Linear dependent types in a call-by-value scenario

Ugo Dal Lago, Barbara Petit

Linear Dependent Types in a Call-by-Value Scenario

Ugo Dal Lago

INRIA Sophia Antipolis & University of Bologna

Barbara Petit

INRIA Rhone Alpes

Abstract

Linear dependent types were introduced recently [1] as a formal system that allows to precisely capture both the extensional behavior and the time complexity of λ-terms, when the latter are evaluated by Krivine's abstract machine. In this work, we show that the same paradigm can be applied to call-by-value computation. A system of linear dependent types for Plotkin's PCF is introduced, called $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, whose types reflect the complexity of evaluating terms in the CEK machine. $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ is proved to be sound, but also relatively complete: every true statement about the extensional and intentional behaviour of terms can be derived, provided all true index term inequalities can be used as assumptions.

Keywords: Complexity Analysis, Implicit Computational Complexity, Type Systems, Linear Logic

1. Introduction

Program verification is one of the most challenging activities in computer science, due to the fundamental limitations recursion theory and complexity

Email addresses: dallago@cs.unibo.it (Ugo Dal Lago), barbara.petit@inria.fr (Barbara Petit)

This is a revised and extended version of a paper which appeared in the proceedings of PPDP 2012

theory pose on it. A variety of methodologies for formally verifying properties of programs have been introduced in the last fifty years. Among them, *type systems* have certain peculiarities. On the one hand, the way type systems are defined makes the task of proving a given program to have a type simple and modular: a type derivation for a compound program usually consists of some type derivations for the components, appropriately glued together in a syntax-directed way (*i.e.*, attributing a type to a program can usually be done *compositionally*). On the other, the kind of specifications that can be expressed through types have traditionally been weak, although stronger properties have recently become of interest, including security properties [2, 3], termination [4], monadic temporal properties [5] or resource bounds [6, 7]. But contrarily to what happens with other formal methods (*e.g.* model checking or program logics), giving a type to a program t is a *sound* but *incomplete* way to prove t to satisfy a specification: there are correct programs which cannot be proved such by way of typing.

In other words, the tension between expressiveness and tractability is particularly evident in the field of type systems, where certain good properties the majority of type systems enjoy (*e.g.* syntax-directedness) are usually considered desirable (if not necessary), but also have their drawbacks: some specifications are intrinsically hard to verify locally and compositionally. One specific research field in which the just-described scenario manifests itself is complexity analysis, in which specifications are concrete or asymptotic bounds on the complexity of the underlying program. Many type systems have been introduced capturing, for instance, the class of polynomial time computable functions [8, 9, 10]. All of them, under mild assumptions, can be employed as tools to certify programs as asymptotically time efficient. However, a tiny slice of the polytime *programs* are generally typable, since the underlying complexity class **FP** is only characterized in a purely extensional sense — for every function in **FP** there is *at least one* typable program computing it.

Gaboardi and the first author have recently introduced [1] a type system for Plotkin's PCF, called $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{N}}$, in which linearity and a restricted form of dependency in the spirit of Xi's DML [11] are present:

- **Linearity** makes it possible to precisely control the number of times subterms are copied during the evaluation of a term t. This number is actually a parameter which accurately reflects the time complexity of evaluating t [12].
- **Dependency** allows to type distinct (virtual) copies of a term with dis-

tinct types. This gives the type system an extra flexibility similar to that of intersection types [13, 14].

When mixed together, these two ingredients allow to precisely capture the extensional behavior of λ-terms *and* the time complexity of their evaluation by the Krivine's Abstract Machine [15] (KAM). Both soundness and relative completeness hold for $\mathsf{d}\ell\mathsf{PCF}_\mathsf{N}$. Noticeably, this not only holds for terms of base type, but also for first-order functions.

One may argue, however, that the practical relevance of these results is quite limited, given that call-by-name evaluation and KAM are very inefficient: why would one be interested in verifying the complexity of evaluating concrete programs in such a setting?

In this work, we show that linear dependent types can also be applied to the analysis of *call-by-value* evaluation of functional programs. This is done by introducing another system of linear dependent types for Plotkin's PCF. The system, called $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$, captures the complexity of evaluating terms by Felleisen and Friedman's CEK machine [16], a simple abstract machine for call-by-value evaluation. $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$ is proved to have the same good properties enjoyed by its sibling $\mathsf{d}\ell\mathsf{PCF}_\mathsf{N}$, namely soundness and relative completeness: every true statement about the intensional (and extensional) behavior of terms can be derived, provided all true index term inequalities can be used as assumptions.

Actually, $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$ is not merely a variation on $\mathsf{d}\ell\mathsf{PCF}_\mathsf{N}$: not only typing rules are different, but also the language of types itself must be modified. Roughly, $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$ and $\mathsf{d}\ell\mathsf{PCF}_\mathsf{N}$ can be thought as being induced by two translations of intuitionistic logic into linear logic: the latter corresponds to Girard's translation $A \Rightarrow B \equiv !A \multimap B$, while the former corresponds to $A \Rightarrow B \equiv !(A \multimap B)$. The strong link between translations of IL into ILL and notions of reduction for the λ-calculus is well-known (see *e.g.* [17]) and has been a guide in the design of $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$ (this is explained in Section. 2.2). Overall, $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$ is arguably simpler than $\mathsf{d}\ell\mathsf{PCF}_\mathsf{N}$: base types need not be annotated with the ! operator.

2. Linear Dependent Types, Intuitively

Consider the following program:

$$\mathsf{dbl} = \ \mathtt{fix}\ f.\lambda x.\ \mathtt{ifz}\ x\ \mathtt{then}\ x\ \mathtt{else}\ \mathtt{s}(\mathtt{s}(f(\mathtt{p}(x)))).$$

In a type system like PCF [18], the term dbl receives type Nat ⇒ Nat. As a consequence, dbl computes a function on natural numbers without "going wrong": it takes in input a natural number, and (possibly) produces in output another natural number. The type Nat ⇒ Nat, however, does not give any information about *which* specific function on the natural numbers dbl computes.

Properties of programs which are completely ignored by ordinary type systems are termination and its most natural refinement, namely termination *within appropriate time bounds* (*e.g.*, polynomial bounds). Typing a term t with Nat ⇒ Nat does not guarantee that t, when applied to a natural number, terminates. Consider, as another example, a slight modification of dbl, namely

$$\mathsf{div} = \ \mathtt{fix}\ f.\lambda x.\ \mathtt{ifz}\ x\ \mathtt{then}\ x\ \mathtt{else}\ \mathtt{s}(\mathtt{s}(f(x))).$$

It behaves as dbl when fed with 0, but it diverges when it receives a strictly positive natural number as argument. But look: div is not so different from dbl. Indeed, the latter can be obtained from the former by feeding not x but $\mathtt{p}(x)$ to the "recursive call" f. And any type system in which dbl and div are somehow recognized as being fundamentally different must be able to detect the presence of p in dbl and deduce termination from it. Indeed, sized types [4] and dependent types [11] are able to do so. Going further, we could ask the type system to be able not only to guarantee termination, but also to somehow evaluate the time or space consumption of programs. For example we could be interested in knowing that, on any natural number, dbl takes a polynomial number of steps to be evaluated in a given machine, and actually some type systems able to control the complexity of higher-order programs exist. Good examples are type systems for amortized analysis [7, 19] or those using ideas from linear logic [9, 10]: in all of them, linearity plays a key role.

$\mathsf{d}\ell\mathsf{PCF}_\mathsf{N}$ [1] combines some of the ideas presented above with the principles of bounded linear logic (BLL [20]): the cost of evaluating a term is measured by counting how many times function arguments need to be copied during evaluation, and different copies can be given distinct, although uniform, types. Making this information explicit in types permits to compute the cost step by step during the type derivation process. Roughly, typing judgments in $\mathsf{d}\ell\mathsf{PCF}_\mathsf{N}$ are statements like

$$\vdash_\mathrm{J} t\ : !_\mathrm{n}\ \mathsf{Nat}[a] \multimap \mathsf{Nat}[\mathrm{I}],$$

where I and J depend on a and n is a natural number capturing the number of times t uses its argument. But this is not sufficient: analogously to what happens in BLL, $\mathsf{d\ell PCF_N}$ makes types more parametric. A type like $!_{\mathrm{n}}\,\sigma \multimap \tau$ is replaced by the more accurate $!_{a<\mathrm{n}}\sigma \multimap \tau$, which tells us that the argument will be used n times, and each instance has type σ *where, however,* the variable a is substituted by a value less than n. This allows to type each copy of the argument differently but uniformly, since all instances of σ have the same PCF skeleton. This form of *uniform linear dependence* is actually crucial in obtaining the result which makes $\mathsf{d\ell PCF_N}$ different from similar type systems, namely completeness. As an example, dbl can be typed as follows in $\mathsf{d\ell PCF_N}$:

$$\vdash_a^{\mathcal{E}} \mathsf{dbl} : !_{b<a+1}\mathsf{Nat}[a] \multimap \mathsf{Nat}[2 \times a].$$

This tells us that the argument will be used $a+1$ times by dbl, and that the cost of evaluation will be itself proportional to a.

2.1. Why Another Type System?

The theory of λ-calculus is full of interesting results, one of them being the so-called Church-Rösser property: both β and $\beta\eta$ reduction are confluent, *i.e.*, if you fire two distinct redexes in a λ-term, you can always "close the diagram" by performing one *or more* rewriting steps. This, however, is only a *local* confluence result, and as such does *not* imply that all reduction strategies are computationally equivalent. Indeed, some of them are normalizing (like normal-order evaluation) while some others are not (like innermost reduction). But how about efficiency?

On the one hand, it is well known that optimal reduction *is* indeed possible [21], even if it gives rise to high overheads [22]. On the other, call-by-name evaluation (CBN) may be highly inefficient. Consider, as an example, the composition of dbl with itself:

$$\mathsf{dbl2} = \lambda x.\mathsf{dbl}(\mathsf{dbl}\ x).$$

The term $\mathsf{dbl2}\ \underline{\mathtt{n}}$ takes quadratic time to be evaluated by the KAM, since the evaluation of $\mathsf{dbl}\ \underline{\mathtt{n}}$ is repeated a linear number of times, whenever it reaches the head position. This actually *can* be seen from within $\mathsf{d\ell PCF_N}$, since

$$\vdash_{\mathrm{J}}^{\mathcal{E}} \mathsf{dbl2} : !_{b<\mathrm{I}}\mathsf{Nat}[a] \multimap \mathsf{Nat}[4 \times a],$$

where both I and J are quadratic in a (more precisely, $\mathrm{I} = 2a^2 + 3a + 1$ and $\mathrm{J} = 4a^2 + 8a + 2$). Call-by-value (CBV) solves this problem, at the price of not being normalizing. Indeed, eager evaluation of $\mathsf{dbl2}$ when fed with a natural number n takes linear time in n. Besides giving examples like the one above, little can be said about the relative efficiency of call-by-value and call-by-name evaluation: the two are clearly incomparable from an analytical point of view (consider, as an another example, the term $(\lambda x.\lambda y.x)\Omega$). A nice informal discussion about the advantages and disadvantages of call-by-value and call-by-name evaluation can be found in [23], together with a comparison with Levy's notion of optimal reduction.

The reason why a step forward $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{N}}$ is needed is not *necessarily* related to efficiency: simply, in many modern functional programming languages (like OCaml and Scheme) terms are call-by-value evaluated. And on the other hand, upper bounds on the time complexity of programs written in functional languages (which can be seen as the long term goal of this line of research) should reflect the underlying evaluation strategy, which is an essential part of their *definition*, and not merely an *implementation* detail.

For the reasons above, we strongly believe that designing a type system in the style of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{N}}$, but able to deal with eager evaluation, is a step forward applying linear dependent types to ML-like programming languages.

2.2. Call-by-Value, Call-by-Name and Linear Logic

Various notions of evaluation for the λ-calculus can be seen as translations of intuitionistic logic (or simply-typed λ-calculi) into Girard's linear logic. This correspondence has been investigated in the specific cases of call-by-name, call-by-value, and call-by-need reduction (*e.g.* see the work of Maraist et al. [17]). In this section, we briefly introduce the main ideas behind the correspondence, explaining why linear logic has guided the design of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$.

The general principle in such translations, is to guarantee that whenever a term *can* possibly be duplicated, it must be mapped to a box in the underlying linear logic proof. In the CBN translation (also called Girard's translation), *any* argument to functions can possibly be substituted for a variable and copied, so *arguments* are banged during the translation:

$$(A \Rightarrow B)^* \;=\; (!A^*) \multimap B^*.$$

Adding the quantitative bound on banged types (as explained in the previous section) gives rise to the type $(!_{a<\mathrm{I}}\sigma) \multimap \tau$ for functions (written $[a < \mathrm{I}]\cdot\sigma \multimap$

τ in [1]). In the same way, *contexts* are banged in the CBN translation: a typing judgment in $\mathsf{d}\ell\mathsf{PCF}_\mathsf{N}$ has the following form:

$$x_1 : !_{a_1<\mathrm{I}_1}\sigma_1, \ldots, x_n : !_{a_n<\mathrm{I}_n}\sigma_n \ \vdash_\mathrm{J} \ t : \tau.$$

In the CBV translation, β-reduction should be performed only if the argument is a value. Thus, arguments are not automatically banged during the translation but values are, so that β-reduction remains blocked until the argument reduces to a value. In the λ-calculus functions are values, hence the translation of the intuitionistic arrow becomes

$$(A \Rightarrow B)^\circ \ = \ !(A^\circ \multimap B^\circ).$$

Accordingly, function types have the form $!_{a<\mathrm{I}}(\sigma \multimap \tau)$ in $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$, and a judgment has the form $x_1 : \sigma_1, \ldots, x_n : \sigma_n \ \vdash_\mathrm{J} \ t : \tau$. The syntax of types varies fairly much between $\mathsf{d}\ell\mathsf{PCF}_\mathsf{N}$ and $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$, and consequently the two type systems are different, although both of them are greatly inspired by linear logic.

In both $\mathsf{d}\ell\mathsf{PCF}_\mathsf{N}$ and $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$, however, the "target" of the translation is not the whole of ILL, but rather a restricted version of it, namely BLL, in which the complexity of normalization is kept under control by shifting from unbounded, infinitary, exponentials to finitary ones. For example, the BLL *contraction* rule allows to merge the first I copies of A, and the following J ones into the first $\mathrm{I} + \mathrm{J}$ copies of A:

$$\frac{\Gamma, !_{a<\mathrm{I}}A, !_{a<\mathrm{J}}A\{\mathrm{I} + a/a\} \vdash B}{\Gamma, !_{a<\mathrm{I}+\mathrm{J}}A \vdash B}$$

We write $\sigma \uplus \tau = !_{a<\mathrm{I}+\mathrm{J}}A$ if $\sigma = !_{a<\mathrm{I}}A$ and $\tau = !_{a<\mathrm{J}}A\{\mathrm{I} + a/a\}$. Whenever a contraction rule is involved in the CBV translation of a type derivation, a sum $\uplus$ appears at the same place in the corresponding $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$ derivation. Similarly, the *dereliction* rule allows to see any banged type as the first copy of itself:

$$\frac{\Gamma, A\{\mathsf{0}/a\} \vdash B}{\Gamma, !_{a<\mathsf{1}}A \vdash B}$$

hence any dereliction rule appearing in the translation of a typing judgment tells us that the corresponding type is copied once. Both contraction and dereliction appear while typing an application in $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$: the PCF typing rule

$$\frac{\Gamma \ \vdash \ t : \ A \Rightarrow B \quad \Gamma \ \vdash \ u : \ A}{\Gamma \ \vdash \ tu : \ B}$$

corresponds to the ILL proof

$$\dfrac{\dfrac{\dfrac{z\colon\ A^\circ \multimap B^\circ \vdash z\colon\ A^\circ \multimap B^\circ}{!z\colon\ !(A^\circ \multimap B^\circ) \vdash z\colon\ A^\circ \multimap B^\circ} \quad \Gamma^\circ \vdash t^\circ\colon\ !(A^\circ \multimap B^\circ)}{\Gamma^\circ \vdash t^\circ\colon\ A^\circ \multimap B^\circ} \qquad \Gamma^\circ \vdash u^\circ\colon\ A^\circ}{\dfrac{\Gamma^\circ,\ \Gamma^\circ \vdash t^\circ u^\circ\colon\ B^\circ}{\Gamma^\circ \vdash t^\circ u^\circ\colon\ B^\circ}}$$

which becomes the following, when appropriately decorated according to the principles of BLL (writing A_0 and B_0 for $A\{0/a\}$ and $B\{0/a\}$, respectively):

$$\dfrac{\dfrac{\dfrac{z\colon\ A_0^\circ \multimap B_0^\circ \vdash z\colon\ A_0^\circ \multimap B_0^\circ}{!z\colon\ !_{a<1}(A^\circ \multimap B^\circ) \vdash z\colon\ A_0^\circ \multimap B_0^\circ} \ \Gamma^\circ \vdash t^\circ\colon\ !_{a<1}(A^\circ \multimap B^\circ)}{\Gamma^\circ \vdash t^\circ\colon\ A_0^\circ \multimap B_0^\circ} \qquad \Gamma^\circ \vdash u^\circ\colon\ A_0^\circ}{\dfrac{\Gamma^\circ,\ \Gamma^\circ \vdash t^\circ u^\circ\colon\ B_0^\circ}{\Gamma^\circ \uplus \Gamma^\circ \vdash t^\circ u^\circ\colon\ B_0^\circ}}$$

The CBV translation of the application rule hence leads to the typing rule for applications in $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$:

$$\dfrac{\Gamma \vdash_{\mathrm{K}} t :!_{a<1}(\sigma \multimap \tau) \quad \Delta \vdash_{\mathrm{H}} u : \sigma\{0/a\}}{\Gamma \uplus \Delta \vdash_{\mathrm{K+H}} tu : \tau\{0/a\}}$$

The same kind of analysis allows to derive the typing rule for abstractions (whose call-by-value translation requires the use of a promotion rule) in $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$:

$$\dfrac{\Gamma, x : \sigma \vdash_{\mathrm{K}} t : \tau}{\sum_{a<\mathrm{I}} \Gamma \vdash_{\mathrm{I}+\sum_{a<\mathrm{I}}\mathrm{K}} \lambda x.t :!_{a<\mathrm{I}}(\sigma \multimap \tau)}$$

One may wonder what the term I represents in this typing rule, and more generally in a judgment such as $\Gamma \vdash_{\mathrm{K}} t :\ !_{a<\mathrm{I}}A$. This is actually the main new idea of $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$: such a judgment intuitively means that the value to which t reduces will be used I times *by the environment*. If t is applied to an argument u, then t must reduce to an abstraction $\lambda x.s$, that is used by the argument without being duplicated. In that case, $\mathrm{I} = 1$, as indicated by the application typing rule. On the opposite, if t is applied to a function $\lambda x.u$, then the type of this function must be of the form (up to a substitution of b) $!_{b<1}((!_{a<\mathrm{I}}A) \multimap \tau)$. This means that $\lambda x.u$ uses I times its arguments, or, that x can appear at most I times in the reducts of u.

This suggests that the type derivation of a term is not unique in general: whether a term t has type $!_{a<\mathrm{I}}A$ or $!_{a<\mathrm{J}}A$ depends on the use we want to make of t. This intuition will direct us in establishing the typing rules for the other PCF constructs (namely conditional branching and fixpoints).

3. $\mathsf{d\ell PCF_V}$, Formally

In this section, the language of programs and a type system $\mathsf{d\ell PCF_V}$ for it will be introduced formally. While terms are just those of a fairly standard λ-calculus (namely Plotkin's PCF), types may include so-called *index terms*, which are first-order terms denoting natural numbers by which one can express properties about the extensional and intentional behavior of programs.

