The geometry of types

Ugo Dal Lago, Barbara Petit

The Geometry of Types

Ugo Dal Lago Barbara Petit
Università di Bologna & INRIA
{dallago,petit}@cs.unibo.it

Abstract

We show that time complexity analysis of higher-order functional programs can be effectively reduced to an arguably simpler (although computationally equivalent) verification problem, namely checking first-order inequalities for validity. This is done by giving an efficient inference algorithm for linear dependent types which, given a PCF term, produces in output both a linear dependent type and a cost expression for the term, together with a set of proof obligations. Actually, the output type judgement is derivable *iff* all proof obligations are valid. This, coupled with the already known *relative completeness* of linear dependent types, ensures that no information is lost, i.e., that there are no false positives or negatives. Moreover, the procedure reflects the difficulty of the original problem: simple PCF terms give rise to sets of proof obligations which are easy to solve. The latter can then be put in a format suitable for automatic or semi-automatic verification by external solvers. Ongoing experimental evaluation has produced encouraging results, which are briefly presented in the paper.

Categories and Subject Descriptors F.3.2 [*Logics And Meanings Of Programs*]: Semantics of Programming Languages—Program Analysis; F.3.1 [*Logics And Meanings Of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Performance, Theory, Verification

Keywords Functional Programming, Higher-order Types, Linear Logic, Resource Consumption, Complexity Analysis

1. Introduction

One of the most crucial non-functional properties of programs is the amount of resources (like time, memory and power) they need when executed. Deriving upper bounds on the resource consumption of programs is crucial in many cases, but is in fact an undecidable problem as soon as the underlying programming language is non-trivial. If the units of measurement in which resources are measured become concrete and close to the physical ones, the problem becomes even more complicated, given the many transformation and optimisation layers programs are applied to before being executed. A typical example is the one of WCET techniques adopted in real-time systems [29], which do not only need to deal with how many machine instructions a program corresponds to, but also with how much time each instruction costs when executed by possibly complex architectures (including caches, pipelining, etc.), a task which is becoming even harder with the current trend towards multicore architectures.

A different approach consists in analysing the *abstract* complexity of programs. As an example, one can take the number of instructions executed by the program as a measure of its execution time. This is of course a less informative metric, which however becomes more accurate if the actual time complexity *of each instruction* is kept low. One advantage of this analysis is the independence from the specific hardware platform executing the program at hand: the latter only needs to be analysed once. A variety of verification techniques have been employed in this context, from abstract interpretation [20] to type systems [23] to program logics [3] to interactive theorem proving[1].

Among the many type-based techniques for complexity analysis, a recent proposal consists in going towards systems of *linear dependent types*, as suggested by Marco Gaboardi and the first author [11]. In linear dependent type theories, a judgement has the form $\vdash_{\mathrm{I}} t : \sigma$, where σ is the type of t and I is its *cost*, an estimation of its time complexity. In this paper, we show that the problem of checking, given a PCF term t and I, whether $\vdash_{\mathrm{I}} t : \sigma$ holds can be efficiently reduced to the one of checking the truth of a set of proof obligations, themselves formulated in the language of a *first-order* equational program. Interestingly, simple λ-terms give rise to simple equational programs. In other words, linear dependent types are not only a sound and *relatively* complete methodology for inferring *time* bounds of programs [11, 13]: they also allow to reduce complexity analysis to an arguably simpler (although computationally equivalent) problem which is much better studied and for which a variety of techniques and concrete tools exist [6]. Noticeably, the bounds one obtains this way translate to bounds on the number of steps performed by evaluation machines for the λ-calculus, which means that the induced metrics are not too abstract after all. The type inference algorithm is described in Section 4.

The scenario, then, becomes similar to the one in Floyd-Hoare program logics for imperative programs, where completeness holds [9] (at least for the simplest idioms [7]) *and* weakest preconditions can be generated automatically (see, e.g., [3]). A benefit of working with functional programs is that type inference — the analogue of generating WPs — can be done compositionally without the need of guessing invariants.

Linear dependent types are simple types annotated with some *index terms*, i.e. first-order terms reflecting the value of data flowing inside the program. Type inference produces in output a type derivation, a set of inequalities (which should be thought of as proof obligations) and an equational program $\mathcal{E}$ giving meaning to function symbols appearing in index terms (see Figure 1). A natural thing to do once $\mathcal{E}$ and the various proof obligations are available is to try to solve them automatically, as an example through SMT solvers. If automatically checking the inequalities for truth does not

POPL'13, January 23–25, 2013, Rome, Italy.

[1] A detailed discussion with related work is in Section 6.

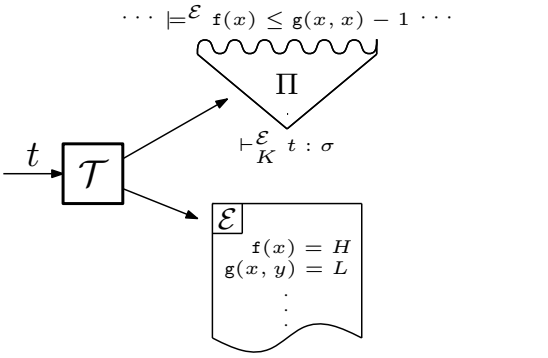


Figure 1. General scheme of the type inference algorithm.

succeed (which must happen, in some cases), one can anyway find useful information in the type derivation, as it tells you precisely *which* data every symbol corresponds to. We elaborate on this issue in Section 5.

But where does linear dependency come from? Linear dependent types can be seen as a way to turn Girard's geometry of interaction (or, equivalently, AJM games [1]) into a type system for the λ-calculus: the equational program one obtains as a result of type inference of a term t is nothing but as a description of a token machine [14] for t. In presence of linear dependency, any term which can possibly be duplicated, can receive different, although uniform, types, similarly to what happens in BLL [19]. As such, this form of dependency is significantly simpler than the one of, e.g., the calculus of inductive constructions.

2. Linear Dependency at a Glance

Traditionally, type systems carry very little information about the *value* of data manipulated by programs, instead focusing on their *nature*. As an example, all (partial recursive) functions from natural numbers to natural numbers can be typed as $\mathtt{Nat} \Rightarrow \mathtt{Nat}$ in the λ-calculus with natural numbers and higher-order recursion, also known as PCF [26]. This is not an intrinsic limit of the type-based analysis of programs, however: much richer type disciplines have flourished in the last twenty years [4, 15, 22]. All of them guarantee stronger properties for typable programs, the price being a more complicated type language and computationally more difficult type inference and checking problems. As an example, sized types [22] are a way to ensure termination of functional programs based on size information. In systems of sized types, a program like

$$t = \lambda x.\lambda y.\mathrm{add}\ (\mathrm{add}\ x\ y)\ (\mathrm{succ}\ y)$$

can be typed as $\mathtt{Nat}_a \Rightarrow \mathtt{Nat}_b \Rightarrow \mathtt{Nat}_{a+2b+1}$, and in general as $\mathtt{Nat}_a \Rightarrow \mathtt{Nat}_b \Rightarrow \mathtt{Nat}_{\mathrm{I}}$, where $\mathrm{I} \geqslant a + 2b + 1$. In other words, the PCF type $\mathtt{Nat}$ is refined into $\mathtt{Nat}_{\mathrm{I}}$ (where I is an arithmetical expression) whose semantics is the set of all natural numbers smaller or equal to I, i.e. the interval $[0, \mathrm{I}] \subseteq \mathbb{N}$. The role of size information is to ensure that all functions terminate, and this is done by restricting the kind of functions of which one is allowed to form fixpoints. Sized types are nonlinear: arguments to functions can be freely duplicated. Moreover, the size information is only approximate, since the expression labelling base types is only an *upper* bound on the size of typable values.

Linear dependent types can be seen as a way to inject precision and linearity into sized types. Indeed, t receives the following type in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ [13]:

$$\mathtt{Nat}[0,a] \overset{c<\mathrm{I}}{\multimap} \mathtt{Nat}[0,b] \overset{d<\mathrm{J}}{\multimap} \mathtt{Nat}[0, a+2b+1]. \qquad (1)$$

As one can easily realise, $\mathtt{Nat}[\mathrm{K},\mathrm{H}]$ is the type of all natural numbers in the interval $[\mathrm{K},\mathrm{H}] \subseteq \mathbb{N}$. Moreover, $\sigma \overset{b<\mathrm{J}}{\multimap} \tau$ is the type of *linear* functions from σ to τ which can be copied by the environment J times. The J copies of the function have types obtained by substituting $0, \ldots, \mathrm{J}-1$ for b in σ and τ. This is the key idea behind linear dependency. The type (1) is imprecise, but can be easily turned into

$$\mathtt{Nat}[a,a] \overset{c<\mathrm{I}}{\multimap} \mathtt{Nat}[b,b] \overset{d<\mathrm{J}}{\multimap} \mathtt{Nat}[a+2b+1, a+2b+1], \qquad (2)$$

itself a type of t. In the following, the singleton interval type $\mathtt{Nat}[\mathrm{K},\mathrm{K}]$ is denoted simply as $\mathtt{Nat}[\mathrm{K}]$.

Notice that linear dependency is not exploited in (2), e.g., d does not appear free in $\mathtt{Nat}[b]$ nor in $\mathtt{Nat}[a+2b+1]$. Yet, (2) precisely captures the functional behaviour of t. If d does not appear free in σ nor in τ, then $\sigma \overset{d<\mathrm{I}}{\multimap} \tau$ can be abbreviated as $\sigma \overset{\mathrm{I}}{\multimap} \tau$. Linear dependency becomes necessary in presence of higher-order functions. Consider, as another example, the term

$$u = \lambda x.\lambda y.\ \mathtt{ifz}\ y\ \mathtt{then}\ \underline{0}\ \mathtt{else}\ xy$$

u has simple type $(\mathtt{Nat} \Rightarrow \mathtt{Nat}) \Rightarrow \mathtt{Nat} \Rightarrow \mathtt{Nat}$. One way to turn it into a linear dependent type is the following

$$(\mathtt{Nat}[a] \overset{\mathrm{H}}{\multimap} \mathtt{Nat}[\mathrm{I}]) \overset{\mathrm{K}}{\multimap} \mathtt{Nat}[a] \overset{\mathrm{L}}{\multimap} \mathtt{Nat}[\mathrm{J}], \qquad (3)$$

where J equals 0 when $a = 0$ and J equals I otherwise. Actually, u has type (3) *for every* I and J, provided the two expressions are in the appropriate relation. Now, consider the term

$$v = (\lambda x.\lambda y.(x\ \mathrm{pred}\ (x\ \mathrm{id}\ y)))\ u.$$

The same variable x is applied to the identity id and to the predecessor pred. Which type should we give to the variable x and to u, then? If we want to preserve precision, the type should reflect *both* uses of x. The right type for u is actually the following

$$(\mathtt{Nat}[a] \overset{1}{\multimap} \mathtt{Nat}[\mathrm{I}]) \overset{c<2}{\multimap} \mathtt{Nat}[a] \overset{1}{\multimap} \mathtt{Nat}[\mathrm{J}], \qquad (4)$$

where both I and J evaluate to a if $c = 0$ and to $a - 1$ otherwise. If id is replaced by succ in the definition of v, then (4) becomes even more complicated: the first "copy" of J is fed not with a but with either 0 or $a + 1$.

Linear dependency precisely consists in allowing different copies of a term to receive types which are indexed differently (although having the same "functional skeleton") and to represent all of them in compact form. This is in contrast to, e.g., intersection types, where the many different ways a function uses its argument could even be structurally different. This, as we will see in Section 3, has important consequences on the kind of completeness results one can hope for: if the language in which *index terms* are written is sufficiently rich, then the obtained system is complete in an intensional sense: a precise type can be given to *every* terminating t *having type* $\mathtt{Nat} \Rightarrow \mathtt{Nat}$.

