2011 26th Annual IEEE Symposium on Logic in Computer Science

Linear Dependent Types and Relative Completeness

Ugo Dal Lago and Marco Gaboardi
Dipartimento di Scienze dell'Informazione & EPI FOCUS
Università di Bologna & INRIA Sophia Antipolis
dallago,gaboardi@cs.unibo.it

***Abstract*—A system of linear dependent types for the lambda calculus with full higher-order recursion, called dℓPCF, is introduced and proved sound and relatively complete. Completeness holds in a strong sense: dℓPCF is not only able to precisely capture the functional behaviour of PCF programs (i.e. how the output relates to the input) but also some of their intensional properties, namely the complexity of evaluating them with Krivine's Machine. dℓPCF is designed around dependent types and linear logic and is parametrized on the underlying language of index terms, which can be tuned so as to sacrifice completeness for tractability.**

I. Introduction

Type systems are powerful tools in the design of programming languages. While they have been employed traditionally to guarantee weak properties of programs (e.g. "well-typed programs cannot go wrong"), it is becoming more and more evident that they can be useful when stronger properties are needed, such as security [1], [2], termination [3], monadic properties [4] or resource bounds [5].

One key advantage of type systems seen as formal methods is their simplicity and their close relationship with programs — checking whether a program has a type or even inferring the (most general) type of a program is often decidable. The price to pay is the incompleteness of most type systems: there are programs satisfying the property at hand which cannot be given a type. This is in contrast with other formal methods, like program logics [6] where completeness is always a desirable feature.

One specific research field in which the just-described scenario manifests itself is implicit computational complexity, in which one aims at defining characterizations of complexity classes by programming languages and logical systems. Many type systems have been introduced capturing, for instance, the polynomial time computable functions [7], [8], [9]. 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 $\mathbf{FP}$ is only characterized in a purely extensional sense — for every function in $\mathbf{FP}$ there is *at least one* typable program computing it.

The main contribution of this paper is a *type system* for the lambda calculus with full recursion, called dℓPCF, which is sound *and complete*. Types of dℓPCF are obtained, in the spirit of DML [10], [11], by decorating types of ordinary PCF [12], [13] with *index terms*. These are first-order terms freely generated from variables, function symbols and a few more term constructs. They are indicated with metavariables like $\mathrm{I}, \mathrm{J}, \mathrm{K}$. Type decoration reflects the standard decomposition of types into *linear types* (as suggested by linear logic [14]), and is inspired by recent works on the expressivity of bounded logics [15].

Index terms and linear types permit to describe program properties with a fine granularity. More precisely, dℓPCF enjoys the following two properties:

- *Soundness*: if t is a program and $\vdash_{\mathrm{K}} t : \mathtt{Nat}[\mathrm{I}, \mathrm{J}]$, then t evaluates to a natural number which lies between I and J and this evaluation takes a linear number of steps in K;
- *Completeness*: if t is typable in PCF and evaluates to a natural number n in m steps, then $\vdash_{\mathrm{I}} t : \mathtt{Nat}[n, n]$ where $\mathrm{I} \leqslant m$.

Completeness of dℓPCF holds not only for programs (i.e. terms of ground types) but also for functions on the natural numbers (see Section V-C for further details). Moreover, typing judgments tell us something about the functional behaviour of programs but also about their non-functional one, namely the number of steps needed to evaluate the term in Krivine's Abstract Machine.

As the title of this paper suggests, completeness of dℓPCF holds in a relative sense. Indeed, the behaviour of programs can be precisely captured only in presence of a complete oracle for the truth of certain assumptions in typing rules. This is exactly what happens in program logics such as Floyd-Hoare's logic, where all the sound partial correctness assertions can be derived *provided* one is allowed to use all the true sentences of first order arithmetic as axioms [16]. In dℓPCF, those assumptions take the form of (in)equalities between index terms, to be verified when function symbols are interpreted as functions on natural numbers according to an equational program $\mathcal{E}$. Actually, the whole of dℓPCF is *parameterized* on $\mathcal{E}$, but while soundness holds independently of the specific $\mathcal{E}$, completeness, as is to be expected, holds only if $\mathcal{E}$ is sufficiently powerful to encode all total computable functions (i.e. if $\mathcal{E}$ is universal). In other words, dℓPCF can be claimed to be not *a* type system, but *a family of* type systems obtained by taking a specific $\mathcal{E}$ as the underlying "logic" of index terms. The simpler $\mathcal{E}$, the easier type checking and type inference are; the more complex $\mathcal{E}$, the larger the class of captured programs.

The design of dℓPCF have been very much influenced by linear logic [14], and in particular by systems of indexed and bounded linear logic [17], [15], which have been recently shown to subsume other ICC systems as for the class of

1043-6871/11 $26.00 © 2011 IEEE
DOI 10.1109/LICS.2011.22

programs they can capture [15]. One of the many ways to "read" dℓPCF is as a variation on the theme of BLL [17] obtained by generalizing polynomials to arbitrary functions. The idea of going beyond a restricted, fixed class of bounds comes from Xi's work on DML [10], [11]. Cost recurrences for first order DML programs have been studied [18]. No similar completeness results for dependent types are known, however.

An extended version with all proofs is available [19].

II. Types and Program Properties: An Informal Account

Consider the following program:

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

In a monomorphic, traditionally designed type system like PCF [12], [13], the term dbl receives type $\mathtt{Nat} \rightarrow \mathtt{Nat}$. As a consequence, dbl computes a function on natural numbers without "going wrong": it takes in input a natural number, and produces in output another natural number (if any). The type $\mathtt{Nat} \rightarrow \mathtt{Nat}$, however, does not give any information about *which* specific function on the natural numbers dbl computes. Indeed, in PCF (and in most real-world programming languages) any program computing a function on natural numbers, being it for instance the identity function or (a unary version of) the Ackermann function, can be typed by $\mathtt{Nat} \rightarrow \mathtt{Nat}$.

Some modern type systems allow one to construct and use types like $\tau = \mathtt{Nat}[a] \rightarrow \mathtt{Nat}[\mathbf{2} \times a]$, which tell not only what set or domain (the interpretation of) the term belongs to, but also which specific element of the domain the term actually denotes. The type τ, for example, could be attributed only to those programs computing the function $n \mapsto 2 \times n$. Types of this form can be constructed in dependent and sized type theories [10], [3]. The type system dℓPCF introduced in this paper offers this possibility, too. But, as a first contribution, it further allows to specify *imprecise* types, like $\mathtt{Nat}[\mathbf{5}, \mathbf{8}]$, which stands for the type of those natural numbers between 5 and 8.

A property of programs which is completely ignored by ordinary type systems is termination, at least if full recursion is in the underlying language. Typing a term t with $\mathtt{Nat} \rightarrow \mathtt{Nat}$ does not guarantee that t, when applied to a natural number, terminates. In PCF this is even worse: t could possibly diverge *itself*! Consider, as another example, a slight modification of dbl, namely

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

It behaves as dbl when fed with 0, but it diverges when it receives a positive natural number as an argument. But look: omega is not so different from dbl. Indeed, the second can be obtained from the first by feeding not x but $\mathtt{p}(x)$ to f. And any type systems in which dbl and omega are somehow recognized as being fundamentally different must be able to detect the presence of p in dbl and deduct termination from it. Indeed, sized types [3] and dependent types [20] 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 dbl takes a polynomial number of steps to be evaluated on any natural number. This cannot be achieved neither using classical type systems nor using systems of sized types, at least when traditionally formulated. However, some type systems able to control the complexity of program exist. Good examples are type systems for amortized analysis [5], [21] or those using ideas from linear logic [8], [9]. In those type systems, typing judgements carry, besides the usual type information, some additional information about the resource consumption of the underlying program. As an example, dbl could be given a type as follows

$$\vdash_{\mathrm{I}} \mathtt{dbl} : \mathtt{Nat} \rightarrow \mathtt{Nat}$$

where I is some cost information for dbl. This way, building a type derivation and inferring resource consumption can be done contextually.

The type system dℓPCF we propose in this paper makes some further steps in this direction. First of all, it combines some of the ideas presented above with the ones of bounded linear logic. BLL allows one to explicitly count the number of times functions use their arguments (in rough notation, $!_m\sigma \multimap \tau$ says that the argument of type σ is used m times). This permits to extract natural cost functions from type derivations. The cost of evaluating a term will be measured by counting how many times function arguments need to be copied during evaluation. Making this information explicit in types permits to compute the cost step by step during the type derivation process. By the way, previous works by the first author [22] show that this way of attributing a cost to (proofs seen as) programs is sound and precise as a way to measure their time complexity. Intuitively, typing judgements in dℓPCF can be thought as:

$$\vdash_{\mathtt{g}(a)} t\ : !_m\ \mathtt{Nat}[a] \multimap \mathtt{Nat}[\mathtt{f}(a)].$$

where $\mathtt{g}$ and $\mathtt{f}$ can be derived while building a type derivation, exploiting the information carried by the modalities. In fact, the quantitative information in $!_m$ allows to statically determine the number of times any subterm will be copied during evaluation. But this is not sufficient: analogously to what happens in BLL, dℓPCF makes types more parametric. A rough type as $!_n\sigma \multimap \tau$ is replaced by the more parametric type $[a < n] \cdot \sigma \multimap \tau$, which tell us that the argument will be used n times, and each instance has type σ *where, however* the variable a is instantiated with 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 dℓPCF different from similar type systems, namely completeness.

