the parametrised variables;
- for every parametrised x, $\Gamma(x)$ is a parametrised annotation of $\hat{\Gamma}(x)$ in $\vec{h}_1(x), h_2(x)$.
- for every parametric x, $\Gamma(x)$ is parametric in the index variables for the parametric index variables to the right of x in Γ.

We can now prove the main theorem of this section: For each simple call-by-push-value typing, there exists a precise $\mathsf{d}f\mathsf{PCF}_{\mathsf{pv}}$ annotation.

Theorem 11.17 (Annotating typings)**.** *Let* Γ *be an annotation of a simple context* $\hat{\Gamma}$ *(in* $\phi = \phi_1, \phi_2$*).*

- *Let* $\hat{\Gamma} \vdash^{\mathsf{v}} v : \hat{A}$ *be a simple* CBPV *typing. Then we can compute a parametric annotation* A *of* $\hat{A}$ *(in* ϕ_1, ϕ_2*) and a precise value typing* $\phi; \emptyset; \Gamma \vdash^{\mathsf{v}} v : A$.
- *Let* $\hat{\Gamma} \vdash^{\mathsf{c}} t : \underline{\hat{B}}$ *be a simple* CBPV *computation typing. Then we can compute a parametric annotation* $\underline{B}$ *of* $\underline{\hat{B}}$ *(in* ϕ_1, ϕ_2*), and a closed index term* M*, together with a precise computation typing* $\phi; \emptyset; \Gamma \vdash^{\mathsf{c}}_{M(\phi_1, \phi_2)} t : \underline{B}$.

(These typings are called parametric annotation of simple (value/computation) typings.)

Proof. By induction on the simple typing.

- The cases constant, application, case distinction, and λ-abstraction are exactly as in $\mathsf{d}f\mathsf{T}$ (see proof of Theorem 10.11).
- Case $v = x$. If x is a parametrised variable, we proceed as in $\mathsf{d}f\mathsf{T}$. Otherwise, $\Gamma(x)$ is already parametric in a subset of the index terms ϕ_1, ϕ_2. We only have to add the missing abstractions.
- Case $v = \mathsf{thunk}\, t$. The inductive hypothesis yields an annotated computation typing $\phi; \emptyset; \Gamma \vdash^{\mathsf{c}}_{K(\phi_1, \phi_2,)} t : \underline{B}$. We use Thunk: $\phi; \emptyset; \Gamma \vdash^{\mathsf{v}} \mathsf{thunk}\, t : \mathsf{U}_{K(\phi_1, \phi_2,)}\, \underline{B}$.
- Case $t = \mathsf{return}\, v$. Similarly to the above; the inductive hypothesis yields an annotated value typing for v. We only have to apply Return.
- Case $t = \mathsf{force}\, v$. The inductive hypothesis yields $\phi; \emptyset; \Gamma \vdash^{\mathsf{v}} v : \mathsf{U}_{K(\phi_1, \phi_2,)}\, \underline{B}$. We apply Force, which also increments the cost:

$$\phi; \emptyset; \Gamma \vdash^{\mathsf{c}}_{(\lambda(\phi_1, \phi_2).\, 1 + K(\phi_1, \phi_2))\, (\phi_1, \phi_2)} \mathsf{force}\, v : \underline{B}$$

- Case $t = \mathsf{bind}\, x \leftarrow t_1 \,\mathsf{in}\, t_2$. We have simple typings $\hat{\Gamma} \vdash^{\mathsf{c}} t_1 : \mathsf{F}\, \hat{A}$ and $x : \hat{A}, \hat{\Gamma} \vdash^{\mathsf{c}} t_2 : \underline{\hat{B}}$. The first inductive hypothesis yields an annotation $\phi; \emptyset; \Gamma \vdash^{\mathsf{c}}_{K_1} t : \mathsf{F}\, A$. Note that $x : A, \Gamma$ is an annotation for $\hat{A}, \hat{\Gamma}$ (where A is parametric). Thus, we can apply the inductive hypothesis on the typing of t_2, which yields the goal.

- Cases $\mathsf{calc}\, x \leftarrow \mathsf{Succ}(v)\, \mathsf{in}\, t$ and $\mathsf{calc}\, x \leftarrow \mathsf{Pred}(v)\, \mathsf{in}\, t$: As the above case.

- Case $\mu x.\, t$. The simple typing is $x : \mathsf{U}\, \hat{\underline{B}}, \hat{\Gamma} \vdash t : \hat{\underline{B}}$. We parametrise the type of x using fresh index variables $i, \vec{h}_1, h_2$. The inductive hypothesis yields the following annotated typing:

$$\phi^* := i, \vec{h}_1, h_2, \phi_1, \phi_2; x : \mathsf{U}_i\, \underline{B}_1, \Gamma \vdash^{\mathsf{c}}_{K(\phi^*)} t : \underline{B}_2$$

We have $pa^-(\underline{B}_1; \vec{h}_1, h_2)$ and $pa^+(\underline{B}_2; i, \vec{h}_1, \phi_1; h_2, \phi_2; \vec{G}_1; G_2)$. As in the iteration case, we unify the parametrised type $\underline{B}_1$ with the parametric type $\underline{B}_2$. For this, we first merge the index variables into a new index variable list: $h^* := i, \vec{h}_1, h_2$. Now, we construct the index term I that takes the tuple h^* as argument and has the index variables ϕ free. The index term I not only 'updates' the refinements in $\underline{B}_2$, but it also computes the cost of t (which may depend on i). Finally, we apply the fixpoint operator on this index term:

$$\begin{aligned}
I &:= \lambda(i, \vec{h}_1, h_2).\\
&\qquad \big\langle K(\phi^*);\\
&\qquad\quad \lambda(\cdots).\, G_{11}(\cdots, \phi^*); \ldots; \lambda(\cdots).\, G_{1n}(\cdots, \phi^*);\\
&\qquad\quad \lambda(\cdots).\, G_2(\cdots, h_2, \phi_2)\big\rangle\\
I^* &:= \mu(i, \vec{h}_1, h_2).\, I(i, \vec{h}_1, h_2)\\
K^* &:= \pi_1(I^*)\\
\underline{B}^* &:= \underline{B}_1(i, \vec{h}_1, h_2 := I^*)
\end{aligned}$$

We can substitute I^* for h^* in the inductive hypothesis. Since I^* is a fixed point (i.e. $I(I^*) \equiv I$), this brings the typing into the right shape to apply Fix:

$$\frac{\phi; \emptyset; x : \mathsf{U}_{K^*}\, \underline{B}^*, \Gamma \vdash^{\mathsf{c}}_{K^*} \underline{B}^*}{\phi; \emptyset; \Gamma \vdash^{\mathsf{c}}_{K^*} \mu x.\, t : \underline{B}^*}$$

After making $\underline{B}^*$ parametric again (by re-introducing λ-abstractions on ϕ again), we are done. □

11.5 Call-by-value version and embedding of dℓT

We can derive an effect system for the call-by-value version of PCF, dℓPCF$_{\mathsf{v}}$. For this, we take the rules of dℓT and substitute the iteration rule with the following rule for fixpoints:

$$\textsc{Fix}\quad \frac{\phi; \Phi; f : \tau, \Gamma \vdash_0 \lambda x.\, t : \tau}{\phi; \Phi; \Gamma \vdash_0 \mu f x.\, t : \tau}$$

Using the translation function $\cdot^{\mathsf{v}}$, we can translate CBV terms to CBPV computations, as in Section 2.3.4. It is easy to embed dℓPCF$_{\mathsf{v}}$ in dℓPCF$_{\mathsf{pv}}$. Moreover, we can embed dℓT in

$\mathsf{d}f\mathsf{PCF_v}$: We only have to introduce iteration in $\mathsf{d}f\mathsf{PCF_v}$ as syntactic sugar, as we did in Section 5.6:

$$\mathsf{iter}\, t_1\, t_2 := \mu f x.\, \mathsf{ifz}\, x\, \mathsf{then}\, t_2\, \mathsf{else}\, t_1\, (f\, (\mathsf{Pred}(x)))$$

Also, it is easy to show that the rule Iter is admissible in $\mathsf{d}f\mathsf{PCF_v}$:

Proof. Abbreviate $K^* := i\cdot(2+M_1)+M_2+\sum_{a<i} K(\vec{h} := \mathsf{iter}\, g\, f\, a)$ and $\rho := \forall i.\, \mathsf{Nat}[i] \xrightarrow{K^*} \tau(\vec{h} := \mathsf{iter}\, g\, f\, i)$. Then we can derive the following typing:

$$\dfrac{\dfrac{\dfrac{i, \phi; i = 0, \Phi; x : \mathsf{Nat}[i]\, , f : \rho, \Gamma \vdash_{M_2} t_2 : \tau(\vec{h} := f) \qquad \dfrac{\begin{array}{c} i, \phi; 0 < i, \Phi; x : \mathsf{Nat}[i]\, , f : \rho, \Gamma \vdash_{M_1} t_1 : \forall \vec{h}.\, \tau \xrightarrow{K} \tau(\vec{h} := g(\vec{h})) \\ \cdots \vdash_{1+K^*\{i-1/i\}} f(\mathsf{Pred}(x)) : \tau(\vec{h} := \mathsf{iter}\, g\, f\, (i-1)) \end{array}}{\cdots \vdash_{2+M_1+K^*\{i-1/i\}} t_1(f(\mathsf{Pred}(x))) : \tau(\vec{h} := \mathsf{iter}\, g\, f\, i)}}{i, \phi; \Phi; x : \mathsf{Nat}[i]\, , f : \rho, \Gamma \vdash_{K^*} \mathsf{ifz}\, x\, \mathsf{then}\, t_2\, \mathsf{else}\, t_1(f(\mathsf{Pred}(x))) : \tau(\vec{h} := \mathsf{iter}\, g\, f\, i)}}{\phi; \Phi; x : \rho, \Gamma \vdash_0 \lambda x.\, \mathsf{ifz}\, x\, \mathsf{then}\, t_2\, \mathsf{else}\, t_1(f(\mathsf{Pred}(x))) : \rho}}{\phi; \Phi; \Gamma \vdash_0 \mathsf{iter}\, t_1\, t_2 : \rho}$$

□

11.6 Annotation examples

In this section, we demonstrate the annotation algorithm on a few examples.

Diverging fixpoint

We can type $\mu x.\, \mathsf{force}\, x$ in $\mathsf{d}f\mathsf{PCF_{pv}}$. For this, we need to introduce two index variables of sort Nat:

$$\dfrac{i, j; \emptyset; x : \mathsf{F\,Nat}[j] \vdash^{\mathsf{c}}_{1+j} \mathsf{force}\, x : \mathsf{F\,Nat}[j]}{\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{\pi_1(I^*)} \mu x.\, \mathsf{force}\, x : \mathsf{F\,Nat}[\pi_2(I^*)]}$$

where $I^* := \mu\langle i; j\rangle.\, \langle 1 + i; j\rangle$. Obviously, this index term diverges, and so does the term.