Noticeably, linear dependency allows to get precise information about the functional behaviour of programs without making the language of types too different from the one of simple types (e.g., one does not need to quantify over index variables, as in sized types). The price to pay, however, is that types, and especially higher-order types, need to be *context aware*: when you type u as a subterm of v (see above) you need to know which arguments u will be applied to. Despite this, a genuinely compositional type inference procedure can actually be designed and is the main technical contribution of this paper.

2.1 Linearity, Abstract Machines and the Complexity of Evaluation

Why dependency, but specially *linearity*, are so useful for complexity analysis? Actually, typing a term using linear dependent types requires finding an upper bound to the number of times each value is copied by its environment, called its *potential*. In the term v from the example above, the variable x is used twice, and accordingly one finds $c < 2$ in (4). Potentials of higher-order values occurring in a term are crucial parameters for the complexity of evaluating the term by abstract mechanisms [10]. The following is an hopefully convincing (but necessarily informal) discussion about why this is the case.

Configurations of abstract machines for the λ-calculus (like Friedman and Felleisen's CEK and Krivine's KAM) can be thought of as being decomposable into two distinct parts:

- First of all, there are *duplicable* entities which are either copied entirely or turned into non-duplicable entities. This includes, in particular, terms in so-called environments. Each (higher-order) duplicable entity is a subterm of the term the computation started from.
- There are *non-duplicable* entities that the machine uses to look for the next redex to be fired. Typically, these entities are the current term and (possibly) the stack. The essential feature of non-duplicable entities is the fact that they are progressively "consumed" by the machine: the search for the next redex somehow consists in traversing the non-duplicable entities until a redex is found *or* a duplicable entity needs to be turned into a non-duplicable one.

As an example, consider the process of evaluating the PCF term

$$(\lambda f.\ \texttt{ifz}\ (f\,\underline{0})\ \texttt{then}\ \underline{0}\ \texttt{else}\ f(f\,\underline{0}))((\lambda x.\lambda y.\mathrm{add}\,x\,y)\ \underline{3})$$

by an appropriate generalisation of the CEK, see Figure 2. Initially, the whole term is non-duplicable. By travelling into it, the machine finds a first redex u; at that point, $\underline{3}$ becomes duplicable. The obtained closure itself becomes part of the environment ξ, and the machine looks into the body of t, ending up in an occurrence of f, which needs to be replaced by a copy of $\xi(f)$. After an *instantiation step*, a new non-duplicable entity $\xi(f)$ indeed appears. Note that, by an easy combinatorial argument, the number of machine steps necessary to reach f is at most (proportional to) the size of the starting term tu, since reaching f requires consuming non-duplicable entities which can only be created through instantiations. After a copy of $\xi(f)$ becomes non-duplicable, some additional "nonduplicable fuel" becomes available, but not too much: $\lambda y.\mathrm{add}\,x\,y$ is after all a subterm of the initial term.

The careful reader should already have guessed the moral of this story: when analysing the time complexity of evaluation, we could limit ourselves to counting how many *instantiation steps* the machine performs (as opposed to counting *all* machine steps). We claim, on the other hand, that the number of instantiation steps *equals* the sum of potentials of all values appearing in the initial term, something that can be easily inferred from the kind of precise linear typing we were talking about at the beginning of this section.

Summing up, once a dependently linear type has been attributed to a term t, the time complexity of evaluating t can be derived somehow for free: not only an expression bounding the number of instantiation steps performed by an abstract machine evaluating t can be derived, but it is part of the underlying type derivation, essentially. As a consequence, reasoning (automatically or not) about it can be done following the structure of the program.

3. Programs and Types, Formally

In this section, we present some details of dℓPCF, a system of *linear dependent* types for PCF [26]. Two versions exist: dℓPCF$_{\mathsf{N}}$ and dℓPCF$_{\mathsf{V}}$, corresponding to *call-by-name* and *call-by-value* evaluation of terms, respectively. The two type systems are different, but the underlying idea is basically the same. We give here the details of the CBV version [13], which better corresponds to widespread intuitions about evaluation, but also provide some indications about the CBN setting [11].

3.1 Terms and Indexes

Terms are given by the usual PCF grammar:

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

A *value* (denoted by v, w etc.) is either a primitive integer $\underline{\mathtt{n}}$, or an abstraction $\lambda x.t$, or a fixpoint $\mathtt{fix}\,x.t$. In addition to the usual terms of the λ-calculus, there are a fixpoint construction, primitive natural numbers with predecessor and successor, and conditional branching with a test for zero. For instance, a simple program computing addition is the following:

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

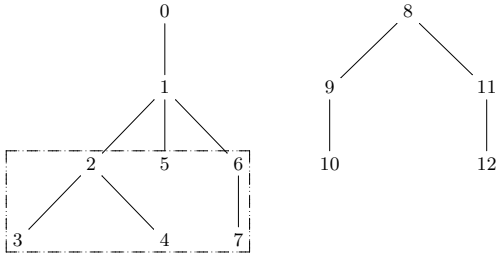
3.1.1 Language of Indexes

As explained informally in Section 2, a type in dℓPCF consists in an annotation of a PCF type, where the annotation consists in some *indexes*. The latter are parametrised by a set of index variables $\mathcal{V} = \{a, b, c, \dots\}$ and an untyped signature Θ of *function symbols*, denoted by f, g, h, etc. We assume Θ contains at least the arithmetic symbols $+$, $-$, 0 and 1, and we write $\underline{n}$ for $1 + \cdots + 1$ (n times). Indexes are then constructed by the following grammar:

$$\mathrm{I,J,K}\ ::=\quad a\ \mid\ \mathrm{f}(\mathrm{I}_1,\dots,\mathrm{I}_n)\ \mid\ \sum_{a<\mathrm{I}}\mathrm{J}\ \mid\ \bigtriangleup_a^{\mathrm{I,J}}\mathrm{K},$$

where n is the arity of f in Θ. Free and bound index variables are defined as usual, taking care that all free occurrences of a in J are bound in both $\sum_{a<\mathrm{I}}\mathrm{J}$ and $\bigtriangleup_a^{\mathrm{I,K}}\mathrm{J}$. The substitution of a variable a by J in I is written $\mathrm{I}\{\mathrm{J}/a\}$. Given an *equational program* $\mathcal{E}$ attributing a *meaning* in $\mathbb{N}^n \rightharpoonup \mathbb{N}$ to some symbols of arity n, and a *valuation* ρ mapping index variables to $\mathbb{N}$, the *semantics* $[\![\mathrm{I}]\!]_\rho^{\mathcal{E}}$ of an index I is either a natural number or undefined. Let us describe how we interpret the last two constructions, namely *bounded sums* and *forest cardinalities*.

Bounded sums have the usual meaning: $\sum_{a<\mathrm{I}}\mathrm{J}$ is simply the sum of all possible values of J with a taking the values from 0 up to I, excluded. Describing the meaning of forest cardinalities, on the other hand, requires some effort. Informally, the index $\bigtriangleup_a^{\mathrm{I,J}}\mathrm{K}$ counts the number of nodes in a forest composed of J trees described using K. Each node in the forest is (uniquely) identified by a natural number, starting from I and visiting the tree in preorder. The index K has the role of describing the number of children of each forest node, e.g. the number of children of the node 0 is $\mathrm{K}\{0/a\}$. Consider the following forest comprising two trees:



and consider an index K with a free index variable a such that $\mathrm{K}\{1/a\} = 3$; $\mathrm{K}\{\underline{n}/a\} = 2$ for $n \in \{2, 8\}$; $\mathrm{K}\{\underline{n}/a\} = 1$ when $n \in$

$t = \lambda f.\, \texttt{ifz}\ (f\ \underline{0})\ \texttt{then}\ \underline{0}\ \texttt{else}\ f(f\ \underline{0});$ $\quad u = (\lambda x.\lambda y.\mathrm{add}\, x\, y)\ \underline{3};$ $\quad \xi = f \mapsto \langle\, \lambda y.\mathrm{add}\, x\, y\,;\, x \mapsto \langle\, \underline{3}\,\rangle\rangle.$

$$
\begin{array}{rrl}
\langle\, tu\,\rangle \star \diamond & \succ & \langle\, t\,\rangle \star \mathsf{arg}\langle\, u\,\rangle \cdot \diamond \\
& \succ & \langle\, u\,\rangle \star \mathsf{fun}\langle\, t\,\rangle \cdot \diamond \\
& \succ & \langle\, \lambda x.\lambda y.\mathrm{add}\, x\, y\,\rangle \star \mathsf{arg}\langle\, \underline{3}\,\rangle \cdot \mathsf{fun}\langle\, t\,\rangle \cdot \diamond \\
& \succ^* & \langle\, \lambda y.\mathrm{add}\, x\, y\,;\, x \mapsto \langle\, \underline{3}\,\rangle\rangle \star \mathsf{fun}\langle\, t\,\rangle \cdot \diamond \\
& \succ & \langle\, \texttt{ifz}\ (f\ \underline{0})\ \texttt{then}\ \underline{0}\ \texttt{else}\ f(f\ \underline{0})\,;\, \xi\,\rangle \star \diamond \\
& \succ & \langle\, f\ \underline{0}\,;\, \xi\,\rangle \star \mathsf{fork}\langle\, \underline{0}\,;\, f(f\ \underline{0})\,;\, \xi\,\rangle \cdot \diamond \\
& \succ & \langle\, f\,;\, \xi\,\rangle \star \mathsf{arg}\langle\, \underline{0}\,;\, \xi\,\rangle \cdot \mathsf{fork}\langle\, \underline{0}\,;\, f(f\ \underline{0})\,;\, \xi\,\rangle \cdot \diamond \\
& \succ & \langle\, \lambda y.\mathrm{add}\, x\, y\,;\, x \mapsto \langle\, \underline{3}\,\rangle\rangle \star \mathsf{arg}\langle\, \underline{0}\,;\, \xi\,\rangle \cdot \mathsf{fork}\langle\, \underline{0}\,;\, f(f\ \underline{0})\,;\, \xi\,\rangle \cdot \diamond \\
& \succ^* & \langle\, \mathrm{add}\, x\, y\,;\, (y \mapsto \langle\, \underline{0}\,;\, \xi\,\rangle, x \mapsto \langle\, \underline{3}\,\rangle)\,\rangle \star \mathsf{fork}\langle\, \underline{0}\,;\, f(f\ \underline{0})\,;\, \xi\,\rangle \cdot \diamond \\
& \succ^* & \langle\, \underline{3}\,\rangle \star \mathsf{fork}\langle\, \underline{0}\,;\, f(f\ \underline{0})\,;\, \xi\,\rangle \cdot \diamond \\
& \succ & \langle\, f(f\ \underline{0})\,;\, \xi\,\rangle \star \diamond \\
& \succ & \langle\, f\,;\, \xi\,\rangle \star \mathsf{arg}\langle\, f\ \underline{0}\,;\, \xi\,\rangle \cdot \diamond \\
& \succ & \langle\, \lambda y.\mathrm{add}\, x\, y\,;\, x \mapsto \langle\, \underline{3}\,\rangle\rangle \star \mathsf{arg}\langle\, f\ \underline{0}\,;\, \xi\,\rangle \cdot \diamond \\
& \succ & \langle\, f\ \underline{0}\,;\, \xi\,\rangle \star \mathsf{fun}\langle\, \lambda y.\mathrm{add}\, x\, y\,;\, x \mapsto \langle\, \underline{3}\,\rangle\rangle \cdot \diamond \\
& \succ^* & \langle\, \underline{3}\,\rangle \star \mathsf{fun}\langle\, \lambda y.\mathrm{add}\, x\, y\,;\, x \mapsto \langle\, \underline{3}\,\rangle\rangle \cdot \diamond \\
& \succ^* & \langle\, \underline{6}\,\rangle \star \diamond
\end{array}
$$

Figure 2. Evaluation of a term in the $\mathsf{CEK}_{\mathsf{PCF}}$ abstract machine.