Finally, as already stressed in the introduction, dℓPCF is also parametric in the class of functions (in the form of an equational program $\mathcal{E}$) that can be used to reason about types and costs. This permits to tune the type system, as described in Section VI below.

Anticipating on the next section, we can say that dbl can be typed as follows in dℓPCF:

$$\vdash^{\mathcal{E}}_{\mathtt{h}(a)} \mathtt{dbl} : [b < a] \cdot \mathtt{Nat}[a] \multimap \mathtt{Nat}[2 \times a].$$

This tells us that the argument will be used a times by dbl, namely a number of times equal to its value. And that the cost of evaluation will be $\mathtt{h}(a)$, itself proportional to a.

III. dℓPCF

In this section, the language of programs and the type system dℓPCF for it will be introduced formally. Some of their basic properties will be described. The type system dℓPCF is based on the notion of index terms whose semantics, in turn, is defined by an equational program. As a consequence, all these notions must be properly introduced and are the subject of Section III-A below.

A. Index Terms and Equational Programs

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

Formally, a *signature* Σ is a pair $(\mathcal{S}, \alpha)$ where $\mathcal{S}$ is a finite set of *function symbols* and $\alpha : \mathcal{S} \to \mathbb{N}$ assigns an *arity* to every function symbol. Index terms on a given signature $\Sigma = (\mathcal{S}, \alpha)$ are generated by the following grammar:

$$\mathrm{I}, \mathrm{J}, \mathrm{K} ::= a \mid \mathtt{f}(\mathrm{I}_1, \ldots, \mathrm{I}_{\alpha(\mathtt{f})}) \mid \sum_{a<\mathrm{I}} \mathrm{J} \mid \mathop{\triangle\mkern-13mu\bigcirc}\limits_{a}^{\mathrm{I},\mathrm{J}} \mathrm{K}$$

where $\mathtt{f} \in \mathcal{S}$ and a is a variable drawn from a set $\mathcal{V}$ of *index variables*. We assume the symbols 0, 1 (with arity 0) and $+$, $\dot{-}$ (with arity 2) are always part of Σ. An index term in the form $\sum_{a<\mathrm{I}} \mathrm{J}$ is a *bounded sum*, while one in the form $\mathop{\triangle\mkern-13mu\bigcirc}_{a}^{\mathrm{I},\mathrm{J}} \mathrm{K}$ is a *forest cardinality*.

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 Σ? It is the one induced by an equational program $\mathcal{E}$. Formally, an *equational program $\mathcal{E}$* over a signature Σ is a set of equations in the form $t = s$ where both t and s are terms built from variables and the symbols in Σ. We are interested in equational programs guaranteeing that, whenever symbols in Σ are interpreted as partial functions over $\mathbb{N}$ and 0, 1, $+$ and $\dot{-}$ are interpreted in the usual way, the semantics of any function symbol $\mathtt{f}$ can be uniquely determined from $\mathcal{E}$. This can be guaranteed by, for example, taking $\mathcal{E}$ as an Herbrand-Gödel scheme [23] or as an orthogonal constructor term rewriting system [24]. One may wonder why the definition of index terms is parametric on Σ and $\mathcal{E}$. As we will see in Section VI, being parametric this way allows us to tune our concrete 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 more effort to be described. Informally, $\mathop{\triangle\mkern-13mu\bigcirc}_{a}^{\mathrm{I},\mathrm{J}} \mathrm{K}$ 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 pre-order, starting from I. The term K has the role of describing the number of children of each forest node n by properly instantiating the variable a, 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:

$$\mathop{\triangle\mkern-13mu\bigcirc}\limits_{a}^{\mathrm{I},0} \mathrm{K} = 0 \tag{1}$$

$$\mathop{\triangle\mkern-13mu\bigcirc}\limits_{a}^{\mathrm{I},\mathrm{J}+1} \mathrm{K} = \left(\mathop{\triangle\mkern-13mu\bigcirc}\limits_{a}^{\mathrm{I},\mathrm{J}} \mathrm{K} \right) + 1 + \left(\mathop{\triangle\mkern-13mu\bigcirc}\limits_{a}^{\mathrm{I}+1+\mathop{\triangle\mkern-13mu\bigcirc}_{a}^{\mathrm{I},\mathrm{J}} \mathrm{K},\, \mathrm{K}\{\mathrm{I}+\mathop{\triangle\mkern-13mu\bigcirc}_{a}^{\mathrm{I},\mathrm{J}} \mathrm{K}/a\}} \mathrm{K} \right) \tag{2}$$

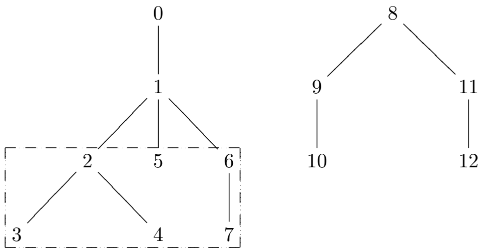
Equation (1) says that a forest of 0 trees contains no nodes. Equation (2) tells us that a forest of $\mathrm{J}+1$ trees contains:

- 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.

In particular, the latter suggest the following decomposition that is a crucial property of the forest cardinalities:

$$\mathop{\triangle\mkern-13mu\bigcirc}\limits_{a}^{1,\mathrm{J}} \mathrm{I} \cong \sum_{b<\mathrm{J}} \mathop{\triangle\mkern-13mu\bigcirc}\limits_{a}^{0,1} \mathrm{I}\{a + 1 + \mathop{\triangle\mkern-13mu\bigcirc}\limits_{a}^{1,b} \mathrm{I}/a\}. \tag{3}$$

To better understand forest cardinalities, consider the following forest comprising two trees:



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

One may wonder what is the role of forest cardinalities in the type systems. 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 *altogether*. 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 $\llbracket \mathrm{I} \rrbracket^{\mathcal{E}}_{\rho}$ denotes the meaning of I, defined by induction along the lines of the previous discussion, where $\rho : \mathcal{V} \rightarrow \mathbb{N}$ 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, $\llbracket \mathrm{I} \rrbracket^{\mathcal{E}}_{\rho}$ can be undefined itself. A *constraint* is an inequality in the form $\mathrm{I} \leqslant \mathrm{J}$. A constraint is *true* in an assignment ρ if $\llbracket \mathrm{I} \rrbracket^{\mathcal{E}}_{\rho}$ and $\llbracket \mathrm{J} \rrbracket^{\mathcal{E}}_{\rho}$ 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

$$\phi; \Phi \models^{\mathcal{E}} \mathrm{I} \leqslant \mathrm{J} \tag{4}$$

denotes the fact that the truth of $\mathrm{I} \leqslant \mathrm{J}$ semantically follows from the truth of the constraints in Φ. The expression $\phi; \Phi \models^{\mathcal{E}} \mathrm{I} = \mathrm{I}$ indicates that (the semantics of) I is *defined* for the relevant values of the variables in ϕ; this is usually written as $\phi; \Phi \models^{\mathcal{E}} \mathrm{I} \Downarrow$.

Before embarking in the description of the type system, a further remark on the role of index terms could be useful. Index terms are not meant to be part of *programs* but of *types*. As a consequence, computation will not be carried out on index terms but on proper terms, which are the subject of Section III-B below.