Minimum

We type the minimum function, which is defined as follows:

$$\begin{aligned} \mathit{min} &:= \mu f.\, \lambda x.\, \lambda y.\, t \\ t &:= \mathsf{ifz}\, x\, \mathsf{then}\, \underline{0}\, \mathsf{else}\, \mathsf{ifz}\, y\, \mathsf{then}\, \underline{0}\, \mathsf{else}\, \mathsf{calc}\, x' \leftarrow \mathsf{Pred}(x)\, \mathsf{in}\, \mathsf{calc}\, y' \leftarrow \mathsf{Pred}(y)\, \mathsf{in}\, t' \\ t' &:= \mathsf{bind}\, z \leftarrow (\mathsf{force}\, f)\, x'\, y'\, \mathsf{in}\, \mathsf{calc}\, z' \leftarrow \mathsf{Succ}(z)\, \mathsf{in}\, \mathsf{return}\, z' \end{aligned}$$

Note that this function is equivalent to the (unthunked) call-by-value translation of:

$$\mu f x.\, \lambda y.\, \mathsf{ifz}\, x\, \mathsf{then}\, 0\, \mathsf{else}\, \mathsf{ifz}\, y\, \mathsf{then}\, 0\, \mathsf{else}\, \mathsf{Succ}\, (f\, (\mathsf{Pred}(x))\, (\mathsf{Pred}(y)))$$

First, we define the parametrised computation type $\underline{B}$ with three free index variables:

$$\underline{B} := \forall a.\, \mathsf{Nat}[a] \xrightarrow{h_{11}(a)} \forall b.\, \mathsf{Nat}[b] \xrightarrow{h_{12}(a,b)} \mathsf{Nat}[h_2(a, b)]$$

We also introduce a fresh index variable i and type $\lambda x.\,\lambda y.\,t$ with $f : \mathsf{U}_i\,\underline{B}$ in the context. For this, we proceed as in dfT. The following typing, for example, is one of the intermediate steps:

$$\begin{aligned}a, b, i, h_{11}, h_{12}, h_2; a > 0; b > 0; x' : \mathsf{Nat}[a-1]\,, y' : \mathsf{Nat}[b-1]\,, x : \mathsf{Nat}[a]\,, y : \mathsf{Nat}[b]\,,\\ f : \mathsf{U}_i\,\underline{B} \vdash_{1+i+h_{11}(a-1)+h_{12}(a-1,b-1)} t' : \mathsf{Nat}[1 + h_2(a-1, b-1)]\end{aligned}$$

Note that in the above typing, the types of the variables x' and y' are *parametric*, since they are binders introduced by Pred. Ultimately, we will arrive at the following typing:

$$i, h_{11}, h_{12}h_{,2}\,; \emptyset; f : \mathsf{U}_i\,\underline{B} \vdash_{\pi_1(I)} \lambda x.\,\lambda y.\,t : \underline{B}(i, h_{11}, h_{12}, h_2 := I)$$

with the following index term:

$$\begin{aligned}I :=\ &\big\langle 0;\\ &\quad \lambda a.\,0;\\ &\quad \lambda(a,b).\,\mathsf{ifz}\,a\,\mathsf{then}\,0\,\mathsf{else}\,\mathsf{ifz}\,b\,\mathsf{then}\,0\,\mathsf{else}\,1 + i + h_{11}(a-1) + h_{12}(a-1, b-1);\\ &\quad \lambda(a,b).\,\mathsf{ifz}\,a\,\mathsf{then}\,0\,\mathsf{else}\,\mathsf{ifz}\,b\,\mathsf{then}\,0\,\mathsf{else}\,1 + h_2(a-1, b-1)\rangle\big\rangle\\ I^* :=\ &\mu\langle i; h_{11}; h_{12}; h_2\rangle.\,I\end{aligned}$$

Now, we can apply the fixpoint typing rule: $\emptyset; \emptyset; \emptyset \vdash_{\pi_1(I^*)} \mathit{min} : \underline{B}(h_2, i, h_{11}, h_{12} := I^*)$. By solving the recurrences, we can simplify the typing:

$$\emptyset; \emptyset; \emptyset \vdash_0 \mathit{min} : \forall a.\,\mathsf{Nat}[a] \xrightarrow{0} \forall b.\,\mathsf{Nat}[b] \xrightarrow{\mathit{min}\,a\,b} \mathsf{Nat}[\mathit{min}\,a\,b]$$

with an implementation of *min* in $\mathcal{L}^f_{idx}$. This means that if we apply the dfPCF$_{\mathsf{pv}}$ function *min* to two constants $\underline{n_1}$ and $\underline{n_2}$ with $m = \mathit{min}\,n_1\,n_2$, the computation terminates in return $\underline{m}$, and the cost (i.e. the number of forcing steps) is $0 + m = m$.

11.7 Extensions of dℓPCF$_{\mathsf{pv}}$

In this last section, we discuss two extensions to dfPCF.

11.7.1 Conjunctives and disjunctives

As we discussed in Section 7.7, we can trivially extend dfPCF$_{\mathsf{pv}}$ with conjunctives and disjunctives. We introduce the following value and computation types:

$$\begin{aligned}A &::= \cdots \mid 1 \mid A_1 \otimes A_2 \mid A_1 \oplus A_2\\ \underline{B} &::= \cdots \mid \underline{B}_1\ {}_{M_1}\&_{M_2}\ \underline{B}_2\end{aligned}$$

Note that the additive conjunction operator & is refined with two index terms that denote the cost of either projection. The rules are shown in Figure 11.2.

$$\textsc{Unit}\qquad \phi;\Phi;\emptyset \vdash^{\mathsf{v}} () : 1$$

$$\textsc{MProd}\qquad \frac{\phi;\Phi;\Gamma \vdash^{\mathsf{v}} v_1 : A_1 \qquad \phi;\Phi;\Gamma \vdash^{\mathsf{v}} v_2 : A_2}{\phi;\Phi;\Gamma \vdash^{\mathsf{v}} (v_1;v_2) : A_1 \otimes A_2}$$

$$\textsc{LetPair}\qquad \frac{\begin{array}{c}\phi;\Phi;\Gamma \vdash^{\mathsf{v}} v : A_1 \otimes A_2 \\ \phi;\Phi;x : A_1, y : A_2, \Gamma \vdash^{\mathsf{c}}_{M} t : \underline{B}\end{array}}{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{M} \mathsf{let}\,(x;y) := v\,\mathsf{in}\,t : \underline{B}}$$

$$\textsc{AProd}\qquad \frac{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{M_1} t_1 : \underline{B}_1 \qquad \phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{M_2} t_2 : \underline{B}_2}{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{0} \langle t_1;t_2\rangle : \underline{B}_1\ {}_{M_1}\&_{M_2}\ \underline{B}_2}$$

$$\textsc{Proj}\qquad \frac{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{M} t : \underline{B}_1\ {}_{M_1}\&_{M_2}\ \underline{B}_2}{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{M+M_i} \pi_i(t) : \underline{B}_i}$$

$$\textsc{Inl}\qquad \frac{\phi;\Phi;\Gamma \vdash^{\mathsf{v}} v : A_1}{\phi;\Phi;\Gamma \vdash^{\mathsf{v}} \mathsf{inl}(v) : A_1 \oplus A_2}$$

$$\textsc{Inr}\qquad \frac{\phi;\Phi;\Gamma \vdash^{\mathsf{v}} v : A_2}{\phi;\Phi;\Gamma \vdash^{\mathsf{v}} \mathsf{inr}(v) : A_1 \oplus A_2}$$

$$\textsc{CaseSum}\qquad \frac{\phi;\Phi;\Gamma \vdash^{\mathsf{v}} v : A_1 \oplus A_2 \qquad \phi;\Phi;x : A_1, \Gamma \vdash^{\mathsf{c}}_{M} t_1 : \underline{B} \qquad \phi;\Phi;y : A_2, \Gamma \vdash^{\mathsf{c}}_{M} t_2 : \underline{B}}{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{M} \mathsf{case}\,v\,[\mathsf{inl}(x) \Rightarrow t_1 \mid \mathsf{inr}(y) \Rightarrow t_2] : \underline{B}}$$

Figure 11.2: Typing rules for conjunctives and disjunctives for dfPCF$_{\mathsf{pv}}$

11.7.2 Polymorphism

Perhaps surprisingly, polymorphism is not supported by dfPCF. For example, consider the following System F typing: $\Lambda.\,\Lambda.\,\lambda f.\,\lambda x.\,f\,x : \forall \alpha\,\beta.\,(\alpha \to \beta) \to (\alpha \to \beta)$. Depending on the 'choice' of α, we need to parametrise the type of x over a different number of index variables. For example, we can assign the following dfPCF$_{\mathsf{v}}$ types to instances of this function:

$$\forall f_1\, f_2.\, (\forall i.\mathsf{Nat}[i] \xrightarrow{f_1(i)} \mathsf{Nat}[f_2(i)]) \xrightarrow{0} (\forall i.\mathsf{Nat}[i] \xrightarrow{f_1(i)+1} \mathsf{Nat}[f_2(i)])$$

$$\begin{aligned}\forall g_{11}\, g_{12}\, g_2.\, (\forall f_1\, f_2.\, (\forall i.\, \mathsf{Nat}[i] \xrightarrow{f_1(i)} \mathsf{Nat}[f_2(i)]) \xrightarrow{g_{11}(i,f_1,f_2)} (\forall i.\, \mathsf{Nat}[i] \xrightarrow{g_{12}(i,f_1,f_2)} \mathsf{Nat}[g_2(i,f_2)])) \xrightarrow{0} \\ (\forall f_1\, f_2.\, (\forall i.\, \mathsf{Nat}[i] \xrightarrow{f_1(i)} \mathsf{Nat}[f_2(i)]) \xrightarrow{1+g_{11}(i,f_1,f_2)} (\forall i.\, \mathsf{Nat}[i] \xrightarrow{g_{12}(i,f_1,f_2)} \mathsf{Nat}[g_2(i,f_2)]))\end{aligned}$$

One solution to this problem could be to abstract over sorts, which we leave for future work.

Part III

Conclusions

Chapter 12

Discussion and conclusions

In this last chapter, we conclude the thesis and discuss related and future work.

12.1 Verification and complexity analysis using (co-)effect-based type systems

We first discuss and compare the main strengths and weaknesses of our two approaches.

Both $\mathsf{d}\ell\mathsf{PCF}$ and $\mathsf{d}f\mathsf{PCF}$ are families of sound and relatively complete refinement type systems for verification and complexity analysis of pure functional programs. The completeness results are *relative* insofar they depend on sufficiently expressive index term languages and logics that support them. Moreover, both approaches also support *gradual* refinements: The type $\mathsf{Nat}[\bot]$ is equivalent to the simple type Nat. This means that one can choose to omit all or some refinements. Moreover, if one restricts the index term language to polynomials, the systems could still be used for (strict) polynomial programs without fixpoints.

Furthermore, we have presented syntax-directed algorithms that take as input simple typings and compute annotated $\mathsf{d}\ell\mathsf{PCF}$ or $\mathsf{d}f\mathsf{PCF}$ typings. Since the annotated typings are *precise*, the computed annotations terminate if and only if the given terms terminate. These algorithms are efficient (polynomial in the size of the derivation tree), but do not attempt to simplify the generated index terms. This could perhaps be done partly automatically and partly manually.

We have already discussed some of the shortcomings of $\mathsf{d}\ell\mathsf{PCF}$ in Chapter 3. Summarising again:

- $\mathsf{d}\ell\mathsf{PCF}$ typings are not abstract. Observationally equivalent typings have different (precise) $\mathsf{d}\ell\mathsf{PCF}$ types. This is a prohibitive issue in practice, since verification would need to be repeated after program transformations. However, we conjecture that $\mathsf{d}\ell\mathsf{PCF}$ typings can be made fully abstract. For this, we would need to extend the systems such that they admit type equalities like $[a < 2] \cdot \underline{B} \equiv [a < 2] \cdot \underline{B}\{(\mathsf{if}\ a <$

$1\,\mathsf{then}\,1\,\mathsf{else}\,0)/a\}$. We also need to change the forcing typing rule:

$$\frac{\phi;\Phi;\Gamma \vdash^{\mathsf{v}}_{K} v : [a < I] \cdot \underline{B} \qquad \phi;\Phi \vDash I' < I}{\phi;\Phi;\Gamma \vdash^{\mathsf{c}}_{K} \mathsf{thunk}\, v : \underline{B}\{I'/a\}}$$

- Cost analysis is not possible for higher-order programs. From a given (precise) dℓPCF typing of a function, we cannot know how many steps a function needs to evaluate to a λ-abstraction. This is because weights of dℓPCF typings account for the actual cost of a term plus the cost of the (potential) applications that are permitted by the typing. We will discuss an easy fix for this below.