$\{0, 6, 9, 11\}$; and $\mathrm{K}\{\underline{n}/a\} = 0$ when $n \in \{3, 4, 7, 10, 12\}$. 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 recursive calls produces a tree-like structure whose size is just the number of times the (recursively defined) function will be used *globally*.

Notice that $[\![\mathrm{I}]\!]_\rho^{\mathcal{E}}$ is undefined whenever the equality between I and any natural number cannot be derived from the underlying equational program. In particular, a forest cardinality may be undefined even if all its subterms are defined: as an example $\mathrm{I} = \triangle_a^{0,1}\,1$ has no value, because the corresponding tree consists of an infinite descending chain and its cardinality is infinite. By the way I is the index term describing the structure of the recursive calls induced by the program $\texttt{fix}\ x.x$.

3.1.2 Semantic Judgements

A *constraint* is an inequality on indexes. A constraint $\mathrm{I} \leqslant \mathrm{J}$ is *valid* for ρ and $\mathcal{E}$ when both $[\![\mathrm{I}]\!]_\rho^{\mathcal{E}}$ and $[\![\mathrm{J}]\!]_\rho^{\mathcal{E}}$ are defined, and $[\![\mathrm{I}]\!]_\rho^{\mathcal{E}} \leqslant [\![\mathrm{J}]\!]_\rho^{\mathcal{E}}$. As usual, we can derive a notion of equality and strict inequality from $\leqslant$. A *semantic judgement* is of the form

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

where Φ is a set of constraints and ϕ is the set of free index variables in Φ, I and J. These semantic judgements are used as axioms in the typing derivations of $\mathsf{d}\ell\mathsf{PCF}$, and the set of constraints Φ, called the *index context*, contains *mainly* some indications of bounds for the free index variables (such as $a < \mathrm{K}$). Such a judgement is valid when, for every valuation $\rho : \phi \to \mathbb{N}$, if all constraints in Φ are valid for $\mathcal{E}$ and ρ then so is $\mathrm{I} \leqslant \mathrm{J}$.

3.2 Types

Remember that $\mathsf{d}\ell\mathsf{PCF}$ is aimed at controlling the complexity of programs. The time complexity *of the evaluation* is thus analysed statically, while typing the term at hand. The grammar for types distinguishes the subclass of *linear types*, which correspond to *non-duplicable* terms (see Section 2.1), and the one of *modal types*, for *duplicable* terms. In $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, they are defined as follows:

$$
\begin{array}{rcll}
A, B & := & \sigma \multimap \tau; & \text{linear types} \\
\sigma, \tau & := & [a < \mathrm{I}] \cdot A \;\big|\; \mathtt{Nat}[\mathrm{I}, \mathrm{J}]. & \text{modal types}
\end{array}
$$

Indeed, CBV evaluation only duplicates values. If such a value has an arrow type, then it is a function (either an abstraction or a fixpoint) that can potentially increase the complexity of the whole program if we duplicate it. Hence we need a bound on the number of times we instantiate it if we want to keep the overall complexity under control. This bound, call the *potential* of the value, is represented by I in the type $[a < \mathrm{I}] \cdot (\sigma \multimap \tau)$ (also written $\sigma \overset{a<\mathrm{I}}{\multimap} \tau$). As explained in Section 2, $\mathtt{Nat}[\mathrm{I}, \mathrm{J}]$ is the type of programs evaluating to a natural number in the closed interval $[\mathrm{I}, \mathrm{J}]$. The potential of natural number values is not specified, as they can be freely duplicated along CBV evaluation.

3.2.1 Summing types

Intuitively, the modal type $\sigma \equiv [a < \mathrm{I}] \cdot A$ is assigned to terms that can be copied I times, the k^{th} copy being of type $A\{\underline{k} - 1/a\}$. For those readers who are familiar with Linear Logic, σ can be thought of as representing the type $A\{0/a\} \otimes \cdots \otimes A\{\mathrm{I} - 1/a\}$.

In the typing rules we are going to define, modal types need to be manipulated in an algebraic way. For this reason, two operations on modal types are required. The first one is a binary operation $\uplus$ on modal types. Suppose that $\sigma \equiv [a < \mathrm{I}] \cdot A\{a/c\}$ and that $\tau \equiv [a < \mathrm{J}] \cdot A\{\mathrm{I} + a/c\}$. In other words, σ consists of the first I instances of A, i.e. $A\{0/c\} \otimes \cdots \otimes A\{\mathrm{I} - 1/c\}$, while τ consists of the next J instances of A, i.e. $A\{\mathrm{I} + 0/c\} \otimes \cdots \otimes A\{\mathrm{I} + \mathrm{J} - 1/c\}$. 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 $\mathtt{Nat}[\mathrm{I}, \mathrm{J}]$. A bounded sum operator on modal types can be defined by generalising the idea above: suppose that

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

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 does not occur free in J nor in K.

3.2.2 Subtyping

Central to $\mathsf{d}\ell\mathsf{PCF}$ is the notion of subtyping. An inequality relation $\sqsubseteq$ between (linear or modal) types can be defined using the

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

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

Figure 3. Subtyping derivation rules of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$.

formal system in Fig. 3. This relation corresponds to lifting index inequalities to the type level. As defined here, $\sqsubseteq$ is a pre-order (i.e. a reflexive and transitive relation), which allows to cope with approximations in the typed analysis of programs. However, in the type inference algorithm we will present in the next section only the symmetric closure $\equiv$ of $\sqsubseteq$, called *type equivalence* will be used. This ensures that the type produced by the algorithm is *precise*.

3.2.3 Typing

A typing judgement 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 (including the potential substitutions by t itself in its evaluation context). The index context Φ is as in a semantic judgement (see Section 3.1.2), and Γ is a (term) 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, $\{x : \sigma; y : \tau\} \uplus \{x : \varrho, z : \xi\} = \{x : \sigma \uplus \varrho; y : \tau; z : \xi\}$). There might be free index variables in Φ, Γ, τ and K, all of them from ϕ. Typing judgements can be derived from the rules of Figure 4.

Observe that, in the typing rule for the abstraction ($\multimap$), I represents the number of times the value $\lambda x.t$ can be copied. Its weight (that is, the number of substitutions involving $\lambda x.t$ or one of its subterms) is then I plus, for each of these copies, the weight of t. In the typing rule (*App*), on the other hand, t is used once as a function, without been copied. Its potential needs to be at least 1. The typing rule for the fixpoint is arguably the most complicated one. As a first approximation, assume that only one copy of $\mathtt{fix}\, x.t$ will be used (that is, $\mathrm{K} \equiv 1$ and a does not occur free in B). 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 the index I (as explained in Section 3.1.1), since each occurrence of x in t stands for a recursive call. It has $\mathrm{H} = \triangle_b^{0,1} \mathrm{I}$ nodes. At each node b of this tree, there is a copy of t in which the a^{th} occurrence of x will be replaced by the a^{th} son of b, i.e. by $b+1+\triangle_b^{b+1,a} \mathrm{I}$. The types thus have to correspond, which is what the second premise of this rule prescribes. Now if $\mathtt{fix}\, x.t$ is in fact aimed at being copied $\mathrm{K} \geqslant 0$ times, then all the copies of t are represented by a forest of K trees described by I.

For the sake of simplicity, we present here the type system with an explicit subsumption rule. The latter allows to relax any bound in the types (and the weight), thereby loosing some precision in the information provided by the typing judgement. However, we could alternatively replace this rule by relaxing the premises of all the other ones (which corresponds to the presentation of the type system given in [13], or in [11] for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{N}}$). Restricting subtyping to type equivalence amounts to considering types up to index equality in the type system of Figure 4 without the rule (*Subs*) — this is what we do in the type inference algorithm in Section 4. In this case we say that the typing judgements are *precise*:

DEFINITION 3.1. *A derivable judgement* $\phi;\Phi;\Gamma \vdash^{\mathcal{E}}_{\mathrm{I}} t : \sigma$ *is* precise *if*

$$\phi;\Phi;\Delta \vdash^{\mathcal{E}}_{\mathrm{J}} t : \tau \textit{ is derivable} \implies \begin{cases} \phi;\Phi \vdash_{\mathcal{E}} \Delta \sqsubseteq \Gamma \\ \phi;\Phi \vdash_{\mathcal{E}} \sigma \sqsubseteq \tau \\ \phi;\Phi \vDash_{\mathcal{E}} \mathrm{I} \leqslant \mathrm{J} \end{cases}$$

3.2.4 Call-by-value vs. Call-by-name

In $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{N}}$, the syntax of terms and of indexes is the same as in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, but the language of types differs:

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

Modal types still represent duplicable terms, except that now not only values but any argument to functions can be duplicated. So modal types only occur in negative position in arrow types. In the same way, one can find them in the *context* of any typing judgement,

$$\phi;\Phi;x_1:\sigma_1,\ldots,x_n:\sigma_n \vdash^{\mathcal{E}}_{\mathrm{K}} t : A.$$

When a term is typed, it is *a priori* not duplicable, and its type is linear. It is turned into a duplicable term when it holds the argument position in an application. As a consequence, the typing rule (*App*) becomes the most "expansive" one (for the weight) in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{N}}$: the whole context used to type the argument has to be duplicated, whereas in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ this duplication of context is "anticipated" in the typing rules for values.

The readers who are familiar with linear logic, could have noted that if we replace modal types by banged types (and we remove all annotations with indexes), then $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{N}}$ corresponds to the target fragment of the CBN translation from simply-typed λ-calculus to LL, and $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ to the target of the CBV translation [25].

In $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{N}}$, the weight K of a typing judgement represents the maximal number of substitutions that may occur in the CBN evaluation of t. We do not detail the typing rules of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{N}}$ here (they can be found in [11]). However, an important remark is that in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{N}}$, just like in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, some semantic judgements can be found in the axioms of a typing derivation, and every typing rule is reversible (except subsumption). The type inference algorithm for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ that we present in Section 4 can be easily adapted to $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{N}}$.

3.3 Abstract Machines

The evaluation of PCF terms can be simulated through an extension $\mathsf{KAM}_{\mathsf{PCF}}$ of Krivine's abstract machine [24] (for CBN evaluation) or through an extension $\mathsf{CEK}_{\mathsf{PCF}}$ of Felleisen and Friedman's CEK machine [16] (for CBV evaluation).

Both these machines have states in the form of *processes*, that are pairs of a *closure* (i.e. a term with an environment defining its free variables) and a *stack*, representing the evaluation context. In the $\mathsf{KAM}_{\mathsf{PCF}}$, these objects are given by the following grammar:

$$\begin{array}{lrcl} \text{Closures:} & \mathsf{c} & := & \langle\, t\,;\,\xi\,\rangle; \\ \text{Environment:} & \xi & := & \{x_1 \mapsto \mathsf{c}_1;\cdots;x_k \mapsto \mathsf{c}_k\}; \\ \text{Stacks:} & \pi & := & \diamond \mid \mathsf{arg}\langle\, t\,;\,\xi\,\rangle\cdot\pi \mid \mathsf{s}\cdot\pi \mid \mathsf{p}\cdot\pi \\ & & & \quad \mid \mathsf{fork}\langle\, t\,;\,u\,;\,\xi\,\rangle\cdot\pi; \\ \text{Processes:} & P & := & \mathsf{c} \star \pi. \end{array}$$

When the environment is empty, we may use the notation $\langle\, t\,\rangle$ instead of $\langle\, t\,;\,\varnothing\,\rangle$ for closures. The evaluation rules of the $\mathsf{KAM}_{\mathsf{PCF}}$ are given in Figure 5. The fourth evaluation rule is said to be an *instantiation step*: the value of a variable x is replaced by the term x maps to in the underlying environment ξ.