B. The Type System

All the definitions in this and the next sections should be understood as parametric on an equational program $\mathcal{E}$ over a signature Σ. For the sake of simplicity, however, we will often avoid to explicitly mention $\mathcal{E}$ and leave it implicit. Terms are generated by the following grammar:

$$\begin{aligned} t ::= & \, x \mid \underline{\mathtt{n}} \mid \mathtt{s}(t) \mid \mathtt{p}(t) \mid \lambda x.t \mid tu \\ & \mid \mathtt{ifz}\ t\ \mathtt{then}\ u\ \mathtt{else}\ v \mid \mathtt{fix}\ x.t \end{aligned}$$

where $\mathtt{n}$ ranges over natural numbers and x ranges over a set of *variables*. Terms should be familiar to the reader acquainted with PCF. A *w*eak-head reduction relation $\rightarrow$ on terms can be easily defined. A term t is said to be *program* if it can be given the PCF type $\mathtt{Nat}$ in the empty context. Basic and modal types are defined as follows:

$$\begin{aligned} \sigma, \tau &::= \mathtt{Nat}[\mathrm{I}, \mathrm{J}] \mid A \multimap \sigma && \text{basic types} \\ A, B &::= [a < \mathrm{I}] \cdot \sigma && \text{modal types} \end{aligned}$$

where I, J range over index terms and a ranges over index variables. $\mathtt{Nat}[\mathrm{I}]$ is syntactic sugar for $\mathtt{Nat}[\mathrm{I}, \mathrm{I}]$. Modal types need some comments. As a first approximation, they can be thought of as quantifiers over type variables. So, a type like $A = [a < \mathrm{I}] \cdot \sigma$ acts as a binder for the index variable a in the basic type σ. Moreover, the condition $a < \mathrm{I}$ says that A consists of all the instances of the basic type σ where the variable a is successively instantiated with the values from $\mathtt{0}$ to (the value of) $\mathrm{I} - 1$, i.e. $\sigma\{\mathtt{0}/a\}, \ldots, \sigma\{\mathrm{I} - 1/a\}$. For those readers who are familiar with linear logic, and in particular with BLL the modal type $[a < \mathrm{I}] \cdot \sigma$ is a generalization of the BLL formula $!_{a<p}\sigma$ to arbitrary index terms. As such it can be thought of as representing the type $\sigma\{\mathtt{0}/a\} \otimes \cdots \otimes \sigma\{\mathrm{I} - 1/a\}$. In analogy to what happens in the standard linear logic decomposition of the intuitionistic arrow, i.e. $!A \multimap B = A \Rightarrow B$, it is sufficient to restrict to modal types appearing in negative position, similarly to DLAL [25]. Finally, for those readers with some knowledge of DML, modal types are in a way similar to DML's subset sort constructions [11].

In the typing rules, modal types need to be 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 $A = [a < \mathrm{I}] \cdot \mu\{a/c\}$ and that $B = [b < \mathrm{J}] \cdot \mu\{\mathrm{I} + b/c\}$. In other words, A consists of the first I instances of μ, i.e. $\mu\{\mathtt{0}/c\}, \ldots, \mu\{\mathrm{I} - 1/c\}$ while B consists of the next J instances of μ, i.e. $\mu\{\mathrm{I} + \mathtt{0}/c\}, \ldots, \mu\{\mathrm{I} + \mathrm{J} - 1/c\}$. Their *sum* $A \uplus B$ is naturally defined as a modal type consisting of the first $\mathrm{I} + \mathrm{J}$ instances of μ, i.e. $[c < \mathrm{I} + \mathrm{J}] \cdot \mu$. An operation of bounded sum on modal types can be defined by generalizing the idea above: suppose that

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

Then its *bounded sum* $\sum_{a<\mathrm{I}} A$ is $[c < \sum_{a<\mathrm{I}} \mathrm{J}] \cdot \sigma$.

To every type σ corresponds a type $(\!|\sigma|\!)$ of ordinary PCF, namely a type built from the basic type $\mathtt{Nat}$ and the arrow operator $\rightarrow$.

Central to dℓPCF is the notion of subtyping. An inequality relation $\sqsubseteq$ between (basic and modal) types can be defined using the formal system in Figure 1. This relation corresponds to lifting index inequalities at the type level. The equality $\phi; \Phi \vdash \sigma \cong \tau$ holds when both $\phi; \Phi \vdash \sigma \sqsubseteq \tau$ and $\phi; \Phi \vdash \tau \sqsubseteq \sigma$ can be derived from the rules in Figure 1.

It is now time to introduce the main object of this paper, namely the type system dℓPCF. *Typing judgements* of dℓPCF are expressions in the form

$$\phi; \Phi; \Gamma \vdash^{\mathcal{E}}_{\mathrm{I}} t : \sigma \tag{5}$$

where Γ is a *typing context*. That is, a set of term variable assignments of the shape $x : A$ where each variable x occurs at most once. The expression (5) can be informally read as follows: for every values of the index variables in ϕ satisfying Φ, t can be given type σ and *cost* I once its free term variables have types as in Γ. In proving this, equations from $\mathcal{E}$ can be used.

Typing rules are in Figure 2, where binary and bounded sums are used in their natural generalization to contexts. A *type derivation* is nothing more than a tree built according to typing rules. A *precise type derivation* is a type derivation such that all premises in the form $\sigma \sqsubseteq \tau$ (respectively, in the form $\mathrm{I} \leqslant \mathrm{J}$) are required to be in the form $\sigma \cong \tau$ (respectively, $\mathrm{I} = \mathrm{J}$).

$$\frac{\begin{array}{c}\phi;\Phi \models^{\mathcal{E}} \mathrm{K} \leqslant \mathrm{I} \\ \phi;\Phi \models^{\mathcal{E}} \mathrm{J} \leqslant \mathrm{H}\end{array}}{\phi;\Phi \vdash^{\mathcal{E}} \mathtt{Nat}[\mathrm{I},\mathrm{J}] \sqsubseteq \mathtt{Nat}[\mathrm{K},\mathrm{H}]} \qquad \frac{\begin{array}{c}\phi;\Phi \vdash^{\mathcal{E}} B \sqsubseteq A \\ \phi;\Phi \vdash^{\mathcal{E}} \sigma \sqsubseteq \tau\end{array}}{\phi;\Phi \vdash^{\mathcal{E}} A \multimap \sigma \sqsubseteq B \multimap \tau} \qquad \frac{\begin{array}{c}\phi,a;\Phi,a<\mathrm{J} \vdash^{\mathcal{E}} \sigma \sqsubseteq \tau \\ \phi;\Phi \models^{\mathcal{E}} \mathrm{J} \leqslant \mathrm{I}\end{array}}{\phi;\Phi \vdash^{\mathcal{E}} [a<\mathrm{I}]\cdot\sigma \sqsubseteq [a<\mathrm{J}]\cdot\tau}$$

Fig. 1. The subtyping relation

$$\frac{\phi;\Phi \vdash^{\mathcal{E}} [a<\mathrm{I}]\cdot\sigma \sqsubseteq [a<1]\cdot\tau}{\phi;\Phi;\Gamma,x:[a<\mathrm{I}]\cdot\sigma \vdash^{\mathcal{E}}_{0} x:\tau\{0/a\}} \qquad \frac{\phi;\Phi;\Gamma,x:[a<\mathrm{I}]\cdot\sigma \vdash^{\mathcal{E}}_{\mathrm{J}} t:\tau}{\phi;\Phi;\Gamma \vdash^{\mathcal{E}}_{\mathrm{J}} \lambda x.t:[a<\mathrm{I}]\cdot\sigma \multimap \tau}\ L \qquad \frac{\begin{array}{c}\phi;\Phi;\Gamma \vdash^{\mathcal{E}}_{\mathrm{J}} t:[a<\mathrm{I}]\cdot\sigma \multimap \tau \\ \phi,a;\Phi,a<\mathrm{I};\Delta \vdash^{\mathcal{E}}_{\mathrm{K}} u:\sigma \\ \phi;\Phi \vdash^{\mathcal{E}} \Sigma \sqsubseteq \Gamma \uplus \sum_{a<\mathrm{I}}\Delta\end{array}}{\phi;\Phi;\Sigma \vdash^{\mathcal{E}}_{\mathrm{J}+\sum_{a<\mathrm{I}}\mathrm{K}+\mathrm{I}} tu:\tau}$$

$$\frac{\begin{array}{c}\phi;\Phi \models^{\mathcal{E}} \mathrm{I} \leqslant n \\ \phi;\Phi \models^{\mathcal{E}} n \leqslant \mathrm{J}\end{array}}{\phi;\Phi;\Gamma \vdash^{\mathcal{E}}_{0} \underline{\mathtt{n}}:\mathtt{Nat}[\mathrm{I},\mathrm{J}]} \qquad \frac{\begin{array}{c}\phi;\Phi \vdash^{\mathcal{E}} \mathtt{Nat}[\mathrm{I}+1,\mathrm{J}+1] \sqsubseteq \mathtt{Nat}[\mathrm{K},\mathrm{H}] \\ \phi;\Phi;\Gamma \vdash^{\mathcal{E}}_{\mathrm{L}} t:\mathtt{Nat}[\mathrm{I},\mathrm{J}]\end{array}}{\phi;\Phi;\Gamma \vdash^{\mathcal{E}}_{\mathrm{L}} \mathtt{s}(t):\mathtt{Nat}[\mathrm{K},\mathrm{H}]} \qquad \frac{\begin{array}{c}\phi;\Phi \vdash^{\mathcal{E}} \mathtt{Nat}[\mathrm{I}\dot{-}1,\mathrm{J}\dot{-}1] \sqsubseteq \mathtt{Nat}[\mathrm{K},\mathrm{H}] \\ \phi;\Phi;\Gamma \vdash^{\mathcal{E}}_{\mathrm{L}} t:\mathtt{Nat}[\mathrm{I},\mathrm{J}]\end{array}}{\phi;\Phi;\Gamma \vdash^{\mathcal{E}}_{\mathrm{L}} \mathtt{p}(t):\mathtt{Nat}[\mathrm{K},\mathrm{H}]}$$