- In the fixpoint typing rule, recursion trees are directly encoded using index terms and the forest cardinality operator. However, reasoning about index terms with forest cardinalities can be tedious in practice. In particular, this can make it complicated to simplify typings generated by the annotation algorithm. Yet, for restricted recursive schemes like (higher-order) iteration, we can derive simpler rules.

- Typing higher-order primitive recursion functions like the Ackermann function is possible and mechanical. However, the result of the algorithm is not easy to understand. Applying the dfPCF annotation algorithm to the Ackermann function, however, yields sensible index terms that resemble the Ackermann function.

- One advantage of dℓPCF, however, is that it readily supports polymorphism, as we discussed in Section 8.2.

Interestingly, the disadvantages of dℓPCF are advantages of dfPCF, and *vice versa*:

- dfPCF offers full abstraction (as an easy corollary of the soundness results). Thus, we can simply replace a sub-program with an observationally equivalent (or more efficient) sub-program, without the need to simplify index terms again.

- Costs are more expressive than weights in dℓPCF. For all terms of all types, the cost of a typing is a static upper bound on the actual execution cost.

- Yet, type polymorphism is not readily supported by dfPCF.

Call-by-push-value Considering a call-by-push-value variant helped the author to better understand the systems. It also yielded technical contributions, since the proofs of soundness and (relative) completeness for dℓPCF$_{\mathsf{pv}}$ are easier. One reason is that dℓPCF$_{\mathsf{v}}$ and dℓPCF$_{\mathsf{n}}$ typing rules are composed of multiple parts in dℓPCF$_{\mathsf{pv}}$. For example, the fixpoint rule in dℓPCF$_{\mathsf{v}}$ embeds the fixpoint rule and the thunk rule of dℓPCF$_{\mathsf{pv}}$. Owing to this, we have to reason about forests instead of only trees.

In hindsight, we should have mechanised dℓPCF$_{\mathsf{pv}}$ instead of dℓPCF$_{\mathsf{v}}$, since it would probably have been easier and the results would have been more general. However, we devised dℓPCF$_{\mathsf{pv}}$ after the mechanisation of dℓPCF$_{\mathsf{v}}$. Nonetheless, the Coq mechanisation also lead to technical insights that could be carried over to dℓPCF$_{\mathsf{pv}}$ (like *typing skeletons* and the *findSlot* function) and also revealed mistakes in the original papers on dℓPCF$_{\mathsf{n}}$ [11] and dℓPCF$_{\mathsf{v}}$ [12].

12.2 Combining dℓPCF and dfPCF

To address the problem that dℓPCF does not support bounding the cost of a function (i.e. the number of forcing steps that it needs to reduce to a λ-abstraction), we can combine dℓPCF with ideas of dfPCF. In the combined system, values do not have any weight (similar to dfPCF, where values do not have a cost). The weight of a thunked computation is annotated using the refinement $[a < I]_M$:

$$\frac{a, \phi; a < I, \Phi; \Delta \vdash^{\mathsf{c}}_{K} t : \underline{B}}{\phi; \Phi; \sum_{a<I} \Delta \vdash^{\mathsf{v}} \mathsf{thunk}\, t : [a < I]_{\sum_{a<I} K} \cdot \underline{B}} \qquad \frac{\phi; \Phi; \Gamma \vdash^{\mathsf{v}} v : [a < 1]_K \cdot \underline{B}}{\phi; \Phi; \Gamma \vdash^{\mathsf{c}}_{1+K} \mathsf{force}\, v : \underline{B}\{0/a\}}$$

$$\frac{\phi; \Phi; \Delta_1 \vdash^{\mathsf{c}}_{K} t : A \multimap \underline{B} \qquad \phi; \Phi; \Delta_2 \vdash^{\mathsf{v}} v : A}{\phi; \Phi; \Delta_1 \uplus \Delta_2 \vdash^{\mathsf{c}}_{K} t\, v : \underline{B}}$$

Modal sums are trivially extended; the weights are just added. Now, for a closed precise typing $\emptyset; \emptyset; \emptyset \vdash^{\mathsf{c}}_{M} t : \mathsf{F}\,([a < I]_K \cdot \underline{B})$, M is the actual cost of t (i.e. the number of forcing steps that t needs until it returns a thunked computation), and the weight K accounts for the (potential) costs of the K executions of the resulting thunked computation.

The fact that values have no weight also has another advantage: We can add quantification over function variables to the type level, as in dfPCF. In other words, we do not have to parametrise typings over function variables, as in Chapter 8.

$$\frac{\phi; j/n, \Sigma; \Phi; \Gamma \vdash^{\mathsf{v}} v : A}{\phi; \Sigma; \Phi; \Gamma \vdash^{\mathsf{v}} v : \forall j/n.\, A}$$

The type inference algorithm can be trivially adapted, and polymorphism works as before.

12.3 Other applications of coeffect and effect systems

Coeffect and effect-based type systems have also been applied in domains other than complexity analysis. For example, [32] discusses a general coeffect calculus based on monoidal indexed comonads. This calculus can be instantiated, for example to track *implicit parameters* (which are similar to POSIX-like environment variables). This work is generalised in [33], where a coeffect is associated to each variable in the typing context. Thus, it is possible to bound reuse of variables (like BLL) or track liveness of variables.

Coeffect systems The type system $\ell\mathcal{R}$PCF [8] is similar to dℓPCF. It is parametrised over a semiring $\mathcal{R}$. Elements of this semiring are used to annotate exponentials. The language is also parametrised over a set of *co-handlers*. Typings can also be associated with a weight, which is used to show semantic soundness. The system can be instantiated with various concrete structures. For example, one implementation of the abstract structures can be used for complexity analysis. However, complexity analysis using $\ell\mathcal{R}$PCF is incomplete, since the system does not refine Nat-types and the fixpoint rule is only an approximation. Other applications include probabilistic analysis and bounding the number of look-ahead operations in signal processing (as discussed in [32]).

Effect systems Effect type systems are perhaps better known than coeffect type systems. For a textbook introduction, see [31, Chapter 5], which discusses, e.g., effect type systems for control flow analysis and side effects (stores with locations). Effect type systems are already used in several production-grade programming languages. For example, Java's `throws` annotation is used to track and document which exceptions a method may raise. *Monads* are used in pure languages such as Haskell to encapsulate code with side effects. Monads can be generalised to *algebraic effects*, which allow the programmer to define arbitrary effects, like catchable exceptions, non-determinism, and state. Algebraic effects and handlers have been implemented in systems like Eff [4] and Koka [26]. However, these systems cannot be used for verification and complexity analysis.

Dependent ML (DML) [39] is a type system similar in spirit to $\mathsf{d}\ell\mathsf{PCF}$. It is parametrised over a language of index terms. Thus, the expressiveness can be fine-tuned. The system has been implemented with an index term language for linear arithmetic and with length refinements for lists. Unlike $\mathsf{d}\ell\mathsf{PCF}$, DML also features *guarded types* $P \supset \tau$ and *assertion types* $P \wedge \tau$. In order to use a guarded type, one first has to prove the assertion P. Guarded types and assertion types can be used to express loop invariants. In addition to universal quantification over index terms, DML also allows existential quantification. Thus, the type $\exists a.\, a > 0 \wedge \mathsf{List}(a)$ encodes non-empty lists. However, there are no cost annotations in DML.

Combination of effects and coeffects Effects and coeffects can be combined in meaningful ways. For example, one could consider the combination of bounded variable reuse and exceptions. Effects are usually modelled using (graded) monads, and, dually, coeffects are modelled using (graded) comonads. For example, the paper [17] formalises the interaction between (graded) effects and coeffects using (graded) distributive laws. They formalise the framework using denotational semantics based on category theory.

12.4 Other approaches to verification and complexity analysis

There are also other approaches for verification and/or complexity analysis. We briefly discuss some of these below.

Program logics *Program logics* like Hoare logic [20] can be used to certify correctness of functional and imperative programs. Hoare logics are also *relatively complete* in the sense that the systems are complete if one assumes that the underlying theory admits every true theorem. There are also extensions of Hoare logic which can be used to prove that programs terminate, and even bound the number of steps that the program needs to evaluate. Various extensions of Hoare logic are well-suited for verification of 'real-life' imperative programming languages. For example, separation logics are used for imperative language that use a heap. *Iris* [23] is a generic system for higher-order separation logic that is implemented in the Coq proof assistant.

Program logics can also be used to certify low-level abstract machines, like Turing machines. For example, a relational program logic has been used to certify functional correctness and specify time and space complexity for multi-tape Turing machines in Coq [15].

Recurrence extraction and simplification The techniques for acquiring complexity bounds from programs that we have discussed in this thesis yield concrete *functions*. However, in complexity analysis, one is often interested in *asymptotic* bounds, e.g. those expressed using the $\mathcal{O}(\cdot)$ notation. *Recurrence extraction* and *simplification* are two orthogonal techniques to this end. First, we extract recurrences, e.g. using a syntactic procedure on a program. For example, this is done in [24], where an algorithm is developed that extracts recurrences from CBPV computations. Note that this is very similar in spirit to our $\mathsf{d}f\mathsf{PCF}_{\mathsf{pv}}$ annotation algorithm. After retrieving these recurrences, one can compute the complexity class. One method for this, which is taught in undergraduate computer science classes, is the *master method*.

For example, consider a functional implementation of the mergesort function, which sorts lists of even length:

```
(* msort : int list -> int list *)
fun msort [] = []
  | msort xs = let val (ys, zs) = split xs in
                 merge (msort ys) (msort zs)
               end
```

where the function `merge` merges two sorted lists of lengths m and n in $m + n$ steps, and `split` splits a list of length $2n$ into two lists of length n each in n steps. From these specifications, we can derive the following bounds on the running time of `msort`, where we assume that the pattern matching on the list and the recursive calls incur some constant additional costs:

$$\begin{aligned} f(0) &:= c_1 \\ f(2n) &:= c_2 + 2n + 2f(n) \end{aligned}$$

Using the master method, we can conclude that the running time complexity of the algorithm is $\mathcal{O}(n \log(n))$, where n is the length of the list.

It should be clear that in order to extract recurrences, we first need to verify some functional properties and the complexity of auxiliary functions. In particular, we would not have been able to bound the complexity of the mergesort function without knowing that `split` halves the length of the list and takes linear time. Thus, if one uses refinement type systems, it would have sufficed to refine the type of lists with their lengths.

Amortised complexity analysis Amortisation [37] is an advanced method for complexity analysis, which was initially used to analyse the complexity of stateful data structures. The basic idea is that instead of considering the worst case running time of operations, one considers the average cost of a sequence of operations. For example, most operations could be ‘cheap’. In addition to paying for the cheap cost, one pays ahead for the (few) later costly operations.

Resource aware ML (RaML) [22, 21] is a system for analysing polynomial worst time complexity of certain resources used by first-order Standard ML programs. The system is parametrised by a resource model, which permits analysing, e.g., heap usage or running time. The system is based on amortised analysis. Since the generated constraints are linear (although the resource bounds are polynomials), they can be automatically solved by off-the-shelf linear programming solvers.