The $\mathsf{CEK}_{\mathsf{PCF}}$, which performs CBV evaluation, is slightly more complex: within closures, the *value closures* are those whose first

$$\frac{}{\phi;\Phi;\Gamma,x:\sigma\vdash^{\mathcal{E}}_{0} x:\sigma}(Ax) \qquad \frac{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{\mathrm{I}} t:\sigma \qquad \phi;\Phi\vdash_{\mathcal{E}}\Delta\sqsubseteq\Gamma \qquad \phi;\Phi\vdash_{\mathcal{E}}\sigma\sqsubseteq\tau \qquad \phi;\Phi\vDash_{\mathcal{E}}\mathrm{I}\leqslant\mathrm{J}}{\phi;\Phi;\Delta\vdash^{\mathcal{E}}_{\mathrm{J}} t:\tau}(Subs)$$

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

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

$$\frac{}{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{0}\underline{\mathtt{n}}:\mathtt{Nat}[\underline{n},\underline{n}]}(n) \qquad \frac{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{\mathrm{M}} t:\mathtt{Nat}[\mathrm{I},\mathrm{J}]}{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{\mathrm{M}}\mathtt{s}(t):\mathtt{Nat}[\mathrm{I}+1,\mathrm{J}+1]}(s) \qquad \frac{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{\mathrm{M}} t:\mathtt{Nat}[\mathrm{I},\mathrm{J}]}{\phi;\Phi;\Gamma\vdash^{\mathcal{E}}_{\mathrm{M}}\mathtt{p}(t):\mathtt{Nat}[\mathrm{I}-1,\mathrm{J}-1]}(p)$$

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

Figure 4. Typing rules of $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$.

component is a value: $\mathsf{v} := \langle\, v\,;\,\xi\,\rangle$ (remember that a *value* v is of the form $\underline{\mathtt{n}}$, $\lambda x.t$ or $\mathtt{fix}\ x.t$). Moreover, environments assign only value closures to variables:

$$\xi \;:=\quad \{x_1\mapsto \mathsf{v}_1;\cdots;x_k\mapsto \mathsf{v}_k\}.$$

The grammar for stacks is the same, with one additional construction ($\mathsf{fun}(\mathsf{v})\cdot\pi$) that is used to encapsulate a function (lambda abstraction or fixpoint) while its argument is computed. Indeed, the latter cannot be substituted for a variable if it is not a value. Evaluation rules for processes are the same as the ones in Figure 5, except that the second and the third ones are replaced by the following:

$$\begin{array}{lcl}
\mathsf{v}\star \quad \mathsf{arg}(\mathsf{c})\cdot\pi & \succ & \mathsf{c}\star\mathsf{fun}(\mathsf{v})\cdot\pi\\
\mathsf{v}\star\ \mathsf{fun}\langle\,\lambda x.t\,;\,\xi\,\rangle\cdot\pi & \succ & \langle\,t\,;\,x\mapsto\mathsf{v}\cdot\xi\,\rangle\star\pi\\
\mathsf{v}\star\mathsf{fun}\langle\,\mathtt{fix}\,x.t\,;\,\xi\,\rangle\cdot\pi & \succ & \\
\multicolumn{3}{r}{\langle\,t\,;\,x\mapsto\langle\,\mathtt{fix}\,x.t\,;\,\xi\,\rangle\cdot\xi\,\rangle\star\mathsf{arg}(\mathsf{v})\cdot\pi}
\end{array}$$

An example of the evaluation of a term by the $\mathsf{CEK}_{\mathsf{PCF}}$ can be found in Figure 2.

We say that a term t *evaluates* to u in an abstract machine when $\langle\,t\,\rangle\star\diamond\succ\langle\,u\,;\,\xi\,\rangle\star\diamond$. Observe that if t is a closed term, then u is necessarily a value. We write $t\Downarrow^{n}_{\mathsf{N}} u$ whenever the $\mathsf{KAM}_{\mathsf{PCF}}$ evaluates t to u in exactly n steps, and $t\Downarrow^{n}_{\mathsf{V}} u$ when the same holds for the $\mathsf{CEK}_{\mathsf{PCF}}$ (we may also omit the exponent n when the number of steps is not relevant).

Abstract Machines and Weight. The *weight* of a typable term was informally presented as the number of instantiation steps in its evaluation. Abstract machines enable a more precise formulation of this idea:

FACT 1. *1. If $t\Downarrow_{\mathsf{N}} u$, and $\vdash^{\mathcal{E}}_{\mathrm{I}} t:A$ is derivable in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{N}}$, then $[\![\mathrm{I}]\!]^{\mathcal{E}}$ is an upper bound for the number of instantiation steps in the evaluation of t by the $\mathsf{KAM}_{\mathsf{PCF}}$.*

2. If $t\Downarrow_{\mathsf{V}} u$, and $\vdash^{\mathcal{E}}_{\mathrm{I}} t:[a<1]\cdot A$ is derivable in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, then $[\![\mathrm{I}]\!]^{\mathcal{E}}$ is an upper bound for the instantiation steps in the evaluation of t by the $\mathsf{CEK}_{\mathsf{PCF}}$.

This can be shown by extending the notion of weight and of typing judgement to stacks and processes [11, 13], and is the main ingredients for proving Intensional Soundness (see Section 3.4).

3.4 Key Properties

In this section we briefly recall the main properties of $\mathsf{d}\ell\mathsf{PCF}$, arguing for its relevance as a methodology for complexity analysis. We give the results for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, but they also hold for $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{N}}$ (all proofs can be found in [11, 13]).

The Subject Reduction Property guarantees as usual that typing is correct with respect to term reduction, but specifies also that the weight of a term cannot increase along reduction:

PROPOSITION 3.1 (Subject Reduction). *For any PCF-terms t,u, if $\phi;\Phi;\varnothing\vdash^{\mathcal{E}}_{\mathrm{I}} t:\tau$ is derivable in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, and if $t\to u$ in* CBV*, then $\phi;\Phi;\varnothing\vdash^{\mathcal{E}}_{\mathrm{J}} u:\tau$ is also derivable for some* J *such that $\phi;\Phi\vDash_{\mathcal{E}}\mathrm{J}\leqslant\mathrm{I}$.*

As a consequence, the weight does not tell us much about the number of reduction steps bringing a (typable) term to its normal form. So-called *Intensional Soundness*, on the other hand, allows to deduce some sensible information about the time complexity of evaluating a typable PCF program *by an abstract machine* from its $\mathsf{d}\ell\mathsf{PCF}$ typing judgement.

PROPOSITION 3.2 (Intensional Soundness). *For any term t, if $\vdash^{\mathcal{E}}_{\mathrm{K}} t:\mathtt{Nat}[\mathrm{I},\mathrm{J}]$ is derivable in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, then t evaluates to $\underline{\mathtt{n}}$ in k steps in the $\mathsf{CEK}_{\mathsf{PCF}}$, with $[\![\mathrm{I}]\!]^{\mathcal{E}}\leqslant n\leqslant[\![\mathrm{J}]\!]^{\mathcal{E}}$ and $k\leqslant|t|\cdot([\![\mathrm{K}]\!]^{\mathcal{E}}+1)$.*

Intensional Soundness guarantees that the evaluation of any program typable in $\mathsf{d}\ell\mathsf{PCF}$ takes (at most) a number of steps directly proportional to both its syntactic size and its weight. A similar theorem holds when t has a functional type: if, as an example, the type of t is $\mathtt{Nat}[a]\overset{1}{\multimap}\mathtt{Nat}[\mathrm{J}]$, then K is parametric on a and $(|t|+2)\cdot([\![\mathrm{K}]\!]^{\mathcal{E}}+1)$ is an upper bound on the complexity of evaluating t when fed with any integer a.

But is $\mathsf{d}\ell\mathsf{PCF}$ powerful enough to type natural complexity bounded programs? Actually, it is as powerful as PCF itself, since any PCF type derivation can be turned into a $\mathsf{d}\ell\mathsf{PCF}$ one (for an expressive enough equational program), as formalised by the type inference algorithm (Section 4). We can make this statement even more precise for terms of base or first order type, 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 representable by some index term. 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 judgements are derived in a given (recursively enumerable) formal system, because this would violate Gödel's Incompleteness Theorem. In fact, in $\mathsf{d}\ell\mathsf{PCF}$ completeness theorems are *relative* to an oracle for the truth of those assumptions, which is precisely what happens in Floyd-Hoare logics [9].

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

Figure 5. $\mathsf{KAM}_{\mathsf{PCF}}$ evaluation rules.

PROPOSITION 3.3 (Relative Completeness). *If $\mathcal{E}$ is universal, then for any* PCF *term t,*

1. *if $t \Downarrow_{\mathsf{v}}^{k} \underline{\mathtt{m}}$, then $\vdash_{\underline{k}}^{\mathcal{E}} t : \mathtt{Nat}[\underline{m}]$ is derivable in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$.*
2. *if, for any $n \in \mathbb{N}$, there exist k_n, m_n such that $t\,\underline{\mathtt{n}} \Downarrow_{\mathsf{v}}^{k_n} \underline{\mathtt{m}_{\mathtt{n}}}$, then there exist* I *and* J *such that*

$$a; \varnothing; \varnothing \vdash_{\mathrm{I}}^{\mathcal{E}} t : [b < 1] \cdot (\mathtt{Nat}[a] \multimap \mathtt{Nat}[\mathrm{J}])$$

is derivable in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$, with $\models_{\mathcal{E}} \mathrm{J}\{\underline{n}/a\} = \underline{m_n}$ and $\models_{\mathcal{E}} \mathrm{I}\{\underline{n}/a\} \leqslant \underline{k_n}$ for all $n \in \mathbb{N}$.

The careful reader should have noticed that there is indeed a gap between the lower bound provided by completeness and the upper bound provided by soundness: this is indeed the reason why our complexity analysis is only meaningful in an asymptotic sense. Sometimes, however, programs with the same asymptotic behavior *can* indeed be distinguished, e.g. when their size is small relative to the constants in their weight.

In the next section, we will see how to make a concrete use of Relative Completeness. Indeed, we will describe an algorithm that, given a PCF term, returns a $\mathsf{d}\ell\mathsf{PCF}$ judgement $\vdash_{\mathrm{K}}^{\mathcal{E}} t : \tau$ for this term, where $\mathcal{E}$ is equational program that is not *universal*, but expressive enough to derive the typing judgement. To cope with the "relative" part of the result (i.e., the very strong assumption that every *true* semantic judgement must be available), the algorithm also returns a set of *side conditions* that have to be checked. These side conditions are in fact semantic judgements that act as axioms (of instances of the subsumption rule) in the typing derivation.

4. Relative Type Inference

Given on the one hand soundness and relative completeness of $\mathsf{d}\ell\mathsf{PCF}$, and on the other undecidability of complexity analysis for PCF programs, one may wonder whether looking for a type inference procedure makes sense at all. As stressed in the Introduction, we will not give a type inference algorithm *per se*, but rather reduce type inference to the problem of checking the validity of a set of inequalities modulo an equational program (see Figure 1). This is the reason why we can only claim type inference to be algorithmically solvable in a *relative* sense, i.e. assuming the existence of an oracle for proof obligations.

