Kavvos, G. A., Morehouse, E., Licata, D., & Danner, N. (2019). Recurrence extraction for functional programs through call-by-push-value. *Proceedings of the ACM on Programming Languages*, *4*(POPL), [15]. https://doi.org/10.1145/3371083

Recurrence Extraction for Functional Programs through Call-by-Push-Value

G. A. KAVVOS*, EDWARD MOREHOUSE, DANIEL R. LICATA, and NORMAN DANNER, Wesleyan University, United States of America

The main way of analyzing the complexity of a program is that of extracting and solving a recurrence that expresses its running time in terms of the size of its input. We develop a method that automatically extracts such recurrences from the syntax of higher-order recursive functional programs. The resulting recurrences, which are programs in a call-by-name language with recursion, explicitly compute the running time in terms of the size of the input. In order to achieve this in a uniform way that covers both call-by-name and call-by-value evaluation strategies, we use Call-by-Push-Value (CBPV) as an intermediate language. Finally, we use domain theory to develop a denotational cost semantics for the resulting recurrences.

CCS Concepts: • **Theory of computation → Program verification**; **Program analysis**; *Denotational semantics*; • **Software and its engineering →** *Functional languages.*

Additional Key Words and Phrases: recurrence extraction, resource analysis, cost semantics, higher order recurrences, denotational semantics, call by push value, general recursion

ACM Reference Format:
G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. 2020. Recurrence Extraction for Functional Programs through Call-by-Push-Value. *Proc. ACM Program. Lang.* 4, POPL, Article 15 (January 2020), 31 pages. https://doi.org/10.1145/3371083

1 INTRODUCTION

Functional programmers typically analyze time, space, or other resource usage of their programs using the *extract-and-solve* method. First, we *extract a recurrence* from the program. In this context, a recurrence is a mathematical object—usually an inequality—that expresses an upper bound for the running time of a program in terms of the *size* of its input. Depending on the task at hand, this notion of size may vary. For example, if the input is a tree, we may define its size to be its number of nodes, its depth, or some more complicated expression. The second step consists of *solving* this recurrence: mathematical methods are used to express it (or a suitably looser version of it) in a non-recursive *closed form* and big-O bound. While this method is taught in introductory textbooks (e.g. Bird [2014, §7]), there is no *formal* connection between the program and the extracted recurrence. In this work we concentrate on the first of those steps: we seek a method to *automatically extract a recurrence* from the syntax of a recursive functional program, in such a way that we can prove a formal *bounding theorem* relating the extracted recurrence to the program's operational cost.

*Current affiliation: Department of Computer Science, Aarhus University

Authors' address: G. A. Kavvos, g.a.kavvos@gmail.com; Edward Morehouse, emorehouse@wesleyan.edu; Daniel R. Licata, dlicata@wesleyan.edu; Norman Danner, ndanner@wesleyan.edu, Department of Mathematics and Computer Science, Wesleyan University, 265 Church Street, Middletown, CT, 06459, United States of America.

2475-1421/2020/1-ART15
https://doi.org/10.1145/3371083

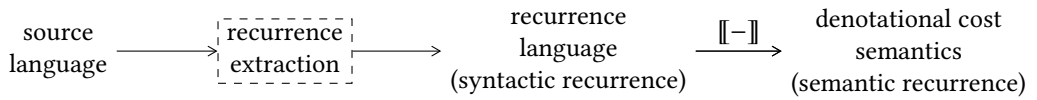

Fig. 1. Recurrence extraction

Danner et al. [2015] present such a method for a call-by-value terminating language with inductive types and structural recursion. Their method consists of the following steps:

- Given a program M in the *source language*—i.e. the language that we wish to analyse—a *syntactic recurrence* $\|M\|$ is extracted. This recurrence is expressed in an appropriate *recurrence language*, which includes primitives for expressing cost. This step is close in spirit to a monadic translation of the program into the writer monad. This approach is able to accommodate higher-order functions, assigning to them a higher-order recurrence expressing their cost in terms of a recurrence for their input.
- A *bounding relation* between source programs and syntactic recurrences is defined by induction on the types of the source language, i.e. as a logical relation. Intuitively, a source program is bounded by a recurrence if the components of the recurrence express *upper bounds* for the attributes of the source program, e.g. evaluation cost, size, etc.
- Following that, a *bounding theorem* is proved. This shows that every source program M is bounded by the extracted recurrence $\|M\|$.
- Finally, a denotational semantics is provided for the recurrence language. Depending on the intended application, this semantics abstracts inductive data types to some notion of size. For example, to consider binary trees up to their height we might interpret them as natural numbers, with the node constructor interpreted as the maximum. Alternatively, the node constructor may be interpreted as addition, thus yielding the number of nodes.

This strategy is shown schematically in Figure 1.

In this previous work, the *recurrence language* was taken to be a call-by-name language, so as to be as 'close to mathematics' as possible. Composing the extraction with the semantic interpretation then yields what one might call a *semantic recurrence*, which is intended to match the recurrence we would informally write when teaching undergraduate students—at least in the context of first-order programs. Thus the entire process gives a formal account and justification for informal cost analysis techniques. Factoring this into a syntactic and a semantic step is a useful tool for obtaining a simplifying account, which is additionally modular in the different notions of size for inductive data types. Nevertheless, it is still somewhat rigid with respect to changes to the *source language*: each source language requires a new and complex logical relations proof.

In this paper, we improve upon the above approach in several ways:

(1) We factor the syntactic phase into a cost-preserving embedding of the source language in an *intermediate language*, followed by a recurrence extraction for the intermediate language in the style of Danner et al. [2015]. The bounding theorem is proved once for the intermediate language, so for each source language we need only prove that the cost-preserving embeddings are correct. This strategy is illustrated Figure 2.
(2) In order to support a wide variety of languages, we choose as an intermediate language Call-by-Push-Value (CBPV) [Levy 2003], which embeds both call-by-value and call-by-name evaluation. We define recurrence extraction and prove a bounding theorem for CBPV, and obtain recurrence extraction and bounding theorems for both call-by-value (CBV) and call-by-name (CBN) source languages by embedding them in CBPV.

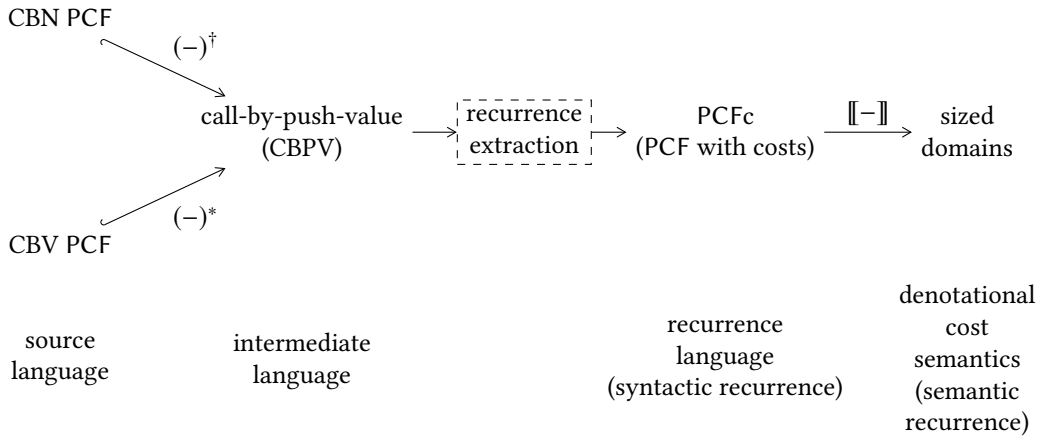

Fig. 2. Recurrence extraction through CBPV

(3) Our intermediate language is a version of CBPV with general recursion. Thus, we obtain a new recurrence extraction and bounding theorem result for CBV with general recursion, and a simpler recurrence extraction and bounding theorem for CBN than Kim [2016]. General recursion is important in our setting for writing divide-and-conquer algorithms in the standard way: in order to give a formal analogue of the standard informal techniques, we should not require the programmer to embed a termination metric in the source program.
(4) We generalize the denotational semantics from sets equipped with a size ordering to domains equipped with a size ordering. This involves some subtle questions regarding the interaction of the size ordering with the information order of the domains. We develop a semantics that identifies the *bottom* element of the domains (non-termination) with the *maximum* element of the size ordering (infinity), and more generally reverse-includes the information order in the size order—a more defined recurrence is a more precise bound. The key features of this semantics are encoded in a syntactic recurrence language PCFc.