$$\frac{\begin{array}{c}\phi;\Phi;\Gamma \vdash^{\mathcal{E}}_{\mathrm{K}} t:\mathtt{Nat}[\mathrm{I},\mathrm{J}] \\ \phi;\Phi,\mathrm{I}\leqslant 0;\Delta \vdash^{\mathcal{E}}_{\mathrm{H}} u:\sigma \\ \phi;\Phi,\mathrm{J}\geqslant 1;\Delta \vdash^{\mathcal{E}}_{\mathrm{H}} v:\sigma \\ \phi;\Phi \vdash^{\mathcal{E}} \Sigma \sqsubseteq \Gamma \uplus \Delta\end{array}}{\phi;\Phi;\Sigma \vdash^{\mathcal{E}}_{\mathrm{K}+\mathrm{H}} \ \mathtt{ifz}\ t\ \mathtt{then}\ u\ \mathtt{else}\ v:\sigma} \qquad \frac{\begin{array}{c}\phi,b;\Phi,b<\mathrm{L};\Gamma,x:[a<\mathrm{I}]\cdot\sigma \vdash^{\mathcal{E}}_{\mathrm{K}} t:\tau \\ \phi;\Phi \vdash^{\mathcal{E}} \tau\{0/b\} \sqsubseteq \mu \\ \phi,a,b;\Phi,a<\mathrm{I},b<\mathrm{L} \vdash^{\mathcal{E}} \tau\{{\bigcirc\!\!\!\!\!\triangle}_{b}^{b+1,a}\,\mathrm{I}+1/b\} \sqsubseteq \sigma \\ \phi;\Phi \vdash^{\mathcal{E}} \Sigma \sqsubseteq \sum_{b<\mathrm{L}+1}\Gamma \\ \phi;\Phi \models^{\mathcal{E}} {\bigcirc\!\!\!\!\!\triangle}_{b}^{0,1}\,\mathrm{I} \leqslant \mathrm{L}\end{array}}{\phi;\Phi;\Sigma \vdash^{\mathcal{E}}_{\mathrm{L}+\sum_{b<\mathrm{L}}\mathrm{K}} \mathtt{fix}\ x.t:\mu}$$

Fig. 2. Typing rules

First of all, observe that the typing rules are syntax-directed: given a term t, all type derivations for t end with the same typing rule, namely the one corresponding to the last syntax rule used in building t. In particular, no explicit subtyping rule exists, but subtyping is applied to the context in every rule. A syntax-directed type system offers a key advantage: it allows one to prove the statements about type derivations by induction on the structure of terms. This greatly simplifies the proof of crucial properties like subject reduction.

Typing rules have premises of three different kinds:

- Of course, typing a term requires typing its immediate subterms, so typing judgements can be rule premises.
- As just mentioned, typing rules allow to subtype the context Γ, so subtyping judgements can be themselves rule premises.
- The application of typing rules (and also of subtyping rules, see Figure 1) sometimes depends on the truth of some inequalities between index terms in the model induced by $\mathcal{E}$.

As a consequence, typing rules can only be applied if some relations between index terms are consequences of the constraints in Φ. These assumptions have a semantic nature, but could of course be verified by any sound formal system. Completeness (see Section V), however, only holds if all *true* inequalities can be used as assumptions. As a consequence, type inference but also type (derivation) checking are bound to be problematic from a computational point of view. See Section VI for a more thorough discussion on this issue.

As a last remark, note that each rule can be seen as a decoration of a rule of ordinary PCF. More: for every dℓPCF type derivation π of $\phi;\Phi;\Gamma \vdash^{\mathcal{E}}_{\mathrm{I}} t:\sigma$ there is a structurally identical derivation in PCF for the same term, i.e. a derivation $(\!|\pi|\!) : (\!|\Gamma|\!) \vdash t : (\!|\sigma|\!)$.

C. Properties

This section is mainly concerned with subject reduction. Subject reduction will only be proved for closed terms, since the language is endowed with a weak notion of reduction and, as a consequence, reduction cannot happen in the scope of a lambda abstraction. The system dℓPCF enjoys some nice properties that are both necessary intermediate steps towards proving subject reduction and essential ingredients for proving soundness and relative completeness. These properties permit to manipulate the judgements being sure that derivability is preserved.

First of all, the constraints Φ in a typing judgement can be made stronger without altering the rest:

Lemma 1 (Constraint Strenghtening): Let $\phi;\Phi;\Gamma \vdash_{\mathrm{I}} t:\sigma$ and $\phi;\Psi \models^{\mathcal{E}} \Phi$. Then, $\phi;\Psi;\Gamma \vdash_{\mathrm{I}} t:\sigma$.

Moreover, subtyping can be freely applied both to the context Γ (contravariantly) and to the type σ (covariantly), leaving the rest of the judgement unchanged:

Lemma 2 (Subtyping): Suppose $\phi;\Phi;x_1 : A_1,\ldots,x_n : A_n \vdash_{\mathrm{I}} t:\sigma$ and $\phi;\Phi \vdash B_i \sqsubseteq A_i$ for $1 \leqslant i \leqslant n$ and $\phi;\Phi \vdash \sigma \sqsubseteq \tau$. Then, $\phi;\Phi;x_1:B_1,\ldots,x_n:B_n \vdash_{\mathrm{I}} t:\tau$.

Another useful transformation on type derivations is substitution of an index variable for an index term:

Lemma 3 (Index Term Substitution): Let $\phi,a;\Phi;\Gamma \vdash_{\mathrm{I}} t : \sigma$. Then we have $\phi;\Phi\{\mathrm{J}/a\};\Gamma\{\mathrm{J}/a\} \vdash_{\mathrm{I}\{\mathrm{J}/a\}} t : \sigma\{\mathrm{J}/a\}$ for every J such that $\phi,\Phi \models^{\mathcal{E}} \mathrm{J} \Downarrow$.

We are now ready to embark on a proof of subject reduction. As usual, the first step is a substitution lemma:

Lemma 4 (Term Substitution): Let $\phi;\Phi,a<\mathrm{I};\varnothing\vdash_{\mathrm{J}} t:\sigma$ and $\phi;\Phi;x:[a<\mathrm{I}]\cdot\sigma,\Delta\vdash_{\mathrm{K}} u:\tau$. Then we have $\phi;\Phi;\Delta\vdash_{\mathrm{H}} u\{t/x\}:\tau$ with $\phi;\Phi\models^{\mathcal{E}}\mathrm{H}\leqslant\mathrm{K}+\sum_{a<\mathrm{I}}\mathrm{J}$.

Proof: As usual, this is an induction on the structure of a type derivation for u. All relevant inductive cases require some manipulation of the type derivation for t. The previous lemmas give exactly the right degree of "malleability". ■

Theorem 1 (Subject Reduction): Let $\phi;\Phi;\varnothing\vdash_{\mathrm{I}} t:\sigma$ and $t\rightarrow u$. Then, $\phi;\Phi;\varnothing\vdash_{\mathrm{J}} u:\sigma$, where $\phi;\Phi\models\mathrm{J}\leqslant\mathrm{I}$.

Proof: If the redex fired in u is an application of an abstraction, of s or of p to their argument, then Lemma 4 quickly leads to the thesis. The only problematic case is the unfolding rule for the fixpoint operator: $\mathtt{fix}\ x.t\rightarrow t\{\mathtt{fix}\ x.t/x\}$. This requires proving that a new type derivation must be built for $\mathtt{fix}\ x.t$ different from the one existing by hypothesis. This is done by exploiting, among others, Equation (3). ■

IV. Intensional Soundness

Subject reduction already implies an *extensional* notion of soundness for programs: if a term t can be typed with $\varnothing;\varnothing;\varnothing\vdash_{\mathrm{K}} t:\mathtt{Nat}[\mathrm{I},\mathrm{J}]$, then its normal form (if any) is a natural number between $[\![\mathrm{I}]\!]$ and $[\![\mathrm{J}]\!]$. However, subject reduction does not tell us whether the evaluation of t terminates, and in how much time. Has K anything to do with the complexity of evaluating t? The only information that can be extracted from the Subject Reduction Theorem is that K does not increase.