Why is solving relative type inference useful? Suppose you have a program $t : \mathtt{Nat} \Rightarrow \mathtt{Nat}$ and you want to prove that it works in a number of steps bounded by a polynomial $p : \mathbb{N} \to \mathbb{N}$ (e.g., $p(x) = 4 \cdot x + 7$). You could of course proceed by building a $\mathsf{d}\ell\mathsf{PCF}$ type derivation for t by hand, or even reason directly on the complexity of t. Relative type inference simplifies your life: it outputs an equational program $\mathcal{E}$, a *precise* type derivation for t whose conclusion is $a; \varnothing; \varnothing \vdash_{\mathrm{I}}^{\mathcal{E}} t : \mathtt{Nat}[a] \overset{1}{\multimap} \mathtt{Nat}[\mathrm{J}]$ and a set $\mathcal{I}$ of inequalities on the same signature as the one of $\mathcal{E}$. Your original problem, then, is reduced to verifying $\models^{\mathcal{E}} \mathcal{I} \cup \{\mathrm{I} \leqslant p(a)\}$. This is arguably an easier problem than the original one: first of all, it has nothing to do with complexity analysis but is rather a problem about the *value* of arithmetical expressions. Secondly it only deals with first-order expressions.

4.1 An Informal Account

From the brief discussion in Section 2, it should be clear that devising a compositional type inference procedure for $\mathsf{d}\ell\mathsf{PCF}$ is nontrivial: the type one assigns to a subterm heavily depends on the ways the rest of the program uses the subterm. The solution we adopt here consists in allowing the algorithm to return *partially unspecified* equational programs: $\mathcal{E}$ as produced in output by $\mathcal{T}$ gives meaning to all the symbols in the output type derivation *except* those occurring in negative position in its conclusion.

To better understand how the type inference algorithm works, let us consider the following term t:

$$uv = (\lambda x.\lambda y.x(xy))(\lambda z.\mathtt{s}(z)).$$

The subterm u can be given type $(\mathtt{Nat} \Rightarrow \mathtt{Nat}) \Rightarrow \mathtt{Nat} \Rightarrow \mathtt{Nat}$ in PCF, while v has type $\mathtt{Nat} \Rightarrow \mathtt{Nat}$. This means t as a whole has type $\mathtt{Nat} \Rightarrow \mathtt{Nat}$ and computes the function $x \mapsto 2 \cdot x$. The type inference algorithm proceeds by giving types to u and to v separately, then assembling the two into one. Suppose we start with v. The type inference algorithm refines $\mathtt{Nat} \Rightarrow \mathtt{Nat}$ into $\sigma = \mathtt{Nat}[\mathrm{f}(a,b)] \overset{b<\mathrm{h}(a)}{\multimap} \mathtt{Nat}[\mathrm{g}(a,b)]$ and the equational program $\mathcal{A}_v$, which gives meaning to g in terms of f:

$$\mathrm{g}(a,b) = \mathrm{f}(a,b) + 1.$$

Observe how both f and h are not specified in $\mathcal{A}_v$, because they appear in *negative* position in σ: $\mathrm{f}(a,b)$ intuitively corresponds to the argument(s) v will be applied to, while $\mathrm{h}(a)$ is the number of times v will be used. Notice that everything is parametrised on a, which is something like a global parameter that will later be set as the input to t. The function u, on the other hand, is given type

$$(\mathtt{Nat}[\mathrm{p}(a,b,c)] \overset{c<\mathrm{j}(a,b)}{\multimap} \mathtt{Nat}[\mathrm{q}(a,b,c)]) \overset{b<\mathrm{k}(a)}{\multimap} \mathtt{Nat}[\mathrm{l}(a,b,c)] \overset{c<\mathrm{m}(a,b)}{\multimap} \mathtt{Nat}[\mathrm{n}(a,b,c)].$$

The newly introduced function symbols are subject to the following equations:

$$\begin{aligned}
\mathrm{j}(a,b) &= 2 \cdot \mathrm{m}(a,b); \\
\mathrm{n}(a,b,c) &= \mathrm{q}(a,b,2c); \\
\mathrm{p}(a,b,2c) &= \mathrm{q}(a,b,2c+1); \\
\mathrm{p}(a,b,2c+1) &= \mathrm{l}(a,b,2c).
\end{aligned}$$

Again, notice that some functions are left unspecified, namely l, m, q and k. Now, a type for uv can be found by just combining the types for u and v, somehow following the typing rule for

applications. First of all, the number of times u needs to be copied is set to 1 by the equation $\mathrm{k}(a) = 1$. Then, the matching symbols of u and v are defined one in terms of the others:

$$\begin{aligned} \mathrm{q}(a, 0, b) &= \mathrm{g}(a, b); \\ \mathrm{f}(a, b) &= \mathrm{p}(a, 0, b); \\ \mathrm{h}(a) &= \mathrm{j}(a, 0). \end{aligned}$$

This is the last step of type inference, so it is safe to stipulate that $\mathrm{m}(a, 0) = 1$ and that $\mathrm{l}(a, 0, c) = a$, thus obtaining a fully specified equational program $\mathcal{E}$ and the following type τ for t:

$$\mathtt{Nat}[a] \overset{c<1}{\multimap} \mathtt{Nat}[\mathrm{n}(a, 0, c)].$$

As an exercise, the reader can verify that the equational program above allows to verify that $\mathrm{n}(a, 0, 0) = a + 2$, and that

$$a; \varnothing; \varnothing \vdash_2^{\mathcal{E}} t : \tau.$$

4.2 Preliminaries

Before embarking on the description of the type inference algorithms, some preliminary concepts and ideas need to be introduced, and are the topic of this section.

4.2.1 Getting Rid of Subsumption

The type inference algorithm takes in input a PCF term t, and returns a typing judgement $\mathcal{J}$ for t, together with a set $\mathscr{R}$ of so-called *side conditions*. We will show below that $\mathcal{J}$ is derivable *iff* all the side conditions in $\mathscr{R}$ are valid. Moreover, in this case $\mathcal{J}$ is *precise* (see Definition 3.1): all occurrences of the base type $\mathtt{Nat}[\mathrm{I}, \mathrm{J}]$ are in fact of the form $\mathtt{Nat}[\mathrm{I}]$, and the weight and all potentials H occurring in a sub-type $[a < \mathrm{H}] \cdot A$ are kept as low as possible. Concretely, this means that there is a derivation for $\mathcal{J}$ in which the subsumption rule is restricted to the following form:

$$\frac{\phi; \Phi; \Gamma \vdash_{\mathrm{I}}^{\mathcal{E}} t : \sigma \qquad \begin{array}{l} \phi; \Phi \vdash_{\mathcal{E}} \Delta \equiv \Gamma \\ \phi; \Phi \vdash_{\mathcal{E}} \sigma \equiv \tau \\ \phi; \Phi \vDash_{\mathcal{E}} \mathrm{I} = \mathrm{J} \end{array}}{\phi; \Phi; \Delta \vdash_{\mathrm{J}}^{\mathcal{E}} t : \tau}$$

The three premises on the right boil down to a set of semantic judgements of the form $\{\phi; \Phi \vDash_{\mathcal{E}} \mathrm{K}_i = \mathrm{H}_i\}$ (see Figure 3), where the K_i's are indexes occurring in σ or Δ (or I itself) and the H_i's occur in τ or Γ (or are J itself). If the equalities $\mathrm{K}_i = \mathrm{H}_i$ can all be derived from $\mathcal{E}$, then the three premises on the right are equivalent to the conjunction (on i) of the following properties:

"$[\![\mathrm{H}_i]\!]_\rho^{\mathcal{E}}$ is defined for any $\rho : \phi \to \mathbb{N}$ satisfying Φ"

(see Section 3.1.2). Given $\mathcal{E}$, this property (called a *side condition*) is denoted by $\phi; \Phi \vDash \mathrm{H}_i \downarrow$. Actually the type inference algorithm does *not* verify any semantic or subtyping judgement coming from (instances of) the subsumption rule. Instead, it turns all index equivalences $\mathrm{H}_i = \mathrm{K}_i$ into rewriting rules in $\mathcal{E}$, and put all side conditions $\phi; \Phi \vDash \mathrm{H}_i \downarrow$ in $\mathscr{R}$. If every side condition in $\mathscr{R}$ is true for $\mathcal{E}$, we write $\mathcal{E} \Vdash \bigwedge \mathscr{R}$. Informally, this means that all subsumptions assumed by the algorithm are indeed valid.

4.2.2 Function Symbols

Types and judgements manipulated by our type inference algorithm have a very peculiar shape. In particular, not every index term is allowed to appear in types, and this property will be crucial when showing soundness and completeness of the algorithm itself:

DEFINITION 4.1 (Primitive Types). *A type is* primitive for ϕ *when it is on the form* $\mathtt{Nat}[\mathrm{f}(\phi)]$*, or* $A \multimap B$ *with* A *and* B *primitive for* ϕ*, or* $[a < \mathrm{f}(\phi)] \cdot A$ *with* $a \notin \phi$ *and* A *primitive for* $a; \phi$*. A type is said to be* primitive *when it is primitive for some* ϕ*.*

As an example, a primitive type for $\phi = a; b$ is

$$\mathtt{Nat}[\mathrm{f}(a, b, c)] \overset{c<\mathrm{g}(a,b)}{\multimap} \mathtt{Nat}[\mathrm{h}(a, b, c)].$$

Informally, then, a type is primitive when the only allowed index terms are function symbols (with the appropriate arity).

4.2.3 Equational Programs

The equational program our algorithm constructs is in fact a *rewriting program*: every equality corresponds to the (partial) definition of a function symbol, and we may write it $\mathrm{f}(a_1, \ldots, a_k) := \mathrm{J}$ (where all free variables of J are in $\{a_1, \ldots, a_k\}$). If there is no such equation in the rewriting program, we say that f is *unspecified*.

An equational program $\mathcal{E}$ is *completely specified* if it allows to deduce a precise meaning (namely a partial recursive function) for each symbol of its underlying signature (written $\Sigma_{\mathcal{E}}$), i.e. none of the symbols in $\Sigma_{\mathcal{E}}$ are unspecified. In other words: a completely specified equational programs has only one *model*. On the other hand, a *partially specified* equational program (i.e. a program where symbols can possibly be unspecified) can have many models, because partial recursive functions can be assigned to unspecified function symbols in many different ways, all of them consistent with its equations. Up to now, we only worked with completely specified programs, but allowing the possibility to have unspecified symbols is crucial for being able to describe the type inference algorithm in a simple way. In the following, $\mathcal{E}$ and $\mathcal{F}$ denote completely specified equational programs, while $\mathcal{A}$ and $\mathcal{B}$ denote rewriting programs that are only partially specified.

DEFINITION 4.2 (Model of a Rewriting Program). *An* interpretation μ *of* $\mathcal{A}$ *in* $\mathcal{E}$ *is simply a map from unspecified symbols of* $\mathcal{A}$ *to indexes on the signature* $\Sigma_{\mathcal{E}}$*, such that if* f *has arity* n*, then* $\mu(\mathrm{f})$ *is a term in* $\Sigma_{\mathcal{E}}$ *with free variables from* $\phi_{\mathrm{f}} = \{a_1, \ldots, a_n\}$*. When such an interpretation is defined, we say that* $\mathcal{E}$ *is a* model *of* $\mathcal{A}$*, and we write* $\mu : \mathcal{E} \models \mathcal{A}$*.*

Notice that such an interpretation can naturally be extended to arbitrary index terms on the signature $\Sigma_{\mathcal{A}}$, and we assume in the following that a rewriting program and its model have disjoint signatures.