λ-amor [35] is a coeffect-based type system that combines potentials with monadic cost effects. It subsumes an univariate version of RaML and $\mathsf{d}\ell\mathsf{PCF}_\mathsf{n}$.

In [28], the Iris framework has been used to analyse upper and lower running time bounds. This has been demonstrated for the *union-find* data structure, where lower amortised running time bounds are crucial to showing the efficiency of the data structure.

Refinement systems *Refinement* is a top-down approach to verification. One starts with an abstract specification of a system. Using several *refinement stages*, one gradually approaches a concrete implementation. Although the intermediate programs are not executable, one verifies that each stage is a refinement of the above stage. This approach has been followed in foundational systems. For example, the Isabelle Refinement Framework [25] has been used to implement a certified satisfiability solver [6].

Relational analysis The systems that we have discussed are used to verify properties about single runs of programs. Relational cost analysis [9, 2] is used to compare execution costs of different programs or different inputs. In one special case, one could show that a program has the same running time for inputs of the same size, which is a crucial property in security and privacy.

12.5 Future work

Although the call-by-name version of the $\mathsf{d}\ell\mathsf{PCF}_\mathsf{n}$ annotation algorithm has been implemented in `OCaml` [13], there are still no experimental results. One crucial component that is missing in this implementation is automated and sound simplification of the generated index terms. It was proposed in [13] to utilise the Why3 framework [7] to enable a combination of automated and manual simplification of index terms. However, this integration has not been fully implemented yet. Our Coq implementation of $\mathsf{d}\ell\mathsf{PCF}_\mathsf{v}$, which we discuss in Appendix B, could serve as an alternative starting point for a verified implementation of the type inference algorithm. Since we implement index terms using a shallow embedding, the whole power of Coq could be utilised for a combination of automated and manual simplification proofs.

Both kinds of systems, $\mathsf{d}\ell\mathsf{PCF}$ and $\mathsf{d}f\mathsf{PCF}$, could be extended with state and other effects and coeffects. For example, we could investigate whether we could combine the ideas of $\ell\mathcal{R}\mathsf{PCF}$ with $\mathsf{d}\ell\mathsf{PCF}$, and algebraic effects with $\mathsf{d}f\mathsf{PCF}$.

Bibliography

[1] Samson Abramsky. Computational interpretations of linear logic. *Theor. Comput. Sci.*, 111(1&2):3–57, 1993.

[2] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. A relational logic for higher-order programs. *J. Funct. Program.*, 29:e16, 2019.

[3] Patrick Baillot, Gilles Barthe, and Ugo Dal Lago. Implicit computational complexity of subrecursive definitions and applications to cryptographic proofs. *J. Autom. Reason.*, 63(4):813–855, 2019.

[4] Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Log. Methods Comput. Sci.*, 10(4), 2014.

[5] Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Stud Logica*, 82(1):25–49, 2006.

[6] Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. *J. Autom. Reason.*, 61(1-4):333–365, 2018.

[7] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. `https://hal.inria.fr/hal-00790310`.

[8] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 351–370. Springer, 2014.

[9] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 316–329. ACM, 2017.

[10] Loïc Colson and Daniel Fredholm. System T, Call-by-Value and the Minimum Problem. *Theor. Comput. Sci.*, 206(1-2):301–315, 1998.

[11] Ugo Dal Lago and Marco Gaboardi. Linear Dependent Types and Relative Completeness. *Logical Methods in Computer Science*, 8, 04 2011.

[12] Ugo Dal Lago and Barbara Petit. Linear Dependent Types in a Call-by-Value Scenario (Long Version). *Science of Computer Programming*, 84, 07 2012.

[13] Ugo Dal Lago and Barbara Petit. The Geometry of Types (Long Version). 10 2012.

[14] Ewen Denney. Refinement types for specification. In David Gries and Willem P. de Roever, editors, *Programming Concepts and Methods, IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods (PROCOMET '98) 8-12 June 1998, Shelter Island, New York, USA*, volume 125 of *IFIP Conference Proceedings*, pages 148–166. Chapman & Hall, 1998.

[15] Yannick Forster, Fabian Kunze, and Maximilian Wuttke. Verified programming of Turing machines in Coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 114–128. ACM, 2020.

[16] Timothy S. Freeman and Frank Pfenning. Refinement types for ML. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 268–277. ACM, 1991.

[17] Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, and Tarmo Uustalu. Combining effects and coeffects via grading. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 476–489. ACM, 2016.

[18] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[19] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: A modular approach to polynomial-time computability. *Theor. Comput. Sci.*, 97(1):1–66, 1992.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[21] Jan Hoffmann. *Types with potential: polynomial resource bounds via automatic amortized analysis.* PhD thesis, Ludwig Maximilians University Munich, 2011.

[22] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 223–236. ACM, 2010.

[23] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.

[24] G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. Recurrence extraction for functional programs through call-by-push-value. *Proc. ACM Program. Lang.*, 4(POPL):15:1–15:31, 2020.

[25] Peter Lammich. Automatic data refinement. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 84–99, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[26] Daan Leijen. Type directed compilation of row-typed algebraic effects. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 486–499. ACM, 2017.

[27] Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation.* Springer, 2004.

[28] Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time credits and time receipts in iris. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2019.

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

[30] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.

[31] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis.* Springer, 1999.

[32] Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: Unified static analysis of context-dependence. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, volume 7966 of *Lecture Notes in Computer Science*, pages 385–397. Springer, 2013.

[33] Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: A calculus of context-dependent computation. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 123–135. ACM, 2014.

[34] Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[35] Vineet Rajani, Marco Gaboardi, and Jan Hoffmann. A unifying type-theory for higher-order (amortized) cost analysis. *Proc. ACM Program. Lang.* (to appear).

[36] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: Reasoning with Multi-Sorted de Bruijn Terms and Vector Substitutions. *8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, 2019.

[37] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[38] The Coq Development Team. The Coq proof assistant, version 8.11.0, January 2020.

[39] Hongwei Xi. Dependent ML: An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.

Appendix A

dℓPCF$_\mathsf{v}$ Proofs

A.1 Completeness

A.1.1 Parametric Joining

Lemma A.1 (Subtyping and bounded sums)**.** *Let I and L be index terms, where c may be free in I but not in L. Furthermore, a and c may be free in Φ, Γ, and τ.*

Define the substitution $\theta := \{a + \sum_{d<c} L\{d/c\}/b\}$ that introduces a and c as free variables.

If $a, c, \phi; a < I, c < L, \Phi\theta \vdash A\theta \sqsubseteq B\theta$, then $b, \phi; b < \sum_{c<L} I, \Phi \vdash A \sqsubseteq B$.

Proof. We define the inverting substitution $\theta^* := \{\pi_1(f^{-1}(b))/c, \pi_2(f^{-1}(b))/a\}$ with $f^{-1} := \mathit{findSlot}_c\, L\, I$. We apply this substitution to the hypothesis and get:

$$b, \phi; \pi_2(f^{-1}(b)) < I\{\pi_1(f^{-1}(b))/c\}, \pi_1(f^{-1}(b)) < L, \Phi\theta\theta^* \vdash A\theta\theta^* \sqsubseteq B\theta\theta^*$$

From Lemma 5.39 (2), it follows that:

$$b, \phi; b < \sum_{c<L} I, \Phi \vDash \pi_2(f^{-1}(b)) < I\{\pi_1(f^{-1}(b))/c\} \wedge \pi_1(f^{-1}(b)) < L \wedge \Phi\theta\theta^*$$

Similarly, we have: $b, \phi; b < \sum_{c<L} I(a), \Phi \vdash A\theta\theta^* \equiv A$ (and the same for B). □

Lemma A.2 (Typing and bounded sums)**.** *Let θ be defined as above. From a (precise) typing $a, c, \phi; a < I, b < L, \Phi\theta; \Gamma\theta \vdash_{M\theta} t : \tau\theta$, we can derive a (precise) typing $b, \phi; b < \sum_{c<L} I, \Phi; \Gamma \vdash_M t : \tau$ (with the same skeleton).*

Proof. As above. □

We can now prove Lemma 5.44:

> Let $c, \phi; c < L, \Phi; \emptyset \vdash_M v : \rho$ be a precise typing. Then there exists a ρ' with $c, \phi; c < L, \Phi \vdash \rho \equiv \rho'$ and a precise typing $\phi; \Phi; \emptyset \vdash_{\sum_{c<L} M} v : \sum_{c<L} \rho'$ (with the same skeleton).

Proof. Case analysis on the value. Without loss of generality, we can assume that no subsumption ($\equiv$) was used.

- Case $v = \underline{n}$: trivial.
- Case $v = \lambda x.\, t$. By inverting the precise typing, we have:

  $$a, c, \phi; a < I, c < L, \Phi; x : \sigma \vdash_K t : \tau \qquad c, \phi; c < L, \Phi \vDash I + \textstyle\sum_{a<I} K = M$$

  $$\rho = [a < I] \cdot (\sigma \multimap \tau)$$

  As in Lemma 5.40, we build a ρ' and a sum $\sum_{c<L} \rho'$: Let $\theta := \{a + \sum_{d<c} L\{d/c\}/b\}$ and $\theta^* := \{\pi_1(f^{-1}(b))/c, \pi_2(f^{-1}(b))/a\}$ with $f^{-1} := \mathit{findSlot}_c\, L\, I$. Then we define:

  $$\begin{aligned} A' &:= (\sigma \multimap \tau)\theta^* = \sigma\theta^* \multimap \tau\theta^* \\ \rho' &:= [a < I] \cdot A'\theta \\ \sum_{c<L} \rho' &= [b < \sum_{c<L} I] \cdot A' \end{aligned}$$

  Furthermore, we have $c, \phi; c < L, \Phi \vdash \sigma\theta^*\theta \multimap \tau\theta^*\theta \equiv \sigma \multimap \tau$, and the same holds for K. Thus, we have:

  $$a, c, \phi; a < I, c < L, \Phi; x : \sigma\theta^*\theta \vdash_{K\theta^*\theta} t : \tau\theta^*\theta$$

  With Lemma A.2 and Lam, we can type:

  $$\frac{b, \phi; b < \sum_{b<L} I, \Phi; x : \sigma\theta^* \vdash_{K\theta^*} t : \tau\theta^*}{\phi; \Phi; \emptyset \vdash_{\sum_{c<L} I + \sum_{b<\sum_{c<I} L} K\theta^*} \lambda x.\, t : [b < \sum_{b<L} I] \cdot (\sigma\theta^* \multimap \tau\theta^*) = \sum_{c<L} \rho'}$$

  Finally, we have:

  $$\begin{aligned} \sum_{c<L} I + \sum_{b<\sum_{c<I} L} K\theta^* &\equiv \sum_{c<L} I + \sum_{b<L} \sum_{a<I} K\theta^*\theta \\ &\equiv \sum_{c<L} I + \sum_{b<L} \sum_{a<I} K \equiv \sum_{c<L} (I + \sum_{a<I} K) \equiv \sum_{c<L} M \end{aligned}$$

- Case $v = \mu f x.\, t$. Inverting of the typing yields:

  $$b, c, \phi; b < H, c < L, \Phi; f : [a < I] \cdot A \vdash_J \lambda x.\, t : [a < 1] \cdot B \tag{A.1}$$

  $$a, b, c, \phi; a < I, b < H, c < L, \Phi \vdash B\{0/a, 1 + b + H_1/b\} \equiv A \tag{A.2}$$

  $$a, \phi; a < K, \Phi \vdash B\{0/a, H_2/b\} \equiv C \tag{A.3}$$

  $$\rho = [a < K] \cdot C \qquad H_1 := \mathop{\triangle}\limits_{c}^{a} I\{1 + b + c/b\} \qquad H_2 := \mathop{\triangle}\limits_{b}^{a} I$$

  Recall that the term $1 + b + H_1$ computes the number of the a^{th} child node of the b^{th} node in the forest described by I (with $c < L$). H_2 is the size of the first $a < K$

trees in the forest $c < L$. The index term I (with $c < L$ as a free index variable) describes forests consisting of K trees and H nodes. We will join the L forests into one forest (consisting of $\sum_{c<L} K$ trees and $\sum_{c<L} H$ nodes).