3.1. PCF

Plotkin introduced PCF [18] as a simply typed lambda calculus with full recursion. Terms of the language also include primitive natural numbers, predecessor and successor operators as well as a branching construct. PCF as in [18] also includes booleans, which we elide here for the sake of simplicity. We here adopt a call-by-value version of PCF. *Values* and *terms* are generated by the following grammar:

$$\begin{array}{lll} \text{Values} & v, w ::= & \underline{\mathtt{n}} \mid \lambda x.t \mid \mathtt{fix}\ x.t; \\ \text{Terms} & s, t, u ::= & x \mid v \mid tu \mid \mathtt{s}(t) \mid \mathtt{p}(t) \\ & & \mid \mathtt{ifz}\ t\ \mathtt{then}\ u\ \mathtt{else}\ s. \end{array}$$

The CBV operational semantics of PCF is generated by the rules in Figure 1, subject to the closure rules in Figure 2. The language is provided with a simple type system. Types (denoted by T, U) are those generated by the basic type `Nat` and the binary type constructor $\Rightarrow$, and typing rules are standard (Figure 3). A term t is said to be a *program* if it can be given the PCF type `Nat` in the empty context.

Notice that a typable PCF term is not guaranteed to terminate. For instance the term $t_\omega \stackrel{\text{def}}{=} \mathtt{fix}\ y.\lambda x.yx$ is typable with $\mathtt{Nat} \Rightarrow \mathtt{Nat}$. However, for any natural number n, the term $t_\omega\,\underline{\mathtt{n}}$ call-by-value reduces to itself in two steps. In $\mathsf{d\ell PCF}$ (Section 3.4), we decorate PCF types with some cost annotations (that are *indexes*, see Section 3.3), so that the evaluation complexity of a term can be deduced from its type. In particular, only terminating terms are typable.

The complexity bound that we will give for a given PCF term t will be

$$
\begin{array}{rcl}
(\lambda x.t)\ v & \to_v & t[x := v] \\
\mathtt{s}(\underline{\mathtt{n}}) & \to_v & \underline{\mathtt{n}+1} \\
\mathtt{p}(\underline{\mathtt{n}+1}) & \to_v & \underline{\mathtt{n}} \\
\mathtt{p}(\underline{0}) & \to_v & \underline{0} \\
\mathtt{ifz}\ \underline{0}\ \mathtt{then}\ t\ \mathtt{else}\ u & \to_v & t \\
\mathtt{ifz}\ \underline{\mathtt{n}+1}\ \mathtt{then}\ t\ \mathtt{else}\ u & \to_v & u \\
(\ \mathtt{fix}\ x.t)\ v & \to_v & (t[x :=\ \mathtt{fix}\ x.t])\ v
\end{array}
$$

Figure 1: Call-by-value Reduction Rules for PCF terms.

$$
\frac{t \to_v s}{tu \to_v su} \qquad \frac{t \to_v s}{ut \to_v us} \qquad \frac{t \to_v s}{\mathtt{s}(t) \to_v \mathtt{s}(s)} \qquad \frac{t \to_v s}{\mathtt{p}(t) \to_v \mathtt{p}(s)}
$$

$$
\frac{t \to_v s}{\mathtt{ifz}\ t\ \mathtt{then}\ u\ \mathtt{else}\ p \to_v\ \mathtt{ifz}\ s\ \mathtt{then}\ u\ \mathtt{else}\ p}
$$

Figure 2: Closure Reduction Rules of PCF terms.

$$
\frac{}{\Pi, x : T \vdash x : T} \qquad \frac{\Pi, x : U \vdash t : T}{\Pi \vdash \lambda x.t : U \Rightarrow T} \qquad \frac{\Pi \vdash t : U \Rightarrow T \quad \Pi \vdash u : U}{\Pi \vdash tu : T}
$$

$$
\frac{}{\Pi \vdash \underline{\mathtt{n}} : \mathtt{Nat}} \qquad \frac{\Pi \vdash t : \mathtt{Nat}}{\Pi \vdash \mathtt{s}(t) : \mathtt{Nat}} \qquad \frac{\Pi \vdash t : \mathtt{Nat}}{\Pi \vdash \mathtt{p}(t) : \mathtt{Nat}}
$$

$$
\frac{\Pi \vdash t : \mathtt{Nat} \quad \Pi \vdash u : T \quad \Pi \vdash s : T}{\Pi \vdash\ \mathtt{ifz}\ t\ \mathtt{then}\ u\ \mathtt{else}\ s : T} \qquad \frac{\Pi, x : T \vdash t : T}{\Pi \vdash\ \mathtt{fix}\ x.t : T}
$$

Figure 3: PCF Typing Rules.

related to its *syntactic size* $|t|$:

$$\begin{aligned} |\underline{\mathtt{n}}| = |x| &= 2; \\ |\lambda x.t| = |\ \mathtt{fix}\ x.t| &= |t| + 2; \\ |tu| &= |t| + |u| + 2; \\ |\mathtt{s}(t)| = |\mathtt{p}(t)| &= |t| + 2; \\ |\ \mathtt{ifz}\ t\ \mathtt{then}\ u\ \mathtt{else}\ s| &= |t| + |u| + |s| + 2. \end{aligned}$$

In other words, every symbol counts for 2 in the syntactic size of a term. In the following, we might also consider the *multiplicative size* of PCF terms. It is defined as their syntactic size, except on values which have null multiplicative size (see Figure 11 for a formal definition). The reason why values are not taken into account by the multiplicative size, is that evaluation by the CEK abstract machine (that will be used to define the complexity of a term, see Section 3.2) proceeds by first *scanning* terms until a value is reached, and the cost of these initial steps is taken into account by the multiplicative size. Then this value is either erased (*e.g.* when a lambda abstraction is given an argument), or duplicated (*e.g.* when it is itself the argument to a lambda abstraction). The cost of this second phase is measured by the type system $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$.

3.2. An Abstract Machine for PCF

The call-by-value evaluation of PCF terms can be faithfully captured by an abstract machine in the style of CEK [16], which is the subject of this section.

The internal state of the $\mathsf{CEK}_{\mathsf{PCF}}$ machine consists of a closure and a stack, interacting following a set of rules (figures 4 and 5). The machine mimics weak call-by-value evaluation of programs: it starts with a closed term (or, formally, a closure) and an empty stack, and scans the term until it reaches its head subterm. Doing this, it records all the other subterms (that constitute its *evaluation context*) in the stacks. Once the head subterm has been reached, one of the reduction rules of Figure 1 is simulated.

Formally, a *value closure* is a pair $\mathsf{v} = \langle\, v\,;\, \xi\,\rangle$ where v is a value and ξ is an *environment*, itself a list of assignments of value closures to variables:

$$\xi ::= \varnothing \ \Big|\ (x \mapsto \mathsf{v}) \cdot \xi.$$

A *closure* is a pair $\mathsf{c} = \langle\, t\,;\, \xi\,\rangle$ where t is a term (and not necessarily a value). Given a closure c, one naturally define the *unfolding* $\mathsf{Unf}(\mathsf{c})$, namely the

$$
\begin{array}{rcrcrcl}
\mathsf{v} & \star & \mathsf{arg}(\mathsf{c}\,,\,\pi) & \succ & \mathsf{c} & \star & \mathsf{fun}(\mathsf{v}\,,\,\pi) \\
\mathsf{v} & \star & \mathsf{fun}(\langle\, \lambda x.t\,;\,\xi \,\rangle, \pi) & \succ & \langle\, t\,;\,(x \mapsto \mathsf{v}) \cdot \xi \,\rangle & \star & \pi \\
\mathsf{v} & \star & \mathsf{fun}(\langle\, \texttt{fix}\ x.t\,;\,\xi \,\rangle, \pi) & \succ & & & \\
 & & & & \langle\, t\,;\,(x \mapsto \langle\, \texttt{fix}\ x.t\,;\,\xi \,\rangle) \cdot \xi \,\rangle & \star & \mathsf{arg}(\mathsf{v}\,,\,\pi) \\
\langle\, \underline{\mathtt{0}}\,;\,\xi' \,\rangle & \star & \mathsf{fork}(t\,,\,u\,,\,\xi\,,\,\pi) & \succ & \langle\, t\,;\,\xi \,\rangle & \star & \pi \\
\langle\, \underline{\mathtt{n}+1}\,;\,\xi' \,\rangle & \star & \mathsf{fork}(t\,,\,u\,,\,\xi\,,\,\pi) & \succ & \langle\, u\,;\,\xi \,\rangle & \star & \pi \\
\langle\, \underline{\mathtt{n}}\,;\,\xi \,\rangle & \star & \mathsf{s}(\pi) & \succ & \langle\, \underline{\mathtt{n}+1}\,;\,\varnothing \,\rangle & \star & \pi \\
\langle\, \underline{\mathtt{n}}\,;\,\xi \,\rangle & \star & \mathsf{p}(\pi) & \succ & \langle\, \underline{\mathtt{n}-1}\,;\,\varnothing \,\rangle & \star & \pi
\end{array}
$$

Figure 4: $\mathsf{CEK}_{\mathsf{PCF}}$ Evaluation Rules for Value Closures.

$$
\begin{array}{rcrcrcl}
\langle\, x\,;\,\xi \,\rangle & \star & \pi & \succ & \xi(x) & \star & \pi \\
\langle\, tu\,;\,\xi \,\rangle & \star & \pi & \succ & \langle\, t\,;\,\xi \,\rangle & \star & \mathsf{arg}(\langle\, u\,;\,\xi \,\rangle, \pi) \\
\langle\, \mathsf{s}(t)\,;\,\xi \,\rangle & \star & \pi & \succ & \langle\, t\,;\,\xi \,\rangle & \star & \mathsf{s}(\pi) \\
\langle\, \mathsf{p}(t)\,;\,\xi \,\rangle & \star & \pi & \succ & \langle\, t\,;\,\xi \,\rangle & \star & \mathsf{p}(\pi) \\
\langle\, \texttt{ifz}\ t\ \texttt{then}\ u\ \texttt{else}\ s\,;\,\xi \,\rangle & \star & \pi & \succ & \langle\, t\,;\,\xi \,\rangle & \star & \mathsf{fork}(u\,,\,s\,,\,\xi\,,\,\pi)
\end{array}
$$

Figure 5: $\mathsf{CEK}_{\mathsf{PCF}}$ Contextual Evaluation Rules.

(closed) term obtained by iteratively substituting variables for terms, i.e.,

$$\mathsf{Unf}(\langle\, t\,;\, (x_1 \mapsto \mathsf{v}_1) \cdot \ldots \cdot (x_n \mapsto \mathsf{v}_n)\,\rangle) = t[x_1 := \mathsf{Unf}(\mathsf{v}_1)] \cdots [x_n := \mathsf{Unf}(\mathsf{v}_n)].$$

Stacks are terms from the following grammar:

$$\pi ::= \diamond \;\Big|\; \mathsf{arg}(\mathsf{c}\,,\,\pi) \;\Big|\; \mathsf{fun}(\mathsf{v}\,,\,\pi) \;\Big|\; \mathsf{fork}(t\,,\,u\,,\,\xi\,,\,\pi) \;\Big|\; \mathsf{s}(\pi) \;\Big|\; \mathsf{p}(\pi).$$

A *process* P is then a pair $\mathsf{c} \star \pi$ of a closure and a stack. The stack π describes the *evaluation context* (or the *continuation*) of the closure c. If it is in the form $\mathsf{arg}(\mathsf{c_o}\,,\,\pi_o)$, it means that c_o is the first argument of c, and π_o is its continuation. Conversely, if π is $\mathsf{fun}(\mathsf{v}_o\,,\,\pi_o)$, it means that after evaluating c, its value will be given as argument to the function represented by v. These constructions are used to simulate call-by-value β-reduction. Indeed, to evaluate the application of a function t to an argument u, the $\mathsf{CEK_{PCF}}$ first computes the value $\lambda x.t_o$ of t, then the value v of u, and last maps the variable x to v in t_o:

$$\begin{array}{rcccc}
\langle\, tu\,;\,\xi\,\rangle \star \pi & \succ & \langle\, t\,;\,\xi\,\rangle & \star & \mathsf{arg}(\langle\, u\,;\,\xi\,\rangle\,,\,\pi) \\
& \succ^* & \langle\, \lambda x.t_o\,;\,\xi\,\rangle & \star & \mathsf{arg}(\langle\, u\,;\,\xi\,\rangle\,,\,\pi) \\
& \succ & \langle\, u\,;\,\xi\,\rangle & \star & \mathsf{fun}(\langle\, \lambda x.t_o\,;\,\xi\,\rangle\,,\,\pi) \\
& \succ^* & \mathsf{v} & \star & \mathsf{fun}(\langle\, \lambda x.t_o\,;\,\xi\,\rangle\,,\,\pi) \\
& \succ & \langle\, t_o\,;\,(x \mapsto \mathsf{v}) \cdot \xi\,\rangle & \star & \pi
\end{array}$$

The last three constructions defining a stack are used to describe the evaluation context of a program whose value is a number. $\mathsf{fork}(t\,,\,u\,,\,\xi\,,\,\pi_o)$ means that t will then be evaluated if this value is zero, and if it is strictly positive then u will be evaluated (in both cases, ξ will be the environment and π_o the next continuation). $\mathsf{s}(\pi_o)$ and $\mathsf{p}(\pi_o)$ mean that the successor and the predecessor of this value will be evaluated, respectively.

Figure 5 gives the formal rules that the $\mathsf{CEK_{PCF}}$ follows to scan a term until the head subterm (which is a value) is reached, while Figure 4 gives the ones that simulate a reduction rule of PCF (Figure 1).

Evaluating a term t in the $\mathsf{CEK_{PCF}}$ consists in iteratively applying any of these rules, starting from the process $\langle\, t\,;\,\varnothing\,\rangle \star \diamond$. Please observe that the behavior of the machine is deterministic, since any process can be the left hand side of at most one evaluation rule. If the evaluation of a correct (*i.e.* typable) closed PCF term terminates, then it leads to a process on the form $\langle\, v\,;\,\xi\,\rangle \star \diamond$. In this case, we may use the following notations:

Definition 3.1 *We say that a term t* evaluates to *a value v and we write $t \Downarrow v$ when $(\langle\, t\,;\, \varnothing\,\rangle \star \diamond) \succ^* (\langle\, w\,;\, \xi\,\rangle \star \diamond)$ and $\mathsf{Unf}(\langle\, w\,;\, \xi\,\rangle) = v$. We also write $t \Downarrow^n v$ to specify that n steps are required for this evaluation to end.*

The following ensures that the $\mathsf{CEK}_{\mathsf{PCF}}$ is an adequate methodology to evaluate PCF terms:

Proposition 3.1 (Adequacy) *If t is a* PCF *term of type* Nat*, then $t \to_v^* \underline{\mathtt{n}}$ iff $t \Downarrow \underline{\mathtt{n}}$.*

Proof. The proof is standard and proceeds as follows:

- On the one hand, one generalizes $\mathsf{Unf}(\cdot)$ to processes in the obvious way. Moreover, the PCF type system is itself lifted to environments, closures, stacks, and ultimately processes.
- On the other hand, it is shown that whenever $\mathrm{P} \succ \mathrm{R}$, then $\mathsf{Unf}(\mathrm{P}) \to_v^* \mathsf{Unf}(\mathrm{R})$.
- Finally, it is shown that a typable process P is irreducible precisely when $\mathsf{Unf}(\cdot)$ is a value.

□

3.3. Index Terms and Equational Programs

Syntactically, index terms are built either from function symbols from a given untyped signature Θ or by applying any of two special term constructs:

$$\mathrm{I}, \mathrm{J}, \mathrm{K} \quad ::= \quad a \;\Big|\; \mathrm{f}(\mathrm{I}_1, \ldots, \mathrm{I}_n) \;\Big|\; \sum_{a<\mathrm{I}} \mathrm{J} \;\Big|\; \bigcirc\!\!\!\!\!\!\bigtriangleup_{a}^{\mathrm{I},\mathrm{J}} \mathrm{K}.$$

Here, f is a symbol of arity n from Θ and a is a variable drawn from a set $\mathcal{V}$ of *index variables*. We assume the symbols 0, 1 (of arity 0) and $+$, $-$ (of arity 2) are always part of Θ (they will be used in the typing rules). An index term in the form $\sum_{a<\mathrm{I}} \mathrm{J}$ is a *bounded sum*, while one in the form $\bigcirc\!\!\!\!\!\!\bigtriangleup_{a}^{\mathrm{I},\mathrm{J}} \mathrm{K}$ is a *forest cardinality*. For every natural number n, the index term n is just $\underbrace{1 + 1 + \ldots + 1}_{n \text{ times}}$.

Index terms are meant to denote natural numbers, possibly depending on the (unknown) values of variables. Variables can be instantiated with other index terms, *e.g.* $\mathrm{I}\{\mathrm{J}/a\}$. So, index terms can also act as first order functions. What is the meaning of the function symbols from the signature Θ? It is the one induced by an equational program $\mathcal{E}$. Formally, an *equational program* $\mathcal{E}$

over a signature Θ is a set of equations in the form $\mathrm{I} = \mathrm{J}$ where both I and J are index terms. We are interested in equational programs guaranteeing that, whenever symbols in Θ are interpreted as partial functions over $\mathbb{N}$ and 0, 1, + and − are interpreted in the usual way, the semantics of any function symbol f can be uniquely determined from $\mathcal{E}$. This can be guaranteed by, for example, taking $\mathcal{E}$ as an Herbrand-Gödel scheme [24] or as an orthogonal constructor term rewriting system [25]. The definition of index terms is parametric on Θ and $\mathcal{E}$: this way one can tune our type system from a highly undecidable but truly powerful machinery down to a tractable but less expressive formal system.

What about the meaning of bounded sums and forest cardinalities? The first is very intuitive: the value of $\sum_{a<\mathrm{I}} \mathrm{J}$ is simply the sum of all possible values of J with a taking the values from 0 up to I, excluded. Forest cardinalities, on the other hand, require some effort to be described. Informally, $\bigcirc\!\!\!\!\triangle_a^{\mathrm{I,J}}\,\mathrm{K}$ is an index term denoting the number of nodes in a forest composed of J trees described using K. All the nodes in the forest are (uniquely) identified by natural numbers. These are obtained by consecutively visiting each tree in preorder, starting from I. The term K has the role of describing the number of children of each forest node, *e.g.*, the number of children of the node $\mathtt{0}$ is $\mathrm{K}\{\mathtt{0}/a\}$. More formally, the meaning of a forest cardinality is defined by the following two equations:

$$\bigcirc\!\!\!\!\triangle_a^{\mathrm{I,0}}\,\mathrm{K} = 0;$$

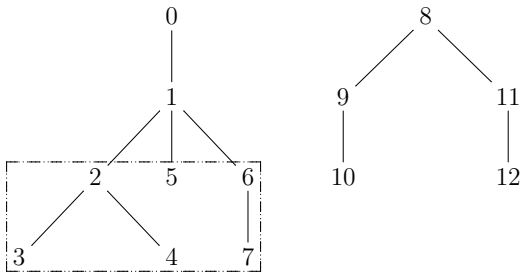
$$\bigcirc\!\!\!\!\triangle_a^{\mathrm{I,J+1}}\,\mathrm{K} = \left(\bigcirc\!\!\!\!\triangle_a^{\mathrm{I,J}}\,\mathrm{K}\right) + 1 + \left(\bigcirc\!\!\!\!\triangle_a^{\mathrm{I+1+}\bigcirc\!\!\!\!\triangle_a^{\mathrm{I,J}}\,\mathrm{K,K}\{\mathrm{I+}\bigcirc\!\!\!\!\triangle_a^{\mathrm{I,J}}\,\mathrm{K}/a\}}\,\mathrm{K}\right).$$

The first equation says that a forest of 0 trees contains no nodes. The second one tells us that a forest of J + 1 trees contains:

- The nodes in the first J trees;
- plus the nodes in the last tree, which are just one plus the nodes in the immediate subtrees of the root, considered themselves as a forest.