In this section, *intensional soundness* (Theorem 2 below) for the type system dℓPCF will be proved. A Krivine's Machine $\mathsf{K}_{\mathsf{PCF}}$ for dℓPCF programs will be first defined in Section IV-A. Given a program (i.e. a close term of ground type), the machine $\mathsf{K}_{\mathsf{PCF}}$ either evaluates it to normal form or diverges. A formal connection between the machine $\mathsf{K}_{\mathsf{PCF}}$ and the type system dℓPCF will be established by means of a weighted typability notion for machine configurations, introduced in Section IV-B. This notion is the fundamental ingredient to keep track of the number of machine steps.

A. The $\mathsf{K}_{\mathsf{PCF}}$ Machine

The Krivine's Machine has been introduced as a natural device to evaluate pure lambda-terms under a weak-head notion of reduction [26]. Here, the standard Krivine's Machine is extended to a machine $\mathsf{K}_{\mathsf{PCF}}$ which handles not only abstractions and applications, but also constants, conditionals and fixpoints.

The *configurations* of the machine $\mathsf{K}_{\mathsf{PCF}}$, ranged over by $C,D,\ldots$, are triples $C=(t,\rho,\xi)$ where ρ and ξ are two additional constructions: ρ is an *environment*, that is a (possibly empty) finite sequence of *closures*; while ξ is a (possibly empty) *stack* of *contexts*. Stacks are ranged over by $\xi,\theta,\ldots$. A *closure*, as usual, is a pair $c=(t,\rho)$ where t is a term and ρ is an environment. A *context* is either a closure, a term s, a term p, or a triple (u,v,ρ) where u,v are terms and ρ is an environment.

The transition steps between configurations of the machine $\mathsf{K}_{\mathsf{PCF}}$ are given in Figure 3. The transition rules need some comments. First of all, a naïve management of name variables is used. A more effective description could be, however, given by using the standard de Bruijn indexes. Note that the triple (u,v,ρ) is used as a context for the conditional construction; moreover, in a recursion step, a copy of the recursive term is put in a closure on the top of the current environment. As usual, the symbol $\rightarrow^{*}$ denotes the reflexive and transitive closure of the transition relation $\rightarrow$. The relation $\rightarrow^{*}$ implements weak-head reduction. Weak-head normal form and the normal form coincide for programs. So the machine $\mathsf{K}_{\mathsf{PCF}}$ is a correct device to evaluate programs. For this reason, the notation $t\Downarrow\underline{\mathrm{n}}$ can be used as a shorthand for $(t,\epsilon,\epsilon)\rightarrow^{*}(\underline{\mathrm{n}},\epsilon,\epsilon)$. Moreover, notations like $C\Downarrow^{n}$ could also be used to stress that C reduces to an irreducible configuration in exactly n steps. The proof of the formal correctness of the implementation is outside the scope of this paper, however it should be clear that it could be obtained as a simple extension of the original one [26].

Intensional soundness will be proved by studying how the weight I of any program t evolves during the evaluation of t by $\mathsf{K}_{\mathsf{PCF}}$. This is possible because every reduction step in t is decomposed into a number of transitions in $\mathsf{K}_{\mathsf{PCF}}$, and this decomposition highlights *when*, precisely, the weight changes. The same would be more difficult when performing plain reduction on terms. Proving intensional soundness this way requires, however, to keep track of the types and weights of all objects in a machine configuration. In other words, the type system should be somehow generalized to an assignment system on *configurations*.

B. Types and Weights for Configurations

Assigning types and weights to configurations amounts to somehow keeping track of the nature of all terms appearing in environments and stacks. This is captured by the following complex, although natural, definitions:

Definition 1: A closure $(t,c_1\cdots c_n)$ is said to be $(\phi;\Phi)$-*typable with type σ and weight* J if

$$\phi;\Phi;x_1:[a<\mathrm{I}_1]\cdot\tau_1,\ldots,x_n:[a<\mathrm{I}_n]\cdot\tau_n\vdash_{\mathrm{K}} t:\sigma$$

and each c_i is $(\phi,a;\Phi,a<\mathrm{I}_i)$-typable with type τ_i and weight H_i $(1\leqslant i\leqslant n)$, and

$$\phi;\Phi\models\mathrm{J}=\mathrm{K}+\mathrm{I}_1+\ldots+\mathrm{I}_n+\sum_{a<\mathrm{I}_1}\mathrm{H}_1+\ldots+\sum_{a<\mathrm{I}_n}\mathrm{H}_n.$$

A stack ξ is said to be $(\phi;\Phi)$-*acceptable for σ with weight* J *and type τ* if either:

- $\xi=\varepsilon$, and $\phi;\Phi\models\mathrm{J}=0$, and $\sigma=\tau$;
- or $\xi=c\cdot\theta$, $\sigma=[a<\mathrm{I}]\cdot\gamma\multimap\mu$, c is $(\phi,a;\Phi,a<\mathrm{I})$-typable with type γ and weight K, θ is $(\phi;\Phi)$-acceptable for μ with weight H and type τ and $\phi;\Phi\models\mathrm{J}=\mathrm{H}+\sum_{a<\mathrm{I}}\mathrm{K}+\mathrm{I}$;

Term	**Environment**	**Stack**		**Term**	**Environment**	**Stack**
tu	ρ	ξ	$\to$	t	ρ	$(u,\rho)\cdot\xi$
$\lambda x.t$	ρ	$c\cdot\xi$	$\to$	t	$c\cdot\rho$	ξ
x	$(t_0,\rho_0)\cdot\dots\cdot(t_n,\rho_n)$	ξ	$\to$	t_x	ρ_x	ξ
$\mathtt{ifz}\ t\ \mathtt{then}\ u\ \mathtt{else}\ v$	ρ	ξ	$\to$	t	ρ	$(u,v,\rho)\cdot\xi$
$\mathtt{fix}\ x.t$	ρ	ξ	$\to$	t	$(\mathtt{fix}\ x.t,\rho)\cdot\rho$	ξ
$\underline{\mathtt{n}}$	ρ	$\mathtt{s}\cdot\xi$	$\to$	$\underline{\mathtt{n+1}}$	ρ	ξ
$\underline{\mathtt{n}}$	ρ	$\mathtt{p}\cdot\xi$	$\to$	$\underline{\mathtt{n-1}}$	ρ	ξ
$\underline{\mathtt{0}}$	ρ	$(t,u,\mu)\cdot\xi$	$\to$	t	μ	ξ
$\underline{\mathtt{n+1}}$	ρ	$(t,u,\mu)\cdot\xi$	$\to$	u	μ	ξ
$\mathtt{s}(t)$	ρ	ξ	$\to$	t	ρ	$\mathtt{s}\cdot\xi$
$\mathtt{p}(t)$	ρ	ξ	$\to$	t	ρ	$\mathtt{p}\cdot\xi$

Fig. 3. The $\mathsf{K}_{\mathsf{PCF}}$ machine transition steps.

- or $\xi = \mathtt{s}\cdot\theta$, $\sigma = \mathtt{Nat}[\mathrm{I},\mathrm{L}]$ and θ is itself $(\phi;\Phi)$-acceptable for some $\mathtt{Nat}[\mathrm{K},\mathrm{H}]$ with weight J and type τ, where $\phi;\Phi \vdash \mathtt{Nat}[\mathrm{I}+1,\mathrm{L}+1] \sqsubseteq \mathtt{Nat}[\mathrm{K},\mathrm{H}]$;
- or $\xi = \mathtt{p}\cdot\theta$, $\sigma = \mathtt{Nat}[\mathrm{I},\mathrm{L}]$ and θ is itself $(\phi;\Phi)$-acceptable for some $\mathtt{Nat}[\mathrm{K},\mathrm{H}]$ with weight J and type τ, where $\phi;\Phi \vdash \mathtt{Nat}[\mathrm{I}\dot{-}1,\mathrm{L}\dot{-}1] \sqsubseteq \mathtt{Nat}[\mathrm{K},\mathrm{H}]$;
- or $\xi = (t,u,\rho)\cdot\theta$, $\sigma = \mathtt{Nat}[\mathrm{I},\mathrm{L}]$, (t,ρ) is $(\phi;\Phi,\mathrm{I}\leqslant 0)$-typable with type μ and weight K, (u,ρ) is $(\phi;\Phi,\mathrm{L}\geqslant 1)$-typable with type μ and weight K, θ is $(\phi;\Phi)$-acceptable for μ with weight H and type τ, and $\phi;\Phi \models \mathrm{J} = \mathrm{K}+\mathrm{H}$.

A configuration (t,ρ,ξ) is $(\phi;\Phi)$-*typable with type* τ *and weight* I if the closure (t,ρ) is $(\phi;\Phi)$-typable with type σ and weight K and the environment ξ is $(\phi;\Phi)$-acceptable for σ with type τ and weight J, and $\phi;\Phi \models \mathrm{I} = \mathrm{J}+\mathrm{K}$.