DEFINITION 4.3 (Validity in a Model). *Given* $\mu : \mathcal{E} \models \mathcal{A}$*, we say that a semantic judgement* $\phi; \Phi \vDash_{\mathcal{A}} \mathrm{I} \leqslant \mathrm{J}$ *is* valid in the model *(notation:* $\phi; \Phi \vDash_{\mu} \mathrm{I} \leqslant \mathrm{J}$*) when* $\phi; \Phi \vDash_{\mathcal{F}} \mathrm{I} \leqslant \mathrm{J}$ *where* $\mathcal{F} = \mathcal{A} \cup \mathcal{E} \cup \{\mathrm{f}(\phi_{\mathrm{f}}) := \mu(\mathrm{f}) \mid \mathrm{f}$ *is unspecified in* $\mathcal{A}\}$*. This definition is naturally extended to side conditions (with* $\mu \Vdash \bigwedge \mathscr{R}$ *standing for* $\mathcal{F} \Vdash \bigwedge \mathscr{R}$*).*

Please note that if $\mathcal{A}$ is a completely specified rewriting program, then any model $\mu : \mathcal{E} \models \mathcal{A}$ has an interpretation μ with an empty domain, and $\mu \Vdash \bigwedge \mathscr{R}$ iff $\mathcal{A} \Vdash \bigwedge \mathscr{R}$ (still assuming that $\Sigma_{\mathcal{A}}$ and $\Sigma_{\mathcal{E}}$ are disjoint).

As already mentioned, the equational programs handled by our type inference algorithm are not necessarily completely specified. Function symbols which are not specified are precisely those occurring in "negative position" in the judgement produced in output. This invariant will be very useful and is captured by the following definition:

DEFINITION 4.4 (Positive and Negative Symbols). *Given a primitive type* τ*, the sets of its* positive *and* negative *symbols (denoted*

by τ^+ and τ^- respectively) are defined inductively by

$$\mathtt{Nat}[\mathrm{i}(\phi)]^+ = \{\mathrm{i}\};$$
$$\mathtt{Nat}[\mathrm{i}(\phi)]^- = \varnothing;$$
$$[a < \mathrm{h}(\phi)] \cdot (\sigma \multimap \tau)^+ = \sigma^- \cup \tau^+;$$
$$[a < \mathrm{h}(\phi)] \cdot (\sigma \multimap \tau)^-; = \{\mathrm{h}\} \cup \sigma^+ \cup \tau^-.$$

Then the set of positive *(resp. negative) symbols of a judgement $\phi; \Phi; (x_i : \sigma_i)_{i \leqslant n} \vdash^{\mathcal{E}}_{\mathrm{I}} t : \tau$ is the union of all negative (resp. positive) symbols of the σ_i's and all positive (resp. negative) symbols of τ.*

Polarities in $\{+, -\}$ are indicated with symbols like p, q. Given such a p, the opposite polarity is $\neg p$.

DEFINITION 4.5 (Specified Symbols, Types and Judgments). *Given a set of function symbols $\mathcal{S}$, a symbol* f *is said to be* $(\mathcal{S}, \mathcal{A})$-specified *when there is a rule $\mathrm{f}(\phi) := \mathrm{J}$ in $\mathcal{A}$ such that any function symbol appearing in* J *is either* f *itself, or in $\mathcal{S}$, or a symbol that is $(\mathcal{S} \cup \{\mathrm{f}\}, \mathcal{A} \backslash \{\mathrm{f}(\phi) := \mathrm{J}\})$-specified. Remember that when there is no rule $\mathrm{f}(\phi) := \mathrm{J}$ in $\mathcal{A}$ the symbol* f *is* unspecified *in $\mathcal{A}$. A primitive type σ is said to be $(p, \mathcal{S}, \mathcal{A})$-*specified *when all function symbols in σ^p are $(\mathcal{S}, \mathcal{A})$-specified and all symbols in $\sigma^{\neg p}$ are unspecified. A judgement $\phi; \Phi; \Gamma \vdash^{\mathcal{A}}_{\mathrm{I}} t : \tau$ is* correctly specified *when τ and all types in Γ are primitive for ϕ, and τ is $(+, \mathcal{N}, \mathcal{A})$-specified, and all types in Γ are $(-, \mathcal{N}, \mathcal{A})$-specified, and all function symbols in* I *are $(\mathcal{N}, \mathcal{A})$-specified where $\mathcal{N}$ is the set of negative symbols of the judgement.*

In other words, a judgement is correctly specified if the underlying equational program (possibly recursively) defines all symbols in positive position depending on those in negative position.

4.3 The Structure of the Algorithm

The type inference algorithm receives in input a PCF term t and returns a dℓPCF judgement $\varnothing \vdash^{\mathcal{E}}_{\mathrm{K}} t : \tau$ for it, together with a set of side conditions $\mathscr{R}$. We will prove that it is *correct*, in the sense that the typing judgement is derivable *iff* the side conditions hold. The algorithm proceeds as follows:

1. Compute d_{PCF}, a PCF type derivation for t;
2. Proceeding by structural induction on d_{PCF}, construct a dℓPCF derivation for t (call it d_{v}) and the corresponding set of side conditions $\mathscr{R}$;
3. Returns $\mathscr{R}$ and the conclusion of d_{v}.

The *skeleton* $(\!|\sigma|\!)$ (or $(\!|A|\!)$) of a modal type σ (*resp.* of a linear type A) is obtained by erasing all its indexes (and its bounds $[a < \mathrm{I}]$). The *skeleton* of a dℓPCF derivation is obtained by replacing each type by its skeleton, and erasing all the subsumption rules. In PCF the type inference problem is decidable, and Step 1 raises no difficulty: actually, one could even assume that the type d_{PCF} attributes to t is principal. The core of the algorithm is of course Step 2. In Section. 4.5 we define a recursive algorithm GEN that build d_{v} and $\mathscr{R}$ by annotating d_{PCF}. The algorithm GEN itself relies on some auxiliary algorithms, which will be described in Section 4.4 below.

All algorithms we will talk about have the ability to generate fresh variables and function symbols. Strictly speaking, then, they should take a counter (or anything similar) as a parameter, but we elide this for the sake of simplicity. Also, we assume the existence of a function $\alpha(\phi; T)$ that, given a set of index variables ϕ and a PCF type T, returns a modal type τ primitive for ϕ, containing only fresh function symbols, and such that $(\!|\tau|\!) = T$.

4.4 Auxiliary Algorithms and Linear Logic

The design of systems of linear dependent types such as dℓPCF$_{\mathsf{V}}$ and dℓPCF$_{\mathsf{N}}$ is strongly inspired by BLL, itself a restriction of linear logic. Actually, the best way to present the type inference algorithm consists in first of all introducing four auxiliary algorithms, each corresponding to a principle regulating the behaviour of the exponential connectives in linear logic. Notice that these auxiliary algorithms are the main ingredients of both dℓPCF$_{\mathsf{V}}$ and dℓPCF$_{\mathsf{N}}$ type inference. Consistently to what we have done so far, we will prove and explain them with dℓPCF$_{\mathsf{V}}$ in mind. All the auxiliary algorithm we will talk about in this section will take a tuple of dℓPCF$_{\mathsf{V}}$ types as first argument; we assume that all of them have the same skeleton and, moreover, that all index terms appearing in them are pairwise distinct.

Dereliction. Dereliction is the following principle: any duplicable object (say, of type $!A$) can be made linear (of type A), that is to say $!A \to A$. In dℓPCF, being duplicable means having a modal type, which also contains some quantitative information, namely how many times the object can be duplicated, at most. In dℓPCF$_{\mathsf{V}}$, dereliction can be simply seen as the principle $[a < 1] \cdot A \to A\{0/a\}$, and is implicitly used in the rules (*App*) and (*Fix*). Along the type inference process, as a consequence, we often need to create "fresh instances" of dereliction in the form of pairs of types being in the correct semantic relation. This is indeed possible:

LEMMA 4.1. *There is an algorithm* DER *such that given two types τ (primitive for ϕ) and σ (primitive for a, ϕ) of the same skeleton,* $\mathrm{DER}((\sigma, \tau); a; \phi; \Phi; p) = (\mathcal{A}, \mathscr{R})$ *where:*

1. *for every $\mathcal{E} \supseteq \mathcal{A}$, if $\mathcal{E} \Vdash \bigwedge \mathscr{R}$ then $\phi; \Phi \vdash_{\mathcal{E}} \sigma\{0/a\} \equiv \tau$;*
2. *whenever $\phi; \Phi \vdash_{\mathcal{E}} \varrho\{0/a\} \equiv \xi$ where $(\!|\varrho|\!) \equiv (\!|\sigma|\!)$, there is $\mu : \mathcal{E} \models \mathcal{A}$ such that $\phi; \Phi \vdash_{\mu} \sigma\{0/a\} \equiv \varrho\{0/a\}$, $\phi; \Phi \vdash_{\mu} \tau \equiv \xi$, and $\mu \Vdash \bigwedge \mathscr{R}$;*
3. *σ is $(p, \tau^p, \mathcal{A})$-specified and τ is $(\neg p, \sigma^{\neg p}, \mathcal{A})$-specified.*

The algorithm DER works by recursion on the PCF type $(\!|\sigma|\!)$ and has thus linear complexity in $|(\!|\sigma|\!)|$. The proof of Lemma 4.1 (see [12]), as a consequence, proceeds by induction on the structure of $(\!|\sigma|\!)$.

Contraction. Another key principle in linear logic is contraction, according to which two copies of a duplicable object can actually be produced, $!A \to !A \otimes !A$. Contraction is used in binary rules like (*App*) or (*If*), in the form of the operator $\uplus$. This time, we need an algorithm CTR which takes *three* linear types A, B and C (all of them primitive for (a, ϕ)) and turn them into an equational program and a set of side conditions:

$$\mathrm{CTR}((A, B, C); (\mathrm{I}, \mathrm{J}); a; \phi; \Phi; p) = (\mathcal{A}, \mathscr{R}).$$

The parameters I and J are index terms capturing the number of times B and C can be copied. A Lemma akin to 4.1 can indeed be proved about CTR. In particular, for any $\mathcal{E} \supseteq \mathcal{A}$, if $\mathcal{E} \Vdash \bigwedge \mathscr{R}$ then

$$\phi; \Phi \vdash_{\mathcal{E}} [a < \mathrm{I} + \mathrm{J}] \cdot A \equiv ([a < \mathrm{I}] \cdot D) \uplus ([a < \mathrm{J}] \cdot E) \quad (5)$$

for some D and E such that $\phi; \Phi, a < \mathrm{I} \vdash_{\mathcal{E}} D \equiv B$ and $\phi; \Phi, a < \mathrm{J} \vdash_{\mathcal{E}} E \equiv C$.

Digging. In linear logic, any duplicable object having type $!A$ can be turned into an object of type $!!A$, namely an object which is the duplicable version of a duplicable object. Digging is the principle according to which this transformation is possible, namely $!A \to !!A$. At the quantitative level, this corresponds to splitting a bounded sum into its summands. This is used in the typing rules for functions, $(\multimap)$ and (*Fix*).

The auxiliary algorithm corresponding to the digging principle takes two linear types and builds, as usual, a rewriting program and a set of side conditions capturing the fact that the first of the two types is the bounded sum of the second:

$$\mathrm{DIG}((A, B); (\mathrm{I}, \mathrm{J}); \phi; (a, b); \Phi; p) = (\mathcal{A}, \mathscr{R}).$$

The correctness of DIG can again be proved similarly to what we did in Lemma 4.1, the key statement being that for every $\mathcal{E} \supseteq \mathcal{A}$ such that $\mathcal{E} \Vdash \bigwedge \mathscr{R}$, the following must hold

$$\phi; \Phi \vdash_{\mathcal{E}} [b < \sum_{a<\mathrm{I}} \mathrm{J}] \cdot A \equiv \sum_{a<I} [b < J] \cdot C$$

for some C such that $\phi; \Phi, a < \mathrm{I}, b < \mathrm{J} \vdash_{\mathcal{E}} C \equiv B$.