In more detail, our technical development proceeds as follows. We first define three variants of PCF, starting in Section 2 with relatively standard CBV and CBN source languages. To isolate the issues involved in considering general recursion, we choose source languages with only natural numbers as data (treated as a flat base type). Next, in Section 3 we define the recurrence language PCF*c*, a version of call-by-name PCF with an additional type of costs. The key axiomatic component of PCF*c* is its *size order* (really, a pre-order relation). The size order codifies the minimum requirements necessary to prove the bounding theorem, while subsequent denotational interpretations of PCF*c* can refine this to specific notions of cost and size. The primary conceptual difficulty in the size order is understanding how it interacts with the implicit information order that arises from models of PCF, which is necessary for verifying the bounding theorem for recurrences extracted from recursively-defined functions. Our approach takes more defined recurrences to be *smaller* in the size order, i.e. to be better bounds. This is encoded in two of the rules that axiomatize the size order: one that asserts that rational chains are decreasing in the size order, and one that asserts that if all the approximants to a fixed point are a bound, then so is the fixed point itself.

Next, we describe the "end-user" recurrence extraction functions and corresponding bounding theorems for CBV and CBN source languages in Section 4. The statements of these theorems for the two source languages are independent of the intermediate language, but we will prove them by

factoring them through it, as indicated in Figure 2. These two examples illustrate the flexibility of our approach: even though they originate from the same general theorem, the the two bounding relations are very different in spirit.

The rest of the paper is largely devoted to proving that the extraction functions in Section 4 are correct using the approach of Figure 2. We define the version of Levy's *Call-by-Push-Value* (CBPV) [Levy 2003] that we use as an intermediate language in Section 5. CBPV is a polarised λ-calculus that is structured around a fundamental dichotomy between *values* and *computations*. The primitives of CBPV provide logical mechanisms that control evaluation. In terms of expressivity, these mechanisms are strong enough to allow a faithful embedding of both CBN and CBV PCF in CBPV. The structure of CBPV ensures compatibility with computational effects such as printing, memory, etc. For the purposes of this paper, we will consider cost to be an effect: the embeddings of CBN and CBV into CBPV will explicitly mention a *command* that incurs evaluation cost whenever some should be incurred according to the source language operational semantics. This makes the embeddings essentially parametric in the evaluation costs: even if we decide to change the costs—e.g. by charging twice as much for a function call—we only need to slightly alter the embedding into CBPV and the accompanying cost-preservation proof. We thus demonstrate that CBPV is an extremely flexible and general intermediate language for recurrence extraction.

The extraction function and bounding relation for CBPV is defined in Section 6. It is here that we reap the benefits of the distinction between values and computations, for it is clear that they require different notions of bounding. This in turn explicates the differences in the bounding relations for our original call-by-value and call-by-name versions of PCF. The cost-preserving embeddings of the two languages that we give in Section 7 are fundamentally different: following Levy [2003], CBN types are translated to CBPV computation types, whereas CBV types are translated to CBPV value types.

We define a denotational semantics for the recurrence language PCFc in Section 8 by introducing *sized domains*, which combine size and information orders. We observe that the standard model of PCF extends to a model in sized domains, in which the natural numbers are given their usual order. In Section 9, we use this model to give an end-to-end example of recurrence extraction for a non-structural recursion, in order to show that the recurrence we obtain indeed matches the one we would expect from an informal analysis. Finally, in Section 10 we sketch an extension of our work to a call-by-value source language with inductive types, following the approach of [Danner et al. 2015], and use it to extract a recurrence from a program implementing merge sort. Both this and the exponentiation example validate our claim that our technique formalizes the usual informal extract-and-solve approach to cost analysis.

2 THE SOURCE LANGUAGES: PCF

Our objective in this paper is to present a completely formal and provably correct way to extract recurrences from general-recursive functional programs. For the sake of clarity, we target a small formal calculus that illustrates the core features of this style of programming. Perhaps the most well-understood calculus of this sort is PCF [Plotkin 1977]. PCF essentially consists of a simply-typed λ-calculus, a base type of natural numbers, and some mechanism for obtaining fixed points. Almost every introductory book on programming language semantics includes a wealth of material on it: see e.g. [Gunter 1992; Mitchell 1996; Streicher 2006]. As discussed in the introduction, we would like to extract recurrences from both a CBN and a CBV variant of PCF.

The types, terms, and typing judgments of PCF are defined in Figure 3. The version we use comes with flat natural numbers, product, and function types. We write $\mathcal{T}_A^{\mathrm{PCF}}$ for the (open and closed) terms of PCF of type A, and $\mathcal{V}_A^{\mathrm{PCF}}$ for the CBV canonical forms of type A.

$$
\begin{array}{rlll}
\textbf{Types} & A, B & ::= & \mathsf{nat} \mid A \times B \mid A \to B \\
\textbf{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A \\
\textbf{Numerical operations} & \mathbf{op} & ::= & + \mid * \mid - \mid \div \mid \mathrm{mod} \\
\textbf{Canonical forms (CBN)} & V & ::= & \underline{n} \mid \langle M, N \rangle \mid \lambda x.\, M \\
\textbf{Canonical forms (CBV)} & V, W, Z & ::= & \underline{n} \mid \langle V, W \rangle \mid \lambda x.\, M \mid \mathsf{rec}\, f(x) = M
\end{array}
$$

$$
\frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{}{\Gamma \vdash \underline{n} : \mathsf{nat}} \qquad \frac{\Gamma \vdash N : \mathsf{nat} \quad \Gamma \vdash P, Q : A}{\Gamma \vdash \mathsf{if}\ N\ \mathsf{then}\ P\ \mathsf{else}\ Q : A} \qquad \frac{\Gamma \vdash M : \mathsf{nat} \quad \Gamma \vdash N : \mathsf{nat}}{\Gamma \vdash M\ \mathbf{op}\ N : \mathsf{nat}}
$$

$$
\frac{\Gamma, x : A \vdash M : A}{\Gamma \vdash \mathsf{fix}\ x.\, M : A} \qquad \frac{\Gamma \vdash M : A_1 \quad \Gamma \vdash N : A_2}{\Gamma \vdash \langle M, N \rangle : A_1 \times A_2} \qquad \frac{\Gamma \vdash P : A_1 \times A_2}{\Gamma \vdash \pi_i(P) : A_i}
$$

$$
\frac{\Gamma, f : A \to B, x : A \vdash M : B}{\Gamma \vdash \mathsf{rec}\, f(x) = M : A \to B} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}
$$

Fig. 3. PCF

Our natural numbers are introduced by constants $\underline{0}, \underline{1}, \ldots$ and correspond to eager, rather than lazy, natural numbers. Instead of the usual predecessor and successor functions, we introduce five arithmetic operations on natural numbers, as well as the zero test if N then P else Q, which behaves like P if N is $\underline{0}$, and like Q otherwise. This choice of primitives gives them the flavor of *machine words*. We sometimes refer to these as "flat" natural numbers, as they admit a domain interpretation with a flat information order (bottom below all numerals, and no other relations).

The terms of the CBN and the CBV variants differ only on fixed points. In the case of CBN we may take fixed points at every type: if we have $M : A$ in terms of $x : A$, we may construct $\mathsf{fix}\ x.\, M : A$. However, in CBV we are only allowed to take fixed points at function types, i.e. to make recursive function definitions: if we have $M : B$ in terms of an argument $x : A$ and a recursive variable $f : A \to B$, then we can obtain a recursively defined function $\mathsf{rec}\, f(x) = M : A \to B$. This formulation of CBV PCF is commonly found in the literature; see e.g. [Levy 2006, §7], [Fiore 1996, §1.2.5], or [Plotkin and Power 2001].

The big-step semantics of the two variants of PCF are defined in Figure 4. Judgments of the form $M \downarrow^n V$ are read as "M evaluates to canonical form V incurring a cost n in the process." Except the presence of the cost superscripts, these rules are standard in the literature on PCF, while the cost annotations follow Blelloch and Greiner [1995]. Each rule sums the costs incurred in the premises and adds additional cost for the evaluation step represented by that rule. For simplicity, we choose to charge one unit of cost for pair projections and (possibly recursive) function applications, and no cost otherwise. This could easily be adjusted to give any constant cost for each reduction, but—as recurrence extraction is syntax-directed—each reduction for the same term constructor must incur the same cost (e.g. we cannot charge 1 for reducing an if to the 0 branch and 2 for reducing it to the other branch). We have that

Theorem 2.1 (Determinacy). *There is at most one pair n, V such that $M \downarrow^n V$.*

We let $M{\downarrow} \stackrel{\text{def}}{\equiv} \exists n, V.\, M \downarrow^n V$ which we pronounce as "M is bounded," and write $M{\uparrow}$ to mean $\neg\,(M{\downarrow})$.