To better understand forest cardinalities, consider the following forest com-

prising two trees:



It is well described by an index term K with a free index variable a such that $\mathrm{K}\{1/a\} = 3$; $\mathrm{K}\{\mathrm{n}/a\} = 2$ for $n \in \{2, 8\}$; $\mathrm{K}\{\mathrm{n}/a\} = 1$ when $n \in \{0, 6, 9, 11\}$; and $\mathrm{K}\{\mathrm{n}/a\} = 0$ when $n \in \{3, 4, 5, 7, 10, 12\}$. That is, K describes the number of children of each node. Then $\triangle_a^{0,2}\,\mathrm{K} = 13$ since it takes into account the entire forest; $\triangle_a^{0,1}\,\mathrm{K} = 8$ since it takes into account only the leftmost tree; $\triangle_a^{8,1}\,\mathrm{K} = 5$ since it takes into account only the second tree of the forest; finally, $\triangle_a^{2,3}\,\mathrm{K} = 6$ since it takes into account only the three trees (as a forest) within the dashed rectangle.

One may wonder what is the role of forest cardinalities in the type system. Actually, they play a crucial role in the treatment of recursion, where the unfolding of recursion produces a tree-like structure whose size is just the number of times the (recursively defined) function will be used *globally*. Note that the value of a forest cardinality could also be undefined. For instance, this happens when infinite trees, corresponding to diverging recursive computations, are considered.

The expression $[\![\mathrm{I}]\!]^{\mathcal{E}}_{\rho}$ denotes the meaning of I, defined by induction along the lines of the previous discussion, where $\rho : \mathcal{V} \to \mathbb{N}$ is an assignment and $\mathcal{E}$ is an equational program giving meaning to the function symbols in I. Since $\mathcal{E}$ does not necessarily interpret such symbols as *total* functions, and moreover, the value of a forest cardinality can be undefined, $[\![\mathrm{I}]\!]^{\mathcal{E}}_{\rho}$ can be undefined itself. A *constraint* is an inequality in the form $\mathrm{I} \leqslant \mathrm{J}$. Such a constraint is *true* (or *satisfied*) in an assignment ρ if $[\![\mathrm{I}]\!]^{\mathcal{E}}_{\rho}$ and $[\![\mathrm{J}]\!]^{\mathcal{E}}_{\rho}$ are *both* defined and the first is smaller or equal to the latter. Now, for a subset ϕ of $\mathcal{V}$, and for a set Φ of constraints involving variables in ϕ, the expression

$$\phi; \Phi \models_{\mathcal{E}} \mathrm{I} \leqslant \mathrm{J}$$

denotes the fact that the truth of $\mathrm{I} \leqslant \mathrm{J}$ *semantically* follows from the truth of the constraints in Φ. To denote that I is well defined for $\mathcal{E}$ and any valuation ρ satisfying Φ, we may write $\phi; \Phi \models_{\mathcal{E}} \mathrm{I} \Downarrow$ instead of $\phi; \Phi \models_{\mathcal{E}} \mathrm{I} \leqslant \mathrm{I}$.

$$\frac{\phi;\Phi \vDash_{\mathcal{E}} \mathrm{K} \leqslant \mathrm{I} \qquad \phi;\Phi \vDash_{\mathcal{E}} \mathrm{J} \leqslant \mathrm{H}}{\phi;\Phi \vdash_{\mathcal{E}} \mathtt{Nat}[\mathrm{I},\mathrm{J}] \sqsubseteq \mathtt{Nat}[\mathrm{K},\mathrm{H}]} \qquad \frac{\phi;\Phi \vdash_{\mathcal{E}} \varrho \sqsubseteq \sigma \qquad \phi;\Phi \vdash_{\mathcal{E}} \tau \sqsubseteq \varphi}{\phi;\Phi \vdash_{\mathcal{E}} \sigma \multimap \tau \sqsubseteq \varrho \multimap \varphi}$$

$$\frac{(a,\phi);(a < \mathrm{J},\Phi) \vdash_{\mathcal{E}} A \sqsubseteq B \qquad \phi;\Phi \vDash_{\mathcal{E}} \mathrm{J} \leqslant \mathrm{I}}{\phi;\Phi \vdash_{\mathcal{E}} [a < \mathrm{I}] \cdot A \sqsubseteq [a < \mathrm{J}] \cdot B}$$

Figure 6: Subtyping Derivation Rules of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$.

3.4. The $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ *Type System*

The Language of Types. The type system $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ can be seen as a refinement of PCF obtained by a linear decoration of its type derivations. *Linear* and *modal types* are defined as follows:

Linear Types	$A, B ::= \ \sigma \multimap \tau;$
Modal Types	$\sigma, \tau ::= \ [a < \mathrm{I}] \cdot A \ \mid \ \mathtt{Nat}[\mathrm{I},\mathrm{J}].$

where I, J range over index terms and a ranges over index variables. Modal types need some comments. Natural numbers are freely duplicable, so $\mathtt{Nat}[\mathrm{I},\mathrm{J}]$ is modal by definition. As a first approximation, $[a < \mathrm{I}] \cdot A$ can be thought of as a universal quantification of A, and so a is bound in the linear type A. Moreover, the condition $a < \mathrm{I}$ says that σ consists of all the instances of the linear type A where the variable a is successively instantiated with the values from $\mathtt{0}$ to $\mathrm{I} - \mathtt{1}$, *i.e.*, $A\{\mathtt{0}/a\}, \ldots, A\{\mathrm{I} - \mathtt{1}/a\}$. For those readers who are familiar with linear logic, and in particular with BLL, the modal type $[a < \mathrm{I}] \cdot A$ is a generalization of the BLL formula $!_{a<p}A$ to arbitrary index terms. As such it can be thought of as representing the type $A\{\mathtt{0}/a\} \otimes \cdots \otimes A\{\mathrm{I} - \mathtt{1}/a\}$. $\mathtt{Nat}[\mathrm{I}]$ is syntactic sugar for $\mathtt{Nat}[\mathrm{I},\mathrm{I}]$. As usual, the variable a is said to be *bound* in the modal type $[a < \mathrm{I}] \cdot A$, and when a does not appear free in σ nor in τ we may write $\sigma \overset{\mathrm{I}}{\multimap} \tau$ instead of $[a < \mathrm{I}] \cdot \sigma \multimap \tau$.

In the typing rules we are going to define, modal types are manipulated in an algebraic way. For this reason, two operations on modal types need to be introduced. The first one is a binary operation $\uplus$ on modal types. Suppose that $\sigma = [a < \mathrm{I}] \cdot A\{a/c\}$ and that $\tau = [b < \mathrm{J}] \cdot A\{\mathrm{I} + b/c\}$. In other words,

$$\frac{}{\phi;\Phi;\Gamma,x:\sigma\vdash^{\mathcal{E}}_{\mathsf{0}} x:\sigma}(Ax) \qquad \frac{}{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{\mathsf{0}} \underline{\mathrm{n}}:\mathtt{Nat}[\mathrm{n},\mathrm{n}]}(n)$$

$$\frac{(a,\phi);(a<\mathrm{I},\Phi);\Gamma,x:\sigma\vdash^{\mathcal{E}}_{\mathrm{K}} t:\tau}{\phi;\Phi;\sum_{a<\mathrm{I}}\Gamma\vdash^{\mathcal{E}}_{\mathrm{I}+\sum_{a<\mathrm{I}}\mathrm{K}} \lambda x.t:[a<\mathrm{I}]\cdot\sigma\multimap\tau}(\multimap)$$

$$\frac{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{\mathrm{K}} t:[a<\mathtt{1}]\cdot\sigma\multimap\tau \qquad \phi;\Phi;\Delta\vdash^{\mathcal{E}}_{\mathrm{H}} u:\sigma\{\mathtt{0}/a\}}{\phi;\Phi;\Gamma\uplus\Delta\vdash^{\mathcal{E}}_{\mathrm{K}+\mathrm{H}} tu:\tau\{\mathtt{0}/a\}}(App)$$

$$\frac{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{\mathrm{M}} t:\mathtt{Nat}[\mathrm{I},\mathrm{J}]}{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{\mathrm{M}} \mathtt{s}(t):\mathtt{Nat}[\mathrm{I}+\mathtt{1},\mathrm{J}+\mathtt{1}]}(s) \quad \frac{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{\mathrm{M}} t:\mathtt{Nat}[\mathrm{I},\mathrm{J}]}{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{\mathrm{M}} \mathtt{p}(t):\mathtt{Nat}[\mathrm{I}-\mathtt{1},\mathrm{J}-\mathtt{1}]}(p)$$

$$\frac{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{\mathrm{M}} t:\mathtt{Nat}[\mathrm{J},\mathrm{K}] \qquad \begin{array}{c}\phi;(\mathrm{J}\leqslant\mathtt{0},\Phi);\Delta\vdash^{\mathcal{E}}_{\mathrm{N}} u:\tau\\ \phi;(\mathrm{K}\geqslant\mathtt{1},\Phi);\Delta\vdash^{\mathcal{E}}_{\mathrm{N}} s:\tau\end{array}}{\phi;\Phi;\Gamma\uplus\Delta\vdash^{\mathcal{E}}_{\mathrm{M}+\mathrm{N}} \ \mathtt{ifz}\ t\ \mathtt{then}\ u\ \mathtt{else}\ s:\tau}(If)$$

$$\frac{\begin{array}{c}(b,\phi);(b<\mathrm{H},\Phi);\Gamma,x:[a<\mathrm{I}]\cdot A\vdash^{\mathcal{E}}_{\mathrm{J}} t:[a<\mathtt{1}]\cdot B\\ (a,b,\phi);(a<\mathrm{I},b<\mathrm{H},\Phi)\vdash_{\mathcal{E}} B\{\mathtt{0}/a\}\{\triangle_b^{b+1,a}\,\mathrm{I}+b+1/b\}\sqsubseteq A\end{array}}{\phi;\Phi;\sum_{b<\mathrm{H}}\Gamma\vdash^{\mathcal{E}}_{\mathrm{H}+\sum_{b<\mathrm{H}}\mathrm{J}} \ \mathtt{fix}\ x.t:[a<\mathrm{K}]\cdot B\{\mathtt{0}/a\}\{\triangle_b^{0,a}\,\mathrm{I}/b\}}(Fix)$$
$$(\text{where } \mathrm{H}=\triangle_b^{0,\mathrm{K}}\,\mathrm{I})$$

$$\frac{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{\mathrm{I}} t:\sigma \quad \phi;\Phi\vdash_{\mathcal{E}}\Delta\sqsubseteq\Gamma \quad \phi;\Phi\vdash_{\mathcal{E}}\sigma\sqsubseteq\tau \quad \phi;\Phi\vDash_{\mathcal{E}}\mathrm{I}\leqslant\mathrm{J}}{\phi;\Phi;\Delta\vdash^{\mathcal{E}}_{\mathrm{J}} t:\tau}(Subs)$$

Figure 7: Typing Rules of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$.

σ consists of the first I instances of A, *i.e.* $A\{\mathtt{0}/c\}, \ldots, A\{\mathrm{I} - \mathtt{1}/c\}$ while τ consists of the next J instances of A, *i.e.* $A\{\mathrm{I}+\mathtt{0}/c\}, \ldots, A\{\mathrm{I}+\mathrm{J}-\mathtt{1}/c\}$. Their *sum* $\sigma \uplus \tau$ is naturally defined as a modal type consisting of the first $\mathrm{I} + \mathrm{J}$ instances of A, *i.e.* $[c < \mathrm{I} + \mathrm{J}] \cdot A$. Furthermore, $\mathtt{Nat}[\mathrm{I}, \mathrm{J}] \uplus \mathtt{Nat}[\mathrm{I}, \mathrm{J}]$ is just $\mathtt{Nat}[\mathrm{I}, \mathrm{J}]$. An operation of bounded sum on modal types can be defined by generalizing the idea above: suppose that

$$\sigma = [b < \mathrm{J}] \cdot A\{b + \sum_{d<a} \mathrm{J}\{d/a\}/c\}.$$

Then its *bounded sum* $\sum_{a<\mathrm{I}} \sigma$ is just $[c < \sum_{a<\mathrm{I}} \mathrm{J}] \cdot A$. Finally, $\sum_{a<\mathrm{I}} \mathtt{Nat}[\mathrm{J}, \mathrm{K}] = \mathtt{Nat}[\mathrm{J}, \mathrm{K}]$, provided a is not free in J nor in K.

Subtyping. Central to $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ is the notion of subtyping. An inequality relation $\sqsubseteq$ between (linear or modal) types can be defined using the formal system in Figure 6. This relation corresponds to lifting index inequalities at the type level. Please observe that $\sqsubseteq$ is a preorder, *i.e.*, a reflexive and transitive relation.

Typing. A typing judgment is of the form

$$\phi; \Phi; \Gamma \vdash^{\mathcal{E}}_{\mathrm{K}} t : \tau,$$

where K is the *weight* of t, that is (informally) the maximal number of substitutions involved in the CBV evaluation of t. Φ is a set of constraints (see Section 3.3) that we call the *index context*, and Γ is a context assigning a modal type to (at least) each free variable of t. Both sums and bounded sums are naturally extended from modal types to contexts (with, for instance, $\{x : \sigma; y : \tau\} \uplus \{x : \zeta, z : \eta\} = \{x : \sigma \uplus \zeta; y : \tau; z : \eta\}$). There might be free index variables in Φ, Γ, τ and K, all of them from ϕ. Typing judgments can be derived from the rules of Figure 7. We are implicitly assuming that all index terms appearing in (derivable) typing judgments are defined in the appropriate index contexts.

Derivation rules for abstractions and applications have been informally presented in Section 2.2. The other ones are then intuitive, except the derivation rule for typing $\mathtt{fix}\ x.t$, that is worth an explanation: to simplify, assume we want to type only one copy of its type (that is, $\mathrm{K} = \mathtt{1}$). To compute the weight of $\mathtt{fix}\ x.t$, we need to know the number of times t will be copied during the evaluation, that is the number of nodes in the tree of its recursive calls. This tree is described by I (as explained in Section 3.3), since each

occurrence of x in t stands for a recursive call. It has, say, $\mathrm{H} = \bigtriangleup_b^{0,1} \mathrm{I}$ nodes. At each node b of this tree, the a^{th} occurrence of x will be replaced by the a^{th} child of b, *i.e.* by $b + 1 + \bigtriangleup_b^{b+1,a} \mathrm{I}$. The types have to match, and that is what the second premise expresses. Finally, the type of $\mathtt{fix}\ x.t$ is the type of the "main" copy of t, at the root of the tree (*i.e.*, at $b = 0$). The weight counts all the recursive calls (*i.e.*, H) plus the weight of each copy of t (*i.e.*, the weight of t for each $b < \mathrm{H}$).

Last, the subsumption rule (*Subs*) allows to relax the precision standard of a typing judgment. One can also restrict the inequalities on indexes to equalities in this rule, this way constructing *precise* typing judgments. Observe that the set of all rules but this one is syntax directed. Moreover, the subsumption rule preserves the PCF skeleton of the types, and so the type system is itself syntax directed *up to* index inequalities.

Weights and the $\mathsf{CEK}_{\mathsf{PCF}}$ *Machine.* As it will be formalized in Section 5.3, an upper bound for the evaluation of a given term in the $\mathsf{CEK}_{\mathsf{PCF}}$ machine can be obtained by multiplying its weight and its syntactic size. This result can be explained as follows: the weight of a program represents the maximal number of substitutions in its CBV evaluation, and thereby the maximal number of steps of the form

$$\mathsf{v} \star \mathsf{fun}(\langle\, \lambda x.t\,;\, \xi\,\rangle, \pi) \succ \langle\, t\,;\, (x \mapsto \mathsf{v}) \cdot \xi\,\rangle \star \pi; \tag{1}$$

$$\mathsf{v} \star \mathsf{fun}(\langle\, \mathtt{fix}\ x.t\,;\, \xi\,\rangle, \pi) \succ \langle\, t\,;\, (x \mapsto \langle\, \mathtt{fix}\ x.t\,;\, \xi\,\rangle) \cdot \xi\,\rangle \star \mathsf{arg}(\mathsf{v}, \pi) \tag{2}$$

in its evaluation with the $\mathsf{CEK}_{\mathsf{PCF}}$. Between two such steps, the use of the other rules is not taken into account by the weight; however the other rules make the *size* of the process smaller.

4. Examples

In this section we will see how to type some common terms in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$. As a byproduct, we will derive a weight for them.

4.1. Addition

In PCF, addition can be computed as follows:

$$\mathtt{add} \;=\; \mathtt{fix}\ x.\lambda yz.\ \mathtt{ifz}\ y\ \mathtt{then}\ z\ \mathtt{else}\ \mathtt{s}(x\,\mathtt{p}(y)\,z),$$

and has type $\mathtt{Nat} \Rightarrow \mathtt{Nat} \Rightarrow \mathtt{Nat}$. A brief analysis of its evaluation, if we apply it to two values v and w in $\mathtt{Nat}$, indicates that a correct annotation for this type in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ would be

$$[a < \mathtt{1}] \cdot (\mathtt{Nat}[d] \multimap [c < \mathtt{1}] \cdot (\mathtt{Nat}[e] \multimap \mathtt{Nat}[d + e])) \,,$$

where d and e are variable symbols representing the values of v and w respectively. Since we directly apply add, without copying this function, the index variables a and c are bounded by $\mathtt{1}$. The type above is indeed derivable for add in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, assuming that the equational program $\mathcal{E}$ is powerful enough to express the following index terms (they all depend on the index variables b and d):

$$\begin{aligned} \mathrm{I} &= \text{if } b < d \text{ then } \mathtt{1} \text{ else } \mathtt{0}; \\ \mathrm{J} &= d - b - 1; \\ \mathrm{H} &= d - b; \\ \mathrm{K} &= d - b + 1. \end{aligned}$$

The derivation is given in Figure 8. We omit all the subsumption steps, but the index equalities they use are easy to check given that the number of nodes in the tree of recursive calls is $\bigtriangleup_b^{0,1} \mathrm{I} = d + \mathtt{1}$. The final weight is equal to $3 \times (d + \mathtt{1})$.