A formal connection between typed terms and typed configurations could be established as expected, moreover, such connection could be shown to be preserved by reduction. However, the following lemma will be sufficient in the sequel.

Lemma 5: Let $t \in \mathcal{P}$. Then, $\phi;\Phi;\varnothing \vdash_{\mathrm{I}} t : \sigma$ if and only if (t,ϵ,ϵ) is $(\phi;\Phi)$-typable with type σ and weight I.

Analogous notions of acceptability and typability can be given for PCF. They can be obtained by simplifying those for dℓPCF and are not given here (see [19]).

C. Measure Decreasing and Intensional Soundness

An important property of Krivine's Machine says that during the evaluation of programs only subterms of the initial program are recorded in the environment. This justifies the notion of *size* for configurations, denoted $|C|$, that will be used in the sequel. This is defined as $|(t,\rho,\xi)| = |t| + |\xi|$. The size $|\xi|$ of a stack ξ is defined as the sum of sizes of its elements, and $|(t,\rho)| = |t|$, $|\mathrm{s}| = 2$, $|\mathrm{p}| = 1$, and $|(t,u,\rho)| = |t| + |u|$. Moreover, another consequence of the same property is the following lemma.

Lemma 6: Let $t \in \mathcal{P}$ and let $C = (t,\epsilon,\epsilon)$. Then, for each $D = (u,\rho,\xi)$ such that $C \to^* D$ and for each v in ρ and ξ, $|v| \leqslant |t|$.

Intensional Soundness (Theorem 2) will express the fact that for a program $t \in \mathcal{P}$ such that $\varnothing;\varnothing;\varnothing \vdash^{\mathcal{E}}_{\mathrm{I}} t : \mathtt{Nat}[\mathrm{J},\mathrm{K}]$, the number $[\![\mathrm{I}]\!]^{\mathcal{E}}_{\rho}$ is a good estimate of the number of steps needed to evaluate the program. Moreover, thanks to subject reduction, the numbers $[\![\mathrm{J}]\!]^{\mathcal{E}}_{\rho}$ and $[\![\mathrm{K}]\!]^{\mathcal{E}}_{\rho}$ give an upper and a lower bound, respectively, to the result of such an evaluation. This is proved by showing that during the reduction a measure, expressed as the combination of the weight and the size of a configuration, decreases.

This requires, in particular, to extend some of the properties in Section III-C from terms to configurations. As an example, substitution holds on configurations, too:

Lemma 7: If (t,ρ) is $(\phi,a;\Phi)$-typable with type σ and weight H, then it is also $(\phi;\Phi\{\mathrm{J}/a\})$-typable with type $\tau\{\mathrm{J}/a\}$ and weight $\mathrm{H}\{\mathrm{J}/a\}$ for every J such that $\phi,\Phi \models^{\mathcal{E}} \mathrm{J}\Downarrow$.

Now we have:

Lemma 8 (Measure Decreasing): Suppose $(t,\epsilon,\epsilon) \quad \to^* D \to E$ and let D be $(\phi;\Phi)$-typable with weight I and type σ. Then one of the following holds:

1. E is $(\phi;\Phi)$-typable with weight J and type σ, $\phi;\Phi \models \mathrm{I} = \mathrm{J}$ but $|D| > |E|$;
2. E is $(\phi;\Phi)$-typable with weight J and type σ, $\phi;\Phi \models \mathrm{I} > \mathrm{J}$ and $|E| < |D| + |t|$;

Proof sketch.: The proof is by cases on the reduction $D \to E$. Condition 1 can be shown to apply to all the cases but the one in which $D = (x, c_1\cdots c_n, \xi)$. In that one, weight decreasing relies on the side condition in the typing rule for variables, while the bound on the size increasing comes from Lemma 6. ■

It is worth noticing that if Φ is inconsistent, the inequality $\phi;\Phi \models \mathrm{I} > \mathrm{J}$ in Lemma 8, point 2, does not necessary imply weight decreasing. Indeed, intensional soundness only holds in presence of an empty set of constraints:

Theorem 2 (Intensional Soundness): Let $\varnothing;\varnothing;\varnothing \vdash_{\mathrm{I}} t : \mathtt{Nat}[\mathrm{J},\mathrm{K}]$ and $t \Downarrow^n \underline{\mathtt{m}}$. Then, $n \leqslant |t|\cdot[\![\mathrm{I}]\!]$.

Proof: By induction on n, making essential use of Lemma 8 and Lemma 6. ■

V. Relative Completeness

This section is devoted to proving *relative completeness* for the type system dℓPCF. In fact, two relative completeness theorems will be presented. The first one (Theorem 4) states relative completeness *for programs*: for each PCF program t that evaluates to a numeral $\underline{\mathtt{n}}$ there is a type derivation in dℓPCF whose index terms capture both the number of reduction steps and the value of $\underline{\mathtt{n}}$. The second one (Theorem 5) states relative completeness *for functions*: for each PCF term $t : \mathtt{Nat} \rightarrow \mathtt{Nat}$ computing a *total* function f in time expressed by a function g there exists a type derivation in dℓPCF whose index terms capture both the extensional behaviour f and the intensional property embedded into g.

Relative completeness does not hold in general. Indeed, it holds only when the underlying equational program $\mathcal{E}$ is *universal*, i.e. when it is sufficiently expressive to encode all total computable functions. A universal equational program is introduced in Section V-A.

Relative completeness for programs will be proved using a weighted form of *subject expansion* (Theorem 3) similar to the one holding in intersection type theories. This will be proved in Section V-B. The proof of relative completeness for functions needs a further step: a *uniformization* result (Lemma 11) relying on the properties of the universal model. This is the subject of Section V-C.

A. Universal Equational Program

Since the class of equational programs is clearly recursively enumerable, it can be put in one-to-one correspondence with natural numbers, using a coding scheme $\ulcorner\cdot\urcorner$ *à la Gödel*. Such a coding, as usual, can be used to define a *universal equational program* $\mathcal{U}$ that is able to simulate all equational programs (including itself).

Let $\ulcorner\mathcal{E},\mathtt{f}\urcorner$ be the natural number coding an equational program $\mathcal{E}$ and a function symbol $\mathtt{f}$ defined on it. This can be easily computed from (a description of) $\mathcal{E}$ and $\mathtt{f}$. A signature $\Sigma_{\mathcal{U}}$ containing just the symbol $\mathtt{empty}$ of arity 0 and the symbols $\mathtt{pair}$ and $\mathtt{eval}$ of arity 2 is sufficient to define the universal program $\mathcal{U}$. For each $\mathtt{f}$ of arity n, the equational program $\mathcal{U}$ satisfies

$$[\![\mathtt{eval}(\ulcorner\mathcal{E},\mathtt{f}\urcorner, \mathit{pairing}_n(x_1,\dots,x_n))]\!]^{\mathcal{U}}_\rho = [\![\mathtt{f}(x_1,\dots,x_n)]\!]^{\mathcal{E}}_\rho$$

where $\mathit{pairing}_n(t_1,\dots,t_n)$ is defined by induction on n:

$$\mathit{pairing}_0 = \mathtt{empty};$$
$$\mathit{pairing}_{n+1}(t_1,\dots,t_{n+1}) = \mathtt{pair}(\mathit{pairing}_n(t_1,\dots,t_n),t_{n+1}).$$

This way, $\mathcal{U}$ acts as an interpreter for any equational program.

The universal equational program $\mathcal{U}$ enjoys some nice properties which are crucial when proving subject expansion. The following lemma says, for example, that sums and bounded sums can always be formed (modulo $\cong$) whenever index terms are built and reasoned about using the universal program:

Lemma 9: 1. For every A and B such that $(\!|A|\!) = (\!|B|\!)$ there are C and D such that $\phi;\emptyset \vdash^{\mathcal{U}} C \cong A$, $\phi;\emptyset \vdash^{\mathcal{U}} D \cong B$ and $C \uplus D$ is defined.

2. For every A and I there is B such that $\phi,a;\emptyset \vdash^{\mathcal{U}} B \cong A$ and $\sum_{a<\mathrm{I}} B$ is defined.

B. Subject Expansion and Programs Relative Completeness

Weighted subject expansion (Theorem 3) says that typing is preserved while weights increase by at most one along any $\mathsf{K}_{\mathsf{PCF}}$ expansion step. This is somehow the converse of the Measure Decreasing Lemma. Weighted subject expansion, however, does not hold in general but only when the universal equational program $\mathcal{U}$ is considered.

In order to prove weighted subject expansion, only typing that carry precise weight information should be considered. Formally, a configuration D is said to be $(\phi;\Phi)$*-precisely-typable with weight* I *and type* σ if it is $(\phi;\Phi)$-typable with weight I and type σ by precise type derivations. The type of a precisely-typable configuration, in other words, carries exact information about the value of the objects at hand.