We need to define two pairs of inverting substitutions: Let $f^{-1} := \mathit{findSlot}_c\, L\, K$ and $g^{-1} := \mathit{findSlot}_c\, L\, H$. For $a' < \sum_{c<L} K$ (i.e. a' is an index of a tree in the joined forest, as in the second subtyping premise for C), $f^{-1}(a')$ computes the number $c < L$ of the forest in which this tree is located, and the offset $a < K$ of this tree in that forest. For $b' < \sum_{c<L} H$ (i.e. node b' is a node in the joined forest), $g^{-1}(b')$ computes the number $c < K$ of the forest and the offset $b < H$ (i.e. b' is the b^{th} node in the c^{th} forest).

$$\theta_1 := \{a + \sum_{d<c} K\{d/c\}/a'\} \qquad \theta_1^* := \{\pi_1(f^{-1}(a'))/c, \pi_2(f^{-1}(a'))/a\}$$
$$\theta_2 := \{b + \sum_{d<c} H\{d/c\}/b'\} \qquad \theta_2^* := \{\pi_1(g^{-1}(b'))/c, \pi_2(g^{-1}(b'))/b\}$$

As in the λ case (but with a' instead of b), we construct the sum over ρ using θ_1:

$$\begin{aligned} C' &:= C\theta_1^* \\ \rho' &:= [a < K] \cdot C'\theta_1 \\ \sum_{c<L} \rho' &= [a' < \sum_{c<L} K] \cdot C' \end{aligned}$$

As before, we have $c, \phi; c < L, \Phi \vdash \rho' \equiv \rho$, which follows from $a, c, \phi; a < K, c < L, \Phi \vdash C \equiv C\theta_1^*\theta_1$.

The joined forest $I^* := I\theta_2^*$ has cardinality $H^* \equiv \triangle_{b'}^{K^*} I^* \equiv \sum_{c<L} H$ with $K^* := \sum_{c<L} K$.

We state the arguments and premises of the typing of $\mu f x.\, t$ and show the premises one-by-one:

$$I^* := I\theta_2^* \qquad K^* := \sum_{c<L} K \qquad H^* := \sum_{c<L} H \qquad H_1^* := H_1\theta_2^*$$

$$H_2^* := (H_2 + \sum_{d<c} H\{d/c\})\theta_1^*$$

$$\frac{\begin{array}{c} b', \phi; b' < H^*, \Phi; f : [a < I^*] \cdot A\theta_2^* \vdash_{J\theta_2^*} \lambda x.\, t : B\theta_2^* \\ a, b', \phi; a < I^*, b' < H^*, \Phi \vdash B\theta_2^*\{0/a, 1 + b' + H_1^*/b'\} \equiv A\theta_2^* \\ \phi; \Phi \vdash [a' < K^*] \cdot B\theta_2^*\{0/a, H_2^*/b'\} \equiv \sum_{c<L} \rho' = [a' < K^*] \cdot C' \\ a, b', \phi; a < I^*, b' < H^*, \Phi \vDash H_1^* \equiv \underset{d}{\overset{a}{\triangle}} I^*\{1 + b' + d/b'\} \\ a', \phi; a' < K^* \vdash H_2^* \equiv \underset{b'}{\overset{a'}{\triangle}} I^* \end{array}}{\phi; \Phi; \emptyset \vdash_{\sum_{b'<H^*} J\theta_2^*} \mu f x.\, t : \sum_{c<L} \rho'}$$

- The typing and first subtyping premise follow by applying the substitution $\theta_2^*\theta$ to all index terms and types in (A.1), (A.2), and then applying Lemma A.2, as in the lambda case.
- The equation for H_1^* follows by the definition of I^* and H_1.
- The equation for H_2^* is quite intuitive: In order to compute the size of the first $a' < K^*$ forests, (using θ_1^*) we first make the decomposition $a' = a + \sum_{d<c} K\{d/c\}$ with $c < J$ and $a < K$. We count the size of the first c forests (by $\sum_{d<c} H\{d/c\}$) and then add the size of the first a trees in the c^{th} forest (using H_2).
- The last subtyping is a bit more complicated. We need to show:

  $$a', \phi; a' < K^*, B\theta_2^*\{0/a, H_2^*/b'\} \equiv C' = C\theta_1^*$$

  We first apply the substitution θ_1^* to the original subtyping (A.3); by transitivity, it suffices to show:

  $$\begin{aligned}&a', \phi; a' < K^*, \Phi \vdash B\theta_2^*\{0/a, H_2^*/b'\}\\ &\qquad\overset{!}{\equiv} B\{0/a, H_2/b\}\theta_1^* = B\{0/a, H_2\theta_1^*/b, \pi_1(f^{-1}(b'))/c\}\end{aligned}$$

  It suffices to show that the two substitutions are equal (under the premise $a' < K^*, \Phi$). Therefore, we have to show that all free index variables of B (a, b, c) are replaced by the same index terms. We make a case distinction over these variables.

  - Case a: Both substitutions substitute 0 for a.
  - Case b: We need to show:

    $$\begin{aligned}a', \phi; a' < K^*, \Phi \vDash \pi_2(g^{-1}(H_2^*)) &\equiv \pi_2(g^{-1}(H_2\theta_2^* + \sum_{d<\pi_1(f^{-1}(a'))} H\{d/c\}))\\ &\equiv H_2\theta_2^*\end{aligned}$$

  - Case c: We need to show: $a', \phi; a' < K^*, \Phi \vDash \pi_1(g^{-1}(H_2^*)) \equiv \pi_1(f^{-1}(a'))$. This is similar to the above.
- Finally, we have to show that the weight is correct:

  $$\sum_{b<H^*} J\theta_2^* = \sum_{b<\sum_{c<L} H} J\theta_2^* \equiv \sum_{c<L}\sum_{b<H} J\theta_2^*\theta_2 \equiv \sum_{c<L}\sum_{b<H} J \equiv \sum_{c<L} M \qquad \square$$

A.1.2 Subject Expansion

It follows a technical lemmas that states that we can always change the order of the index variables in ϕ. It is an instance of the generic index term substitution lemma (Lemma 5.5). However, for technical reasons, we had to prove this lemmas in Coq separately, as we will explain in Appendix B.

Lemma A.3 (Swapping lemma)**.** *Let a and b be index variables and let θ be the substitution $\{a/b, b/a\}$. If $\Phi\theta; \Gamma\theta \vdash_{M\theta} t : \tau\theta \mathbin{@} s$, then $\Phi; \Gamma \vdash_M t : \tau \mathbin{@} s$.*

Proof (sketch). We first prove the converse: If $\Phi; \Gamma \vdash_M t : \tau @ s$, then $\Phi\theta; \Gamma\theta \vdash_{M\theta} t : \tau\theta @ s$. From this, the goal follows since θ is an involution. □

Lemma A.4 (Uniformisation of subtyping)**.** *Let $\phi; \Phi\{n/a\} \vdash \sigma\{n/a\} \sqsubseteq \sigma\{n/a\}$ for all constants n. Then $a, \phi; \Phi \vdash \sigma \sqsubseteq \tau$. The same holds for linear types.*

Proof. By induction on the shape of the types. □

Corollary A.5 (Introducing the constraint $a < 1$ in a subtyping)**.** *Let $\phi; \Phi\{0/a\} \vdash \sigma\{0/a\} \sqsubseteq \sigma\{0/a\}$. Then $a, \phi; a < 1, \Phi \vdash \sigma \sqsubseteq \tau$. The same holds for linear types.*

Corollary A.6 (Introducing the constraint $a < 1$ in a typing)**.** *Let $\phi; \Phi\{0/a\}; \Gamma\{0/a\} \vdash_{M\{0/a\}} t : \tau\{0/a\} \mathbin{@} s$, then $a, \phi; a < 1, \Phi; \Gamma \vdash_M t : \tau \mathbin{@} s$.*

Proof (sketch). The goal follows by induction on the given typing. Note that in the λ and fixpoint cases, new variables are introduced before a, ϕ, but (formally) the inductive hypothesis only applies if a is the first index variable in the list of index variables. This matters if we use *de Bruijn indexes* to formalise binders in index terms (as we do in our Coq implementation, see Appendix B). This problem is solved using Lemma A.3. □

We can now show the two non-trivial cases of subject expansion.

Lemma A.7 (Subject expansion (λ case))**.** *Let $\emptyset \vdash (\lambda x.\, t)\, v : \llparenthesis\tau\rrparenthesis @ s$ be a* PCF *typing and let s' be the successor skeleton of this typing. Assume a precise* dℓPCF$_\mathsf{v}$ *typing $\phi; \Phi; \emptyset \vdash_M t\{v/x\} : \tau \mathbin{@} s'$. Then we can precisely type $\phi; \Phi; \emptyset \vdash_{1+M} (\lambda x.\, t)\, v : \tau \mathbin{@} s$.*

Proof. First we invert the PCF typing and get:

$$x : \hat{\sigma} \vdash t : \hat{\tau} \mathbin{@} s_1 \qquad\qquad \emptyset \vdash v : \hat{\sigma} \mathbin{@} s_2 \qquad\qquad s = \mathsf{App}\, \hat{\sigma}\, (\mathsf{Lam}\, s_1)\, s_2$$

Thus, $s' = \mathit{subst}(x; t; s_1; s_2)$. Converse substitution (Lemma 5.45) yields:

$$\phi; \Phi; x : \sigma \vdash_{N_1} t : \tau \mathbin{@} s_1 \qquad \phi; \Phi; \emptyset \vdash_{N_2} v : \sigma \mathbin{@} s_2 \qquad \phi; \Phi \vDash N_1 + N_2 \equiv M \qquad \llparenthesis\sigma\rrparenthesis = \hat{\sigma}$$

Let a be a fresh index variable. We type $(\lambda x.\, t)\, v$ using Corollary A.6:

$$\dfrac{\dfrac{\dfrac{\phi; \Phi; x : \sigma \vdash_{N_1} t : \tau \mathbin{@} s_1}{a, \phi; a < 1, \Phi; x : \sigma \vdash_{N_1} t : \tau \mathbin{@} s_1}}{\phi; \Phi \vdash_{1+N_1} \lambda x.\, t : [a < 1] \cdot (\sigma \multimap \tau) \mathbin{@} \mathsf{Lam}\, s_1} \qquad \phi; \Phi; \emptyset \vdash_{N_2} v : \sigma = \sigma\{0/a\} \mathbin{@} s_2}{\phi; \Phi; \emptyset \vdash_{1+M \equiv 1+N_1+N_2} (\lambda x.\, t)\, v : \tau = \tau\{0/a\} \mathbin{@} s}$$

□

As we did in subject reduction, we can reduce a part of the fixpoint case to the λ case.