Rules for all variants

$$\frac{}{\underline{n} \downarrow^0 \underline{n}} \qquad \frac{M \downarrow^a \underline{m} \quad N \downarrow^b \underline{n}}{M \mathbf{op} N \downarrow^{a+b} \underline{m \mathbf{op} n}} \qquad \frac{}{\lambda x.\, M \downarrow^0 \lambda x.\, M}$$

$$\frac{N \downarrow^a \underline{0} \quad P \downarrow^b V}{\mathsf{if}\ N\ \mathsf{then}\ P\ \mathsf{else}\ Q \downarrow^{a+b} V} \qquad \frac{N \downarrow^a \underline{n+1} \quad Q \downarrow^b V}{\mathsf{if}\ N\ \mathsf{then}\ P\ \mathsf{else}\ Q \downarrow^{a+b} V}$$

Call-by-name PCF

$$\frac{}{\langle M, N\rangle \downarrow^0 \langle M, N\rangle} \qquad \frac{P \downarrow^a \langle M_1, M_2\rangle \quad M_i \downarrow^b V_i}{\pi_i(P) \downarrow^{a+b+1} V_i} \qquad \frac{M \downarrow^m \lambda x.\, P \quad P[N/x] \downarrow^n V}{MN \downarrow^{m+n+1} V} \qquad \frac{M[\mathsf{fix}\ x.\, M/x] \downarrow^n V}{\mathsf{fix}\ x.\, M \downarrow^{n+1} V}$$

Call-by-value PCF

$$\frac{}{\mathrm{rec}\, f(x) = P \downarrow^0 \mathrm{rec}\, f(x) = P} \qquad \frac{M \downarrow^a V \quad N \downarrow^b W}{\langle M, N\rangle \downarrow^{a+b} \langle V, W\rangle} \qquad \frac{P \downarrow^n \langle V_1, V_2\rangle}{\pi_i(P) \downarrow^{n+1} V_i}$$

$$\frac{M \downarrow^m \mathrm{rec}\, f(x) = P \quad N \downarrow^n W \quad P[\mathrm{rec}\, f(x) = P/f, W/x] \downarrow^k V}{MN \downarrow^{m+n+k+1} V} \qquad \frac{M \downarrow^m \lambda x.\, P \quad N \downarrow^n W \quad P[W/x] \downarrow^k V}{MN \downarrow^{m+n+k+1} V}$$

Fig. 4. Big-step semantics for PCF

$$\widehat{0} \stackrel{\text{def}}{=} \mathbf{0} \qquad \widehat{n+1} \stackrel{\text{def}}{=} \mathbf{1} \boxplus \widehat{n}$$

Types $\quad A, B ::= \cdots \mid \mathbb{C}$

Canonical forms (CBN only) $\quad V ::= \cdots \mid \widehat{n}$

Typing rules: The CBN rules of Figure 3, plus $\dfrac{\widehat{n} \in \{\mathbf{0}, \mathbf{1}\}}{\Gamma \vdash \widehat{n} : \mathbb{C}}$ and $\dfrac{\Gamma \vdash M : \mathbb{C} \quad \Gamma \vdash N : \mathbb{C}}{\Gamma \vdash M \boxplus N : \mathbb{C}}$

Big-step semantics: The CBN rules of Figure 4, plus $\dfrac{\widehat{n} \in \{\mathbf{0}, \mathbf{1}\}}{\widehat{n} \downarrow \widehat{n}}$ and $\dfrac{M \downarrow \widehat{m} \qquad N \downarrow \widehat{n}}{M \boxplus N \downarrow \widehat{m+n}}$

Fig. 5. PCF with costs

3 THE RECURRENCE LANGUAGE: PCF WITH COSTS

The recurrences we aim to extract from PCF terms will themselves be expressed in an appropriate *recurrence language*, which will also be a version of PCF, and which we call *PCF with costs* (PCFc). As mentioned before, we would like the recurrence language to behave in a way that is as close to mathematics as possible. Hence, we choose PCFc to be a CBN language.

In addition to standard constructs, PCFc also sports an additional type of costs, which we denote by $\mathbb{C}$. We assume as little as possible about costs: there are constants for no cost and unit cost, denoted $\mathbf{0}$ and $\mathbf{1}$. Furthermore, PCFc comes with an operator that adds arbitrary costs together: if $M : \mathbb{C}$ and $N : \mathbb{C}$ we have $M \boxplus N : \mathbb{C}$. We define $\widehat{n}$ to be the right-associated sum $\mathbf{1} \boxplus \ldots \boxplus \mathbf{0}$ of n unit costs. We write $\mathcal{T}_A^{\mathrm{PCFc}}$ for the (open and closed) terms of PCFc of type A.

The operational semantics of PCFc consists of a judgment $M \downarrow V$, which is read as "term M evaluates to canonical form V." Notice that this judgment does not record any evaluation costs, as we are not interested in the running time of recurrences. The rules for $M \downarrow V$ consist of all the CBN rules of Figure 4—with the cost annotations erased—along with two additional rules that handle the evaluation of costs: one which allows $\mathbf{0}$ and $\mathbf{1}$ to evaluate to themselves, and one which handles the summation of costs in the expected way. These facts are summarised in Figure 5.

However, the operational semantics of PCFc play a very limited rôle: they are only used to define the predicate $M{\downarrow}$ for PCFc. Instead, the central device used alongside PCFc is the *size order*. It consists of judgments of the form $\Gamma \vdash M \leqslant N : A$, which we read as "$M$ is bounded above by N at type A in context Γ." The size order is used to compare the costs denoted by recurrences, as well as the sizes of values.

The full inductive definition of the size order is introduced in Figure 6. To explain its definition, we first need to discuss three auxiliary notions.

Syntactic Unfolding. The *nth syntactic unfolding* of a fixed point term, denoted by $\mathsf{fix}_n\ x.\ M$, is in a sense the "nth approximation" to the fixed point $\mathsf{fix}\ x.\ M$. Intuitively, $\mathsf{fix}_n\ x.\ M$ is the term that runs for up to n recursive calls, and in the next call decides to diverge. It is defined by

$$\mathsf{fix}_0\ x.\ M \stackrel{\text{def}}{=} \mathsf{fix}\ x.\ x$$

$$\mathsf{fix}_{n+1}\ x.\ M \stackrel{\text{def}}{=} M[\mathsf{fix}_n\ x.\ M/x]$$

It is easy to see that

Proposition 3.1. *If* $\Gamma, x : A \vdash M : A$ *then* $\Gamma \vdash \mathit{fix}_n\ x.\ M : A$ *for any* $n \geq 0$.

Monotone contexts. As is usual, PCF term contexts are defined in a manner similar to PCF terms. The difference is that PCF contexts have had one of their subterms replaced by a *hole*, denoted by $[]$. For example, the context $C \stackrel{\text{def}}{=} \lambda x.M([] + 3)$ is obtained by replacing the term N in $\lambda x.M(N + 3)$ by a hole $[]$. We may recover the original term $\lambda x.M(N + 3)$ by *filling* the hole with N, which we write as $C[N]$ and define in the usual way. Recall also that—unlike substitution—filling a hole may result in variable capture. For example, the variable x is no longer free in $C[x] \equiv \lambda x.M(x + 3)$.

Of the contexts, we choose certain ones to be *monotone*, which means that they preserve the size order $\leqslant$. In order to prove the bounding theorem about recurrence extraction into PCFc, we will need that the monotone contexts include at least the principal positions of elimination forms, as well as introduction forms for negative types (i.e. the bodies of functions and pairs), as stated in Figure 6. It is always permissible, and sometimes desirable, to make more contexts monotone (e.g. all of them). However, this places more requirements on semantic models of PCFc, so in this figure we only include those that are necessary for proving the bounding theorem. Monotone contexts are introduced by a typing judgment $C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash B)$. The meaning of this judgment is this: the hole in C is assumed to represent a term $\Gamma \vdash M : A$, i.e. a term of type A in context Γ. When we fill the hole with a $\Gamma \vdash M : A$, the resultant term $C[M]$ has type B in context Δ. Note that the context may change, as C might bind some free variables.

Proposition 3.2 (Filling Typing). $\Gamma \vdash M : A \ \wedge\ C :: (\Gamma \vdash A) \rightsquigarrow_{mon.} (\Delta \vdash B) \implies \Delta \vdash C[M] : B$

Eliminative contexts. Of the monotone contexts, the (negative) *eliminative contexts* consist of a series of applications and projections. They are defined by

$$\mathcal{E} \ ::= \ [] \mid \pi_i(\mathcal{E}) \mid \mathcal{E}M$$

Eliminative contexts are strict in effects, and in particular they preserve divergence:

Lemma 3.3 (Preservation of Infinity). *For an eliminative context* $\mathcal{E}$, $M{\uparrow}$ *implies* $\mathcal{E}[M]{\uparrow}$.