4.2. Fibonacci

The Fibonacci function can easily be defined using addition:

$$\begin{aligned} \mathsf{fibo} \;=\; & \mathtt{fix}\ f.\lambda x.\ \mathtt{ifz}\ x\ \mathtt{then}\ \underline{\mathtt{1}}\ \mathtt{else} \\ & \quad \mathtt{ifz}\ \mathtt{p}(x)\ \mathtt{then}\ \underline{\mathtt{1}}\ \mathtt{else}\ \mathsf{add}\ (f\,\mathtt{p}(x))\ (f\,\mathtt{p}(\mathtt{p}(x))). \end{aligned}$$

In order to derive a type for this function, we assume that the equational program allows to derive the following index (in)equalities (we later show a

$$
\begin{array}{rcl}
A & = & \mathtt{Nat}[\,\mathrm{J}] \multimap [c < \mathtt{1}] \cdot (\mathtt{Nat}[e] \multimap \mathtt{Nat}[\mathrm{J} + e]) \\
C & = & \mathtt{Nat}[\mathrm{H}] \multimap [c < \mathtt{1}] \cdot (\mathtt{Nat}[e] \multimap \mathtt{Nat}[\mathrm{H} + e]) \\
\Gamma & = & \{x : [a < \mathrm{I}] \cdot A,\ y : \mathtt{Nat}[\mathrm{H}],\ z : \mathtt{Nat}[e]\} \\
\phi & = & \{a, b, c, d, e\} \\
\Phi & = & \{b < d + \mathtt{1}, a < \mathtt{1}, c < \mathtt{1}\}
\end{array}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\phi; (\mathrm{H} \geqslant \mathtt{1}, \Phi); x : [a < \mathrm{I}] \cdot A \vdash^{\mathcal{E}}_{0} x : [a < \mathrm{I}] \cdot A \qquad \dfrac{\phi; (\mathrm{H} \geqslant \mathtt{1}, \Phi); y : \mathtt{Nat}[\mathrm{H}] \vdash^{\mathcal{E}}_{0} y : \mathtt{Nat}[\mathrm{H}]}{\phi; (\mathrm{H} \geqslant \mathtt{1}, \Phi); y : \mathtt{Nat}[\mathrm{H}] \vdash^{\mathcal{E}}_{0} \mathtt{p}(y) : \mathtt{Nat}[\mathrm{J}]}\,(p)}{\phi; (\mathrm{H} \geqslant \mathtt{1}, \Phi); x : [a < \mathrm{I}] \cdot A, y : \mathtt{Nat}[\mathrm{H}] \vdash^{\mathcal{E}}_{0} x\ \mathtt{p}(y) : [c < \mathtt{1}] \cdot (\mathtt{Nat}[e] \multimap \mathtt{Nat}[\mathrm{J} + e])}\,(App) \qquad \phi; (\mathrm{H} \geqslant \mathtt{1}, \Phi); z : \mathtt{Nat}[e] \vdash^{\mathcal{E}}_{0} z : \mathtt{Nat}[e]}{\phi; (\mathrm{H} \geqslant \mathtt{1}, \Phi); \Gamma \vdash^{\mathcal{E}}_{0} x\ \mathtt{p}(y)\, z : \mathtt{Nat}[\mathrm{J} + e]}\,(App)}{\phi; (\mathrm{H} \geqslant \mathtt{1}, \Phi); \Gamma \vdash^{\mathcal{E}}_{0} \mathtt{s}(x\ \mathtt{p}(y)\, z) : \mathtt{Nat}[\mathrm{H} + e]}\,(s)
$$

$$
\dfrac{\dfrac{\dfrac{\phi; \Phi; y : \mathtt{Nat}[\mathrm{H}] \vdash^{\mathcal{E}}_{0} y : \mathtt{Nat}[\mathrm{H}] \qquad \phi; (\mathrm{H} \leqslant \mathtt{0}, \Phi); \Gamma \vdash^{\mathcal{E}}_{0} z : \mathtt{Nat}[\mathrm{H} + e] \qquad \vdots}{\phi; \Phi; \Gamma \vdash^{\mathcal{E}}_{0}\ \mathtt{ifz}\ y\ \mathtt{then}\ z\ \mathtt{else}\ \mathtt{s}(x\ \mathtt{p}(y)\, z) : \mathtt{Nat}[\mathrm{H} + e]}\,(If)}{(b, a, d, e); (b < d + \mathtt{1}, a < \mathtt{1}); (x : [a < \mathrm{I}] \cdot A; y : \mathtt{Nat}[\mathrm{H}]) \vdash^{\mathcal{E}}_{1} \lambda z.\ \mathtt{ifz}\ y\ \mathtt{then}\ z\ \mathtt{else}\ \mathtt{s}(x\ \mathtt{p}(y)\, z) : [c < \mathtt{1}] \cdot (\mathtt{Nat}[e] \multimap \mathtt{Nat}[\mathrm{H} + e])}\,(\multimap)}{(Fix)\ \dfrac{b, d, e; b < d + \mathtt{1}; x : [a < \mathrm{I}] \cdot A \vdash^{\mathcal{E}}_{1+1} \lambda y z.\ \mathtt{ifz}\ y\ \mathtt{then}\ z\ \mathtt{else}\ \mathtt{s}(x\ \mathtt{p}(y)\, z) : [a < \mathtt{1}] \cdot C \qquad b, d, e; b < d + \mathtt{1} \vDash_{\mathcal{E}} C\{b + \mathtt{1}/b\} \equiv A}{d, e; \varnothing; \varnothing \vdash^{\mathcal{E}}_{d+1+\sum_{b<d+1}(1+1)} \mathtt{add} : [a < \mathtt{1}] \cdot \mathtt{Nat}[d] \multimap [c < \mathtt{1}] \cdot (\mathtt{Nat}[e] \multimap \mathtt{Nat}[d + e])}
$$

Figure 8: A type derivation for add.

concrete example of such a model):

$$\begin{aligned}
&\mathrm{f}(b) \geqslant 2 \Rightarrow \mathrm{f}(b+\mathtt{1}) = \mathrm{f}(b) - \mathtt{1};\\
&\mathrm{f}(b) \geqslant 2 \Rightarrow \mathrm{f}(b+\mathtt{1}+\overset{b+\mathtt{1},\mathtt{1}}{\underset{b}{\textcircled{\triangle}}}\,\mathrm{i}(b)) = \mathrm{f}(b) - 2;\\
&\mathrm{f}(b) \geqslant 2 \Rightarrow \mathrm{h}(b) = \mathrm{h}(b+\mathtt{1}) + \mathrm{h}(b+\mathtt{1}+\overset{b+\mathtt{1},\mathtt{1}}{\underset{b}{\textcircled{\triangle}}}\,\mathrm{i}(b))\ ;\\
&\mathrm{f}(b) \leqslant \mathtt{1} \Rightarrow \mathrm{h}(b) = \mathtt{1};\\
&\mathrm{f}(b) \geqslant 2 \Rightarrow \mathrm{i}(b) = 2;\\
&\mathrm{f}(b) \leqslant \mathtt{1} \Rightarrow \mathrm{i}(b) = \mathtt{0};\\
&\mathrm{f}(b) \geqslant 2 \Rightarrow \mathrm{j}(b) = 3 \cdot (\mathrm{h}(b+\mathtt{1}) + \mathtt{1});\\
&\mathrm{f}(b) \leqslant \mathtt{1} \Rightarrow \mathrm{j}(b) = \mathtt{0};\\
&\qquad \mathrm{k} = \overset{\mathtt{0},\mathtt{1}}{\underset{b}{\textcircled{\triangle}}}\,\mathrm{i}(b);\\
&\mathrm{m}(\mathtt{0},b) = \mathrm{f}(b) - \mathtt{1};\\
&\mathrm{m}(\mathtt{1},b) = \mathrm{f}(b) - 2;\\
&\mathrm{n}(\mathtt{0},b) = \mathrm{h}(b+\mathtt{1});\\
&\mathrm{n}(\mathtt{1},b) = \mathrm{h}(b) - \mathrm{h}(b+\mathtt{1}).
\end{aligned}$$

We also use the previous type derivation for $\mathtt{add}$ with an instantiation of the variables d and e:

$$b; \varnothing; \varnothing \vdash^{\mathcal{E}}_{3(\mathrm{n}(\mathtt{0},b)+\mathtt{1})} \mathtt{add} : \mathtt{Nat}[\mathrm{n}(\mathtt{0},b)] \overset{\mathtt{1}}{\multimap} (\mathtt{Nat}[\mathrm{n}(\mathtt{1},b)] \overset{\mathtt{1}}{\multimap} \mathtt{Nat}[\mathrm{n}(\mathtt{0},b)+\mathrm{n}(\mathtt{1},b)]).$$

The type derivation of fibo is given in Figure 9. For the sake of readability the semantic hypothesis of the subsumption derivation steps are not written, but they easily follow from the listed index equations. Subsumption steps are denoted with a dashed horizontal line between the premise and the conclusion. To apply the fixpoint derivation rule (double horizontal line, at the root of the derivation tree), the following semantic judgments have to hold:

$$a, b; b < \mathrm{k}, a < \mathrm{i}(b) \vDash_{\mathcal{E}} \mathtt{Nat}[\mathrm{m}(a,b)] \sqsubseteq \mathtt{Nat}[\mathrm{f}(\mathtt{1}+b+\overset{b+\mathtt{1},a}{\underset{b}{\textcircled{\triangle}}}\,\mathrm{i}(b))];$$

$$a, b; b < \mathrm{k}, a < \mathrm{i}(b) \vDash_{\mathcal{E}} \mathtt{Nat}[\mathrm{h}(\mathtt{1}+b+\overset{b+\mathtt{1},a}{\underset{b}{\textcircled{\triangle}}}\,\mathrm{i}(b))] \sqsubseteq \mathtt{Nat}[\mathrm{n}(a,b)].$$

The validity of both judgments follows from the index equations we have assumed (the types are in fact pairwise equivalent). We will now delineate a model that validates those equations.

Fibonacci trees. The *Fibonacci tree* of size n, written $\mathcal{T}_n$, is defined inductively by

$$\begin{aligned}\mathcal{T}_0 &= \mathsf{Leaf}_0;\\ \mathcal{T}_1 &= \mathsf{Leaf}_1;\\ \mathcal{T}_{n+2} &= \mathsf{Node}_{n+2}(\mathcal{T}_{n+1}, \mathcal{T}_n).\end{aligned}$$

The usual notions on trees (leaves, descendants, right or left child, etc.) are defined in the standard way, and each node is naturally labeled by a natural number. As an example, the only node in $\mathcal{T}_1$ is labeled with 1. Observe that for every n, the root of $\mathcal{T}_n$ is labeled by n.

Consider now the Fibonacci tree of size $n_o \in \mathbb{N}$. Its nodes are totally ordered by the usual tree preorder. We define $\mathrm{f}(b)$ as the label of the b^{th} node of $\mathcal{T}_{n_o}$ (in particular, $\mathrm{f}(0) = n_o$, because the preorder starts at the root), and $\mathrm{h}(b)$ as the number of leaves among the descendants of the b^{th} node. We then define k as the total number of nodes in $\mathcal{T}_{n_o}$, and $\mathrm{i}(b)$ as 0 if $\mathrm{f}(b) \in \{0, 1\}$, and 2 otherwise. Please observe that $\mathrm{i}(b)$ is the symbol describing the tree $\mathcal{T}_{n_o}$ as explained in Section 3.3 (in particular, $\bigtriangleup_b^{b+1,1} \mathrm{i}(b)$ is the number of descendants of the node b if it is not a leaf, and $b + 1 + \bigtriangleup_b^{b+1,1} \mathrm{i}(b)$ is the number of its right children). We also define $\mathrm{m}(a, b)$ and $\mathrm{n}(a, b)$ for $a \in \{0, 1\}$ by:

$$\begin{aligned}\mathrm{m}(\mathtt{0}, b) &= \mathrm{f}(b) - \mathtt{1};\\ \mathrm{n}(\mathtt{0}, b) &= \mathrm{h}(b + \mathtt{1});\\ \mathrm{m}(\mathtt{1}, b) &= \mathrm{f}(b) - 2;\\ \mathrm{n}(\mathtt{1}, b) &= \mathrm{h}(b) - \mathrm{h}(b + \mathtt{1}).\end{aligned}$$

Finally, $\mathrm{j}(b)$ is zero if $\mathrm{f}(b) \leqslant 1$, and $3 \cdot (\mathrm{h}(b + 1) + 1)$ otherwise. This model ensures the validity of all the index equations required to derive the type $\mathtt{Nat}[\mathrm{f}(\mathtt{0})] \overset{\mathtt{1}}{\multimap} \mathtt{Nat}[\mathrm{h}(\mathtt{0})]$ and the weight $\mathrm{k} + \sum_{b<\mathrm{k}}(\mathtt{1} + \mathrm{j}(b))$ for fibo. Notice that if fibo is applied to a natural number $\mathrm{n}_o = \mathrm{f}(0)$, then the result is the number of leaves in $\mathcal{T}_{n_o}$, which is actually the value of the Fibonacci function in n_o. In fact, the typing of the Fibonacci function has reduced the analysis of its code and its complexity to a set of first order equations that we can check semantically.

$$
\begin{array}{rl}
\Phi_o & = \{b < \mathrm{k}; \mathrm{f}(b) - 1 \geqslant 1\} \qquad\qquad \mathsf{fibo} \;=\; \mu f.\lambda x.\ \mathtt{ifz}\ x\ \mathtt{then}\ \underline{1}\ \mathtt{else}\ \big(\ \mathtt{ifz}\ \mathtt{p}(x)\ \mathtt{then}\ \underline{1}\ \mathtt{else}\ \mathtt{add}\ (f\,\mathtt{p}(x))\ (f\,\mathtt{p}(\mathtt{p}(x)))\big)\\
\Gamma_1 & = \{\ f : [a < 1] \cdot \mathtt{Nat}[\mathrm{m}(a,b)] \multimap \mathtt{Nat}[\mathrm{n}(a,b)]\ ,\ x : \mathtt{Nat}[\mathrm{f}(b)]\ \}\\
\Gamma_2 & = \{\ f : [a < 1] \cdot \mathtt{Nat}[\mathrm{m}(a+1,b)] \multimap \mathtt{Nat}[\mathrm{n}(a+1,b)]\ ,\ x : \mathtt{Nat}[\mathrm{f}(b)]\ \}\\
\Gamma_1 \uplus \Gamma_2 & = \{\ f : [a < 2] \cdot \mathtt{Nat}[\mathrm{m}(a,b)] \multimap \mathtt{Nat}[\mathrm{n}(a,b)]\ ,\ x : \mathtt{Nat}[\mathrm{f}(b)]\ \}\\
\Gamma_o & = \{\ f : [a < \mathrm{i}(\mathrm{b})] \cdot \mathtt{Nat}[\mathrm{m}(a,b)] \multimap \mathtt{Nat}[\mathrm{n}(a,b)]\ ,\ x : \mathtt{Nat}[\mathrm{f}(b)]\ \}
\end{array}
$$

$$
\dfrac{b; \Phi_o; f : [a < 1] \cdot \mathtt{Nat}[\mathrm{m}(a,b)] \multimap \mathtt{Nat}[\mathrm{n}(a,b)] \vdash_0^{\mathcal{E}} f : [a < 1] \cdot \mathtt{Nat}[\mathrm{m}(a,b)] \multimap \mathtt{Nat}[\mathrm{n}(a,b)] \qquad \dfrac{\dfrac{b; \Phi_o; x : \mathtt{Nat}[\mathrm{f}(b)] \vdash_0^{\mathcal{E}} x : \mathtt{Nat}[\mathrm{f}(b)]}{b; \Phi_o; x : \mathtt{Nat}[\mathrm{f}(b)] \vdash_0^{\mathcal{E}} \mathtt{p}(x) : \mathtt{Nat}[\mathrm{f}(b) - 1]}}{b; \Phi_o; x : \mathtt{Nat}[\mathrm{f}(b)] \vdash_0^{\mathcal{E}} \mathtt{p}(x) : \mathtt{Nat}[\mathrm{m}(0,b)]}}{b; \Phi_o; f : [a < 1] \cdot \mathtt{Nat}[\mathrm{m}(a,b)] \multimap \mathtt{Nat}[\mathrm{n}(a,b)], x : \mathtt{Nat}[\mathrm{f}(b)] \vdash_0^{\mathcal{E}} f\,\mathtt{p}(x) : \mathtt{Nat}[\mathrm{n}(0,b)]}
$$

$$
\dfrac{b; \Phi_o; f : [a < 1] \cdot \mathtt{Nat}[\mathrm{m}(a+1,b)] \multimap \mathtt{Nat}[\mathrm{n}(a+1,b)] \vdash_0^{\mathcal{E}} f : [a < 1] \cdot \mathtt{Nat}[\mathrm{m}(a+1,b)] \multimap \mathtt{Nat}[\mathrm{n}(a+1,b)] \qquad \vdots}{b; \Phi_o; f : [a < 1] \cdot \mathtt{Nat}[\mathrm{m}(a+1,b)] \multimap \mathtt{Nat}[\mathrm{n}(a+1,b)], x : \mathtt{Nat}[\mathrm{f}(b)] \vdash_0^{\mathcal{E}} f\,\mathtt{p}(\mathtt{p}(x)) : \mathtt{Nat}[\mathrm{n}(1,b)]}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{b; \Phi_o; x : \mathtt{Nat}[\mathrm{f}(b)] \vdash_0^{\mathcal{E}} x : \mathtt{Nat}[\mathrm{f}(b)]}{b; \Phi_o; x : \mathtt{Nat}[\mathrm{f}(b)] \vdash_0^{\mathcal{E}} \mathtt{p}(x) : \mathtt{Nat}[\mathrm{f}(b) - 1]}}{b; \Phi_o; x : \mathtt{Nat}[\mathrm{f}(b)] \vdash_0^{\mathcal{E}} \mathtt{p}(\mathtt{p}(x)) : \mathtt{Nat}[\mathrm{f}(b) - 2]}}{b; \Phi_o; x : \mathtt{Nat}[\mathrm{f}(b)] \vdash_0^{\mathcal{E}} \mathtt{p}(\mathtt{p}(x)) : \mathtt{Nat}[\mathrm{m}(1,b)]}
$$

$$
\dfrac{b; \Phi_o; \varnothing \vdash^{\mathcal{E}}_{3(\mathrm{n}(0,b)+1)} \mathsf{add} : \mathtt{Nat}[\mathrm{n}(0,b)] \xrightarrow{1} (\mathtt{Nat}[\mathrm{n}(1,b)] \xrightarrow{1} \mathtt{Nat}[\mathrm{n}(0,b) + \mathrm{n}(1,b)]) \qquad b; \Phi_o; \Gamma_1 \vdash_0^{\mathcal{E}} f\,\mathtt{p}(x) : \mathtt{Nat}[\mathrm{n}(0,b)]}{b; \Phi_o; \Gamma_1 \vdash^{\mathcal{E}}_{3(\mathrm{n}(0,b)+1)} \mathsf{add}\,(f\,\mathtt{p}(x)) : \mathtt{Nat}[\mathrm{n}(1,b)] \xrightarrow{1} \mathtt{Nat}[\mathrm{n}(0,b) + \mathrm{n}(1,b)]}
$$

$$
\dfrac{\begin{array}{r} b; b < \mathrm{k}, \mathrm{f}(b) \geqslant 1 ; x : \mathtt{Nat}[\mathrm{f}(b)] \quad \vdash_0^{\mathcal{E}} \mathtt{p}(x) : \mathtt{Nat}[\mathrm{f}(b) - 1] \\ b; b < \mathrm{k}, \mathrm{f}(b) - 1 = 0 ; \Gamma_o \quad \vdash^{\mathcal{E}}_{\mathrm{j}(b)} \underline{1} : \mathtt{Nat}[\mathrm{h}(b)] \end{array} \qquad \dfrac{\vdots \qquad b; \Phi_o; \Gamma_2 \vdash_0^{\mathcal{E}} f\,\mathtt{p}(\mathtt{p}(x)) : \mathtt{Nat}[\mathrm{n}(1,b)]}{b; \Phi_o; \Gamma_1 \uplus \Gamma_2 \vdash^{\mathcal{E}}_{3(\mathrm{n}(0,b)+1)} \mathsf{add}\ (f\,\mathtt{p}(x))\ (f\,\mathtt{p}(\mathtt{p}(x))) : \mathtt{Nat}[\mathrm{n}(0,b) + \mathrm{n}(1,b)]}}{b; b < \mathrm{k}, \mathrm{f}(b) - 1 \geqslant 1 ; \Gamma_o \vdash^{\mathcal{E}}_{\mathrm{j}(b)} \mathsf{add}\ (f\,\mathtt{p}(x))\ (f\,\mathtt{p}(\mathtt{p}(x))) : \mathtt{Nat}[\mathrm{h}(b)]}
$$

$$
\dfrac{\dfrac{b; b < \mathrm{k}; x : \mathtt{Nat}[\mathrm{f}(b)] \vdash_0^{\mathcal{E}} x : \mathtt{Nat}[\mathrm{f}(b)] \qquad \vdots \qquad b; b < \mathrm{k}, \mathrm{f}(b) = 0 ; \Gamma_o \vdash^{\mathcal{E}}_{\mathrm{j}(b)} \underline{1} : \mathtt{Nat}[\mathrm{h}(b)]}{b; b < \mathrm{k}; \Gamma_o \vdash^{\mathcal{E}}_{\mathrm{j}(b)} \ \mathtt{ifz}\ x\ \mathtt{then}\ \underline{1}\ \mathtt{else}\ \big(\ \mathtt{ifz}\ \mathtt{p}(x)\ \mathtt{then}\ \underline{1}\ \mathtt{else}\ \mathtt{add}\ (f\,\mathtt{p}(x))\ (f\,\mathtt{p}(\mathtt{p}(x)))\big) : \mathtt{Nat}[\mathrm{h}(b)]}}{\begin{array}{l} b\ ;\ b < \mathrm{k}\ ;\ f : [a < \mathrm{i}(\mathrm{b})] \cdot \mathtt{Nat}[\mathrm{m}(a,b)] \multimap \mathtt{Nat}[\mathrm{n}(a,b)] \\ \qquad \vdash^{\mathcal{E}}_{1+\mathrm{j}(b)} \lambda x.\ \mathtt{ifz}\ x\ \mathtt{then}\ \underline{1}\ \mathtt{else}\ \big(\ \mathtt{ifz}\ \mathtt{p}(x)\ \mathtt{then}\ \underline{1}\ \mathtt{else}\ \mathtt{add}\ (f\,\mathtt{p}(x))\ (f\,\mathtt{p}(\mathtt{p}(x)))\big) : [a < 1] \cdot \mathtt{Nat}[\mathrm{f}(b)] \multimap \mathtt{Nat}[\mathrm{h}(b)] \end{array}}
$$

$$
\dfrac{\varnothing \vdash^{\mathcal{E}}_{\mathrm{k}+\sum_{b<\mathrm{k}}(1+\mathrm{j}(b))} \mathsf{fibo} : [a < 1] \cdot \mathtt{Nat}[\mathrm{f}(\triangle_b^{0,a}\,\mathrm{i}(b))] \multimap \mathtt{Nat}[\mathrm{h}(\triangle_b^{0,a}\,\mathrm{i}(b))]}{\varnothing \vdash^{\mathcal{E}}_{\mathrm{k}+\sum_{b<\mathrm{k}}(1+\mathrm{j}(b))} \mathsf{fibo} : \mathtt{Nat}[\mathrm{f}(0)] \xrightarrow{1} \mathtt{Nat}[\mathrm{h}(0)]}
$$

Figure 9: A type derivation for fibo.