Lemma A.8 (Subject expansion (fixpoint case))**.** *Let $\emptyset \vdash (\mu f x.\, t)\, v : \llparenthesis\tau\rrparenthesis \mathbin{@} s$ be a* PCF *typing and let s' be the successor skeleton of this typing. Assume a precise* dℓPCF$_\mathsf{v}$ *typing $\phi; \Phi; \emptyset \vdash_M t\{\mu f x.\, t/f, v/x\} : \tau \mathbin{@} s'$. Then we can precisely type $\phi; \Phi; \emptyset \vdash_{1+M} (\mu f x.\, t)\, v : \tau \mathbin{@} s$.*

Proof. We first invert the simple typing:

$$f : \hat{\sigma} \to \hat{\tau} \vdash \lambda x.\, t : \hat{\sigma} \to \hat{\tau} \mathbin{@} \mathsf{Lam}\, s_1 \qquad \emptyset \vdash v : \hat{\sigma} \mathbin{@} s_2 \qquad s = \mathsf{App}\, \hat{\sigma}\, (\mathsf{Fix}\, (\mathsf{Lam}\, s_1))\, s_2$$

We note that the following skeleton reduces to the same target:

$$\begin{aligned}&((\lambda x.\, t\{\mu f x.\, t/f\})\, v; \mathsf{App}\, \hat{\sigma}\, (\mathit{subst}(f; \lambda x.\, t; \mathsf{Lam}\, s_1; \mathsf{Fix}\, (\mathsf{Lam}\, s_1)))\, s_2)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \succ_1 (t\{\mu f x.\, t/f, v/x\}; s')\end{aligned}$$

Thus, by the λ case (Lemma A.7), we have:

$$\phi; \Phi; \emptyset \vdash_{1+M} (\lambda x.\, t\{\mu f x.\, t/f\})\, v : \tau \mathbin{@} \mathsf{App}\, \hat{\sigma}\, (\mathit{subst}(f; \lambda x.\, t; \mathsf{Lam}\, s_1; \mathsf{Fix}\, (\mathsf{Lam}\, s_1)))\, s_2$$

After inverting this typing, we get (with a as a fresh index variable):

$$\phi; \Phi; \emptyset \vdash_N \lambda x.\, t\{\mu f x.\, t/f\} : [a < 1] \cdot (\sigma \multimap \tau) \mathbin{@} \mathit{subst}(f; \lambda x.\, t; \mathsf{Lam}\, s_1; \mathsf{Fix}\, (\mathsf{Lam}\, s_1)) \quad \text{(A.4)}$$

$$\phi; \Phi; \emptyset \vdash_{N'} v : \sigma\{0/a\} = \sigma \mathbin{@} s_2 \qquad\qquad \phi; \Phi \vDash N + N' = 1 + M$$

By applying rule App again, it suffices to show:

$$\phi; \Phi; \emptyset \vdash_N \mu f x.\, t : [a < 1] \cdot (\sigma \multimap \tau) \mathbin{@} \mathsf{Fix}\, (\mathsf{Lam}\, s_1)$$

Now we can forget everything about v (and s_2, N') and focus on the fixpoint. By applying converse substitution on (A.4), we get:

$$\phi; \Phi; f : \sigma_\mu \vdash_{M_1} \lambda x.\, t : [a < 1] \cdot (\sigma \multimap \tau) \mathbin{@} \mathsf{Lam}\, s_1 \qquad \text{(A.5)}$$

$$\phi; \Phi; \emptyset \vdash_{M_2} \mu f x.\, t : \sigma_\mu \mathbin{@} \mathsf{Fix}\, (\mathsf{Lam}\, s_1) \qquad \text{(A.6)}$$

$$\phi; \Phi \vDash N = M_1 + M_2$$

Because the goal is interesting enough, we continue the proof in the following lemma. □

Lemma A.9 (Subject expansion (fix case, part 2))**.** *Assume* (A.5) *and* (A.6)*. Then:*

$$\phi; \Phi; \emptyset \vdash_{M_1 + M_2} \mu f x.\, t : [a < 1] \cdot (\sigma \multimap \tau) \mathbin{@} \mathsf{Fix}\, (\mathsf{Lam}\, s_1)$$

Proof. The skeletons are not a complication any more; we will leave them out. The general idea of this proof is that the fixpoint typing in (A.6) already provides a recursion forest consisting of K trees. We just need to add a root node on top of this forest – the K trees are the children of this new node.

First, we invert the fixpoint typing:

$$b, \phi; b < H, \Phi; f : [a < I] \cdot A, \Delta \vdash_J \lambda x.\, t : [a < 1] \cdot B \qquad \text{(A.7)}$$

$$a, b, \phi; a < I, b < H, \Phi \vdash B\{0/a, 1 + b + \left(\overset{a}{\underset{c}{\triangle}} I\{1 + b + c/b\}\right)/b\} \sqsubseteq A \qquad \text{(A.8)}$$

$$\sigma_\mu = [a < K] \cdot C \qquad a, \phi; a < K, \Phi \vdash B\{0/a, \overset{a}{\underset{b}{\triangle}} I/b\} \equiv C \qquad \phi; \Phi \vDash H \equiv \overset{K}{\underset{b}{\triangle}} I$$

We apply rule Fix with the following arguments:

$$I^* := \mathsf{ifz}\, b\, \mathsf{then}\, K\, \mathsf{else}\, I\{1+b/b\} \qquad H^* := 1 + H \qquad K^* := 1$$

$$A^* := \mathsf{ifz}\, b\, \mathsf{then}\, C\, \mathsf{else}\, A\{1+b/b\} \qquad B^* := \mathsf{ifz}\, b\, \mathsf{then}\, \sigma \multimap \tau\, \mathsf{else}\, B\{1+b/b\}$$

We prove both the (sub)typing goals by case distinction on $b = 0$ (Lemma 5.41).

- Case $b = 0$. Follows from (A.5).
- Case $1 \leq b$. Follows by substituting $1 + b$ for b in Equations (A.7) and (A.8).

The final subtyping is trivial:

$$\phi; \Phi \vdash [a < 1] \cdot B^*\{0/a, \overset{a}{\underset{b}{\triangle}} I^*/b\} \equiv [a < 1] \cdot (\sigma \multimap \tau) \qquad \square$$

Appendix B

Coq formalisation of dℓPCF$_\mathsf{v}$

We have formalised dℓPCF$_\mathsf{v}$ in the proof assistant Coq [38]. The code can be downloaded from the following URL:

`https://gitlab.mpi-sws.org/FCS/dpcf-public-releases/`

We target Coq version `8.11`; it has not been ported to later versions of Coq. In this appendix, we outline the key points of our formalisation and discuss some of its challenges.

B.1 Preliminaries

We make extensive use the dependent type `Vector.t X n` that stands for lists of length `n`. However, Coq's standard library lacks a lot of useful definitions and lemmas about vectors. One crucial operation is *casting*: We can convert a vector `Vector.t X m` into a vector `Vector.t X n` if we can prove that `m = n`. This operation is called `cast` in Coq's standard library. We often need to reason about equalities of vectors. Since this can be complicated in the presence of man casting operations, we have implemented a tactic that reduces the goal into a corresponding goal on (non-dependent) lists.

```
(* Prepend an element to a vector. Notation: [x ::: xs] *)
Definition cons : ∀ (X : Type) (n : nat), X → Vector.t X n → Vector.t X (S n).
Definition hd : ∀ (X : Type) (n : nat), Vector.t X (S n) → X. (* head element *)
Definition tl : ∀ (X : Type) (n : nat), Vector.t X (S n) → Vector.t X n. (* tail *)
Lemma eta : ∀ (X : Type) (n : nat) (xs : Vector.t X (S n)), xs = hd xs ::: tl xs.
Definition cast : ∀ (X : Type) (m : nat), Vector.t X m →
    ∀ (n : nat), m = n → Vector.t X n.
```

B.2 Syntax and semantics of PCF

We use a *deep embedding* for the syntax of PCF. This means that we define an inductive data type for PCF terms. To formalise binders and variables, we use *de Bruijn* indexes. This means that variables are not represented by names but by natural numbers. The number denotes how many binders have to be 'skipped': For example, $\mu f x.\, \lambda y.\, f\, x\, (\mathsf{Pred}(y))$ is encoded as $\mu\lambda.\, 2\, 1\, (\mathsf{Pred}(0))$. Note that fixpoints introduce two binders.

Formally, terms of PCF are defined using the following inductive type. Values are implemented as an inductive predicate on terms.

```
Inductive tm : Type :=
| Var : nat → tm
| Lam : tm → tm
| Fix : tm → tm
| App : tm → tm → tm
| Ifz : tm → tm → tm → tm
| Const : nat → tm
| Pred : tm → tm
| Succ : tm → tm.
```

```
Inductive val : tm → Prop :=
| val_lam t : val (Lam t)
| val_fix t : val (Fix t)
| val_const k : val (Const k).
```

For example, the term $\lambda x.\,\lambda y.\,\mathsf{ifz}\,x\,\mathsf{then}\,y\,\mathsf{else}\,\underline{0}$ is encoded as `Lam (Lam (Ifz (Var 1) (Var 0) (Const 0)))`, and $\mu f x.\,\mathsf{Succ}(f\,(\mathsf{Pred}(x)))$ is written as `Fix (Succ (App (Var 1) (Pred (Var 0))))`.

To implement substitution, we initially used *Autosubst 2* [36], which is a code generator that implements a parallel substitution function and a simplification tactic. However, we later switched to *naive substitution*, which does not avoid variable capturing. This is not a problem in our setup, since we always substitute closed terms for variables, i.e. we do not “reduce under binders”.

Naive substitution is implemented as follows. `nsubst t x s` substitutes the (closed) term `s` for every occurrence of the variable with index `x` in `t`:

```
Fixpoint nsubst (t : tm) (x : nat) (s : tm) : tm :=
  match t with
  | Var y => if Nat.eq_dec x y then s else Var y
  | Lam t => Lam (nsubst t (S x) s)
  | Fix t => Fix (nsubst t (S (S x)) s)
  | App t1 t2 => App (nsubst t1 x s) (nsubst t2 x s)
  | Ifz t1 t2 t3 => Ifz (nsubst t1 x s) (nsubst t2 x s) (nsubst t3 x s)
  | Const k => Const k
  | Pred t => Pred (nsubst t x s)
  | Succ t => Succ (nsubst t x s)
  end.
```

The inductive predicate for small steps, written `t ≻^κ t'`, is parametrised by a *step kind* κ which can either be β (for β-substitution steps, i.e. cost 1) or ϵ for any other step (without an associated cost). We also define a predicate `t ≻^(k) t'` that stands for sequences of steps with exactly k β-steps. Further, we define an inductive predicate for big steps that is parametrised by the cost. The proof of the equivalence between these semantics is standard.

```
Lemma big_step_to_small_steps (t v : tm) i :
  t ⇓(i) v → t ≻^(i) v.
Lemma small_steps_to_big_step (i : nat) (t v : tm) :
  t ≻^(i) v → val v → t ⇓(i) v.
```

B.3 Index terms, constraints, and types

We use a *shallow embedding* for index terms. This means that we use Coq's dependently typed term language (also known as *Galina*) itself to define index terms and constraints. This has several advantages:

- We do not have to formalise binders for index terms, since we use Coq's binders;
- we can use automation tactics like `lia` to discharge many arithmetic goals;
- since we assume the axiom of *functional extensionality*, extensionally equal types are considered equal. For example, we have $I_1 + I_2 = I_2 + I_1$.

However, since all Coq term terminate, this means that all index terms have to be well-defined. Consequently, we do not support diverging index terms and can thus only type terminating programs.