$$\frac{}{[] :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Gamma \vdash A)} \qquad \frac{C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash \mathsf{nat}) \quad \Delta \vdash M, N : C}{\mathsf{if}\ C\ \mathsf{then}\ M\ \mathsf{else}\ N :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash C)}$$

$$\frac{\Delta \vdash M : \mathbb{C} \quad C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash \mathbb{C})}{M \boxplus C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash \mathbb{C})} \qquad \frac{C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash \mathbb{C}) \quad \Delta \vdash N : \mathbb{C}}{C \boxplus N :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash \mathbb{C})}$$

$$\frac{C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta, x : B \vdash C)}{\lambda x.\, C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash B \to C)} \qquad \frac{C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash B \to C) \quad \Delta \vdash N : B}{CN :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash C)}$$

$$\frac{\Delta \vdash M : B_1 \quad C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash B_2)}{\langle M, C\rangle :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash B_1 \times B_2)} \qquad \frac{C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash B_1) \quad \Delta \vdash N : B_2}{\langle C, N\rangle :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash B_1 \times B_2)}$$

$$\frac{C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash A_1 \times A_2)}{\pi_i(C) :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash A_i)} \qquad \frac{C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash \mathsf{nat}) \quad \Delta \vdash N : \mathsf{nat}}{C\ \mathbf{op}\ N :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash \mathsf{nat})}$$

$$\frac{C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash \mathsf{nat}) \quad \Delta \vdash N : \mathsf{nat} \quad \mathbf{op}^+ \in \{+, *\}}{M\ \mathbf{op}^+\ C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash \mathsf{nat})}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M \leqslant M : A}\ (\text{refl}) \qquad \frac{\Gamma \vdash M \leqslant N : A \quad \Gamma \vdash N \leqslant P : A}{\Gamma \vdash M \leqslant P : A}\ (\text{trans})$$

$$\frac{\Gamma \vdash M : \mathbb{C}}{\Gamma \vdash M \leqslant \mathbf{0} \boxplus M : \mathbb{C}}\ (\text{zero}) \qquad \frac{\Gamma \vdash M : \mathbb{C} \quad \Gamma \vdash N : \mathbb{C} \quad \Gamma \vdash P : \mathbb{C}}{\Gamma \vdash M \boxplus (N \boxplus P) \leqslant (M \boxplus N) \boxplus P : \mathbb{C}}\ (\text{assoc})$$

$$\frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash N : A}{\Gamma \vdash N[M/x] \leqslant (\lambda x.\, M)N : B}\ (\to_\beta) \qquad \frac{\Gamma \vdash M_1 : A_1 \quad \Gamma \vdash M_2 : A_2}{\Gamma \vdash M_i \leqslant \pi_i(\langle M_1, M_2\rangle) : A_i}\ (\times_\beta)$$

$$\frac{\Gamma \vdash P : A \quad \Gamma \vdash Q : A}{\Gamma \vdash P \leqslant \mathsf{if}\ \underline{0}\ \mathsf{then}\ P\ \mathsf{else}\ Q : A}\ (\text{if}_{\mathsf{tt}}) \qquad \frac{\Gamma \vdash P : A \quad \Gamma \vdash Q : A}{\Gamma \vdash Q \leqslant \mathsf{if}\ \underline{n+1}\ \mathsf{then}\ P\ \mathsf{else}\ Q : A}\ (\text{if}_{\mathsf{ff}})$$

$$\frac{}{\Gamma \vdash \underline{n}\ \mathbf{op}\ \underline{m} \leqslant \underline{n}\ \mathbf{op}\ \underline{m} : \mathsf{nat}}\ (\text{num}_\beta) \qquad \frac{\Gamma \vdash N : \mathsf{nat}}{\Gamma \vdash \underline{m} \bmod N \leqslant N - 1 : \mathsf{nat}}\ (\text{mod})$$

$$\frac{\Gamma \vdash M \leqslant N : A \quad C :: (\Gamma \vdash A) \rightsquigarrow_{\text{mon.}} (\Delta \vdash B)}{\Delta \vdash C[M] \leqslant C[N] : B}\ (\text{ctx}) \qquad \frac{\Gamma, x : A \vdash E : A}{\Gamma \vdash \mathsf{fix}_{n+1}\ x.\, E \leqslant \mathsf{fix}_n\ x.\, E : A}\ (\text{rat})$$

$$\frac{\Gamma, x : A \vdash E : A \quad \Delta, z : A \vdash \mathcal{E}[z] : B \quad \forall n \geq 0.\ \Delta \vdash M \leqslant \mathcal{E}[\mathsf{fix}_n\ x.\, E] : B}{\Delta \vdash M \leqslant \mathcal{E}[\mathsf{fix}\ x.\, E] : B}\ (\text{cpind})$$

Fig. 6. Monotone contexts and the size order for PCFc

The rules of the size order. The rules of the size order given in Figure 6 consist of exactly those that are needed in order to prove the bounding theorem of §6. For example, we do not assume that $\underline{n} \leqslant \underline{n+1} : \mathsf{nat}$ or $\mathbf{0} \leqslant \mathbf{1} : \mathbb{C}$, so that we can consider models where $\leqslant$ is actually equality, in addition to models where these inequalities do hold (such as the one given in Section 8).

The rules for $\leqslant$ can be naturally organised in three groups. The first group is the one that makes $\leqslant$ a preorder—i.e. (refl), (trans)—and preserved by monotone contexts (ctx). The second

group contains rules that encode small-step β-reduction: these ensure that $N \leqslant M$ whenever we would have had $M \longrightarrow_\beta N$. This group contains the rules (zero) and (assoc), which ensure that $\widehat{n+m} \leqslant \widehat{n} + \widehat{m}$ for all $n, m \in \mathbb{N}$, as a simple induction shows.

Finally, the third and most interesting group consists of the two rules that concern fixed points. Both of these rules deal with sequences of terms that we call *rational chains*, i.e. sequences of the form $(\mathsf{fix}_i\ x.\ M)_{i\in\omega}$, which semantically correspond to chains of the form $\left(f^i(\bot)\right)_{i\in\omega}$. Rational chains are ubiquitous in syntactic and intensional models of PCF: see e.g. [Abramsky et al. 1996; Escardó and Ho 2009; Milner 1977; Pitts 1997].

The rule (rat) ensures that rational chains are *decreasing* in the size order, i.e. that

$$\ldots \leqslant \mathsf{fix}_2\ x.\ M \leqslant \mathsf{fix}_1\ x.\ M \leqslant \mathsf{fix}_0\ x.\ M \stackrel{\text{def}}{=} \infty$$

Recall that the $(n+1)$th syntactic unfolding may make more recursive calls than the nth, so it is a more defined term. Hence, this rule intuitively states that *a more defined recurrence is a tighter bound*. Note that as $\mathsf{fix}_{n+1}\ x.\ M$ is defined to be $M[\mathsf{fix}_n\ x.\ M/x]$, this rule is analogous to β for fix.[1]

Finally, the rule (cpind) is a form of *computational induction* for PCFc. It ensures that this process of iterative tightening of bounds does not 'overshoot': if we can prove that $\mathsf{fix}_n\ x.\ E$ is an upper bound for M for every $n \geq 0$, then so is $\mathsf{fix}\ x.\ E$. Moreover, this rule ensures that computational induction can take place under any eliminative context $\mathcal{E}$.

We immediately obtain the following results.

Lemma 3.4 (Infinite Loop is a Top Element (Infinity)). $\Gamma \vdash M \leqslant \mathit{fix}\, x.\ x : A$ *for all* $\Gamma \vdash M : A$.

Proof. Use the rule (rat), $n = 0$ for a fresh variable $x \notin \textsc{Vars}(\Gamma)$. □

Lemma 3.5 (Bounded terms are a lower set). $M{\downarrow} \ \wedge\ N \leqslant M : A \implies N{\downarrow}$

Proof. By induction on $N \leqslant M : A$. □

4 RECURRENCE EXTRACTION FOR CALL-BY-NAME AND CALL-BY-VALUE

We now have enough details in place to show how to extract recurrences for both call-by-name and call-by-value PCF.

4.1 Call-by-Value

The extraction procedure for CBV is very similar to that of [Danner et al. 2015, 2013], and is displayed in Figure 7. To begin, we map each type A of CBV PCF to a type

$$\|A\| \stackrel{\text{def}}{=} \mathbb{C} \times \langle\!\langle A \rangle\!\rangle$$

of PCFc, where $\langle\!\langle A \rangle\!\rangle$ is defined by induction on A. We call this the type of *complexities for* A. A complexity for A consists of a pair: its first component is the *cost* of evaluating a term of type A to a value, and its second component is the *potential* of the resultant value. If $E : \|A\|$ we write $E_c \stackrel{\text{def}}{=} \pi_1(E) : \mathbb{C}$ and $E_p \stackrel{\text{def}}{=} \pi_2(E) : \langle\!\langle A \rangle\!\rangle$ to refer to these two components respectively.