Furthermore, only particular typing transformations should be considered, namely those that leave the weight information unaltered. In order to achieve this, some properties of $(\phi;\Phi)$-typability for the $\mathsf{K}_{\mathsf{PCF}}$ machine should be exploited. As an example, if a closure (t,ρ) is $(\phi;\Phi)$-typable with type σ and weight I; then it is also $(\phi;\Phi)$-typable with type τ and weight J for every τ and J such that $\phi;\Phi \vdash \sigma \cong \tau$ and $\phi;\Phi \models \mathrm{I} = \mathrm{J}$.

Moreover, it is worth noticing that by considering an inconsistent set of constraints Φ, it is possible to make a closure (t,ρ) typable with type σ (in the sense of PCF) to be also $(\phi;\Phi)$-typable with type τ and weight I for each τ such that $(\!|\tau|\!) = \sigma$ and for each weight I. This says that inconsistent sets cover a role similar to the ω-rule in intersection type systems.

Lemma 9 permits to prove some results which allow to manipulate contexts and types in a way which is somehow dual to the usual one, e.g. to the one of Lemma 6. An example is the following:

Lemma 10: Let (t,ρ) be $(\phi,a;\Phi,a<\mathrm{I})$-typable with type σ and weight H and $(\phi,a;\Phi,a<\mathrm{J})$-typable with type $\sigma\{a+\mathrm{I}/a\}$ and weight $\mathrm{H}\{a+\mathrm{I}/a\}$. Then, (t,ρ) is also $(\phi,a;\Phi,a<\mathrm{I}+\mathrm{J})$-typable with type σ and weight H.

It is now time to state weighted subject expansion, since all the necessary ingredients have been introduced:

Theorem 3 (Weighted Subject Expansion): If D is $(\phi;\Phi)$-precisely-typable with weight I and type σ and C is typable with type $(\!|\sigma|\!)$. Then, $C \rightarrow D$ implies that C is $(\phi;\Phi)$-precisely-typable with weight J and type σ where $\phi;\Phi \models \mathrm{J} \leqslant \mathrm{I}+1$.

Relative completeness for programs is a direct consequence of weighted subject expansion:

Theorem 4 (Relative Completeness for Programs): Let t be a PCF program such that $t \Downarrow^n \underline{\mathtt{m}}$. Then, there exist two index terms I and J such that $[\![\mathrm{I}]\!]^{\mathcal{U}} \leqslant n$ and $[\![\mathrm{J}]\!]^{\mathcal{U}} = m$ and such that the term t is typable in dℓPCF as $\emptyset;\emptyset;\emptyset \vdash^{\mathcal{U}}_{\mathrm{I}} t : \mathtt{Nat}[\mathrm{J}]$.

Proof: By induction on n using the Weighted Subject Expansion Theorem and Lemma 5. ■

C. Uniformization and Relative Completeness for Functions

It is useful to recall that by *relative completeness for functions* we mean the following: for each PCF term t computing a total function f in time expressed by a function g there exists a type derivation in dℓPCF whose index terms capture both the extensional functional behaviour f and the intensional property g. Anticipating on what follows, and using an intuitive notation, this can be expressed by a typing judgement similar to $x : \mathtt{Nat}[a] \vdash_{\mathtt{g}(a)} t : \mathtt{Nat}[\mathtt{f}(a)]$. In order to show this form of relative completeness, a *uniformization* result for type derivations needs to be proved.

Suppose that $\{\pi\}_{n\in\mathbb{N}}$ is a sufficiently "regular" (i.e. recursively enumerable) family of type derivations such that any π_n is mapped by $(\!|\cdot|\!)$ to the *same* PCF type derivation. Uniformization tells us that with the hypothesis above, there is a *single* type derivation π which captures the whole family $\{\pi_n\}_{n\in\mathbb{N}}$. In other words, uniformization is as an extreme form of polymorphism. Note that, for instance, uniformization does not hold in intersection types, where *uniform typing* permits only to define small classes of functions [27], [28], [29].

More formally, a family $\{\pi_n\}_{n\in\mathbb{N}}$ of type derivations is said to be *recursively enumerable* if there is a computable function f which, on input n, returns (an encoding of) π_n. Similarly, recursively enumerable families of index terms, types and modal types can be defined.

Lemma 11 (Uniformizing Type Derivations): Suppose $\{\pi_n\}_{n\in\mathbb{N}}$ recursively enumerable, and that:

1. for every $n \in \mathbb{N}$, $\pi_n : \phi; \Phi_n; \Gamma_n \vdash^{\mathcal{U}}_{\mathrm{I}_n} t : \sigma_n$;
2. for every $n \in \mathbb{N}$, $\Phi_n = \mathrm{J}^n_1 \leqslant \mathrm{K}^n_1, \ldots, \mathrm{J}^n_m \leqslant \mathrm{K}^n_m$ where m does not depend on n;
3. for every $n, m \in \mathbb{N}$, $(\!|\Gamma_n|\!) = (\!|\Gamma_m|\!)$;
4. for every $n, m \in \mathbb{N}$, $(\!|\sigma_n|\!) = (\!|\sigma_m|\!)$.

Then there is one type derivation π such that:

1. $\pi : \phi, a; \Phi; \Gamma \vdash^{\mathcal{U}}_{\mathrm{I}} t : \sigma$;
2. $\Phi = \mathrm{J}_1 \leqslant \mathrm{K}_1, \ldots, \mathrm{J}_m \leqslant \mathrm{K}_m$ where $\phi; \varnothing \models^{\mathcal{U}} \mathrm{J}_p\{n/a\} \simeq \mathrm{J}^n_p$ and $\phi; \varnothing \models^{\mathcal{U}} \mathrm{K}_p\{n/a\} \simeq \mathrm{K}^n_p$ for every $n \in \mathbb{N}$ and for every $1 \leqslant p \leqslant m$.
3. $\phi; \varnothing \vdash^{\mathcal{U}} \sigma\{n/a\} \cong \sigma_n$ for every $n \in \mathbb{N}$;
4. $\phi; \varnothing \vdash^{\mathcal{U}} \Gamma\{n/a\} \cong \Gamma_n$ for every $n \in \mathbb{N}$;
5. $\phi; \varnothing \models^{\mathcal{U}} \mathrm{I}\{n/a\} = \mathrm{I}_n$ for ever $n \in \mathbb{N}$.

Proof: By induction on the structure of t using the properties of the universal equational program $\mathcal{U}$. ■

Uniformization is the key to prove relative completeness for functions from relative completeness for programs:

Theorem 5 (Relative Completeness for Functions): Suppose that t is a PCF term such that $\vdash t : \mathtt{Nat} \rightarrow \mathtt{Nat}$. Moreover, suppose that there are two (total and computable) functions $f, g : \mathbb{N} \rightarrow \mathbb{N}$ such that $t\ \underline{\mathtt{n}} \Downarrow^{g(n)} \underline{\mathtt{f}(\mathtt{n})}$, there are terms $\mathrm{I}, \mathrm{J}, \mathrm{K}$ with $[\![\mathrm{I} + \mathrm{J}]\!] \leqslant g$ and $[\![\mathrm{K}]\!] = f$, such that

$$a; \varnothing; \varnothing \vdash^{\mathcal{U}}_{\mathrm{I}} t : [b < \mathrm{J}] \cdot \mathtt{Nat}[a] \multimap \mathtt{Nat}[\mathrm{K}].$$

Proof: A consequence of relative completeness for programs (Theorem 4) and Lemma 11. Indeed, a type derivation for $a; \varnothing; \varnothing \vdash_{\mathrm{I}} t : [b < \mathrm{J}] \cdot \mathtt{Nat}[a] \multimap \mathtt{Nat}[\mathrm{K}]$ can be obtained simply by uniformizing all type derivations π_n for programs in the form $t\underline{\mathtt{n}}$. In turn, those type derivations can be built effectively by way of subject expansion. ■

VI. On the Undecidability of Type Checking

As we have seen in the last two sections, dℓPCF is not only sound, but complete: all true typing judgements involving programs can be derived, and this can be indeed lifted to first-order functions, as explained in Section V-C.

There is a price to pay, however. Checking a type derivation for correctness is undecidable in general, simply because it can rely on semantic assumptions in the form of inequalities between index terms, or on subtyping judgements, which themselves rely on the properties of the underlying equational program $\mathcal{E}$. If $\mathcal{E}$ is sufficiently involved, e.g. if we work with $\mathcal{U}$, there is no hope to find a decidable complete type checking procedure. In this sense, dℓPCF is a non-standard type system.