Instead of a context ϕ of (named) index variables, we just use a natural number ϕ : `nat`. Index terms are defined as functions from vectors of length ϕ to natural numbers. Constraints are implemented analogusly, and thus the definition of entailments is trivial, since we simply use Coq's implications.

```
(* Index terms with [φ] free index variables *)
Definition idx (φ : nat)     : Set  := Vector.t nat φ → nat.
(* Constraints with [φ] free index variables *)
Definition constr (φ : nat) : Type := Vector.t nat φ → Prop.
(* Constant index term. Note that [φ] is implicit. It will be inferred automatically
   from the context where [iConst n] is used. *)
Definition iConst {φ} (n : nat) : idx φ := fun _ => n.
(* Entailment, written [sem! Φ ⊨ Ψ] *)
Definition entails {φ} (Φ : constr φ) (Ψ : Vector.t nat φ → Prop) :=
  ∀ xs : Vector.t nat φ, Φ xs → Ψ xs.
```

Types Modal and linear types with ϕ free index variables are defined by mutual inductions. All operations and lemmas on/about types are thus mutually inductive.

```
(* Linear and modal types *)
Inductive lty (φ : nat) : Type :=
| Arr (τ1 : mty φ) (τ2 : mty φ) : lty φ (* Written τ1 ⊸ τ2 *)
with mty (φ : nat) : Type :=
| Nat (i : idx φ) : mty φ
| Quant (i : idx φ) (A : lty (S φ)) : mty φ. (* Written [<i]·A *)
```

Index term substitution Index term substitution is somewhat complicated. A *substitution* is a function `f` that maps an index terms with m free variables to an index term with n free variables.

```
Fixpoint subst_lty {m n : nat} (A : lty m) (f : idx m → idx n) { struct A } : lty n :=
  match A with
  | Arr τ1 τ2 => Arr (subst_mty τ1 f) (subst_mty τ2 f)
  end
```

```
with subst_mty {m n :  nat} (τ : mty m) (f : idx m → idx n) { struct τ } : mty n :=
  match τ with
  | Nat i => Nat (f i) (* Apply [f] to the index term *)
  | Quant i A =>
    Quant (f i) (* Apply [f] to the index term *)
          (subst_lty A (* Recursively apply substitution on [A] *)
             (* Here we build a new substitution function of type
                [idx (S m) → idx (S n)] using [f : idx m → idx n].
                Note that we do not use the [i] from the input [Quant i A] any more. *)
             (fun (i' : idx (S m)) (xs : Vector.t nat (S n)) =>
                 f (fun ys : Vector.t nat m => i' (hd xs ::: ys)) (tl xs)))
  end.
```

We define several classes of substitution functions. For example, the following function is used to substitute the 0^{th} index variable with an index term `i` that depends on the other index variables:

```
Definition subst_beta_ground_fun {X: Type} {ϕ} (i : idx ϕ) :
    (Vector.t nat (S ϕ) → X) → (Vector.t nat ϕ → X) :=
  fun (f : Vector.t nat (S ϕ) → X) (xs : Vector.t nat ϕ) => f (i xs ::: xs).
```

We defined an abstract substitution function, in which we substitute the x^{th} index variable with a new index term i that introduces y additional variables:

```
Definition subst_var_beta_fun {X: Type} {ϕ} (x : Fin.t (S ϕ)) (y : nat)
           (i : idx (y + (ϕ -' fin_to_nat x))) :
           (Vector.t nat (S ϕ) → X) → (Vector.t nat (y + ϕ) → X).
```

Here `-'` is a custom variant of the substraction function in which the equality `x -' 0 = x` holds by conversion. This helps avoiding many uses of `cast`. We have also defined substitution functions that swap and clone index variables.

Forest cardinality Since forest cardinality is a partial function and since we use a shallow embedding for index terms, we define forest cardinality relationally. To this end, we use the standard technique called *fuel*. The first-order function `forestCard` returns `None` if the fuel did not suffice. Finally, we define a relation `isForestCard` and prove several lemmas about this relation.

```
(* Auxiliary function ("bind" operation for the option monad) *)
Definition bind_option {A B : Type} : (A → option B) → option A → option B :=
  fun f a => match a with | None => None | Some x => f x end.

Fixpoint forestCard (K : nat → nat) (fuel : nat) (j : nat) : option nat :=
  match j with
  | 0 => Some 0
  | S j => match fuel with
           | 0 => None
           | S fuel => bind_option
                         (fun x => bind_option
                                     (fun y => Some (S (x + y)))
                                     (forestCard (fun a => K (S (x + a))) fuel j))
```

```
                         (forestCard (fun a => K (S a)) fuel (K 0))
           end
  end.

Definition isForestCard (K : nat → nat) (j : nat) (x : nat) :=
  ∃ fuel, forestCard K fuel j = Some x.
```

B.4 dℓPCF$_{\mathsf{v}}$ typing rules

Subtyping Subtyping is defined by mutual induction on modal/linear types:

```
Inductive sublty {ϕ : nat} (Φ : constr ϕ) : lty ϕ → lty ϕ → Prop :=
| sublty_arr τ1 τ2 σ1 σ2 :
    mty! Φ ⊢ σ2 ⊑ σ1 →
    mty! Φ ⊢ τ1 ⊑ τ2 →
    lty! Φ ⊢ (σ1 ⊸ τ1) ⊑ (σ2 ⊸ τ2)
where "lty! Φ ⊢ A ⊑ B" := (sublty Φ A B)
with submty (ϕ : nat) (Φ : constr ϕ) : mty ϕ → mty ϕ → Prop :=
| submty_Nat (k1 k2 : idx ϕ) :
    (sem! Φ ⊨ (fun xs => k1 xs = k2 xs)) →
    mty! Φ ⊢ Nat k1 ⊑ Nat k2
| submty_Quant (i j : idx ϕ) (A B : lty (S ϕ)) :
    (lty! (fun xs : Vector.t X (S ϕ) => hd xs < j (tl xs) ∧ Φ (tl xs)) ⊢ A ⊑ B) →
    (sem! Φ ⊨ fun xs => j xs <= i xs) →
    mty! Φ ⊢ Quant i A ⊑ Quant j B
where "mty! Φ ⊢ A ⊑ B" := (submty Φ A B).
```

We show that subtypings are symmetric and transitive.

Contexts Unlike in our ‘on-paper’ definition of contexts (as lists of type assignments), we have defined contexts as total mappings of type `nat -> mty ϕ`, where the number stands for a de Bruijn index. The contextual definitions, like modal sums and subtyping over contexts, are parametrised over a term. They are lifted to the *free* term variables of that term. We write `ctx! t; Φ ⊢ Γ ⊑ Γ'` for context subtyping w.r.t. the term t. In particular, if `t` is closed, the context subtyping hold vacuously.

We use the notation `τ .: Γ` for the context that maps 0 to τ and `S x` to `Γ x`. Usage of the axiom of functional extensionality simplifies reasoning about contexts. For example, we can show η equality for the operation `.:`.

```
Definition ctx {ty : Type} := (nat -> ty).
Lemma scons_eta {ty : Type} (Γ : @ctx ty) : Γ = scons (Γ 0) (S >> Γ).
```

Modal sums Since we use a shallow embedding for index terms, the notion of syntactic equivalence does not apply. Therefore, we us extensional equivalence.

```
(* Binary modal sum *)
Definition msum {ϕ} (τ1 τ2 : mty ϕ) (τ : mty ϕ) : Prop :=
  match τ1, τ2, τ with
  | Nat i, Nat j, Nat k => i = j ∧ j = k
```

```
  | Quant i A, Quant j B, Quant k C =>
    A = C ∧ (* A and C are extensionally equal *)
    B = lty_shift_add C i ∧ (* B simply is A shifted by i *)
    (∀ xs, k xs = i xs + j xs)
  | _, _, _ => False (* Incompatible shapes *)
  end.

(* Bounded modal sum *)
Definition bsum_Nat {ϕ} (I : idx ϕ) (σ : mty (S ϕ)) (τ : mty ϕ) : Type :=
  { J : idx ϕ | σ = subst_mty (Nat J) (fun f xs => f (tl xs)) ∧ (τ = Nat J) } % type.

Definition bsum_Quant {ϕ} (I : idx ϕ) (σ : mty (S ϕ)) (τ : mty ϕ) : Type :=
  ∃ST J & A, (* [Type] version of [∃ J A, ...] *)
    σ = [<J] · (subst_lty_beta_two (* Implementation of the shift {b + ∑_{d<a}J{d/c}} *)
                  A
                  (fun xs => let b := hd xs in let a := hd (tl xs) in
                             let xs' := tl (tl xs) in b + ∑_{d<a} (J (d ::: xs')))) ∧
    (τ = [< fun xs => ∑_{a < I xs} J (a ::: xs)] · A).

Definition bsum {ϕ} (I : idx ϕ) (σ : mty (S ϕ)) (τ : mty ϕ) : Type :=
  bsum_Nat I σ τ + bsum_Quant I σ τ. (* [Type] version of ∨ *)
```

Bounded sums are defined in the universe `Type` instead of `Prop` for technical reasons.

(Sub)typing and substitution The following lemmas could be proved using an axiom that exploits the parametricity of `f`. Here, `f` acts both as a substitution for index terms (`X := nat`) and constraints (`X := Prop`).

```
Lemma sublty_subst {ϕ1 ϕ2} (f : ∀ {X}, (Vector.t nat ϕ1 → X) → (Vector.t nat ϕ2 → X))
      (Φ : constr ϕ1) (A B : lty ϕ1) :
  (lty! Φ ⊢ A ⊑ B) → (lty! f Φ ⊢ subst_lty A f ⊑ subst_lty B f)
with submty_subst {ϕ1 ϕ2} (f : ∀ {X}, (Vector.t nat ϕ1 → X) → (Vector.t nat ϕ2 → X))
      (Φ : constr ϕ1) (σ τ : mty ϕ1) :
  (mty! Φ ⊢ σ ⊑ τ) → (mty! f Φ ⊢ subst_mty σ f ⊑ subst_mty τ f).
Abort. (* Not proved, but these lemmas would follow from the axiom below. *)
```

Instead of relying on an axiom, we have shown instances of these lemmas (and the corresponding typing lemma) for several instances of `f`. In particular, we derive generic substitution lemmas using the function `subst_var_beta_fun` and derive from this instances for concrete values of x and y. We also prove substitution lemmas for swapping and cloning of index variables. This resulted in a lot of boilerplate code; perhaps using the axiom would have been a more elegant solution.

```
Axiom parametricity :
  ∀ ϕ1 ϕ2 (f : ∀ {X}, ((Vector.t nat ϕ1) → X) → ((Vector.t nat ϕ2) → X)),
  ∃ g : (Vector.t nat ϕ2) → (Vector.t nat ϕ1),
    ∀ (X : Type) (i : Vector.t nat ϕ1 → X) (ys : Vector.t nat ϕ2),
      f i ys = i (g ys).
```