The notion of potential is crucial in CBV extraction. There are varying interpretations about the nature of potentials: in one reading, they—directly or indirectly—encode information about the *size* of values; in another, they represent the future cost of using that value, to which we often refer as *use-cost*. The definition of $\langle\!\langle A \rangle\!\rangle$ is also given in Figure 7. Perhaps the most interesting clause is $\langle\!\langle A \to B \rangle\!\rangle \stackrel{\text{def}}{=} \langle\!\langle A \rangle\!\rangle \to \|B\|$. We may read this as follows. The cost of using a value of function

[1] We do not require β for fix itself as a size order axiom in order prove the bounding theorem, essentially because in the case of the bounding theorem for fixed points, we will apply a fixed point induction principle to instead reason about fix_n. The β axioms are generally used in the proof of the bounding theorem to head expand, showing that a redex is in the relation if its reduct is, but this particular reduction does not come up.

$$\|A\| \stackrel{\text{def}}{=} \mathbb{C} \times \langle\!\langle A \rangle\!\rangle$$

$$\langle\!\langle \text{nat} \rangle\!\rangle \stackrel{\text{def}}{=} \text{nat}$$

$$\langle\!\langle A_1 \times A_2 \rangle\!\rangle \stackrel{\text{def}}{=} \langle\!\langle A_1 \rangle\!\rangle \times \langle\!\langle A_2 \rangle\!\rangle$$

$$\langle\!\langle A \to B \rangle\!\rangle \stackrel{\text{def}}{=} \langle\!\langle A \rangle\!\rangle \to \|B\|$$

$$\|x\| \stackrel{\text{def}}{=} \langle \mathbf{0}, x \rangle$$

$$\|\underline{n}\| \stackrel{\text{def}}{=} \langle \mathbf{0}, \underline{n} \rangle$$

$$\|M\{+,*\}N\| \stackrel{\text{def}}{=} \left\langle \|M\|_c \boxplus \|N\|_c \,,\, \|M\|_p \,\{+,*\}\, \|N\|_p \right\rangle$$

$$\left\|M\{-,\div\}\underline{n}\right\| \stackrel{\text{def}}{=} \left\langle \|M\|_c \,,\, \|M\|_p \,\{-,\div\}\, \underline{n} \right\rangle$$

$$\|M\{-,\div\}N\| \stackrel{\text{def}}{=} \left\langle \|M\|_c \boxplus \|N\|_c \,,\, \|M\|_p \right\rangle (\text{if } N \neq \underline{n})$$

$$\|M \bmod N\| \stackrel{\text{def}}{=} \left\langle \|M\|_c \boxplus \|N\|_c \,,\, \|N\|_- \underline{1} \right\rangle$$

$$\|\text{if } N \text{ then } P \text{ else } Q\| \stackrel{\text{def}}{=} \|N\|_c +_{\text{c}} \text{if } \|N\|_p \text{ then } \|P\| \text{ else } \|Q\|$$

$$\|\langle M, N \rangle\| \stackrel{\text{def}}{=} \left\langle \|M\|_c \boxplus \|N\|_c \,,\, \left\langle \|M\|_p \,,\, \|N\|_p \right\rangle \right\rangle$$

$$\|\pi_i(M)\| \stackrel{\text{def}}{=}_c \left\langle \mathbf{1} \boxplus \|M\|_c \,,\, \pi_i \left(\|M\|_p \right) \right\rangle$$

$$\|\lambda x.\ M\| \stackrel{\text{def}}{=} \langle \mathbf{0}, \lambda x.\ \|M\| \rangle$$

$$\|M\,N\| \stackrel{\text{def}}{=} \mathbf{1} \boxplus \|M\|_c \boxplus \|N\|_c +_{\text{c}} (\|M\|_p \ \|N\|_p)$$

$$\|\text{rec } f(x) = M\| \stackrel{\text{def}}{=} \langle \mathbf{0}, \text{fix } f.\ \lambda x.\ \|M\| \rangle$$

$$\text{If } \Gamma \vdash E : \|A\| \text{ and } \Gamma \vdash C : \mathbb{C} \text{ we write } \begin{cases} \Gamma \vdash E_c \stackrel{\text{def}}{=} \pi_1(E) : \mathbb{C} \\ \Gamma \vdash E_p \stackrel{\text{def}}{=} \pi_2(E) : \langle\!\langle A \rangle\!\rangle \\ \Gamma \vdash C +_{\text{c}} E \stackrel{\text{def}}{=} \langle C \boxplus \pi_1(E), \pi_2(E) \rangle : \|A\| \end{cases}$$

Fig. 7. Recurrence extraction for CBV PCF

type—i.e. a λ-expression—is expressed as a function itself. This function that maps the size/use-cost of a value of type A to a pair of an evaluation cost (the cost of evaluating the application of that λ-expression to a value with that use-cost) and the use-cost of a value of type B. This is consistent with the idea that in CBV *variables represent values*.

Following that, we define a map that extracts a recurrence from each term of CBV PCF, which we also denote by $\|-\|$ (see Danner et al. [2015] for an explanation of most of the cases). Arithmetic operations M **op** N are chosen to have zero cost in addition to the cost of evaluating their inputs (since for big-O bounds we mainly count recursive calls), but for their potential we distinguish some cases, guided by two constraints needed by our proof of the bounding theorem: the potential must be an upper bound on the value of the operation, and must be monotone in the potentials of the operands wherever mathematically possible Addition/multiplication are monotone in both M and N, so we use addition/multiplication to combine the potentials of M and N, which will be an upper bound on $M\{+,*\}N$. In general, subtraction and division are monotone in M but antimonotone in N, so we take the potential to be that of M, which is an upper bound on $M\{-,\div\}N$. However, for subtraction or division with N a numeric constant there will be no monotonicity obligation for N (because it does not vary), so we can perform the $\{-,\div\}$ in the potential. This precision is useful for analyzing algorithms that subtract/divide by a constant in recursive calls. Finally, mod is not monotone in either position, so we cannot use mod in the potential; but it is bounded by both M and $N-1$, and we somewhat arbitrarily choose the later. For algorithms (e.g. GCD) that recur on $x\{-,\div,\text{mod}\}y$, we would need more sophisticated tracking of monotone and antimonotone

positions in the recurrence language. The other new case is for recursive functions, and interprets them using fixed points in PCFc.

Extending $\langle\!\langle - \rangle\!\rangle$ to contexts pointwise, we have

Theorem 4.1. *If* $\Gamma \vdash M : A$ *then* $\langle\!\langle \Gamma \rangle\!\rangle \vdash \|M\| : \|A\|$.

It remains to state a theorem to the effect that each extracted recurrence $\|M\| : \|A\|$ encodes an upper bound for the evaluation of M. Because of the presence of function types, this is stated as a logical relation:

Theorem 4.2 (CBV extraction). *There exist relations*

$$(M : \mathcal{T}_A^{PCF,closed}) \sqsubseteq_A (E : \mathcal{T}_{\|A\|}^{PCFc,closed}) \quad \text{and} \quad (V : \mathcal{V}_A^{PCF,closed}) \sqsubseteq_A^{\mathrm{val}} (E : \mathcal{T}_{\langle\!\langle A \rangle\!\rangle}^{PCFc,closed})$$

such that

$$\begin{aligned}
M \sqsubseteq_A E \quad &\Longrightarrow \quad \text{if } E_c\downarrow \text{ then } \exists n, V.\ \begin{cases} M \downarrow^n V \\ \widehat{n} \leqslant E_c \\ V \sqsubseteq_A^{\mathrm{val}} E_p \end{cases} \\
\underline{n} \sqsubseteq_{nat}^{\mathrm{val}} E \quad &\Longrightarrow \quad \underline{n} \leqslant E \\
\langle V_1, V_2 \rangle \sqsubseteq_{A_1 \times A_2}^{\mathrm{val}} E \quad &\Longrightarrow \quad \begin{cases} V_1 \sqsubseteq_{A_1}^{\mathrm{val}} \pi_1(E) \\ V_2 \sqsubseteq_{A_2}^{\mathrm{val}} \pi_2(E) \end{cases} \\
\lambda x.M \sqsubseteq_{A\to B}^{\mathrm{val}} E \quad &\Longrightarrow \quad \forall (N \sqsubseteq_A^{\mathrm{val}} X).\ M[N/x] \sqsubseteq_B E\,X \\
rec\, f(x) = P \sqsubseteq_{A\to B}^{\mathrm{val}} E \quad &\Longrightarrow \quad \forall (N \sqsubseteq_A^{\mathrm{val}} X).\ P[rec\, f(x) = P/f, N/x] \sqsubseteq_B E\,X
\end{aligned}$$

and, moreover,

(1) $V \sqsubseteq_A^{\mathrm{val}} \|V\|_p$ *for any CBV PCF value* $\cdot \vdash V : A$,
(2) $M \sqsubseteq_A \|M\|$ *for any closed CBV PCF term* $\cdot \vdash M : A$.

We prove this as a corollary of recurrence extraction for CBPV below. Intuitively, the *expression bounding relation* $M \sqsubseteq_A E$ says that the cost and value of a CBV PCF program M are predicted by E in the following sense: if the cost component of E terminates, then so does M, and the evaluation cost of M is bounded by the cost component of E according to the size order. Moreover, the value is bounded by the potential component of E, according to the value bounding relation $\sqsubseteq_A^{\mathrm{val}}$. The value bounding relation is type-directed: at nat, it says that n is bounded by E according to the size order; for pairs, it says that the components are bounded; and for functions it says that the applications are bounded.