Indeed, dℓPCF is not actually a type system, but rather a *framework* in which various distinct type systems can be defined. Concrete type systems can be developed along two axis: on the one hand by concretely instantiating $\mathcal{E}$, on the other by choosing specific and sound formal systems for the verification of semantic assumptions. This way sound and possibly decideable type systems can be derived. Even if completeness can only be achieved if $\mathcal{E}$ is universal, soundness holds for every equational program $\mathcal{E}$. Choosing a simple equational program $\mathcal{E}$ results in a (probably incomplete) type system for which the problem of checking the inequalities can be much easier, if not decidable. And even if $\mathcal{E}$ remains universal, assumptions could be checked using techniques such as abstract interpretation or theorem proving.

By the way, the just described problem is not peculiar to dℓPCF. Unsurprisingly, program logics have the same problem, since the rule

$$\frac{p \Rightarrow r \quad \{r\}P\{s\} \quad s \Rightarrow q}{\{p\}P\{q\}}$$

is part of most relatively complete Hoare-Floyd logics and, of course, the premises $p \Rightarrow r$ and $s \Rightarrow q$ have to be taken semantically for completeness to hold.

VII. dℓPCF and Implicit Computational Complexity

One of the original motivations for the studies which lead to the definition of dℓPCF came from Implicit Computational Complexity. There, one aims at giving characterizations of complexity classes which can often be turned into type systems or static analysis methodologies for the verification of resource usage of programs. Historically [30], [31], what prevented most ICC techniques to find concrete applications along this line was 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. This is true despite the fact that ICC systems are *extensionally* complete — they capture complexity classes seen as classes of *functions*.

The kind of intensional completeness enjoyed by dℓPCF is much stronger: all PCF programs with a certain complexity can be proved to be so by deriving a typing judgement for them.

Of course, dℓPCF is not at all an implicit system: bounds appear everywhere! On the other hand, dℓPCF allows to analyze the time complexity of higher-order functional programs directly, without translating them into low level programs. In other words, dℓPCF can be viewed as an abstract framework where to experiment new implicit computational complexity techniques.

VIII. Related Work

Other type systems can be proved to satisfy completeness properties similar to the ones enjoyed by dℓPCF.

The first example that comes to mind is the one of intersection types. In intersection type disciplines, the class of strongly and weakly normalizable lambda terms can be captured [32]. Recently, these results have been refined in such a way that the actual complexity of reduction of the underlying term can be read from its type derivation [33], [34]. What intersection types lack is the possibility to analyze the behaviour of a functional term in one single type derivation — all function calls must be typed separately [27], [28], [29]. This is in contrast with Theorem 5 which gives a unique type derivation for every PCF program computing a total function on the natural numbers.

Another example of type theories which enjoy completeness properties are refinement type theories [35], as shown in [36]. Completeness, however, only holds in a logical sense: any property which is true in all Henkin models can be captured by refinement types. The kind of completeness we obtain here is clearly more operational: the result of evaluating a program and the time complexity of the process can both be read off from its type.

As already mentioned in the Introduction, linear logic has been a great source of inspiration for the authors. Actually, it is not a coincidence that linear logic was a key ingredient in the development of one of the earliest fully-abstract game model for PCF. Indeed, dℓPCF can be seen as a way to internalize history-free game semantics [37] into a type system. And already BLL and QBAL, both precursors of dℓPCF, have been designed being greatly inspired by the geometry of interaction. dℓPCF is a way to study the extreme consequences of this idea, when bounds are not only polynomials but arbitrary first-order total functions on natural numbers.

Acknowledgment

The second author would like to thank Frank Pfenning for a brief but very insightful conversation about DML properties.

References

[1] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," *JCS*, vol. 4, no. 2/3, pp. 167–188, 1996.

[2] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE JSAC*, vol. 21, no. 1, pp. 5–19, 2003.

[3] G. Barthe, B. Grégoire, and C. Riba, "Type-based termination with sized products," in *CSL*, ser. LNCS, vol. 5213. Springer, 2008, pp. 493–507.

[4] N. Kobayashi and 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.

[5] S. Jost, K. Hammond, H.-W. Loid, and M. Hofmann, "Static Determination of Quantitative Resource Usage for Higher-Order Programs," in *ACM POPL*, Madrid, Spain, 2010.

[6] K. R. Apt, F. S. de Boer, and E.-R. Olderog, *Verification of Sequential and Concurrent Programs*, ser. T. in Comp. Sci. Springer-Verlag, 2009.

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

[8] P. Baillot and K. Terui, "Light types for polynomial time computation in lambda calculus," *I & C*, vol. 207, no. 1, pp. 41–62, 2009.

[9] P. Baillot, M. Gaboardi, and V. Mogbil, "A polytime functional language from light linear logic," in *ESOP*, ser. LNCS, vol. 6012. Springer, 2010, pp. 104–124.

[10] H. Xi and F. Pfenning, "Dependent types in practical programming," in *ACM POPL*, 1999, pp. 214–227.

[11] H. Xi, "Dependent ml an approach to practical programming with dependent types," *J. of Funct. Progr.*, vol. 17, no. 2, pp. 215–286, 2007.

[12] G. D. Plotkin, "LCF considerd as a programming language," *Theor. Comp. Sci.*, vol. 5, pp. 225–255, 1977.

[13] C. A. Gunter, *Semantics of Programming Languages: Structures and Techniques*, ser. Found. of Comp. Series. MIT Press, 1992.

[14] J.-Y. Girard, "Linear logic," *Theor. Comp. Sci.*, vol. 50, pp. 1–102, 1987.

[15] U. Dal Lago and M. Hofmann, "Bounded linear logic, revisited," in *TLCA*, ser. LNCS, vol. 5608. Springer, 2009, pp. 80–94.

[16] S. A. Cook, "Soundness and completeness of an axiom system for program verification," *SIAM J. on Computing*, vol. 7, pp. 70–90, 1978.

[17] J. Girard, A. Scedrov, and P. Scott, "Bounded linear logic," *Theor. Comp. Sci.*, vol. 97, no. 1, pp. 1–66, 1992.

[18] B. Grobauer, "Cost recurrences for DML programs," in *ICFP*, 2001, pp. 253–264.

[19] U. Dal Lago and M. Gaboardi, "Linear dependent types and relative completeness," 2011, available at http://arxiv.org/abs/1104.0193.

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

[21] J. Hoffmann, K. Aehlig, and M. Hofmann, "Multivariate Amortized Resource Analysis," in *ACM POPL*, 2011, pp. 357–370.

[22] U. Dal Lago, "Context semantics, linear logic and computational complexity," *ACM TOCL*, vol. 10, no. 4, 2009.

[23] P. Odifreddi, *Classical Recursion Theory: the Theory of Functions and Sets of Natural Numbers*, ser. Studies in Logic and the Foundations of Mathematics. North-Holland, 1989, no. 125.

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

[25] P. Baillot and K. Terui, "Light types for polynomial time computation in lambda calculus," *Inf. Comput.*, vol. 207, no. 1, pp. 41–62, 2009.

[26] J.-L. Krivine, "A call-by-name lambda-calculus machine," *Higher-Order and Symbolic Computation*, vol. 20, no. 3, pp. 199–207, 2007.

[27] D. Leivant, "Discrete polymorphism," in *ACM LFP*. ACM Press, 1990, pp. 288–297.

[28] A. Bucciarelli, S. D. Lorenzis, A. Piperno, and I. Salvo, "Some computational properties of intersection types," in *LICS*. IEEE Comp. Soc., 1999, pp. 109–118.

[29] A. Bucciarelli, A. Piperno, and I. Salvo, "Intersection types and lambda-definability," *MSCS*, vol. 13, no. 1, pp. 15–53, 2003.

[30] M. Hofmann, "Programming languages capturing complexity classes," *ACM SIGACT News*, vol. 31, pp. 31–42, 2000.

[31] J.-Y. Marion, "Complexité implicite des calculs, de la théorie à la pratique," Habilitation thesis, Université Nancy 2, 2000.

[32] M. Dezani-Ciancaglini, E. Giovannetti, and U. de' Liguoro, "Intersection Types, Lambda-models and Böhm Trees," in *"Theories of Types and Proofs"*. Math. Soc. of Japan, 1998, vol. 2, pp. 45–97.

[33] D. de Carvalho, "Execution time of lambda-terms via denotational semantics and intersection types," *CoRR*, vol. abs/0905.4251, 2009.

[34] A. Bernadet and S. Lengrand, "Complexity of strongly normalising λ-terms via non-idempotent intersection types," in *FOSSACS*, ser. LNCS, vol. 6604. Springer, 2011, pp. 88–107.

[35] T. Freeman and F. Pfenning, "Refinement types for ML," in *PLDI*, 1991, pp. 268–277.

[36] E. Denney, "Refinement types for specification," in *IFIP-PROCOMET*, 1998, pp. 148–166.

[37] S. Abramsky, R. Jagadeesan, and P. Malacaria, "Full abstraction for PCF," *I & C*, vol. 163, no. 2, pp. 409–470, 2000.