Typing rules We define the typing rules as an inductive predicate and declare a notation. Although the rules are technical, all of them, except the fixpoint rule, are straightforward translations of the 'on paper' typing rules. In the fixpoint case, we assume auxiliary index terms for the forest cardinalities, using the relation `isForestCard`. We use a variant of the typing rules where subsumption is 'built into' every rule; subsumption is proved as a lemma. This simplifies inversion of typings, since we can simply use the tactic `inversion`. We give explicit names to the premises, so that `inversion` automatically takes these names. With an explicit subsumption rule, we would have to (inductively) prove inversion lemmas for each rule. We only show some of the typing rules in the listing below.

```
(* Written [ty! Φ; Γ ⊢(i) t : τ]. Note that [ϕ] is implicit. *)
Inductive hasty {ϕ} (Φ : constr ϕ) (Γ : ctx ϕ) (M : idx ϕ) : tm → mty ϕ → Prop :=
| ty_Var x ρ :
    (* Subtyping *)
    ∀ (Hsub: mty! Φ ⊢ Γ x ⊑ ρ),
      (ty! Φ; Γ ⊢(M) (Var x) : ρ)
| ty_Lam (t : tm) (I : idx ϕ) (Δ : ctx (S ϕ)) (σ τ : mty (S ϕ)) (K : idx (S ϕ))
         (ρ : mty ϕ) (* The type after subtyping *)
         (Γ' : ctx ϕ) : (* The context sum of [Δ], before subtyping *)
    ∀ (Hty: ty! (fun xs => hd xs < I (tl xs) ∧ Φ (tl xs)); (σ .: Δ) ⊢(K) : t τ)
      (Hbsum: ctxBSum (Lam t) I Δ Γ') (* Γ' = ∑_{a<I}Δ *)
      (* Subtyping (The variable x is excluded from the subtyping.) *)
      (HΓ: ctx! (Lam t); Φ ⊢ Γ ⊑ Γ')
      (HM: sem! Φ ⊨ fun xs => (I xs + ∑_{a<I xs} K (a ::: xs)) <= M xs)
      (Hρ: mty! Φ ⊢ [<I] · σ ⊸ τ ⊑ ρ),
      (ty! Φ; Γ ⊢(M) Lam t : ρ)
| ty_App (t1 t2 : tm) (Δ1 Δ2 : ctx ϕ) (σ τ : mty (S ϕ)) (K1 K2 : idx ϕ)
         (Γ' : ctx ϕ)  (* The context sum before subtyping *)
         (ρ : mty ϕ) : (* The type after subtyping *)
    ∀ (Hty1: ty! Φ; Δ1 ⊢(K1) t1 : [<iConst 1] · (σ ⊸ τ))
      (Hty2: ty! Φ; Δ2 ⊢(K2) t2 : (subst_mty_beta_ground σ (iConst 0)))
      (Hmsum: ctxMSum (App t1 t2) Δ1 Δ2 Γ') (* Γ' = Δ1 ⊎ Δ2 *)
      (* Subtyping *)
      (HΓ: ctx! (App t1 t2); Φ ⊢ Γ ⊑ Γ')
      (HM: sem! Φ ⊨ fun xs => (K1 xs + K2 xs) <= M xs)
      (Hρ: mty! Φ ⊢ subst_mty_beta_ground τ (iConst 0) ⊑ ρ),
      (ty! Φ; Γ ⊢(M) t1 t2 : ρ)
(* ... *)
```

B.5 Soundness

We prove soundness in the same way as in the paper. For the same reason as for subtyping, we could not derive a generic index term substitution lemma. Therefore, we prove substitution lemmas for the same classes of substitutions. The following are the key lemmas:

```
(* [Γ] is like [Γ'] (for all [m] free variables except [x]), but it has [x : σ] *)
Definition ctxExtends {ϕ} (Φ : constr ϕ) (Γ : ctx ϕ) (m : nat) (x : nat) (σ : mty ϕ)
    (Γ' : ctx ϕ) : Prop :=
  (∀ y, y < m → x ≠ y → mty! Φ ⊢ Γ' y ⊑ Γ y) ∧ Γ x = σ.

(* Value substitution lemma *)
Lemma typepres_nsubst {ϕ} (m : nat) (Φ : constr ϕ) (x : nat) (Γ Γ' Σ : ctx ϕ)
      (σ : mty ϕ) t τ v (M N : idx ϕ) :
      (∀ xs, { Φ xs } + { ¬ Φ xs }) → (* [Φ] is decidable for all valuations *)
```

```
  hasty Φ Γ M t τ →
  bound m t → (* This means that [t] has no more than [m] free variables *)
  (ty! Φ; Σ ⊢(N) v : σ) → (* Σ is an arbitrary context, i.e. "∅" *)
  val v → closed v →
  ctxExtends Φ Γ m x σ Γ' →
  ∃ (K : idx ϕ),
    hasty Φ Γ' K (nsubst t x v) τ ∧
    sem! Φ ⊨ fun xs => K xs <= M xs + N xs.

(* After a beta substitution, the weight decreases by at least one. *)
Lemma preservation_beta {ϕ} (Φ : constr ϕ) (Γ : ctx ϕ) M t τ t' :
  (∀ xs, { Φ xs } + { ¬ Φ xs }) →
  hasty Φ Γ M t τ →
  closed t →
  t ≻(β) t' →
  ∃ N, hasty Φ Γ N t' τ ∧
  sem! Φ ⊨ fun xs => N xs < M xs.

(* After a nat computation, the cost doesn't decrease (but the term size) *)
Lemma preservation_nat {ϕ} (Φ : constr ϕ) (Γ : ctx ϕ) M t τ t' :
  hasty Φ Γ M t τ →
  closed t →
  t ≻(ϵ) t' →
  hasty Φ Γ M t' τ.
```

These lemmas are proved by induction on the typing and inversion on the step. Throughout the subject reduction proof, we need to assume that all valuations of the constraint Φ are decidable. We need this for technical reasons when we want to 'split' a forest cardinality $\triangle_b^{I_1+I_2} K$ into two parts, as in Fact 5.2. Ultimately, we instantiate `Φ := fun xs : Vector.t nat 0 => True` (for `ϕ := 0`), which is obviously decidable. The fixpoint subject reduction case is the most complicated lemma; we have outlined the proof in Lemma 5.11. From subject reduction, we easily derive a normalisation theorem.

```
Theorem normalisation (Γ : ctx 0) (M : idx 0) (t : tm) (τ : mty 0) :
  (ty! (fun xs => True); Γ ⊢(M) t : τ) →
  closed t → normalising t.

Theorem cost_soundness (Γ : ctx 0) :
  ∀ t v (i : nat) (N : idx 0) (τ : mty 0),
    t ⇓(i) v → closed t →
    (ty! (fun xs => True); Γ ⊢(N) t : τ) →
    i <= N [||]. (* [||] is the empty vector. N [||] simply evaluates the index term *)
```

The proposition `normalising t` is defined as the wellfoundedness of the step relation. Together with the progress lemma for simple typings (Lemma 2.8), the first theorem entails that every well-typed term terminates. The second theorem (which is also a corollary of subject reduction) bounds the number of steps.

B.6 Completeness

The simple PCF typing rules are defined in the universe `Prop`. However, we need to distinguish simple typings by their skeletons. Thus, since *proof irrelevance* is consistent with the logic of Coq, we have also defined a `Type` variant of the simple typing rules, together with a function `PCF.strip` that returns the skeleton of the simple typing. (Alternatively, we could have defined a predicate `Γ ⊢ t : A @ s` for simple typings.) We have shown that typings that have the same skeleton are unique (Fact 5.25), but this fact is not needed.

Precise dℓPCF$_\mathsf{v}$ typings are defined as a separate inductive predicate. For technical reason (with future work in mind), we defined the predicate in the universe `Type`. We write `Ty! Φ; Γ ⊢(i) t : τ @ s` for a precise typing with skeleton `s`. The proof scripts of the index term substitution lemmas have been copy-pasted for precise typings.

The outline for the completeness proof is exactly as we have explained in Section 5.5. The following are the key lemmas:

```
(** Increase the cost [i] if [κ = β], else return [i] *)
Definition costAfter (i : nat) (κ : stepKind) : nat :=
  match κ with | β => S i | ϵ => i end.

Theorem subject_expansion {ϕ} (Φ : constr ϕ) (Γ : ctx ϕ) (M : idx ϕ)
      (t t' : tm) (κ : stepKind) (s s' : skel)
      (ρ : mty ϕ) (ρ_PCF : PCF.ty)
      (pcfTy : PCF.hastyT (stripCtx Γ) t ρ_PCF) :
  (Ty! Φ; Γ ⊢(M) t' : ρ @ s') →
  stepT t κ t' →                          (* Variant of [t ≻^κ t'] defined in [Type] *)
  ρ_PCF = mty_strip ρ →                   (* the shape of ρ *)
  s = PCF.strip pcfTy →                   (* The skeleton of the simple typing [pcfTy] *)
  s' = PCF.skel_red t s →                 (* The successor skeleton *)
  closed t →
  ∃S (N : idx ϕ),
    (Ty! Φ; Γ ⊢(N) t : ρ @ s) **          (* [**] is the [Type] variant of [∧] *)
    (sem! Φ ⊨ fun xs => N xs = costAfter (M xs) κ).

Theorem completeness_for_values {ϕ} (Φ : constr ϕ) (Γ : ctx ϕ)
      (t v : tm) (k i : nat) (ρ_PCF : PCF.ty)
      (pcfTy : PCF.hastyT (stripCtx Γ) t ρ_PCF) :
  starBT' k i t v → (* k steps in total, of which i β steps *)
  closed t → val v →
  ∃S (ρ : mty ϕ),
    (Ty! Φ; Γ ⊢(iConst i) t : ρ @ strip pcfTy) **
    (Ty! Φ; Γ ⊢(iConst 0) v : ρ @ skel_reds t (strip pcfTy) k) **
    mty_strip ρ = ρ_PCF.

Corollary completeness_for_programs {ϕ} (Φ : constr ϕ) (Γ : ctx ϕ)
      (t : tm) (n : nat) (k i : nat) (ρ_PCF : PCF.ty)
      (pcfTy : PCF.hastyT (stripCtx Γ) t ρ_PCF) :
  starBT' k i t (Const n) → closed t →
  Ty! Φ; Γ ⊢(iConst i) t : Nat (iConst n) @ strip pcfTy.

(** Completeness of the non-precise version *)
Corollary completeness_for_programs' {ϕ} (Φ : constr ϕ) (Γ : ctx ϕ) (M : idx ϕ) (t : tm)
```

```
    (n : nat) (i : nat) (ρ_PCF : PCF.ty) :
PCF.hastyT (stripCtx Γ) t ρ_PCF → t ≻^(i) (Const n) → closed t →
ty! Φ; Γ ⊢(iConst i) t : Nat (iConst n).
```

B.7 Statistics

The overall lines of code (counted with the tool `coqwc` that is part of the Coq distribution) are shown in Table B.1. In total, one person has been working on the proofs (roughly) between March and July of 2020.

We have used the following axioms. Functional extensionality is technically not needed, but it simplified the development. The axiom `JMeq.JMeq_eq` was automatically used in tactic for dependent inversion. We could probably also have removed this axiom.

```
Print Assumptions completeness_for_programs'.
(* - FunctionalExtensionality.functional_extensionality_dep *)
(* - JMeq.JMeq_eq *)
```

During the formalisation, many technical details had to be considered. Reasoning about forest cardinalities was particularly difficult; even the on-paper proofs are very tedious. The most annoying thing was that we could not derive a general substitution lemma for (sub-)typings. We have only realised after completing the formalisation that we could have derived these lemmas using the parametricity axiom mentioned above. Using this axiom would have saved circa one thousand lines of technical boilerplate code.

B.8 Future mechanisation opportunities

As mentioned earlier, it would probably have been easier to formalise the call-by-push-value variant of dℓPCF, which we developed after finishing the Coq mechanisation. Proving the splitting and joining lemmas would have been considerably easier, and we would have mechanised more general results.

We have started but not completed a proof of the *uniformisation lemma* (Lemma 5.55). The corresponding subtyping lemma has been proved (as a variant of this has also been used in the completeness proof).

	spec	proof	comments
Init	425	314	57
PCF	1327	1288	167
dlPCF	2190	3138	492
Soundness	426	1650	171
Completeness	1673	4530	395
Total	6041	10920	1282

Table B.1: Lines of code