Relative to the bounding relation in Danner et al. [2015], the key change here for supporting general recursion involves the quantifiers in expression bounding. In that work, expression bounding was defined as "if $M \downarrow^n V$ then $\widehat{n} \leqslant E_c$ and $V \sqsubseteq_A^{\mathrm{val}} E_p$" (if the source program evaluates, then the recurrence's prediction is correct), though because all programs in the language considered there terminate, this is equivalent to "$M \downarrow^n V$ and $\widehat{n} \leqslant E_c$ and $V \sqsubseteq_A^{\mathrm{val}} E_p$." Here, we assert this same guarantee *only if* the cost component of the recurrence itself terminates. We view a recurrence whose cost diverges as predicting an infinite cost for the program, so the expression bounding relation is vacuously true in this case, expressing that any program meets this bound.

4.2 Call-by-Name

The extraction procedure for CBN PCF is very different to the one for CBV. We will discover the precise reasons for that through CBPV in §7, but for now we will satisfy ourselves with the

$$
\begin{aligned}
\|\mathsf{nat}\| &\stackrel{\text{def}}{=} \mathbb{C} \times \mathsf{nat} \\
\|A_1 \times A_2\| &\stackrel{\text{def}}{=} \|A_1\| \times \|A_2\| \\
\|A \to B\| &\stackrel{\text{def}}{=} \|A\| \to \|B\|
\end{aligned}
\qquad
\begin{aligned}
\|x\| &\stackrel{\text{def}}{=} x \\
\left\|\underline{n}\right\| &\stackrel{\text{def}}{=} \langle \mathbf{0}, \underline{n} \rangle \\
\|M \ \mathbf{op}\ N\| &\stackrel{\text{def}}{=} \text{(same as in CBV)} \\
\|\mathsf{if}\ N\ \mathsf{then}\ P\ \mathsf{else}\ Q\| &\stackrel{\text{def}}{=} \|N\|_c +_A \mathsf{if}\ \|N\|_p\ \mathsf{then}\ \|P\|\ \mathsf{else}\ \|Q\| \\
\|\langle M, N \rangle\| &\stackrel{\text{def}}{=} \left\langle \mathbf{1} +_{A_1} \|M\|, \mathbf{1} +_{A_2} \|N\| \right\rangle \\
\|\pi_i(M)\| &\stackrel{\text{def}}{=} \pi_i\,(\|M\|) \\
\|\lambda x.\ M\| &\stackrel{\text{def}}{=} \lambda x.\ \mathbf{1} +_B \|M\| \\
\|M\,N\| &\stackrel{\text{def}}{=} \|M\|\ \|N\| \\
\|\mathsf{fix}\ x.\ M\| &\stackrel{\text{def}}{=} \mathsf{fix}\ x.\ \mathbf{1} +_A \|M\|
\end{aligned}
$$

$$
\begin{aligned}
c : \mathbb{C}, x : \mathbb{C} \times \mathsf{nat} \vdash \alpha_{\mathsf{nat}}(c, x) &\stackrel{\text{def}}{=} \langle c \boxplus \pi_1(x), \pi_2(x) \rangle &&: \mathbb{C} \times \mathsf{nat} \\
c : \mathbb{C}, p : \|A_1\| \times \|A_2\| \vdash \alpha_{A_1 \times A_2}(c, p) &\stackrel{\text{def}}{=} \left\langle \alpha_{A_1}(c, \pi_1(p)), \alpha_{A_2}(c, \pi_2(p)) \right\rangle &&: \|A_1\| \times \|A_2\| \\
c : \mathbb{C}, f : \|A\| \to \|B\| \vdash \alpha_{A \to B}(c, f) &\stackrel{\text{def}}{=} \lambda a.\ \alpha_B\,(c, f(a)) &&: \|A\| \to \|B\|
\end{aligned}
$$

Fig. 8. Recurrence extraction for CBN PCF

following intuitions. When studying sundry versions of PCF we often speak of certain types as being *observable*, in the sense that the successful normalization of terms at those types provides some manifest information to the user. For example, every term $M : \mathsf{nat}$ either diverges or is observed to converge to a constant $\underline{n}$. However, in CBN we are unable to observe a function: the only action we may perform is that of applying it to an argument. In contrast, in CBV *every type is observable*:[2] for example, when evaluating a term at function type we either expect divergence or convergence to a λ-expression.

In the case of CBN we only have one observable type, i.e. that of natural numbers. This is to say that we expect the end user to only evaluate terms of type nat. It follows that only complexities for those terms should contain cost components. Complexities at any other type will need to come with some mechanism for "pushing costs down to nat." Thus, a complexity for a CBN type will not merely be a type of PCFc, but will also come with some kind of algebra structure that enables us to do that at every type.

Definition 4.3 (Cost algebra). A (PCFc) *cost algebra* $A = (A^\bullet, \alpha_A)$ consists of a type $A^\bullet$ and a PCFc term $c : \mathbb{C}, x : A^\bullet \vdash \alpha_A(c, x) : A^\bullet$ such that

$$c : \mathbb{C}, x : A^\bullet \vdash \alpha_A(c_1 \boxplus c_2, x) \leqslant \alpha_A(c_1, \alpha_A(c_2, x)) : A^\bullet$$

A PCFc cost algebra[3] $A = (A^\bullet, \alpha_A)$ consists of a type $A^\bullet$, which we call its *carrier*, and a *structure map* $\alpha_A(c, x)$, which is expressed as a term in two free variables. The structure map allows one to "add costs" to any term of carrier type. From this point onwards we will abuse notation by not making a formal distinction between algebras $A = (A^\bullet, \alpha_A)$ and their carriers $A^\bullet$.

[2]There is a third paradigm which is close to CBN, but where termination at function type is observable. This is usually called the *lazy* paradigm; see [Abramsky 1990], [Riecke 1993], and [Levy 2003, §1.7.3].

[3]The analogous equation $x : A^\bullet \vdash x \leqslant \alpha_A(\mathbf{0}, x)$ is true if $\leqslant$ additionally includes certain η-contractions, but this property is not used here so we omit these η rules for maximum generality.

The extraction procedure for CBN may be found in Figure 8. We begin by defining an algebra $(\|A\|, \alpha_A)$ for each type A of CBN PCF. In particular, nat is mapped to the *"free algebra"* $(\mathbb{C}\times\mathsf{nat}, \alpha_{\mathsf{nat}})$ on nat. For simplicity we use the notation $L +_A M \stackrel{\text{def}}{=} \alpha_A(L, M)$ whenever $L : \mathbb{C}$ and $M : \|A\|$.

We then inductively define a map $\|-\|$ from *typed* terms of PCF to terms of PCFc. This definition uses the aforementioned algebras, and so the output depends on the type of each input term.[4] This definition uses the algebras to push the cost of evaluation towards the ground types. Indeed, costs here are not added at the elimination rules, but *at the introduction rules*. Consider the lazy product type nat × nat as an example. A pair $\langle M, N\rangle$ of this type is not directly observable, but contains two observable computations M and N. A computation that uses this pair costs one unit more than a computation that uses either M or N directly, because it must do the product projection.

This translation is well-typed:

Theorem 4.4. *If* $\Gamma \vdash M : A$ *then* $\|\Gamma\| \vdash \|M\| : \|A\|$.

Furthermore, it satisfies a bounding theorem given as a logical relation:

Theorem 4.5 (CBN extraction). *There exists a relation*

$$(M : \mathcal{T}_A^{PCF,\,closed}) \;\sqsubseteq\; (E : \mathcal{T}_{\|A\|}^{PCFc,\,closed})$$

such that

$$M \sqsubseteq_{nat} E \quad\Longrightarrow\quad \text{if } E_c\downarrow \text{ then } \exists n, V.\ \begin{cases} M \downarrow^n V \\ \widehat{n} \leqslant E_c : \mathbb{C} \\ V \leqslant E_p : nat \end{cases}$$

$$M \sqsubseteq_{A_1\times A_2} E \quad\Longrightarrow\quad \begin{cases} \pi_1(M) \sqsubseteq_{A_1} \pi_1(E) \\ \pi_2(M) \sqsubseteq_{A_2} \pi_2(E) \end{cases}$$

$$M \sqsubseteq_{A\to B} E \quad\Longrightarrow\quad \forall (N \sqsubseteq_A X).\ M\,N \sqsubseteq_B E\,X$$

and, moreover, $M \sqsubseteq_A \|M\|$ *for any PCF term* $\cdot \vdash M : A$.

Unlike in CBV, we do not distinguish between value and expression relations. The CBN relation for nat (or other observable types, if we had them) is analogous to the CBV expression relation, while the clauses for negative types simply say that the elimination forms preserve relatedness. Again, rather than proving this bounding theorem directly, we will obtain it as a corollary of a general theorem about CBPV.