Weakening. Weakening means that duplicable objects can also be erased, even when the underlying index is 0. Weakening is useful in the rules (*Ax*) and (*n*). Once a fresh $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ type A is produced, the only thing we need to do is to produce an equational program $\mathcal{A}$ specifying (in an arbitrary way) the symbols in A^p, this way preserving the crucial invariants about the equational programs manipulated by the algorithm. Formally, it means that there is an algorithm WEAK such that

$$\textsc{weak}(A; \phi; a; p) = \mathcal{A},$$

where A is $(p, \varnothing, \mathcal{A})$-specified Observe how no sets of constraints is produced in output by WEAK, contrarily to DER, CTR, and DIG.

4.5 The Type Inference Procedure

In this section, we will describe the core of our type inference algorithm. This consists in a recursive algorithm GEN which decorates a PCF type derivation d_{PCF}, producing in output a $\mathsf{d}\ell\mathsf{PCF}$ judgement, together with an equational program and a set of side conditions. In order to correctly create fresh symbols and to format side conditions properly, the main recursive function GEN also receives a set of index variables ϕ and a set of constraints Φ in input. Thus, it has the following signature:

$$\textsc{gen}(\phi; \Phi; d_{\mathsf{PCF}}) \;=\; (\Gamma \vdash_{\mathrm{I}} t : \tau; \mathcal{A}; \mathscr{R}).$$

We will prove that the the output of GEN satisfies the following two invariants:

- *Decoration*. d_{PCF} has conclusion $(\![\Gamma]\!] \vdash t : (\![\tau]\!])$.
- *Polarity*. $\phi; \Phi; \Gamma \vdash_{\mathrm{I}}^{\mathcal{A}} t : \tau$ is correctly specified (see Definition 4.5).

The algorithm GEN proceeds by inspecting d_{PCF} in an inductive manner. It first annotates the types in the conclusion judgement with fresh function symbols to get a $\mathsf{d}\ell\mathsf{PCF}$ judgement $\mathcal{J}$. Then a recursive call is performed on the immediate sub-derivations of d_{PCF}, this way obtaining some $\mathsf{d}\ell\mathsf{PCF}$ typing judgement $\mathcal{J}_i$. Finally GEN generates, calling the auxiliary algorithms, the equations on function symbols that allow to derive $\mathcal{J}$ from the $\mathcal{J}_i$'s, The equations are written in $\mathcal{A}$, and the required assumptions of index convergence in $\mathscr{R}$.

Decoration and Polarity are the invariants of the algorithm GEN. In particular, the auxiliary algorithms are always called with the appropriate parameters, this way enforcing Polarity.

The algorithm computing GEN proceeds by case analysis on d_{PCF}. We give some cases here, the other ones are developed in [12].

- Suppose that d_{PCF} is

$$\frac{}{y_1 : U_1, \ldots, y_k : U_k \vdash \underline{\mathtt{n}} : \mathtt{Nat}}$$

For each i, let $B_i = \alpha(\phi; U_i)$ and $\mathcal{A}_i = \textsc{weak}(B_i; \phi; b_i; -)$ (where all the b_i's are fresh). Let $\mathrm{i}(\phi)$ be a fresh function symbol. Then return $(\Gamma \vdash_0 \underline{\mathtt{n}} : \mathtt{Nat}[\mathrm{i}(\phi)]; \mathcal{A}; \varnothing)$ where the h_i's are fresh symbols and

$$\mathcal{A} = \bigcup_i (\mathcal{A}_i \cup \{\mathrm{h}_i(\phi) := 0\}) \cup \{\, \mathrm{i}(\phi) := \underline{n} \,\};$$
$$\Gamma = \{y_i : [b_i < \mathrm{h}_i(\phi)] \cdot B_i\}_{1 \leqslant i \leqslant k}.$$

- If d_{PCF} is on the form

$$\frac{e^1_{\mathsf{PCF}} : \Pi \vdash t : U \Rightarrow T \quad e^2_{\mathsf{PCF}} : \Pi \vdash u : U}{\Pi \vdash t\, u : T}$$

let $(\Gamma_1 \vdash_{\mathrm{K}} t : [a < \mathrm{f}(\phi)] \cdot (\sigma_1 \multimap \sigma_2); \mathcal{A}_1; \mathscr{R}_1) = \textsc{gen}(\phi; \Phi; e^1_{\mathsf{PCF}})$, and $(\Gamma_2 \vdash_{\mathrm{H}} u : \tau; \mathcal{A}_2; \mathscr{R}_2) = \textsc{gen}(\phi; \Phi; e^2_{\mathsf{PCF}})$. Let $(\mathcal{B}, \mathscr{S}) = \textsc{der}((\sigma_1, \tau); a; \phi; \Phi; +)$. We then annotate T: let $\tau_2 = \alpha(\phi; T)$, and let $(\mathcal{C}, \mathscr{U}) = \textsc{der}((\sigma_2, \tau_2); a; \phi; \Phi; -)$. Then we build a context equivalent to $\Gamma_1 \uplus \Gamma_2$: by the decoration property, Γ_1 and Γ_2 have the same skeleton Π, so for any $y : [b_y < \mathrm{i}_y(\phi)] \cdot B_y$ in Γ_1, there is some $y : [b_y < \mathrm{j}_y(\phi)] \cdot C_y$ in Γ_2 (possibly after some α-conversion). Then let $A_y = \alpha((b_y, \phi); (\![B_y]\!])$, and $(\mathcal{A}_y; \mathscr{R}_y) = \textsc{ctr}((A_y, B_y, C_y); (\mathrm{i}_y(\phi), \mathrm{j}_y(\phi)); b_y; \phi; \Phi; -)$. There are $\Delta_i \equiv \Gamma_i$ (for $i = 1, 2$) such that $\phi; \Phi \vdash_{\mathcal{E}} \{y : [b_y < \mathrm{i}_y(\phi) + \mathrm{j}_y(\phi)] \cdot A_y\}_y \equiv \Delta_1 \uplus \Delta_2$, for every $\mathcal{E} \supseteq \bigcup_y \mathcal{A}_y$ such that $\mathcal{E} \Vdash \bigwedge(\bigcup_y \mathscr{R}_y)$. Thus, let h_y's be fresh symbols and return $(\Delta \vdash_{\mathrm{K+H}} t\, u : \tau_2; \mathcal{A}; \mathscr{R})$ in output, where

$$\mathscr{R} = \mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{S} \cup \mathscr{U} \cup \bigcup_y \big(\mathscr{R}_y \cup \{\phi; \Phi \vDash \mathrm{i}_y(\phi) + \mathrm{j}_y(\phi)\downarrow\}\big);$$
$$\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{B} \cup \mathcal{C} \cup \{\mathrm{f}(\phi) := 1\} \cup \bigcup_y \big(\mathcal{A}_y \cup \{\mathrm{h}_y(\phi) := \mathrm{i}_y(\phi) + \mathrm{j}_y(\phi)\}\big);$$
$$\Delta = \{y : [b_y < \mathrm{h}_y(\phi)] \cdot A_y\}_y.$$

- Assume that d_{PCF} is

$$\frac{e_{\mathsf{PCF}} : \Pi, x : U \vdash t : T}{\Pi \vdash \lambda x.t : U \Rightarrow T}$$

Let a be a fresh index variable, and $\mathrm{i}(\phi)$ be a fresh function symbol, and compute $(\Gamma, x : \sigma \vdash_{\mathrm{K}} t : \tau; \mathcal{B}; \mathscr{S}) = \textsc{gen}(e_{\mathsf{PCF}}, (a, \phi), (a < \mathrm{i}(\phi), \Phi))$. We build a context equivalent to $\sum_{a<\mathrm{i}(\phi)} \Gamma$: for every $y : [b_y < \mathrm{j}_y(a, \phi)] \cdot B_y \in \Gamma$, let $A_y = \alpha((b_y, \phi); (\![B_y]\!])$, let $\mathrm{h}_y(\phi)$ be a fresh symbol, and let $(\mathcal{A}_y, \mathscr{R}_y)$ be $\textsc{dig}((A_y, B_y); (\mathrm{i}(\phi), \mathrm{j}_y(a, \phi)); \phi; (a, b_y); \Phi; -)$. Then return

$$(\Delta \vdash_{\mathrm{i}(\phi) + \sum_{a<\mathrm{i}(\phi)} \mathrm{K}} \lambda x.t : [a < \mathrm{i}(\phi)] \cdot (\sigma \multimap \tau); \mathcal{A}; \mathscr{R})$$

where

$$\mathscr{R} = \mathscr{S} \cup \bigcup_y (\mathscr{R}_y \cup \{\phi; \Phi \vDash_{\mathcal{E}} \sum_{a<\mathrm{i}(\phi)} \mathrm{j}_y(a, \phi)\downarrow\});$$
$$\mathcal{A} = \mathcal{B} \cup \cup_y (\mathcal{A}_y \cup \{\mathrm{h}_y(\phi) := \sum_{a<\mathrm{i}(\phi)} \mathrm{j}_y(a, \phi)\});$$
$$\Delta = \{y : [b_y < \mathrm{h}_y(\phi)] \cdot A_y\}_y.$$

LEMMA 4.2. *For every ϕ, Φ, and every* PCF *derivation d_{PCF}, the output of* $\textsc{gen}(\phi; \Phi; d_{\mathsf{PCF}})$ *satisfies Decoration and Polarity.*

4.6 Correctness

The algorithm we have just finished describing needs to be proved sound and complete with respect to $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ typing. As usual, this is not a trivial task. Moreover, linear dependent types have a semantic nature which makes the task of formulating (if not proving) the desired results even more challenging.

4.6.1 Soundness

A type inference procedure is *sound* when the inferred type can actually be derived by way of the type system at hand. As already remarked, GEN outputs an equational program $\mathcal{A}$ which possibly

contains unspecified symbols and which, as a consequence, cannot be exploited in typing. Moreover, the role of the set of proof obligations in $\mathscr{R}$ may be unclear at first. Actually, soundness holds for *every* completely specified $\mathcal{E} \supseteq \mathcal{A}$ which makes the proof obligations in $\mathscr{R}$ true:

THEOREM 4.3 (Soundness). *If d_{PCF} is a* PCF *derivation for t, then for any ϕ and Φ, $\textsc{gen}(\phi;\Phi;d_{\mathsf{PCF}}) = (\Gamma \vdash_{\mathrm{I}} t : \tau;\mathcal{A};\mathscr{R})$ where $\phi;\Phi;\Gamma \vdash_{\mathrm{I}}^{\mathcal{A}} t : \tau$ is correctly specified and for any $\mathcal{E} \supseteq \mathcal{A}$,*

$$\mathcal{E} \Vdash \bigwedge \mathscr{R} \quad \Longrightarrow \quad \phi;\Phi;\Gamma \vdash_{\mathrm{I}}^{\mathcal{E}} t : \tau \text{ is derivable and precise.}$$

Soundness can be proved by induction on the structure of d_{PCF}, exploiting auxiliary results like Lemma 4.1.

4.6.2 Completeness

But are we sure that *at least* one type derivation can be built from the outcome of GEN *if* one such type derivation exists? Again, it is nontrivial to formulate the fact that this is actually the case.

THEOREM 4.4 (Completeness). *If $\phi;\Phi;\Delta \vdash_{\mathrm{J}}^{\mathcal{E}} t : \sigma$ is a precise* $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ *judgement derivable by d_{v}, then $\textsc{gen}(\phi;\Phi;([\![d_{\mathsf{v}}]\!])$ is of the form $(\Gamma \vdash_{\mathrm{I}} t : \tau;\mathcal{A};\mathscr{R})$, and there is $\mu : \mathcal{E} \models \mathcal{A}$ such that $\mu \Vdash \bigwedge \mathscr{R}$.*

Completeness can be proved by an induction on the structure of d_{v}; its statement, however, needs to be appropriately enriched for induction to work. Again, results like Lemma 4.1 greatly help here: Point 2 states completeness of DER, and the latter is called by GEN many times.