5. The Metatheory of $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$

In this section, some metatheoretical results about $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$ are presented. More specifically, type derivations are shown to be modifiable in many different ways, all of them leaving the underlying term unaltered. These manipulations, described in Section 5.1, form a basic toolkit which is essential to achieve the main results of this paper, namely intentional soundness and completeness (which are presented in Section 5.3 and Section 5.4, respectively). Types are preserved by call-by-value reduction, as proved in Section 5.2.

5.1. Manipulating Type Derivations

First of all, the constraints Φ in index, subtyping and typing judgments can be made stronger without altering the rest:

Lemma 5.1 (Strengthening) *If $\phi; \Psi \models_{\mathcal{E}} \Phi$, then the following implications hold:*
1. If $\phi; \Phi \models_{\mathcal{E}} \mathrm{I} \leqslant \mathrm{J}$, then $\phi; \Psi \models_{\mathcal{E}} \mathrm{I} \leqslant \mathrm{J}$;
2. If $\phi; \Phi \vdash_{\mathcal{E}} \sigma \sqsubseteq \tau$, then $\phi; \Psi \vdash_{\mathcal{E}} \sigma \sqsubseteq \tau$;
3. If $\phi; \Phi; \Gamma \vdash^{\mathcal{E}}_{\mathrm{I}} t : \sigma$, then $\phi; \Psi; \Gamma \vdash^{\mathcal{E}}_{\mathrm{I}} t : \sigma$.

Proof. Point 1. is a trivial consequence of transitivity of implication in the metalogic. Point 2. can be proved by induction on the structure of the proof of $\phi; \Phi \vdash_{\mathcal{E}} \sigma \sqsubseteq \tau$, using 1. Point 3. can be proved by induction on a proof of $\phi; \Phi; \Gamma \vdash^{\mathcal{E}}_{\mathrm{I}} t : \sigma$, using 1 and 2. □

Whatever appears on the right of $\vdash_{\mathcal{E}}$ should hold for all values of the variables in ϕ satisfying Φ, so strengthening corresponds to making the judgment weaker, which is always possible. Fresh term variables can be added to the context Γ, leaving the rest of the judgment unchanged:

Lemma 5.2 (Context Weakening) *$\phi; \Phi; \Gamma \vdash^{\mathcal{E}}_{\mathrm{I}} t : \tau$ implies $\phi; \Phi; \Gamma, \Delta \vdash^{\mathcal{E}}_{\mathrm{I}} t : \tau$.*

Proof. Again, this is an induction on the structure of a derivation for $\phi; \Phi; \Gamma \vdash^{\mathcal{E}}_{\mathrm{I}} t : \tau$. □

Please note that Δ is completely arbitrary. Another useful transformation on type derivations consists in substituting index variables for (defined) index terms.

Lemma 5.3 (Index Substitution) *If* $\phi; \Phi \vDash_{\mathcal{E}} \mathrm{I} \Downarrow$*, then the following implications hold:*

1. *If* $(a, \phi); \Phi, \Psi \vDash_{\mathcal{E}} \mathrm{J} \leqslant \mathrm{K}$*, then* $\phi; \Phi, \Psi\{\mathrm{I}/a\} \vDash_{\mathcal{E}} \mathrm{J}\{\mathrm{I}/a\} \leqslant \mathrm{K}\{\mathrm{I}/a\}$*;*
2. *If* $(a, \phi); \Phi, \Psi \vdash_{\mathcal{E}} \sigma \sqsubseteq \tau$*, then* $\phi; \Phi, \Psi\{\mathrm{I}/a\} \vdash_{\mathcal{E}} \sigma\{\mathrm{I}/a\} \sqsubseteq \tau\{\mathrm{I}/a\}$*;*
3. *If* $(a, \phi); \Phi, \Psi; \Gamma \vdash^{\mathcal{E}}_{\mathrm{J}} t : \sigma$*, then* $\phi; \Phi, \Psi\{\mathrm{I}/a\}; \Gamma\{\mathrm{I}/a\} \vdash^{\mathcal{E}}_{\mathrm{J}\{\mathrm{I}/a\}} t : \sigma\{\mathrm{I}/a\}$*.*

Proof. 1. Assume that $\phi; \Phi \vDash_{\mathcal{E}} \mathrm{I} \Downarrow$ and $(a, \phi); \Phi, \Psi \vDash_{\mathcal{E}} \mathrm{J} \leqslant \mathrm{K}$, and let ρ be an assignment satisfying $\Phi, \Psi\{\mathrm{I}/a\}$. In particular, ρ satisfies Φ, thus $[\![\mathrm{I}]\!]^{\mathcal{E}}_{\rho}$ is defined, say equal to n. For any index H, $[\![\mathrm{H}\{\mathrm{I}/a\}]\!]^{\mathcal{E}}_{\rho} = [\![\mathrm{H}]\!]^{\mathcal{E}}_{\rho, a \mapsto n}$. Hence $(\rho, a \mapsto n)$ satisfies Φ, Ψ, and then it also satisfies $\mathrm{J} \leqslant \mathrm{K}$. So $[\![\mathrm{J}\{\mathrm{I}/a\}]\!]^{\mathcal{E}}_{\rho} = [\![\mathrm{J}]\!]^{\mathcal{E}}_{\rho, a \mapsto n} \leqslant [\![\mathrm{K}]\!]^{\mathcal{E}}_{\rho, a \mapsto n} = [\![\mathrm{K}\{\mathrm{I}/a\}]\!]^{\mathcal{E}}_{\rho}$, and ρ satisfies $\mathrm{J}\{\mathrm{I}/a\} \leqslant \mathrm{K}\{\mathrm{I}/a\}$. Thus $\phi; \Phi, \Psi\{\mathrm{I}/a\} \vDash_{\mathcal{E}} \mathrm{J}\{\mathrm{I}/a\} \leqslant \mathrm{K}\{\mathrm{I}/a\}$.
2. By induction on the subtyping derivation, using 1.
3. By induction on the typing derivation, using 1. and 2. □

Observe that the only hypothesis is that $\phi; \Phi \vDash_{\mathcal{E}} \mathrm{I} \Downarrow$ (see Section 3.3 for a definition): we do not require I to be a value of a that satisfies Ψ. If it is not the constraints in $\Phi, \Psi\{\mathrm{I}/a\}$ become inconsistent, and the obtained judgments are vacuous.

5.2. Subject Reduction

Subject Reduction is a property any reasonable type system satisfies: types should be preserved along reduction. Actually, $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ is no exception:

Proposition 5.4 (Subject Reduction) *If* $t \to_v u$ *and* $\phi; \Phi; \varnothing \vdash^{\mathcal{E}}_{\mathrm{M}} t : \tau$*, then* $\phi; \Phi; \varnothing \vdash^{\mathcal{E}}_{\mathrm{M}} u : \tau$*.*

Subject Reduction can be proved in a standard way, by going through a Substitution Lemma, which only needs to be proved when the term being substituted is a *value*. Preliminary to the Substitution Lemma are two auxiliary results stating that derivations giving types to values can, if certain conditions hold, be split into two (Lemma 5.5), or put in parametric form (Lemma 5.6):

Lemma 5.5 (Splitting) *If* $\phi; \Phi; \Gamma \vdash^{\mathcal{E}}_{\mathrm{M}} v : \tau_1 \uplus \tau_2$*, then there exist two indexes* $\mathrm{N}_1, \mathrm{N}_2$*, and two contexts* Γ_1, Γ_2*, such that* $\phi; \Phi; \Gamma_i \vdash^{\mathcal{E}}_{\mathrm{N}_i} v : \tau_i$*, and* $\phi; \Phi \vDash_{\mathcal{E}} \mathrm{N}_1 + \mathrm{N}_2 \leqslant \mathrm{M}$ *and* $\phi; \Phi \vdash_{\mathcal{E}} \Gamma \sqsubseteq \Gamma_1 \uplus \Gamma_2$*.*

Proof. If v is a primitive integer $\underline{\mathtt{n}}$, the result is trivial as the only possible decomposition of a type for integers is $\mathtt{Nat}[\mathrm{I},\mathrm{J}] = \mathtt{Nat}[\mathrm{I},\mathrm{J}] \uplus \mathtt{Nat}[\mathrm{I},\mathrm{J}]$. If $v = \lambda x.t$, then its typing judgment derives from:

$$(a,\phi);(a<\mathrm{I},\Phi);\Delta,x:\sigma \vdash^{\mathcal{E}}_{\mathrm{K}} t:\tau; \tag{3}$$

$$\phi;\Phi \vdash_{\mathcal{E}} \Gamma \sqsubseteq \sum_{a<\mathrm{I}} \Delta; \tag{4}$$

with $\tau_1 \uplus \tau_2 = [a<\mathrm{I}]\cdot\sigma \multimap \tau$ and $\mathrm{M} = \mathrm{I} + \sum_{a<\mathrm{I}} \mathrm{K}$. Hence $\mathrm{I} = \mathrm{I}_1 + \mathrm{I}_2$, and $\tau_1 = [a<\mathrm{I}_1]\cdot\sigma \multimap \tau$, and $\tau_2 = [a<\mathrm{I}_2]\cdot\sigma\{\mathrm{I}_1+a/a\} \multimap \tau\{\mathrm{I}_1+a/a\}$. Since $(a,\phi);(a<\mathrm{I}_1,\Phi) \vDash_{\mathcal{E}} (a<\mathrm{I},\Phi)$, we can strength the hypothesis in (3) by Lemma 5.1 and derive

$$\frac{(a,\phi);(a<\mathrm{I}_1,\Phi);\Delta,x:\sigma \vdash^{\mathcal{E}}_{\mathrm{K}} t:\tau}{\phi;\Phi;\sum_{a<\mathrm{I}_1}\Delta \vdash^{\mathcal{E}}_{\mathrm{I}_1+\sum_{a<\mathrm{I}_1}\mathrm{K}} \lambda x.t : [a<\mathrm{I}_1]\cdot\sigma \multimap \tau}$$

On the other hand, we can substitute a with $a+\mathrm{I}_1$ in (3) by Lemma 5.3, and derive

$$\frac{(a,\phi);(a<\mathrm{I}_2,\Phi);\Delta\{a+\mathrm{I}_1/a\},x:\sigma\{a+\mathrm{I}_1/a\} \vdash^{\mathcal{E}}_{\mathrm{K}\{a+\mathrm{I}_1/a\}} t:\tau\{a+\mathrm{I}_1/a\}}{\phi;\Phi;\sum_{a<\mathrm{I}_2}\Delta\{a+\mathrm{I}_1/a\} \vdash^{\mathcal{E}}_{\mathrm{N}_2} \lambda x.t : [a<\mathrm{I}_2]\cdot\sigma\{a+\mathrm{I}_1/a\} \multimap \tau\{a+\mathrm{I}_1/a\}}$$

where $\mathrm{N}_2 = \mathrm{I}_2 + \sum_{a<\mathrm{I}_2}\mathrm{K}\{a+\mathrm{I}_1/a\}$. Hence we can conclude with $\Gamma_1 = \sum_{a<\mathrm{I}_1}\Delta$, $\Gamma_2 = \sum_{a<\mathrm{I}_2}\Delta\{a+\mathrm{I}_1/a\}$, and $\mathrm{N}_1 = \mathrm{I}_1 + \sum_{a<\mathrm{I}_1}\mathrm{K}$. Now, if $v = \mathtt{fix}\ x.t$, then its typing judgment derives from

$$(b,\phi);(b<\mathrm{H},\Phi);\Delta,x:[a<\mathrm{I}]\cdot A \vdash^{\mathcal{E}}_{\mathrm{J}} t:[a<1]\cdot B \tag{5}$$

$$\phi;\Phi \vDash_{\mathcal{E}} \mathrm{H} \geqslant \textcircled{\triangle}^{0,\mathrm{K}}_{b}\,\mathrm{I} \tag{6}$$

$$(a,b,\phi);(a<\mathrm{I},b<\mathrm{H},\Phi) \vdash_{\mathcal{E}} B\{\mathtt{0}/a\}\{\textcircled{\triangle}^{b+1,a}_{b}\,\mathrm{I}+b+1/b\} \sqsubseteq A \tag{7}$$

$$(a,\phi);(a<\mathrm{K},\Phi) \vdash_{\mathcal{E}} B\{\mathtt{0}/a\}\{\textcircled{\triangle}^{0,a}_{b}\,\mathrm{I}/b\} \sqsubseteq C \tag{8}$$

$$\phi;\Phi \vdash_{\mathcal{E}} \Gamma \sqsubseteq \textstyle\sum_{b<\mathrm{H}}\Delta \tag{9}$$

with $\tau_1 \uplus \tau_2 = [a<\mathrm{K}]\cdot C$, and $\mathrm{M} = \mathrm{H} + \sum_{b<\mathrm{H}}\mathrm{J}$. Hence $\mathrm{K} = K_1 + \mathrm{K}_2$, with $\tau_1 = [a<\mathrm{K}_1]\cdot C$, and $\tau_2 = [a<\mathrm{K}_2]\cdot C\{a+\mathrm{K}_1/a\}$. Let $\mathrm{H}_1 = \textcircled{\triangle}^{0,\mathrm{K}_1}_{b}\,\mathrm{I}$ and $\mathrm{H}_2 = \textcircled{\triangle}^{\mathrm{H}_1,\mathrm{K}_2}_{b}\,\mathrm{I}$. Then $\mathrm{H}_1 + \mathrm{H}_2 = \textcircled{\triangle}^{0,\mathrm{K}}_{b}\,\mathrm{I}$, and H_2 is also equal to $\textcircled{\triangle}^{0,\mathrm{K}_2}_{b}\,\mathrm{I}\{\mathrm{H}_1+b/b\}$. Just like the previous case, we can strengthen the hypothesis in (5), (7) and (8) and derive

$$\frac{\begin{array}{c}(b,\phi);(b<\mathrm{H}_1,\Phi);\Delta,x:[a<\mathrm{I}]\cdot A\vdash^{\mathcal{E}}_{\mathrm{J}} t:[a<1]\cdot B\\ (a,b,\phi);(a<\mathrm{I},b<\mathrm{H}_1,\Phi)\vdash_{\mathcal{E}} B\{0/a\}\{\textcircled{\triangle}_b^{b+1,a}\,\mathrm{I}+b+1/b\}\sqsubseteq A\\ (a,\phi);(a<\mathrm{K}_1,\Phi)\vdash_{\mathcal{E}} B\{0/a\}\{\textcircled{\triangle}_b^{0,a}\,\mathrm{I}/b\}\sqsubseteq C\end{array}}{\phi;\Phi;\sum_{b<\mathrm{H}_1}\Delta\vdash^{\mathcal{E}}_{\mathrm{H}_1+\sum_{b<\mathrm{H}_1}\mathrm{J}}\ \mathtt{fix}\ x.t:[a<\mathrm{K}_1]\cdot C}$$

Moreover, if we substitute b with $b+\mathrm{H}_1$ in (7) and we strengthen the constraints (since (6) implies $\phi;\Phi,b<\mathrm{H}_2\vDash_{\mathcal{E}}\Phi,b+\mathrm{H}_1<\mathrm{H}$), we get

$$(a,b,\phi);(a<\mathrm{I},b<\mathrm{H}_2,\Phi)\vdash_{\mathcal{E}} B\{0/a\}\{\textcircled{\triangle}_b^{b+1,a}\,\mathrm{I}+b+1/b\}\{\mathrm{H}_1+b/b\}\sqsubseteq A\{\mathrm{H}_1+b/b\}.$$

But $\big(\textcircled{\triangle}_b^{b+1,a}\,\mathrm{I}+b+1\big)\{\mathrm{H}_1+b/b\}=\textcircled{\triangle}_b^{\mathrm{H}_1+b+1,a}\,\mathrm{I}+\mathrm{H}_1+b+1$ and $\textcircled{\triangle}_b^{\mathrm{H}_1+b+1,a}\,\mathrm{I}=\textcircled{\triangle}_b^{b+1,a}(\mathrm{I}\{\mathrm{H}_1+b/b\})$. Hence $B\{0/a\}\{\textcircled{\triangle}_b^{b+1,a}\,\mathrm{I}+b+1/b\}\{\mathrm{H}_1+b/b\}=B\{\mathrm{H}_1+b/b\}\{0/a\}\{\textcircled{\triangle}_b^{b+1,a}(\mathrm{I}\{\mathrm{H}_1+b/b\})+b+1/b\}$. In the same way we can substitute a with $a+\mathrm{K}_1$ in (8):

$$(a,\phi);(a<\mathrm{K}_2,\Phi)\vdash_{\mathcal{E}} B\{0/a\}\{\textcircled{\triangle}_b^{0,a+\mathrm{K}_1}\,\mathrm{I}/b\}\sqsubseteq C\{a+\mathrm{K}_1/a\}.$$

But $\textcircled{\triangle}_b^{0,a+\mathrm{K}_1}\,\mathrm{I}=\mathrm{H}_1+\textcircled{\triangle}_b^{\mathrm{H}_1,a}\,\mathrm{I}=\mathrm{H}_1+\textcircled{\triangle}_b^{0,a}\,\mathrm{I}\{\mathrm{H}_1+b/b\}$, and then $B\{0/a\}\{\textcircled{\triangle}_b^{0,a+\mathrm{K}_1}\,\mathrm{I}/b\}$ is equivalent to $B\{\mathrm{H}_1+b/b\}\{0/a\}\{\textcircled{\triangle}_b^{0,a}\,\mathrm{I}\{\mathrm{H}_1+b/b\}/b\}$. Finally, by substituting also b with $b+\mathrm{H}_1$ in (5) we can derive

$$\frac{\begin{array}{c}(b,\phi)\ ;\ (b<\mathrm{H}_2,\Phi)\ ;\ \Delta\{\mathrm{H}_1+b/b\}\ ,\ x:([a<\mathrm{I}]\cdot A)\{\mathrm{H}_1+b/b\}\ \vdash^{\mathcal{E}}_{\mathrm{J}\{\mathrm{H}_1+b/b\}}\ t\ :\ [a<1]\cdot B\{\mathrm{H}_1+b/b\}\\ (a,b,\phi)\ ;\ (a<\mathrm{I},b<\mathrm{H}_2,\Phi)\ \vdash_{\mathcal{E}}\ B\{\mathrm{H}_1+b/b\}\{0/a\}\{\textcircled{\triangle}_b^{b+1,a}(\mathrm{I}\{\mathrm{H}_1+b/b\})+b+1/b\}\ \sqsubseteq\ A\{\mathrm{H}_1+b/b\}\\ (a,\phi)\ ;\ (a<\mathrm{K}_2,\Phi)\ \vdash_{\mathcal{E}}\ B\{\mathrm{H}_1+b/b\}\{0/a\}\{\textcircled{\triangle}_b^{0,a}\,\mathrm{I}\{\mathrm{H}_1+b/b\}/b\}\ \sqsubseteq\ C\{a+\mathrm{K}_1/a\}\end{array}}{\phi\ ;\ \Phi\ ;\ \textstyle\sum_{b<\mathrm{H}_2}\Delta\{\mathrm{H}_1+b/b\}\ \vdash^{\mathcal{E}}_{\mathrm{H}_2+\sum_{b<\mathrm{H}_2}\mathrm{J}\{\mathrm{H}_1+b/b\}}\ \mathtt{fix}\ x.t\ :\ [a<\mathrm{K}_2]\cdot C\{a+\mathrm{K}_1/a\}}$$