5 THE INTERMEDIATE LANGUAGE: LEVY'S CALL-BY-PUSH-VALUE (CBPV)

We aim to prove both theorems of Section 4 at once. In order to do so, we will embed both CBN and CBV PCF in an *intermediate language*, and we will do so in a cost-preserving manner. Finally, we will define a recurrence extraction function for this intermediate language, and prove that it is correct. The composite of these two processes will prove the results of Section 4.

Our intermediate language of choice of Levy's *Call-by-Push-Value* (CBPV) [Levy 2003]. CBPV is a polarised λ-calculus whose type structure provides ways of controlling evaluation. The typing judgments of the version of CBPV that we will use in this paper—which includes natural numbers and recursion—may be found in Figure 9.

CBPV is structured around a fundamental dichotomy between values and computations. In particular, it comes with two sorts of types: *value types*, usually denoted by A, and *computation*

[4] Formally, the function is defined on typing derivations/intrinsically typed syntax $\|\Gamma \vdash M : A\|$, but we elide the annotations for brevity.

Value Types	A	::=	$\mathsf{nat} \mid A_1 \times A_2 \mid U\underline{B}$
Computation Types	$\underline{B}$	::=	$FA \mid \underline{B}_1 \,\&\, \underline{B}_2 \mid A \to \underline{B}$
Contexts	Γ	::=	$\cdot \mid \Gamma, x : A$
Numerical operations	**op**	::=	$+ \mid - \mid * \mid \div \mid \mathrm{mod}$
Terminal Computations	T, S	:=	$\mathsf{return}\ V \mid \langle M, N\rangle \mid \lambda x.\ M$

$$\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{v}} x : A}\ (\mathsf{var}) \qquad \frac{}{\Gamma \vdash_{\mathsf{v}} \widetilde{n} : \mathsf{nat}}\ (\mathsf{nat}_{\widetilde{n}})$$

$$\frac{\Gamma \vdash_{\mathsf{c}} M : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{charge}.\, M : \underline{B}}\ (\mathsf{ch}) \qquad \frac{\Gamma, x : U\underline{B} \vdash_{\mathsf{c}} M : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{fix}\ x.\ M : \underline{B}}\ (\mathsf{fix})$$

$$\frac{\Gamma \vdash_{\mathsf{v}} M, N : \mathsf{nat} \quad \Gamma, z : \mathsf{nat} \vdash_{\mathsf{c}} Q : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{calc}\ z \Leftarrow M\ \mathbf{op}\ N\ \mathsf{in}\ Q : \underline{B}}\ (\mathsf{nat}_{\mathbf{op}}) \qquad \frac{\Gamma \vdash_{\mathsf{v}} N : \mathsf{nat} \quad \Gamma \vdash_{\mathsf{c}} P : \underline{B} \quad \Gamma \vdash_{\mathsf{c}} Q : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{ifz}(N;\ P;\ Q) : \underline{B}}\ (\mathsf{nat}_{\mathsf{ifz}})$$

$$\frac{\Gamma \vdash_{\mathsf{v}} V_1 : A_1 \quad \Gamma \vdash_{\mathsf{v}} V_2 : A_2}{\Gamma \vdash_{\mathsf{v}} (V_1, V_2) : A_1 \times A_2}\ (\times\mathcal{I}) \qquad \frac{\Gamma \vdash_{\mathsf{v}} V : A_1 \times A_2 \quad \Gamma, x_1 : A_1, x_2 : A_2 \vdash_{\mathsf{c}} N : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{split}(V; (x_1, x_2).\ N) : \underline{B}}\ (\times\mathcal{E})$$

$$\frac{\Gamma \vdash_{\mathsf{c}} M : \underline{B}}{\Gamma \vdash_{\mathsf{v}} \mathsf{thunk}\ M : U\underline{B}}\ (U\mathcal{I}) \qquad \frac{\Gamma \vdash_{\mathsf{v}} V : U\underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{force}\ V : \underline{B}}\ (U\mathcal{E})$$

$$\frac{\Gamma \vdash_{\mathsf{v}} V : A}{\Gamma \vdash_{\mathsf{c}} \mathsf{return}\ V : FA}\ (F\mathcal{I}) \qquad \frac{\Gamma \vdash_{\mathsf{c}} M : FA \quad \Gamma, x : A \vdash_{\mathsf{c}} N : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{bind}\ x \Leftarrow M\ \mathsf{in}\ N : \underline{B}}\ (F\mathcal{E})$$

$$\frac{\Gamma \vdash_{\mathsf{c}} M : \underline{B}_1 \quad \Gamma \vdash_{\mathsf{c}} N : \underline{B}_2}{\Gamma \vdash_{\mathsf{c}} \langle M, N\rangle : \underline{B}_1 \,\&\, \underline{B}_2}\ (\&\mathcal{I}) \qquad \frac{\Gamma \vdash_{\mathsf{c}} M : \underline{B}_1 \,\&\, \underline{B}_2}{\Gamma \vdash_{\mathsf{c}} \pi_i(M) : \underline{B}_i}\ (\&\mathcal{E})$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{c}} M : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \lambda x.\ M : A \to \underline{B}}\ (\to\mathcal{I}) \qquad \frac{\Gamma \vdash_{\mathsf{c}} M : A \to \underline{B} \quad \Gamma \vdash_{\mathsf{v}} N : A}{\Gamma \vdash_{\mathsf{c}} M\,N : \underline{B}}\ (\to\mathcal{E})$$

Fig. 9. Call-by-Push-Value (CBPV) with natural numbers and recursion

types, usually denoted by $\underline{B}$. Value types often include certain ground types, *positive* products, and *thunks*. Computation types include *free* computation types, *negative* product types, and function types. Like in CBV, variables in CBPV may only denote values, and are thus assigned value types.

The terms at value types are exactly values: the introduction rules encode all the possible ways of constructing them. For example, if $V : A_1$ and $W : A_2$, we have $(V, W) : A_1 \times A_2$. However, we may *not* project the components of that pair back into value types. But if we have a term $N : \underline{B}$ at some computation type with two free variables $x_1 : A_1$ and $x_2 : A_2$, then the elimination rule allows us to split a pair of values into N, yielding $\mathsf{split}((V, W); (x_1, x_2).\ N) : \underline{B}$. It is in this way that the invariant of value types is maintained.

As their name suggests, the terms at computation type may demonstrate some non-trivial computational behaviour, including effects (e.g. printing, output, etc). In the words of Paul Levy, *"a value is, a computation does."* The most basic such type is the *free* (computation) type FA, where A is a value type. A term $M : FA$ is a *returner*, which may engage in some effectful behaviour before returning some value by normalizing to the form $\mathsf{return}\ V$ where $V : A$. Given $M : FA$ and as $x : A \vdash N : \underline{B}$ we may form $\mathsf{bind}\ x \Leftarrow M\ \mathsf{in}\ N : \underline{B}$. This term will first run M, effecting some changes and returning a value which will then be substituted for x. Computation types also include *negative products*, which we write as $\underline{B}_1 \,\&\, \underline{B}_2$, and which come with the usual projections. Finally,

$$\frac{}{\lambda x.\, M \Downarrow^0 \lambda x.\, M} \qquad \frac{}{\mathsf{return}\ V \Downarrow^0 \mathsf{return}\ V} \qquad \frac{}{\langle M, N\rangle \Downarrow^0 \langle M, N\rangle}$$

$$\frac{M \Downarrow^m T}{\mathsf{force}\ (\mathsf{thunk}\ M) \Downarrow^m T} \qquad \frac{N[V_1/x_1, V_2/x_2] \Downarrow^m T}{\mathsf{split}((V_1, V_2); (x_1, x_2).\ N) \Downarrow^m T} \qquad \frac{M[\mathsf{thunk}\ (\mathsf{fix}\ x.\ M)/x] \Downarrow^n T}{\mathsf{fix}\ x.\ M \Downarrow^n T}$$

$$\frac{P \Downarrow^m T}{\mathsf{ifz}(\widetilde{0};\ P;\ Q) \Downarrow^m T} \qquad \frac{Q \Downarrow^m T}{\mathsf{ifz}(\widetilde{n+1};\ P;\ Q) \Downarrow^m T} \qquad \frac{P[\widetilde{m\ \mathbf{op}\ n}/z] \Downarrow^m T}{\mathsf{calc}\ z \Leftarrow \widetilde{m}\ \mathbf{op}\ \widetilde{n}\ \mathsf{in}\ P \Downarrow^m T}$$

$$\frac{M \Downarrow^n T}{\mathsf{charge}.\, M \Downarrow^{n+1} T} \qquad \frac{M \Downarrow^m \mathsf{return}\ V \qquad N[V/x] \Downarrow^n T}{\mathsf{bind}\ x \Leftarrow M\ \mathsf{in}\ N \Downarrow^{m+n} T}$$

$$\frac{P \Downarrow^m \langle M_1, M_2\rangle \qquad M_i \Downarrow^n T}{\pi_i(P) \Downarrow^{m+n} T} \qquad \frac{M \Downarrow^m \lambda x.\, P \qquad P[V/x] \Downarrow^n T}{MV \Downarrow^{m+n} T}$$

Fig. 10. Big-step semantics for CBPV

they include function types: given a computation $M : \underline{B}$ in a free value variable $x : A$, we may form $\lambda x.\, M : A \to \underline{B}$. Notice that $A \to \underline{B}$ is a 'mixed type,' which mirrors the fact that variables may only stand for values.