A direct consequence of Soundness and Completeness (and the remark on Definition 4.3) is the following:

COROLLARY 4.5. *If a closed term t is typable in* PCF *with type* Nat *and a derivation d_{PCF}, then*

$$\textsc{gen}(\varnothing;\varnothing;d_{\mathsf{PCF}}) \;=\; (\vdash_{\mathrm{I}} t : \mathtt{Nat}[\mathrm{f}];\mathcal{E};\mathscr{R})$$

and t is typable in $\mathsf{d}\ell\mathsf{PCF}_{\mathsf{V}}$ *iff $\mathcal{E} \Vdash \bigwedge \mathscr{R}$.*

5. Type Inference at Work

The type inference algorithm presented in the previous section has been implemented in OCAML[2]. Programs, types, equational programs and side conditions become values of appropriately defined inductive data structures in OCAML, while the functional nature of the latter makes the implementation effort easier. This section is devoted to discussing the main issues we have faced along the process, which is still ongoing.

The core of our implementation is an OCAML function called `CheckBound`. Taking a closed term t having PCF type T in input, `CheckBound` returns a typing derivation d_{v}, an equational program $\mathcal{E}$ and a set of side conditions $\mathscr{R}$. The conclusion of d_{v} is a $\mathsf{d}\ell\mathsf{PCF}$ typing judgement for the input term. If T is a first-order type, then the produced judgement is derivable *iff* all the side conditions in $\mathscr{R}$ are valid. To do so, `CheckBound` calls (an implementation of) GEN on t and a context ϕ consisting of n unconstrained index variables, where n is the arity of t. This way, `CheckBound` obtains $\mathcal{A}$ and $\mathscr{R}$ as results, and then proceeds as follows:

- If T is `Nat`, then $\mathcal{A}$ is already completely specified and Corollary 4.5 ensures that we already have what we need.
- If T has a strictly positive arity, then some of the symbols in $\mathcal{A}$ are unspecified, and appropriate equations for them need to be added to $\mathcal{A}$. Take for instance a term s of type $\mathtt{Nat} \Rightarrow \mathtt{Nat}$. `CheckBound`($s$) returns $\mathcal{A}$, $\mathscr{R}$, and a typing judgement of the form

$$a;\varnothing;\varnothing \vdash_{\mathrm{K}}^{\mathcal{A}} t : \mathtt{Nat}[\mathrm{g}(a,b)] \overset{b<\mathrm{f}(a)}{\multimap} \mathtt{Nat}[\mathrm{j}(a,b)],$$

  where j is a positive symbol while f and g are negative, thus unspecified in $\mathcal{A}$. $\mathcal{A}$ can be appropriately "completed" by adding the equations $\mathrm{f}(a) := \underline{1}$ and $\mathrm{g}(a,b) := a$ to it. This way, we are insisting on the behaviour of t when fed with *any* natural number (represented by a) and when the environment needs t only once.

How about complexity analysis? Actually, we are already there: the problem of proving the number of machine reduction steps needed by t to be at most $p : \mathbb{N} \to \mathbb{N}$ (where p is, e.g. a polynomial) becomes the problem of checking $\mathcal{E} \Vdash \bigwedge \mathscr{S}$ where $\mathcal{E}$ is the appropriate completion of $\mathcal{A}$, and $\mathscr{S}$ is $\mathscr{R} \cup \{(\mathrm{K}+1)(|t|+2) \leqslant p(a)\}$ (Proposition 3.2).

Simplifying Equations. Equational programs obtained in output from `CheckBound` contains many equations which are trivial (such as $\mathrm{f}(a) := \underline{n}$ or $\mathrm{f}(a) := \mathrm{g}(a)$), and as such can be eliminated. Moreover, instances of forest cardinalities and bounded sums can sometime be greatly simplified. As an example, $\sum_{a<\underline{0}} \mathrm{J}$ can always be replaced by $\underline{0}$. This allows, in particular, to turn $\mathcal{A}$ into a set of fewer and simpler rules, thus facilitating the next phase.

A basic simplification procedure has already been implemented, and is called by `CheckBound` on the output of GEN. However, automatically treating the equational program by an appropriate prover would of course be desirable. For this purpose, the possibility for `CheckBound` to interact with MAUDE [8], a system supporting equational and rewriting logic specification, is currently investigated.

Checking Side Conditions. As already stressed, once `CheckBound` has produced a pair $(\mathcal{A},\mathscr{R})$, the task we started from, namely complexity analysis of t, is not finished, yet: checking proof obligations in $\mathscr{R}$ is as undecidable as analysing the complexity of t directly, since most of the obligations in $\mathscr{R}$ are termination statements anyway. There is an important difference, however: statements in $\mathscr{R}$ are written in a language (the first-order equational logic) which is more amenable to be treated by already existing automatic and semi-automatic tools.

Actually, the best method would be to first call as many existing automatic provers as possible on the set of side conditions, then asking the programmer to check those which cannot be proved automatically by way of an interactive theorem prover. For this purpose, we have implemented an algorithm translating a pair in the form $(\mathcal{A},\mathscr{R})$ into a WHY3 theory [6].

6. Related Work

Complexity analysis of higher-order programs has been the object of many studies. We can for example mention the proposals for type systems for the λ-calculus which have been shown to correspond in an *extensional sense* to, e.g. polynomial time computable functions. Many of them 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 synthesised. The problem with these systems, however, is that they are usually very weak from an *intentional* point of view, since the class of typable programs is quite restricted compared to the class of all terms working within the prescribed resource bounds.

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 [27]: 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 [5] higher-order programs are translated into higher-order equations,

[2] the source code is available at `http://lideal.cs.unibo.it`.

and the latter are turned into first-order ones; both steps, and in particular the second one, are not completeness-preserving. Recent works on amortised resource analysis are either limited to first-order programs [21] or to linear bounds [23]. A recent proposal by Amadio and Régis-Gianas [2] allows to reason on the the cost of higher-order functional programs by way of so-called cost-annotations, being sure that the actual behaviour 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.

Ghica's slot games [17] are maybe the work which is closest to ours, among the many in the literature. Slot Games are simply ordinary games in the sense of game semantics, which are however instrumented so as to reflect not only the observable behaviour of (higher-order) programs, but also their performance. Indeed, slot games are fully abstract with respect to an operational theory of improvements due do Sands [28]: this can be seen as the counterpart of our relative completeness theorem. An aspect which has not been investigated much since Ghica's proposal is whether slot games provides a way to perform actual verification of programs, maybe via some form of model checking. As we have already mentioned, linear dependency can be seen as a way to turn games and strategies into types, so one can see the present work also as an attempt to keep programs and strategies closer to each other, this way facilitating verification. Another recent work which seems to be quite close to ours is Geometry of Synthesis, in particular when the latter takes the form of type inference [18].

7. Conclusions

A type inference procedure for $\mathsf{d}\ell\mathsf{PCF}$ has been introduced which, given a PCF term, reduces the problem of finding a type derivation for it to the one of solving proof obligations on an equational program, itself part of the output. Truth of the proof obligations correspond to termination of the underlying program. Any type derivation in $\mathsf{d}\ell\mathsf{PCF}$ comes equipped with an expression bounding the complexity of evaluating the underlying program. Noticeably, proof obligations and the related equational program can be obtained in polynomial time in the size of the input PCF program.

The main contribution of this paper consists in having shown that linear dependency is not only a very powerful tool for the analysis of higher-order functional programs, but is also a way to effectively and efficiently turn a complex problem (that of evaluating the time complexity of an higher-order program) into a much easier one (that of checking a set of proof obligations for truth).

Although, as explained in Section 5, experimental evaluation shows that proof obligations can potentially be handled by modern tools, much remains to be done about the technical aspects of turning proof obligations into a form which is suitable to automatic or semi-automatic solving. Actually, many different tools could conceivably be of help here, each of them requiring a specific input format. This implies, however, that the work described in this paper, although not providing a fully-fledged out-of-the-box methodology, has the merit of allowing to factor a complex non-well-understood problem into a much-better-studied problem, namely verification of first-order inequalities on the natural numbers.

References

[1] Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. I & C 163(2), 409–470 (2000)

[2] Amadio, R.M., Régis-Gianas, Y.: Certifying and reasoning on cost annotations of functional programs. CoRR abs/1110.2350 (2011)

[3] de Bakker, J.W.: Mathematical Theory of Program Correctness. Prentice-Hall (1980)

[4] Barthe, G., Grégoire, B., Riba, C.: Type-based termination with sized products. In: CSL 2008. LNCS, vol. 5213, pp. 493–507. Springer (2008)

[5] Benzinger, R.: Automated higher-order complexity analysis. Theor. Comput. Sci. 318(1-2), 79–103 (2004)

[6] Bobot, F., Filliatre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: First International Workshop on Intermediate Verification Languages. pp. 53–64 (2011)

[7] Clarke, E.M.: Programming language constructs for which it is impossible to obtain good hoare axiom systems. J. ACM 26(1), 129–147 (1979)

[8] Clavel, M., Durán, F., Eker, S., Lincoln, P., Mart´i-Oliet, N., Meseguer, J., Talcott, C.: The Maude 2.0 system. In: RTA 2003. LNCS, vol. 2706, pp. 76–87 (2003)

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

[10] Dal Lago, U.: Context semantics, linear logic and computational complexity. In: LICS. pp. 169–178 (2006)

[11] Dal Lago, U., Gaboardi, M.: Linear dependent types and relative completeness. In: LICS. pp. 133–142 (2011)

[12] Dal Lago, U., Petit, B.: The geometry of types (long version) (2012), available at `http://arxiv.org/abs/1210.6857`

[13] Dal Lago, U., Petit, B.: Linear dependent types in a call-by-value scenario. In: ACM PPDP 2012. pp. 115–126 (2012)

[14] Danos, V., Regnier, L.: Reversible, irreversible and optimal lambda-machines. Theor. Comput. Sci. 227(1-2), 79–97 (1999)

[15] Denney, E.: Refinement types for specification. In: IFIP-PROCOMET. pp. 148–166 (1998)

[16] Felleisen, M., Friedman, D.P.: Control operators, the SECD-machine and the λ-calculus. Tech. Rep. 197, Computer Science Department, Indiana University (1986)

[17] Ghica, D.R.: Slot games: a quantitative model of computation. In: ACM POPL 2005. pp. 85–97 (2005)

[18] Ghica, D.R., Smith, A.: Geometry of synthesis III: resource management through type inference. In: ACM POPL 2011. pp. 345–356 (2011)

[19] Girard, J.Y., Scedrov, A., Scott, P.: Bounded linear logic. Theor. Comp. Sci. 97(1), 1–66 (1992)

[20] Gulwani, S.: Speed: Symbolic complexity bound analysis. In: CAV. pp. 51–62 (2009)

[21] Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. In: ACM POPL 2011. pp. 357–370 (2011)

[22] Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: ACM POPL 1996. pp. 410–423 (1996)

[23] Jost, S., Hammond, K., Loid, H.W., Hofmann, M.: Static determination of quantitative resource usage for higher-order programs. In: ACM POPL 2010. Madrid, Spain (2010)

[24] Krivine, J.L.: A call-by-name lambda-calculus machine. Higher-Order and Symbolic Computation 20(3), 199–207 (2007)

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

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

[27] Sands, D.: Complexity analysis for a lazy higher-order language. In: ESOP 1990. LNCS, vol. 432, pp. 361–376 (1990)

[28] Sands, D.: Operational theories of improvement in functional languages (extended abstract). In: Functional Programming. pp. 298–311 (1991)

[29] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenstrom, P.: The worst case execution time problem - overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. (2008)