Thus we can conclude with $\Gamma_1=\sum_{a<\mathrm{H}_1}\Delta$, $\Gamma_2=\sum_{a<\mathrm{H}_2}\Delta\{a+\mathrm{H}_1/a\}$, $\mathrm{N}_1=\mathrm{H}_1+\sum_{a<\mathrm{H}_1}\mathrm{J}$ and $\mathrm{N}_2=\mathrm{H}_2+\sum_{a<\mathrm{H}_2}\mathrm{J}\{a+\mathrm{H}_1/a\}$. □

If splitting tells us that that the *sum* of two types can be split, parametric splitting allows to handle *bounded sums*:

Lemma 5.6 (Parametric Splitting) *If $\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{\mathrm{M}} v:\sum_{c<\mathrm{J}}\sigma$ is derivable, then there exist an index N and a context Δ such that one can derive $c,\phi;c<\mathrm{J},\Phi;\Delta\vdash^{\mathcal{E}}_{\mathrm{N}} v:\sigma$, and $\phi;\Phi\vDash_{\mathcal{E}}\sum_{c<\mathrm{J}}\mathrm{N}\leqslant\mathrm{M}$ and $\phi;\Phi\vdash_{\mathcal{E}}\Gamma\sqsubseteq\sum_{c<\mathrm{J}}\Delta$.*

Proof. The proof uses the same technique as for Lemma 5.5. If v is a lambda abstraction or a fixpoint, then $\sum_{c<\mathrm{J}}\sigma$ is on the form $[a<\sum_{c<\mathrm{J}}\mathrm{L}]\cdot C$, where $[a<\mathrm{L}]\cdot C\{a+\sum_{d<c}\mathrm{L}\{d/c\}/a\}=\sigma$. Then the result also follows

from Strengthening (Lemma 5.1) and Index Substitution (Lemma 5.3): for the lambda abstraction, substitute a with $a + \sum_{d<c} \mathrm{L}\{d/c\}$ in (3). For the fixpoint consider the index H_1 satisfying the equations $\mathrm{H}_1\{\mathtt{0}/c\} = \bigtriangleup_b^{\mathtt{0},\mathrm{L}\{\mathtt{0}/c\}} \mathrm{I}$ and $\mathrm{H}_1\{\mathrm{i}+\mathtt{1}/c\} = \bigtriangleup_b^{\mathrm{H}\{\mathrm{i}/c\},\mathrm{L}\{\mathrm{i}+\mathtt{1}/c\}} \mathrm{I}$. Then substitute b with $b + \sum_{d<c} \mathrm{H}_1\{d/c\}$ (and add the constraint $c < \mathrm{J}$ in the context) in (5) and (7), and substitute a with $a + \sum_{d<c} \mathrm{L}\{d/c\}$ in (8) to derive the result. □

One can already realize *why* these results are crucial for subject reduction: whenever the substituted value flows through a type derivation, there are various places where its type changes, namely when it reaches instances of the typing rules (*App*), (⊸), (*If*) and (*Rec*): in all these cases the type derivation for the value must be modified, and the splitting lemmas certify that this is possible, indeed. We can this way reach the key intermediate result:

Lemma 5.7 (Substitution) *If* $\phi; \Phi; \Gamma, x : \sigma \vdash^{\mathcal{E}}_{\mathrm{M}} t : \tau$ *and* $\phi; \Phi; \varnothing \vdash^{\mathcal{E}}_{\mathrm{N}} v : \sigma$ *are both derivable, then there is an index* K *such that* $\phi; \Phi; \Gamma \vdash^{\mathcal{E}}_{\mathrm{K}} t[x := v] : \tau$ *and* $\phi; \Phi \vDash_{\mathcal{E}} \mathrm{K} \leqslant \mathrm{M} + \mathrm{N}$.

Proof. The proof goes by induction on the derivation of the typing judgment $\phi; \Phi; \Gamma, x : \sigma \vdash^{\mathcal{E}}_{\mathrm{M}} t : \tau$, making intense use of Lemma 5.5 and Lemma 5.6. □

Given Lemma 5.7, proving Proposition 5.4 is routine: the only two nontrivial cases are those where the fired redex is a β-redex or the unfolding of a recursively-defined function, and both consist in a substitution.

Observe how Subject Reduction already embeds a form of *extensional* soundness for $\mathsf{d}\ell\mathsf{PCF_V}$: simply, types are preserved by reduction. As an example, if one builds a type derivation for $\vdash^{\mathcal{E}}_{\mathrm{I}} t : \mathtt{Nat}[2, 7]$, then the normal form of t (if it exists) is guaranteed to be a constant between 2 and 7. Observe, on the other hand, that nothing is known about the *complexity* of the underlying computational process yet, since the weight I does not necessarily decrease along reduction, although it cannot increase. In which sense, then, I is a measure of the complexity of evaluating the underlying term t? This is the topic of the following section.

5.3. Intentional Soundness

In this section, we prove the following result:

$$\frac{\phi;\Phi \vdash_{\mathcal{E}} \sigma \sqsubseteq \tau \quad \phi;\Phi \vDash_{\mathcal{E}} \mathrm{I} \Downarrow}{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \diamond : (\sigma,\tau)}$$

$$\frac{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{J}} \mathsf{c} : \sigma\{0/a\} \quad \phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{K}} \rho : (\tau\{0/a\},\varphi) \quad \phi;\Phi \vDash_{\mathcal{E}} \mathrm{I} = \mathrm{J} + \mathrm{K}}{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \mathsf{arg}(\mathsf{c}\,,\,\rho) : ([a < 1] \cdot (\sigma \multimap \tau),\varphi)}$$

$$\frac{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{J}} \mathsf{v} : [a < 1] \cdot (\sigma \multimap \tau) \quad \phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{K}} \rho : (\tau\{0/a\},\varphi) \quad \phi;\Phi \vDash_{\mathcal{E}} \mathrm{I} = \mathrm{J} + \mathrm{K}}{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \mathsf{fun}(\mathsf{v}\,,\,\rho) : (\sigma\{0/a\},\varphi)}$$

$$\frac{\begin{array}{l}\phi;\mathrm{N} = 0,\Phi \;\; \vdash^{\mathcal{E}}_{\mathrm{J}} \langle\, t\,;\,\xi\,\rangle : \sigma \\ \phi;\mathrm{M} \geqslant 1,\Phi \;\; \vdash^{\mathcal{E}}_{\mathrm{J}} \langle\, u\,;\,\xi\,\rangle : \sigma \qquad \phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{K}} \rho : (\sigma,\tau) \quad \phi;\Phi \vDash_{\mathcal{E}} \mathrm{I} = \mathrm{J} + \mathrm{K}\end{array}}{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \mathsf{fork}(t\,,\,u\,,\,\xi\,,\,\rho) : (\mathtt{Nat}[\mathrm{M},\mathrm{N}],\tau)}$$

$$\frac{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \pi : (\mathtt{Nat}[\mathrm{M}+1,\mathrm{N}+1],\tau)}{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \mathsf{s}(\pi) : (\mathtt{Nat}[\mathrm{M},\mathrm{N}],\tau)} \qquad \frac{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \pi : (\mathtt{Nat}[\mathrm{M}-1,\mathrm{N}-1],\tau)}{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \mathsf{p}(\pi) : (\mathtt{Nat}[\mathrm{M},\mathrm{N}],\tau)}$$

Figure 10: $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$: Lifting Typing to Stacks

Theorem 5.8 (Intensional soundness) *For any term t, if $\vdash^{\mathcal{E}}_{\mathrm{H}} t : \mathtt{Nat}[\mathrm{I},\mathrm{J}]$, then $t \Downarrow^n \underline{\mathtt{m}}$ where $n \leqslant |t| \cdot (\llbracket \mathrm{H} \rrbracket^{\mathcal{E}} + 1)$ and $\llbracket \mathrm{I} \rrbracket^{\mathcal{E}} \leqslant m \leqslant \llbracket \mathrm{J} \rrbracket^{\mathcal{E}}$.*

Roughly speaking, this means that $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ also gives us some sensible information about the time complexity of evaluating typable PCF programs. The path towards Theorem 5.8 is not straightforward: it is necessary to lift $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ to a type system for closures, environments and processes, as defined in Section 3.2. Actually, the type system can be easily generalized to closures by the rule below:

$$\frac{\phi;\Phi;x_1 : \sigma_1,\ldots,x_n : \sigma_n \vdash^{\mathcal{E}}_{\mathrm{K}} t : \tau \qquad \begin{array}{l}\phi;\Phi \;\; \vdash^{\mathcal{E}}_{\mathrm{J}_i} \mathsf{v}_i : \sigma_i \text{ (for all } i) \\ \phi;\Phi \;\; \vDash_{\mathcal{E}} \mathrm{I} = \mathrm{K} + \sum_{i \leqslant n} \mathrm{J}_i\end{array}}{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \langle\, t\,;\,\{x_1 \mapsto \mathsf{v}_1;\cdots;x_n \mapsto \mathsf{v}_n\}\,\rangle : \tau}$$

We might write $\phi;\Phi \vdash^{\mathcal{E}}_{\mathbf{J}} \xi : \Gamma$ when $\Gamma = x_1 : \sigma_1,\ldots,x_n : \sigma_n$ and $\xi = \{x_1 \mapsto \mathsf{v}_1;\cdots;x_n \mapsto \mathsf{v}_n\}$ and $\mathbf{J} = (\mathrm{J}_1,\ldots,\mathrm{J}_n)$ and, for every $i \leqslant n$, $\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{J}_i} \mathsf{v}_i : \sigma_i$. Hence the typing rule for a closure can also be written:

$$\frac{\phi;\Phi;\Gamma \vdash^{\mathcal{E}}_{\mathrm{K}} t : \tau \qquad \phi;\Phi \vdash^{\mathcal{E}}_{\mathbf{J}} \xi : \Gamma \qquad \phi;\Phi \vDash_{\mathcal{E}} \mathrm{I} = \mathrm{K} + \sum \mathbf{J}}{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \langle\, t\,;\,\xi\,\rangle : \tau}$$

Lifting everything to stacks, on the other hand, requires more work (see Figure 10). We say that a stack π is $(\phi;\Phi)$-*acceptable* for σ with type τ and cost I (notation: $\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \pi : (\sigma,\tau)$) when it interacts well with closures of type σ to produce a process of type τ. Finally, a *process* can be typed as follows:

$$\frac{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{K}} \mathsf{c} : \sigma \qquad \phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{J}} \pi : (\sigma,\tau) \qquad \phi;\Phi \vDash_{\mathcal{E}} \mathrm{I} = \mathrm{K} + \mathrm{J}}{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \mathsf{c} \star \pi : \tau}$$

This lifted type system is sound *w.r.t.* subtyping:

Lemma 5.9 (Subtyping is Derivable) *1. The following typing rule for closures is derivable:*

$$\frac{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \mathsf{c} : \sigma \qquad \phi;\Phi \vDash_{\mathcal{E}} \mathrm{I} \leqslant \mathrm{J} \qquad \phi;\Phi \vdash_{\mathcal{E}} \sigma \sqsubseteq \tau}{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{J}} \mathsf{c} : \tau}$$

2. The following typing rule for stacks is derivable:

$$\frac{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \pi : (\sigma,\tau) \quad \phi;\Phi \vdash_{\mathcal{E}} \varrho \sqsubseteq \sigma \quad \phi;\Phi \vdash_{\mathcal{E}} \tau \sqsubseteq \varphi \quad \phi;\Phi \vDash_{\mathcal{E}} \mathrm{I} \leqslant \mathrm{J}}{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{J}} \pi : (\varrho,\varphi)}$$

3. The following typing rule for processes is derivable:

$$\frac{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \mathrm{P} : \sigma \qquad \phi;\Phi \vDash_{\mathcal{E}} \mathrm{I} \leqslant \mathrm{J} \qquad \phi;\Phi \vdash_{\mathcal{E}} \sigma \sqsubseteq \tau}{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{J}} \mathrm{P} : \tau}$$

Proof. The first point directly follows from the typing rule *Subs* for terms. The second one is proved by induction on the stack typing derivation, using 1. The last point follows from the first two. □

In the meantime, the notion of weight has been lifted to processes too, with the hope that it strictly decreases at every evaluation step. Unfortunately, this does not hold: sometimes, evaluation leaves the weight of a process unchanged. However, in that case another parameter is guaranteed to decrease, namely the process *size*. The size $|\mathrm{P}|$ of a process $\mathrm{P} = \mathsf{c} \star \pi$ is defined as $|\mathsf{c}| + |\pi|$, where:

- The size $|\mathsf{c}|$ of a closure $\langle\, t\,;\,\xi\,\rangle$ is the *multiplicative* size of t (see Section 3.1).
- The size of $|\pi|$ is the sum of the sizes of all closures appearing in π plus the number of occurrences of symbols (different from $\diamond$ and fun) in π.

Term Multiplicative Size

$$\begin{array}{rcl@{\qquad}rcl} \|x\| & = & 2; & \|\underline{\mathtt{n}}\| & = & \|\lambda x.t\| = \| \mathtt{fix}\ x.t\| = 0; \\ \|tu\| & = & \|t\| + \|u\| + 2; & \|\mathtt{s}(t)\| & = & \|\mathtt{p}(t)\| = \|t\| + 2; \\ \multicolumn{3}{r@{\qquad}}{} & \multicolumn{3}{l}{\| \mathtt{ifz}\ t\ \mathtt{then}\ u\ \mathtt{else}\ s\| = \|t\| + \|u\| + \|s\| + 2;} \end{array}$$

Closure Size

$$|\langle\, t\,;\, \xi\,\rangle| \;=\; \|t\|;$$

Stack Size

$$\begin{array}{rcl@{\qquad}rcl} |\diamond| & = & 0; & |\mathsf{fun}(\mathsf{v}\,,\,\pi)| & = & |\mathsf{v}| + |\pi|; \\ |\mathsf{s}(\pi)| & = & |\mathsf{p}(\pi)| = |\pi| + 1; & |\mathsf{arg}(\mathsf{c}\,,\,\pi)| & = & |\mathsf{c}| + |\pi| + 1; \\ \multicolumn{6}{l}{|\mathsf{fork}(t\,,\,u\,,\,\xi\,,\,\pi)| = \|t\| + \|u\| + |\pi| + 1;} \end{array}$$

Process Size

$$|\mathsf{c} \star \pi| \;=\; |\mathsf{c}| + |\pi|.$$

Figure 11: Size of Processes.

The formal definition of $|\mathsf{c} \star \pi|$ is in Figure 11. The size of a process decreases by any evaluation steps, except the ones performing a substitution (1) and (2). However, these two reduction rules make the *weight* of a process to decrease, as formalized by the following proposition. By the way, these are the cases in which a box is opened up in the underlying linear logic proof.

Proposition 5.10 (Weighted Subject Reduction) *Assume* $\mathrm{P} \succ \mathrm{R}$ *and* $\phi; \Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \mathrm{P} : \tau$. *Then* $\phi; \Phi \vdash^{\mathcal{E}}_{\mathrm{J}} \mathrm{R} : \tau$ *and:*

- *either* $\phi; \Phi \models_{\mathcal{E}} \mathrm{I} = \mathrm{J}$ *and* $|\mathrm{P}| > |\mathrm{R}|$,
- *or* $\phi; \Phi \models_{\mathcal{E}} \mathrm{I} > \mathrm{J}$ *and* $|\mathrm{P}| + |s| > |\mathrm{R}|$, *where* s *is a term appearing in* P.

Proof. First, if $\mathrm{P} \succ \mathrm{R}$ with a non substitution rule (any rule of Figure 4 or Figure 5 except (1) and (2)), then it is easy to check that $|\mathrm{P}| > |\mathrm{R}|$. Moreover, in all these cases P and R have the same type and the same weight. We detail some cases:

- If $\mathrm{P} = \mathsf{v} \star \mathsf{arg}(\mathsf{c}\,,\,\pi) \succ \mathsf{c} \star \mathsf{fun}(\mathsf{v}\,,\,\pi) = \mathrm{R}$, then the typing of P derives from

$$\dfrac{\begin{array}{l}\phi;\Phi\vDash_{\mathcal{E}} \mathrm{I}=\mathrm{J}+\mathrm{K}\\ \phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{K}} \mathsf{v}:[a<1]\cdot(\sigma_0\multimap\tau_0)\end{array}\qquad\dfrac{\begin{array}{l}\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{H}} \mathsf{c}:\sigma_0\{0/a\}\\ \phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{L}} \pi:(\tau_0\{0/a\},\tau)\\ \phi;\Phi\vDash_{\mathcal{E}} \mathrm{J}=\mathrm{H}+\mathrm{L}\end{array}}{\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{J}} \mathsf{arg}(\mathsf{c}\,,\,\pi):([a<1]\cdot(\sigma_0\multimap\tau_0),\tau)}}{\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{I}} \mathsf{v}\ \star\ \mathsf{arg}(\mathsf{c}\,,\,\pi):\tau}$$

Hence we can derive for R:

$$\dfrac{\begin{array}{l}\phi;\Phi\vDash_{\mathcal{E}} \mathrm{I}=\mathrm{H}+\mathrm{L}+\mathrm{K}\\ \phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{H}} \mathsf{c}:\sigma_0\{0/a\}\end{array}\qquad\dfrac{\begin{array}{l}\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{K}} \mathsf{v}:[a<1]\cdot(\sigma_0\multimap\tau_0)\\ \phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{L}} \pi:(\tau_0\{0/a\},\tau)\end{array}}{\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{L+K}} \mathsf{fun}(\mathsf{v}\,,\,\pi):(\sigma_0\{0/a\},\tau)}}{\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{I}} \mathsf{c}\ \star\ \mathsf{fun}(\mathsf{v}\,,\,\pi):\tau}$$

- If $\mathrm{P}=\langle\, tu\,;\,\xi\,\rangle\star\pi\ \succ\ \langle\, t\,;\,\xi\,\rangle\star\mathsf{arg}(\langle\, u\,;\,\xi\,\rangle,\pi)\ =\mathrm{R}$, then the typing of P derives from

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{l}\phi;\Phi;\Gamma_1\vdash^{\mathcal{E}}_{\mathrm{K}_1} t:[a<1]\cdot\sigma_0\multimap\tau_0\\ \phi;\Phi;\Gamma_2\vdash^{\mathcal{E}}_{\mathrm{K}_2} u:\sigma_0\{0/a\}\end{array}}{\phi;\Phi;\Gamma_1\uplus\Gamma_2\vdash^{\mathcal{E}}_{\mathrm{K}_1+\mathrm{K}_2} tu:\tau_0\{0/a\}}\quad\begin{array}{l}\phi;\Phi\vdash_{\mathcal{E}}\Delta\sqsubseteq\Gamma_1\uplus\Gamma_2\\ \phi;\Phi\vdash_{\mathcal{E}}\sigma\sqsubseteq\tau_0\{0/a\}\\ \phi;\Phi\vDash_{\mathcal{E}}\mathrm{K}_1+\mathrm{K}_2\leqslant\mathrm{H}\end{array}}{\phi;\Phi;\Delta\vdash^{\mathcal{E}}_{\mathrm{H}} tu:\sigma}\quad\begin{array}{l}\phi;\Phi\vdash^{\mathcal{E}}_{\mathbf{J}}\xi:\Delta\\ \phi;\Phi\vDash_{\mathcal{E}}\mathrm{H}+\sum\mathbf{J}=\mathrm{H}_1\end{array}}{\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{H}_1}\langle\, tu\,;\,\xi\,\rangle:\sigma}\quad\begin{array}{l}\phi;\Phi\ \ \vdash^{\mathcal{E}}_{\mathrm{L}}\pi:(\sigma,\tau)\\ \phi;\Phi\ \ \vDash_{\mathcal{E}}\mathrm{I}=\mathrm{L}+\mathrm{H}_1\end{array}}{\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{I}}\langle\, tu\,;\,\xi\,\rangle\star\pi:\tau}$$

In particular, since subtyping is derivable for closures (Lemma 5.9), $\phi;\Phi\vdash^{\mathcal{E}}_{\mathbf{J}}\xi:\Gamma_1\uplus\Gamma_2$. By Lemma 5.5 (that can be trivially extended to closures), it means that there are some vectors of indexes $\mathbf{M},\mathbf{N}$ such that

$$\begin{array}{l}\phi;\Phi\vdash^{\mathcal{E}}_{\mathbf{M}}\xi:\Gamma_1\\ \phi;\Phi\vdash^{\mathcal{E}}_{\mathbf{N}}\xi:\Gamma_2\\ \phi;\Phi\vDash_{\mathcal{E}}\mathbf{M}+\mathbf{N}=\mathbf{J}\end{array}$$

(where, as usual, the sum of vectors has to be understood componentwise). In the same way, since subtyping is derivable for stacks (Lemma 5.9), then $\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{L}}\pi:(\sigma,\tau)$ and $\phi;\Phi\vdash_{\mathcal{E}}\sigma\sqsubseteq\tau_0\{0/a\}$ entail

$$\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{L}}\pi:(\tau_0\{0/a\},\tau)$$

Hence we can derive:

$$\dfrac{\dfrac{\phi;\Phi;\Gamma_1 \vdash^{\mathcal{E}}_{\mathrm{K}_1} t:[a<1]\cdot\sigma_0\multimap\tau_0 \qquad \phi;\Phi \vdash^{\mathcal{E}}_{\mathbf{M}} \xi:\Gamma_1}{\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{K}_1+\sum\mathbf{M}}\langle\, t\,;\,\xi\,\rangle:[a<1]\cdot\sigma_0\multimap\tau_0} \qquad \dfrac{\dfrac{\phi;\Phi;\Gamma_2 \vdash^{\mathcal{E}}_{\mathrm{K}_2} u:\sigma_0\{0/a\} \qquad \phi;\Phi \vdash^{\mathcal{E}}_{\mathbf{N}} \xi:\Gamma_2}{\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{K}_2+\sum\mathbf{N}}\langle\, u\,;\,\xi\,\rangle:\sigma_0\{0/a\}} \qquad \phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{L}}\pi:(\tau_0\{0/a\},\tau)}{\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{L}+\mathrm{K}_2+\sum\mathbf{N}}\mathsf{arg}(\langle\, u\,;\,\xi\,\rangle,\pi):([a<1]\cdot\sigma_0\multimap\tau_0,\tau)}}{\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{K}_1+\sum\mathbf{M}+\mathrm{L}+\mathrm{K}_2+\sum\mathbf{N}}\langle\, t\,;\,\xi\,\rangle \star \mathsf{arg}(\langle\, u\,;\,\xi\,\rangle,\pi):\tau}$$

Finally we can conclude that $\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \mathrm{R}:\tau$ with Lemma 5.9 since $\phi;\Phi \vDash_{\mathcal{E}} \mathrm{I} \geqslant \mathrm{L}+\mathrm{K}_1+\mathrm{K}_2+\sum\mathbf{M}+\sum\mathbf{N}$.

Now, if $\mathrm{P} \succ \mathrm{R}$ with a substitution rule (1) or (2), then the weight of P is strictly higher than the one of R:

- If $\mathrm{P} = \mathsf{v} \star \mathsf{fun}(\langle\, \lambda x.t\,;\,\xi\,\rangle,\pi)$ and $\mathrm{R} = \langle\, t\,;\,(x\mapsto\mathsf{v})\cdot\xi\,\rangle \star \pi$, then the typing of P comes from a derivation of the form:

$$\dfrac{\phi;\Phi \vdash^{\mathcal{E}}_{\mathrm{K}} \mathsf{v}:\sigma\{0/a\} \qquad \dfrac{\dfrac{\dfrac{a,\phi;a<1,\Phi;\Gamma,x:\sigma\vdash^{\mathcal{E}}_{\mathrm{M}} t:\sigma_1}{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{1+\mathrm{M}\{0/a\}}\lambda x.t:[a<1]\cdot(\sigma\multimap\sigma_1)} \qquad \phi;\Phi\vdash^{\mathcal{E}}_{\mathbf{J}}\xi:\Gamma}{\phi;\Phi\vdash^{\mathcal{E}}_{1+\mathrm{M}\{0/a\}+\sum\mathbf{J}}\langle\,\lambda x.t\,;\,\xi\,\rangle:[a<1]\cdot(\sigma\multimap\sigma_1)} \qquad \phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{H}}\pi:(\sigma_1\{0/a\},\tau)}{\phi;\Phi\vdash^{\mathcal{E}}_{1+\mathrm{M}\{0/a\}+\sum\mathbf{J}+\mathrm{H}}\mathsf{fun}(\langle\,\lambda x.t\,;\,\xi\,\rangle,\pi):(\sigma\{0/a\},\tau)}}{\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{I}}\mathsf{v}\star\mathsf{fun}(\langle\,\lambda x.t\,;\,\xi\,\rangle,\pi):\tau}$$

  (with $\phi;\Phi \vDash_{\mathcal{E}} \mathrm{K}+1+\mathrm{M}\{0/a\}+\sum\mathbf{J}+\mathrm{H} = \mathrm{I}$). If $a,\phi;a<1,\Phi;\Gamma,x:\sigma\vdash^{\mathcal{E}}_{\mathrm{M}} t:\sigma_1$ is derivable, then $\phi;\Phi;\Gamma,x:\sigma\{0/a\}\vdash^{\mathcal{E}}_{\mathrm{M}\{0/a\}} t:\sigma_1\{0/a\}$ is derivable too (see Lemma 5.3). Hence we can derive:

$$\dfrac{\begin{array}{rl}\phi;\Phi;\Gamma,x:\sigma\{0/a\} & \vdash^{\mathcal{E}}_{\mathrm{M}\{0/a\}} t:\sigma_1\{0/a\}\\ \phi;\Phi & \vdash^{\mathcal{E}}_{\mathbf{J}}\xi:\Gamma\\ \phi;\Phi & \vdash^{\mathcal{E}}_{\mathrm{K}}\mathsf{v}:\sigma\{0/a\}\end{array}}{\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{M}\{0/a\}+\sum\mathbf{J}+\mathrm{K}}\langle\, t\,;\,(x\mapsto\mathsf{v})\cdot\xi\,\rangle:\sigma_1\{0/a\} \qquad \phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{H}}\pi:(\sigma_1\{0/a\},\tau)}{\phi;\Phi\vdash^{\mathcal{E}}_{\mathrm{J}}\langle\, t\,;\,(x\mapsto\mathsf{v})\cdot\xi\,\rangle\star\pi:\tau}$$

  (with $\phi;\Phi \vDash_{\mathcal{E}} \mathrm{J} = \mathrm{K}+\mathrm{M}\{0/a\}+\sum\mathbf{J}+\mathrm{H}$, and thus $\phi;\Phi\vDash_{\mathcal{E}}\mathrm{J}=\mathrm{I}-1$). Moreover, $|\mathrm{P}| = |\mathsf{v}|+\|\lambda x.t\|+|\pi| = |\mathsf{v}|+|\pi|$, while $|\mathrm{R}| = \|t\|+|\pi|$. Hence, $|\mathrm{P}|+|t| \geqslant |\mathrm{R}|$ (as $|u| \geqslant \|u\|$ for any term u).

□

Splitting and parametric splitting play a crucial role here, once appropriately generalized to value closures.

Remark 5.1 *The proof of this theorem formalizes the intuition that the weight of a program is the number of substitutions performed during its* CBV *evaluation. More precisely, if a typed process* P *has weight* I *and* $\mathrm{P} \succ \mathrm{R}$*, then* R *can be typed with weight* J *such that*

- $\mathrm{J} = \mathrm{I} - \mathtt{1}$ *if* $\mathrm{P} \succ \mathrm{R}$ *is a substitution step* (1) *or* (2),
- $\mathrm{J} = \mathrm{I}$ *otherwise.*

The weight of a process is thus an upper bound on the number of substitution steps in its evaluation.

Corollary 5.11 *Let* $\mathrm{P} = \langle\, t\,;\, \varnothing \,\rangle\ \star\ \diamond$ *and* $|t| = m$*. If* $\phi; \Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \mathrm{P} : \tau$ *and* $\mathrm{P} \succ^{n} \mathrm{R}$*, then* $\phi; \Phi \vdash^{\mathcal{E}}_{\mathrm{J}} \mathrm{R} : \tau$*, and* $\phi; \Phi \vDash_{\mathcal{E}} (\mathrm{I} - \mathrm{J} + \mathtt{1}) \times \mathrm{m} \geqslant \mathrm{n}$*.*

Proof. First observe that if $\mathrm{P} \succ^{*} \mathrm{S}$ then any term s appearing in S is either a subterm of t or a natural number $\underline{\mathtt{n}}$, and thus $|s| \leqslant m$. Hence by induction on n we can prove using Proposition 5.10 that $\mathrm{P} \succ^{n} \mathrm{R}$ implies $n \leqslant |\mathrm{P}| - |\mathrm{R}| + (\llbracket \mathrm{I} \rrbracket^{\mathcal{E}}_{\rho} - \llbracket \mathrm{J} \rrbracket^{\mathcal{E}}_{\rho}) \cdot |t|$ for any valuation ρ defined on ϕ and satisfying Φ. Then we can conclude since $|\mathrm{P}| = m$ and $|\mathrm{R}| \geqslant 0$. □

Given Corollary 5.11, Theorem 5.8 is within reach: if $\vdash^{\mathcal{E}}_{\mathrm{H}} t : \mathtt{Nat}[\mathrm{I}, \mathrm{J}]$ then $\langle\, t\,;\, \varnothing \,\rangle\ \star\ \diamond$ terminates in at most $(\llbracket \mathrm{H} \rrbracket^{\mathcal{E}}_{\rho} + 1) \cdot |t|$ steps in the $\mathsf{CEK}_{\mathsf{PCF}}$, since weights cannot be negative. It reaches an irreducible process, that is a value closure together with an empty stack:

$$\langle\, t\,;\, \varnothing \,\rangle\ \star\ \diamond \succ^{*} \langle\, v\,;\, \varnothing \,\rangle\ \star\ \diamond\,.$$

By Proposition 5.10, it means that $\langle\, v\,;\, \varnothing \,\rangle\ \star\ \diamond$ has type $\mathtt{Nat}[\mathrm{I}, \mathrm{J}]$, and so has v. So we can conclude that v is a natural number in the interval $[\llbracket \mathrm{I} \rrbracket^{\mathcal{E}}; \llbracket \mathrm{J} \rrbracket^{\mathcal{E}}]$.

5.4. (Relative) Completeness

In this section, we will prove some results about the expressive power of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, seen as a tool to prove intentional (but also extensional) properties of PCF terms. Actually, $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ is extremely powerful: every first-order PCF program computing the function $f : \mathbb{N} \to \mathbb{N}$ in a number of steps bounded by $g : \mathbb{N} \to \mathbb{N}$ can be proved to enjoy these properties by way of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, provided two conditions are satisfied:

- On the one hand, the equational program $\mathcal{E}$ needs to be *universal*, meaning that every partial recursive function is expressible by some index terms. This can be guaranteed, as an example, by the presence of a universal program in $\mathcal{E}$.
- On the other hand, all *true* statements in the form $\phi; \Phi \vDash_{\mathcal{E}} \mathrm{I} \leqslant \mathrm{J}$ must be "available" in the type system for completeness to hold. In other words,

one cannot assume that those judgments are derived in a given (recursively enumerable) formal system, because this would violate Gödel's Incompleteness Theorem. In fact, our completeness theorems are *relative* to an oracle for the truth of those assumptions, which is precisely what happens in similar results for Floyd-Hoare logics [26].

PCF *Typing.* The first step towards completeness is quite easy: propositional type systems in the style of PCF for terms, closures, stacks and processes need to be introduced. All of them can be easily obtained by erasing the index information from $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$. As an example, the typing rule for the application looks like

$$\frac{\Gamma \vdash_{\mathsf{PCF}} t : \alpha \Rightarrow \beta \qquad \Gamma \vdash_{\mathsf{PCF}} u : \alpha}{\Gamma \vdash_{\mathsf{PCF}} tu : \beta}$$

while processes can be typed by the following rule

$$\frac{\vdash_{\mathsf{PCF}} \pi : (\alpha, \beta) \qquad \vdash_{\mathsf{PCF}} \mathsf{c} : \alpha}{\vdash_{\mathsf{PCF}} \mathsf{c} \star \pi : \beta}$$

Given any type σ (respectively, any type derivation δ) of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, the PCF type (respectively, the PCF type derivation) obtained by erasing all the index information will be denoted by $(\!|\sigma|\!)$ (respectively, by $(\!|\delta|\!)$). Of course both terms and processes enjoy subject reduction theorems with respect to PCF typing, and their proofs are much simpler than those for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$. As an example, given a type derivation δ for $\vdash_{\mathsf{PCF}} \mathrm{P} : \mathtt{Nat}$ (we might write $\delta \rhd \vdash_{\mathsf{PCF}} \mathrm{P} : \mathtt{Nat}$) and $\mathrm{P} \succ \mathrm{R}$, a type derivation γ for $\vdash_{\mathsf{PCF}} \mathrm{R} : \mathtt{Nat}$ can be easily built by manipulating in a standard way δ; we write $\delta \succ \gamma$.

Weighted Subject Expansion. The key ingredient for completeness is a dualisation of Weighted Subject Reduction:

Proposition 5.12 (Weighted Subject Expansion) *Suppose that* $\delta \rhd \vdash_{\mathsf{PCF}} \mathrm{P} : \alpha$*, that* $\delta \succ \gamma$*, and that* $\epsilon \rhd \phi; \Phi \vdash^{\mathcal{E}}_{\mathrm{I}} \mathrm{R} : \tau$ *where* $(\!|\epsilon|\!) = \gamma$*. Then there is* $\theta \rhd \phi; \Phi \vdash^{\mathcal{E}}_{\mathrm{J}} \mathrm{P} : \tau$ *with* $(\!|\theta|\!) = \delta$ *and* $\phi; \Phi \vDash_{\mathcal{E}} \mathrm{J} \leqslant \mathrm{I} + 1$*. Moreover,* θ *can be effectively computed from* δ*,* ϵ *and* γ*.*

Proving Proposition 5.12 requires a careful analysis of the evolution of the $\mathsf{CEK}_{\mathsf{PCF}}$ machine, similarly to what happened for Weighted Subject *Reduction*. But while in the latter it is crucial to be able to (parametrically) *split* type derivations for terms (and thus closures), here we need to be able to *join* them:

Lemma 5.13 (Joining) *Suppose $\mathcal{E}$ is universal. If $\delta_i \rhd \phi; \Phi; \Gamma_i \vdash^{\mathcal{E}}_{\mathrm{N}_i} v : \tau_i$, $(\!|\delta_1|\!) = (\!|\delta_2|\!)$, $\phi; \Phi \vdash_{\mathcal{E}} \Gamma \sqsubseteq \Gamma_1 \uplus \Gamma_2$, $\phi; \Phi \vdash_{\mathcal{E}} \tau_1 \uplus \tau_2 \sqsubseteq \tau$ and $\phi; \Phi \vDash_{\mathcal{E}} \mathrm{N}_1 + \mathrm{N}_2 \leqslant \mathrm{M}$, then $\phi; \Phi; \Gamma \vdash^{\mathcal{E}}_{\mathrm{M}} v : \tau$.*

Lemma 5.14 (Parametric Joining) *Suppose that $\mathcal{E}$ is universal. If $a, \phi; a < \mathrm{I}, \Phi; \Delta \vdash^{\mathcal{E}}_{\mathrm{N}} v : \sigma$, $\phi; \Phi \vdash_{\mathcal{E}} \Gamma \sqsubseteq \sum_{a<\mathrm{I}} \Delta$, $\phi; \Phi \vdash_{\mathcal{E}} \sum_{a<\mathrm{I}} \sigma \sqsubseteq \tau$ and $\phi; \Phi \vDash_{\mathcal{E}} \sum_{a<\mathrm{I}} \mathrm{N} \leqslant \mathrm{M}$, then $\phi; \Phi; \Gamma \vdash^{\mathcal{E}}_{\mathrm{M}} v : \tau$.*

Observe that the Joining Lemma requires the two type derivations to be joined to have the same PCF "skeleton". This is essential, because otherwise it would not be possible to unify them into one single type derivation.

Completeness for Programs. We now have all the necessary ingredients to obtain a first completeness result, namely one about programs (which are terms of type $\mathtt{Nat}$). Suppose that t is a PCF program and that $t \to_v^* m$, where m is a natural number. By Proposition 3.1, there is a sequence of processes

$$\mathrm{P}_1 \succ \mathrm{P}_2 \succ \ldots \succ \mathrm{P}_n,$$

where $\mathrm{P}_1 = (\langle\, t \,;\, \varnothing \,\rangle \star \diamond)$ and $\mathrm{P}_n = (\langle\, m \,;\, \varnothing \,\rangle \star \diamond)$. Of course, $\vdash_{\mathsf{PCF}} \mathrm{P}_i : \mathtt{Nat}$ for every i. For obvious reasons, $\vdash^{\mathcal{E}}_0 \mathrm{P}_n : \mathtt{Nat}[m]$. Moreover, by Weighted Subject Expansion, we can derive each of $\vdash^{\mathcal{E}}_{\mathrm{I}_i} \mathrm{P}_i : \mathtt{Nat}[m]$, until we reach $\vdash^{\mathcal{E}}_{\mathrm{I}_1} \mathrm{P}_1 : \mathtt{Nat}[m]$, where $\mathrm{I}_1 \leqslant n$. See Figure 12 for a graphical representation of the above argument. It should be now clear that one can reach the following:

Theorem 5.15 (Completeness for Programs) *Suppose that $\vdash_{\mathsf{PCF}} t : \mathtt{Nat}$, that $t \Downarrow^n \underline{\mathtt{m}}$ and that $\mathcal{E}$ is universal. Then, $\vdash^{\mathcal{E}}_{\mathrm{k}} t : \mathtt{Nat}[\mathrm{m}]$, where $k \leqslant n$.*

Uniformization and Completeness for Functions. Completeness for programs, however, is not satisfactory: the fact (normalizing) PCF terms of type $\mathtt{Nat}$ can all be analyzed by $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ is not so surprising, and other type systems (like non-idempotent intersection types [14]) have comparable expressive power. Suppose we want to generalize relative completeness to first-order functions: we would like to prove that every term t having a PCF type $\mathtt{Nat} \Rightarrow \mathtt{Nat}$ (which terminates when fed with any natural number) can be typed in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$. How could we proceed? First of all, observe that the argument in Figure 12 could be applied to all *instances* of t, namely to all terms in $\{t\ \underline{\mathtt{n}} \mid n \in \mathbb{N}\}$. This way one can obtain, for every $n \in \mathbb{N}$, a type derivation δ_n for

$$\vdash^{\mathcal{E}}_{\mathrm{I}_n} t : [a < \mathrm{J}_n] \cdot \mathtt{Nat}[\mathrm{K}_n] \multimap \mathtt{Nat}[\mathrm{H}_n],$$

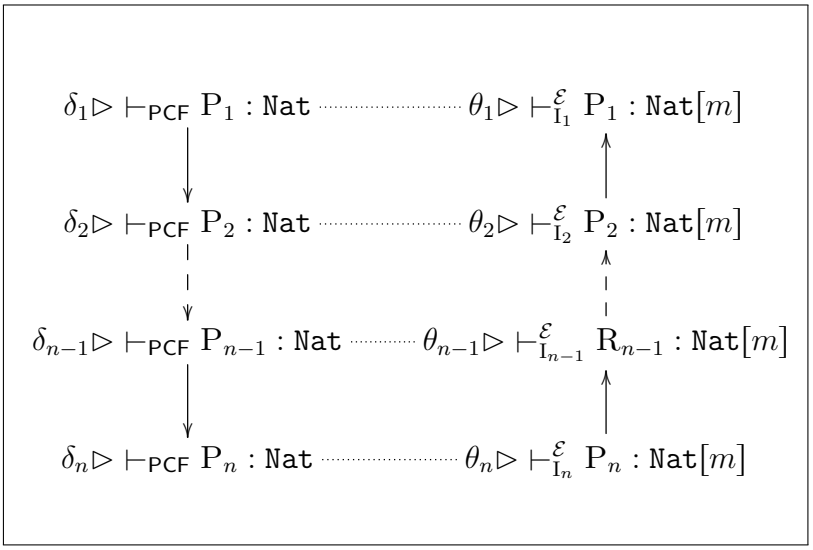


Figure 12: Completeness for programs, sketch of the Proof.

where J_n can be assumed to be 1, while K_n can be assumed to be n. Moreover, the problem of obtaining δ_n from n is recursive, *i.e.*, can be solved by an algorithm. Surprisingly, the infinitely many type derivations in $\{\delta_n \mid n \in \mathbb{N}\}$ can be turned into one:

Proposition 5.16 (Uniformization of Type Derivations) *Suppose that $\mathcal{E}$ is universal and that $\{\delta_n\}_{n\in\mathbb{N}}$ is a recursively enumerable collection of type derivations satisfying the following constraints:*

1. *For every $n \in \mathbb{N}$, $\delta_n \triangleright \vdash^{\mathcal{E}}_{\mathrm{I}_n} t : \sigma_n$;*
2. *all derivations in $\{\delta_n\}_{n\in\mathbb{N}}$ have the same skeleton, i.e., for every $n, m \in \mathbb{N}$, $(\!|\delta_n|\!) = (\!|\delta_m|\!)$.*

Then there is a type derivation $\theta \triangleright a; \varnothing; \varnothing \vdash^{\mathcal{E}}_{\mathrm{I}} t : \sigma$ such that $\models_{\mathcal{E}} \mathrm{I}\{\mathrm{n}/a\} = \mathrm{I}_n$ and $\models_{\mathcal{E}} \sigma\{\mathrm{n}/a\} \equiv \sigma_n$ for all n.

Proof. The proof proceeds by first proving three lemmas:

- First of all, one can prove that any family of semantic judgment in the form $\{\models_{\mathcal{E}} \mathrm{I}\{\mathrm{n}/a\} = \mathrm{J}\{\mathrm{n}/a\}\}_{n\in\mathrm{N}}$ can be turned into one semantic judgment $\models_{\mathcal{E}} \mathrm{I} = \mathrm{J}$.
- Then, it is possible to uniformize index terms themselves: if $\{\mathrm{I}_n\}_{n\in\mathrm{N}}$ is a family of index terms, then there is *one* index term I such that $\models_{\mathcal{E}} \mathrm{I}\{\mathrm{n}/a\} \cong \mathrm{I}_n$.

• Subtyping judgments can be made uniform themselves.

The three lemmas are exactly the necessary ingredients to carry out an induction on the structure of any δ_n (they all have the same structure, anyway). For more details, one can consult [27]. □

Uniformization of type derivations should be seen as an extreme form of joining: not only a finite number of type derivations for the same term can be unified into one, but any recursively enumerable class of them can. Again, the universality of $\mathcal{E}$ is crucial here. We are now ready to give the following:

Theorem 5.17 (Completeness for Functions) *Suppose that* $\vdash_{\mathsf{PCF}} t : \mathtt{Nat} \Rightarrow \mathtt{Nat}$*, that* $t\,\underline{\mathtt{n}} \Downarrow^{k_n} \underline{\mathtt{m}_n}$ *for all* $n \in \mathbb{N}$ *and that* $\mathcal{E}$ *is universal. Then, there are index terms* I *and* H *such that* $a; \varnothing; \varnothing \vdash^{\mathcal{E}}_{\mathrm{I}} t : [b < \mathtt{1}] \cdot \mathtt{Nat}[a] \multimap \mathtt{Nat}[\mathrm{H}]$*, where* $\vDash_{\mathcal{E}} \mathrm{I}\{\mathrm{n}/a\} \leqslant \mathrm{k}_n$ *and* $\vDash_{\mathcal{E}} \mathrm{H}\{\mathrm{n}/a\} = \mathrm{m}_n$*.*

5.5. Weight and Time Complexity

The intensional soundness and relative completeness results show a tight relation between the weight of a program and its evaluation time in the $\mathsf{CEK}_{\mathsf{PCF}}$ machine. Notice that, because of the subsumption typing rule, the weight of a term can be arbitrarily increased. However, the following corollary emphasizes that the lowest weight that can be associated to a program provides a rather tight bound of its complexity.

Corollary 5.18 *Assume* t *is a* PCF *program that evaluates to* $\underline{\mathtt{m}}$ *with* n *steps in the* $\mathsf{CEK}_{\mathsf{PCF}}$*, and* $\mathcal{E}$ *is a universal equational program. Then one can derive* $\vdash^{\mathcal{E}}_{\mathrm{k}} t : \mathtt{Nat}[\underline{\mathtt{m}}]$ *in* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$*, with* $k \leqslant n \leqslant k \cdot |t|$*.*

Proof. The typing judgment with the lower bound $k \leqslant n$ are given by Theorem 5.15. The upper bound $n \leqslant k \cdot |t|$ follows from Theorem 5.8. □

Actually, the minimum weight of a program t represents the number of substitutions in its CBV evaluation or, formally, the number of steps of kind (1) or (2) in its evaluation by the $\mathsf{CEK}_{\mathsf{PCF}}$ machine (Remark 5.1). It is thus a lower bound on its time complexity in $\mathsf{CEK}_{\mathsf{PCF}}$. Between two such substitution steps, the machine only scans the term to reach the head subterm (or to select one branch of the test for zero, when a `fork` rule is applied). At most $|t|$ scanning steps can be done consecutively (during the evaluation,

the machine only contains subterms[2] of the initial term t), and so the total number of steps is bounded by the weight multiplied by $|t|$.

6. Perspectives and Developments

6.1. $\mathsf{d\ell PCF_V}$, $\mathsf{d\ell PCF_N}$ *and their Peculiarities*

In this section, we briefly describe in which senses $\mathsf{d\ell PCF_V}$, the type system we introduce here, differs from $\mathsf{d\ell PCF_N}$ [1].

As already pointed out in Section 2.2, in $\mathsf{d\ell PCF_N}$ all *arguments* to functions are marked as duplicable, while in $\mathsf{d\ell PCF_V}$ the same role is played by *functions* themselves. This, coupled with linear dependency, implies that:

- When typing a term t in $\mathsf{d\ell PCF_N}$ any subterm u of t which can potentially be used as an argument to a function is typed parametrically on an index variable a, each value of a corresponding to a possible "use" of u;
- If t is typed in $\mathsf{d\ell PCF_V}$, the same holds for any λ-abstraction $\lambda x.u$ in t: all possible uses of $\lambda x.u$ need to be taken into account in the underlying type if we want the latter to be precise and, ultimately, if we care about relative completeness.

Saying it another way, both $\mathsf{d\ell PCF_N}$ and $\mathsf{d\ell PCF_V}$ are non-modular, in that the way one attributes a type (and ultimately a weight) to certain subterms can possibly depend on how they are used by the environment. This is indeed true if one builds a type derivation by way of a fixed (maybe weak) equational program. Typing a term and defining the underlying equational program *at the same time*, leaving some of the index terms unspecified, is a way to alleviate this problem, and is actually the main idea behind a linear dependent type inference algorithm [28].

Technically, the metatheory of $\mathsf{d\ell PCF_V}$ can be explored with tools similar to those employed in $\mathsf{d\ell PCF_N}$. However, the authors claim that most results about the former system, soundness and completeness *in primis*, are not easy consequences of the one enjoyed by the latter. An example are splitting lemmas, which are much easier to be proved in $\mathsf{d\ell PCF_N}$ (see [1] for more details).

[2] To claim this, we formally need to consider an extension of the subterm relation where all natural numbers $\underline{\mathtt{n}}$ are equivalent.

6.2. Further Developments

Relative completeness of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, especially in its stronger form (Theorem 5.17) can be read as follows. Suppose that a sound, finitary formal system C deriving judgments in the form $\phi; \Phi \vdash_{\mathcal{E}} \mathrm{I} \leqslant \mathrm{J}$ is fixed and "plugged" into $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$. What you obtain is a sound, but necessarily incomplete formal system, due to Gödel's incompleteness. However, this incompleteness is *only* due to C and not to the rules of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, which are designed so as to reduce the problem of proving properties of programs to checking inequalities over $\mathcal{E}$ *without any loss of information.*

In this scenario, it is of paramount importance to devise techniques to *automatically* reduce the problem of checking whether a program satisfies a given intentional or extensional specification to the problem of checking whether a given set of inequalities over an equational program $\mathcal{E}$ holds. Indeed, many techniques and concrete tools are available for the latter problem (take, as an example, the immense literature on SMT solving), while the same cannot be said about the former problem. The situation, in a sense, is similar to the one in the realm of program logics for imperative programs, where logics are indeed very powerful [26], and great effort have been directed to devise efficient algorithms generating weakest preconditions [29].

Actually, at the time of writing, the authors are actively involved in the implementation of *relative type inference* algorithms for both $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{N}}$ and $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, which can be seen as having the same role as algorithms computing weakest preconditions. Efficient type inference algorithms *can indeed* be defined [28]: the idea is to start with a PCF type derivation and decorate it with index terms, defining an appropriate equational program along the way. A detailed description of the type inference process is however outside the scope of this paper.

6.3. Related Work

Complexity analysis of higher-order programs has been (and continues to be) an object of study in the programming language research community.

Among the many works in this direction, we can mention the proposals for type systems for the λ-calculus characterizing in an *extensional sense*, *e.g.* polynomial time computable functions. Many of these techniques can be seen as static analysis methodologies: once a program is assigned a type, an upper bound to its time complexity is relatively easy to be synthesized. The problem with these systems, however, is that they are usually very weak

from an *intentional* point of view: the class of typable programs is quite restricted.

More powerful static analysis methodologies can actually be devised. All of them, however, are either limited to very specific forms of resource bounds or to a peculiar form of higher-order functions or else they do not get rid of higher-order as the underlying logic. Consider, as an example, one of the earliest work in this direction, namely Sands's system of cost closures [30]: the class of programs that can be handled includes the full lazy λ-calculus, but the way complexity is reasoned about remains genuinely higher-order, being based on closures and contexts. In Benzinger's framework [31] higher-order programs are translated into higher-order equations, and the latter are turned into first-order ones; both steps, and in particular the second one, are not completeness-preserving. More generally, there is an inherent risk in doing program analysis by way of program transformation (see [32, 33]): if the *automatic* analysis of the transformed program fails, one is left with an unintelligible piece of code, which can hardly be analyzed manually or semi-automatically. In this respect, type systems constitute a very good compromise.

Recent works on amortized resource analysis are either limited to first-order programs [19] or to linear bounds [7]. A recent proposal by Amadio and Régis-Gianas [34] allows to reason on the the cost of higher-order functional programs by way of so-called cost-annotations, being sure that the actual behavior of compiled code somehow reflects the annotation. The logic in which cost annotations are written, however, is a *higher-order* Hoare logic. None of the proposed systems, on the other hand, are known to be (relatively) complete in the sense we use here.

Complexity analysis of first-order functional programs, or of programs written in other programming paradigms is a much more developed research area. Examples are the work by Crary And Weirich [35] or the SPEED project [36]

Linear dependency can be seen as a way to turn games and strategies into types, this way facilitating verification. A recent work going in the same direction is Geometry of Synthesis, in particular when the latter is supported by type systems [37].

6.4. $\mathsf{d\ell PCF_V}$ *and Implicit Complexity*

There is a price to pay for the kind of relative completeness $\mathsf{d\ell PCF_V}$ (and $\mathsf{d\ell PCF_N}$) enjoys: checking a type derivation for correctness is undecidable

in general, simply because it relies on semantic assumptions in the form of inequalities between index terms, or on subtyping judgments, which themselves rely on the properties of the underlying equational program $\mathcal{E}$. Indeed, $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$ should *not* be thought of as a type system, but rather as a framework in which various distinct type systems can be defined. Concrete type systems can be crafted by either instantiating $\mathcal{E}$, or by choosing specific and sound formal systems for the verification of semantic assumptions. By the way, the just described problem is not peculiar to $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$: Floyd-Hoare program logics are themselves undecidable.

The main motivation behind the introduction of linear dependent types comes from implicit computational complexity. Traditionally, what prevents (most) ICC techniques to find concrete applications along this line is their poor expressive power: the class of programs which can be recognized as being efficient by (tools derived from) ICC systems is often very small and does not include programs corresponding to natural, well-known algorithms. In this respect, $\mathsf{d}\ell\mathsf{PCF}_\mathsf{N}$ and $\mathsf{d}\ell\mathsf{PCF}_\mathsf{V}$ are fundamentally different: *all* PCF programs with a certain complexity can be proved to be so by deriving a typing judgment for them.

6.5. Conclusions

Linear dependent types are shown to be applicable to the analysis of intentional and extensional properties of functional programs when the latter are call-by-value evaluated. More specifically, soundness and relative completeness results are proved for both programs and functions. This generalizes previous work by Gaboardi and the first author [1], who proved similar results in the call-by-name setting. This paper shows that linear dependency not only provides an expressive formalism, but is also robust enough to be adaptable to calculi whose notions of reduction are significantly different (and in many cases more efficient) than normal order evaluation.

Topics for future work include some further analysis about the applicability of linear dependent types to languages with more features, including some form of inductive data types, or ground type references. Another interesting problem is tuning linear dependent types in such a way as to capture the complexity of *call-by-need* evaluation.

References

[1] U. Dal Lago, M. Gaboardi, Linear dependent types and relative completeness, in: LICS, IEEE Comp. Soc., 2011, pp. 133–142.

[2] D. M. Volpano, C. E. Irvine, G. Smith, A sound type system for secure flow analysis, JCS 4 (1996) 167–188.

[3] A. Sabelfeld, A. C. Myers, Language-based information-flow security, IEEE JSAC 21 (2003) 5–19.

[4] G. Barthe, B. Grégoire, C. Riba, Type-based termination with sized products, in: CSL, volume 5213 of *LNCS*, Springer, 2008, pp. 493–507.

[5] N. Kobayashi, C.-H. L. Ong, A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes, in: LICS, IEEE Comp. Soc., 2009, pp. 179–188.

[6] Karlbary, S. Weirich, Resource bound certification, in: POPL, ACM Press, 2000, pp. 184–198.

[7] S. Jost, K. Hammond, H.-W. Loid, M. Hofmann, Static Determination of Quantitative Resource Usage for Higher-Order Programs, in: POPL, ACM Press, 2010, pp. 223–236.

[8] M. Hofmann, Linear types and non-size-increasing polynomial time computation, in: LICS, IEEE Comp. Soc., 1999, pp. 464–473.

[9] P. Baillot, K. Terui, Light types for polynomial time computation in lambda calculus, Inf. Comput. 207 (2009) 41–62.

[10] P. Baillot, M. Gaboardi, V. Mogbil, A polytime functional language from light linear logic, in: ESOP, volume 6012 of *LNCS*, Springer, 2010, pp. 104–124.

[11] H. Xi, Dependent types for program termination verification, in: LICS, IEEE Comp. Soc., 2001, pp. 231–246.

[12] U. Dal Lago, Context semantics, linear logic, and computational complexity, ACM TOCL 10 (2009).

[13] M. Coppo, M. Dezani-Ciancaglini, An extension of the basic functionality theory for the λ-calculus, Notre Dame J. Formal Logic 21 (1980) 685–693.

[14] D. de Carvalho, Execution time of lambda-terms via denotational semantics and intersection types, 2009. Available at `http://arxiv.org/abs/0905.4251`.

[15] J.-L. Krivine, A call-by-name lambda-calculus machine, Higher-Order and Symbolic Computation 20 (2007) 199–207.

[16] M. Felleisen, D. P. Friedman, Control Operators, the SECD-Machine and the λ-Calculus, Technical Report 197, Computer Science Department, Indiana University, 1986.

[17] J. Maraist, M. Odersky, D. N. Turner, P. Wadler, Call-by-name, call-by-value, call-by-need and the linear lambda calculus, Electr. Notes Theor. Comput. Sci. 1 (1995) 370–392.

[18] G. D. Plotkin, LCF considerd as a programming language, Theor. Comput. Sci. 5 (1977) 225–255.

[19] J. Hoffmann, K. Aehlig, M. Hofmann, Multivariate Amortized Resource Analysis, in: POPL, ACM Press, 2011, pp. 357–370.

[20] J.-Y. Girard, A. Scedrov, P. J. Scott, Bounded linear logic: A modular approach to polynomial-time computability, Theor. Comput. Sci. 97 (1992) 1–66.

[21] J. Lamping, An algorithm for optimal lambda calculus reduction, in: POPL, ACM Press, 1990, pp. 16–30.

[22] A. Asperti, H. G. Mairson, Parallel beta reduction is not elementary recursive, Inf. Comput. 170 (2001) 49–80.

[23] A. Asperti, S. Guerrini, The Optimal Implementation of Functional Programming Languages, Cambridge University Press, 1998.

[24] P. Odifreddi, Classical Recursion Theory: the Theory of Functions and Sets of Natural Numbers, number 125 in Studies in Logic and the Foundations of Mathematics, North-Holland, 1989.

[25] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.

[26] S. A. Cook, Soundness and completeness of an axiom system for program verification, SIAM J. on Computing 7 (1978) 70–90.

[27] U. Dal Lago, M. Gaboardi, Linear dependent types and relative completeness, Logical Methods in Computer Science 8 (2012).

[28] U. Dal Lago, B. Petit, The geometry of types, in: POPL, ACM Press, 2013, pp. 167–178.

[29] J. W. de Bakker, A. de Bruin, J. Zucker, Mathematical theory of program correctness, Prentice-Hall international series in computer science, Prentice Hall, 1980.

[30] D. Sands, Complexity analysis for a lazy higher-order language, in: ESOP 1990, volume 432 of *LNCS*, Springer, 1990, pp. 361–376.

[31] R. Benzinger, Automated higher-order complexity analysis, Theor. Comput. Sci. 318 (2004) 79–103.

[32] M. Rosendahl, Automatic complexity analysis, in: FPCA, pp. 144–156.

[33] N. Danner, J. Paykin, J. S. Royer, A static cost analysis for a higher-order language, in: PLPV, ACM Press, 2013, pp. 25–34.

[34] N. Ayache, R. M. Amadio, Y. Régis-Gianas, Certifying and reasoning on cost annotations in c programs, in: FMICS, pp. 32–46.

[35] K. Crary, S. Weirich, Resource bound certification, in: POPL, ACM Press, 2000, pp. 184–198.

[36] S. Gulwani, Speed: Symbolic complexity bound analysis, in: CAV, volume 5643 of *LNCS*, Springer, 2009, pp. 51–62.

[37] D. R. Ghica, A. Smith, Geometry of synthesis III: resource management through type inference, in: POPL, ACM Press, 2011, pp. 345–356.