Only one thing remains, and that is to overcome the restriction of variables to value types. This is achieved by introducing a value type $U\underline{B}$, the so-called type of *thunks* of $\underline{B}$, for every computation type $\underline{B}$. Each computation $M : \underline{B}$ may be *thunked* as $\mathsf{thunk}\ M : U\underline{B}$, and each $N : U\underline{B}$ may be *forced* into a computation $\mathsf{force}\ M : \underline{B}$. Thunk types are crucial in simulating CBN. They are also necessary in the CBPV fixed point rule, which assumes that the recursive call is given in the form of a thunk.

Our version of CBPV also comes with 'flat' natural numbers as a value type. As in PCF, these are introduced by an infinite number of constants $\widetilde{0}, \widetilde{1}, \ldots$. We may use natural numbers in either of two ways. First, using the term $\mathsf{ifz}(\widetilde{n};\ P;\ Q)$ we may query them on whether they are zero, following the computation P if so, and Q otherwise. Second, using the construct $\mathsf{calc}\ v \Leftarrow \widetilde{m}\ \mathbf{op}\ \widetilde{n}\ \mathsf{in}\ N$ we may *calculate* one of four numerical operations on them, and substitute the result into some variable $v : \mathsf{nat}$ that is free in the 'continuation' N.

Finally, we equip our version of CBPV with a single effect, namely the one that incurs a unit cost: for each computation $M : \underline{B}$ we may form a computation $\mathsf{charge}.\, M : \underline{B}$. In section 7 we will use this effect in our embedding of CBN and CBV into CBPV to record the places where the +1's appear in the operational semantics of PCF.

Figure 10 introduces big-step semantics for CBPV. The judgment $M \Downarrow^n T$ means that the closed CBPV term M evaluates to the *terminal computation* T, incurring a cost n in the process. Terminal computations include $\mathsf{return}\ V$, pairs, and λ-expressions. Unlike the operational semantics of PCF, costs are incurred here only when they are explicitly mentioned as effects: with the exception of the rule for $\mathsf{charge}.\, M$, all the other rules sum the costs of their premises.

6 RECURRENCE EXTRACTION FOR CBPV & THE BOUNDING THEOREM

Having introduced our intermediate language, we will now show how to extract recurrences from its terms. Furthermore, we will prove the central result of this paper, the *Bounding Theorem*, which ensures that extracted recurrences express upper bounds for the running times of CBPV terms.

Compared with the results of [Danner et al. 2015], our results are a significant technical improvement. First, the distinction between *potentials* and *complexities* is now formally motivated: our

techniques demonstrate that *values have potentials*, whereas *computations have complexities*. We thus get a very clear conceptual basis for higher-order recurrences. Second, the fact that our version of CBPV comes with an explicit cost effect makes this result reusable and extensible: our theorem applies not just to a specific source language, but to any source language that can be faithfully embedded in CBPV. All we need to do is encode its terms in CBPV in a way that explicitly records where evaluation costs are incurred, and then invoke the bounding theorem.

6.1 CBPV Recurrence Extraction

The extraction procedure for CBPV incorporates elements of the extraction procedures for both CBV and CBN given in §4. It is rather close to the monad/algebra categorical semantics of CBPV given in [Levy 2006, §3] in the special case of the writer monad $(\mathbb{C} \times -)$.

To begin, we translate CBPV types to PCFc types as follows: a value type A is translated to a PCF type $\langle\!\langle A \rangle\!\rangle$, its so-called type of *potentials*, and a computation type $\underline{B}$ is translated to a *cost algebra* $\left\|\underline{B}\right\| \stackrel{\text{def}}{=} \left(\left\|\underline{B}\right\|^{\bullet}, \alpha_{\underline{B}}\right)$. Again, the idea is that values only have potential (future, indirect use-cost), whereas computations may be evaluated now, hence incurring (immediate, direct) costs. Because of the interdependence between value and computation types introduced by $F(-)$ and $U(-)$, the translation—which may be found in Figure 11—is defined in a mutually recursive manner. For the sake of precision we have maintained the formal distinction between algebras $\left\|\underline{B}\right\|$ and their carriers $\left\|\underline{B}\right\|^{\bullet}$. As before, we abbreviate $\alpha_{\underline{B}}(L, E)$ by $L +_{\underline{B}} E$.

Next, we give a translation of CBPV terms to PCFc terms. In Figure 11 we introduce the judgments $\Gamma \vdash_{\mathsf{v}} V \searrow P : A$ and $\Gamma \vdash_{\mathsf{c}} M \searrow Q : \underline{B}$. The first one is read as "the CBPV term V of value type A translates to the PCFc term P in context Γ." The interpretation of the second judgment is similar, but involves terms of computation type. We have that

Theorem 6.1 (CBPV-to-PCFc translation).

(1) If $\Gamma \vdash_v V : A$ *then there is a unique PCFc term* P *such that* $\Gamma \vdash_v V \searrow P : A$, *and* $\langle\!\langle \Gamma \rangle\!\rangle \vdash P : \langle\!\langle A \rangle\!\rangle$.
(2) If $\Gamma \vdash_c M : \underline{B}$ *then there is a unique PCFc term* Q *such that* $\Gamma \vdash_c M \searrow Q : \underline{B}$, *and* $\langle\!\langle \Gamma \rangle\!\rangle \vdash Q : \left\|\underline{B}\right\|^{\bullet}$.

Hence, whenever $\Gamma \vdash_{\mathsf{v}} V \searrow P : A$ we know that this P is unique. We write $\langle\!\langle V \rangle\!\rangle$ for it, so that $\langle\!\langle \Gamma \rangle\!\rangle \vdash \langle\!\langle V \rangle\!\rangle : \langle\!\langle A \rangle\!\rangle$. We also write $\|M\|$ for the unique Q such that $\Gamma \vdash_{\mathsf{c}} M \searrow Q : \underline{B}$. In that case we have $\langle\!\langle \Gamma \rangle\!\rangle \vdash \|M\| : \left\|\underline{B}\right\|^{\bullet}$. From this point onwards we again confuse algebras with their carriers.

The main ideas of this recurrence extraction are: implementing the effect charge. M in CBPV as adding unit cost using the algebra on the computation type $\underline{B}$ of M, and undoing some of the finer operational distinctions that are made in CBPV but not in PCFc. For example, both eager and lazy products in CBPV are interpreted as products in PCFc, and both split and projections in CBPV are interpreted as projections in PCFc. Because of the latter, it may be surprising that we can obtain a bounding theorem that works for the CBV features of CBPV. A precedent for this kind of translation of CBV into a CBN-like setting is the adequacy theorem of [Plotkin and Power 2001] for CBV PCF with algebraic effects: the main requirement in that result is a strong monad with a *strict* strength, and the writer monad $\mathbb{C} \times -$ is such a monad.

6.2 The Bounding Theorem

We now wish to argue that the extraction procedure for CBPV is correct via a logical relation that generalizes the CBV and CBN ones. We define two relations

$$(V : \mathcal{T}_A^{\text{CBPV,closed}}) \lesssim_A^{\text{val}} (E : \mathcal{T}_{\langle\!\langle A \rangle\!\rangle}^{\text{PCFc,closed}}) \qquad (M : \mathcal{T}_{\underline{B}}^{\text{CBPV,closed}}) \lesssim_{\underline{B}}^{\text{c}} (E : \mathcal{T}_{\left\|\underline{B}\right\|}^{\text{PCFc,closed}})$$

by induction on types; the full definitions are shown in Figure 12. $V \lesssim_A^{\text{val}} E$ is read as "the potential of the value $V : A$ is bounded above by E," and $M \lesssim_{\underline{B}}^{\text{c}} E$ is read as "the complexity of the computation

$$\langle\!\langle \mathsf{nat} \rangle\!\rangle \stackrel{\text{def}}{=} \mathsf{nat} \qquad \|FA\| \stackrel{\text{def}}{=} (\mathbb{C} \times \langle\!\langle A \rangle\!\rangle, \alpha_{FA})$$
$$\langle\!\langle A_1 \times A_2 \rangle\!\rangle \stackrel{\text{def}}{=} \langle\!\langle A_1 \rangle\!\rangle \times \langle\!\langle A_2 \rangle\!\rangle \qquad \|A \to \underline{B}\| \stackrel{\text{def}}{=} (\langle\!\langle A \rangle\!\rangle \to \|\underline{B}\|^\bullet, \alpha_{A \to \underline{B}})$$
$$\langle\!\langle U\underline{B} \rangle\!\rangle \stackrel{\text{def}}{=} \|\underline{B}\|^\bullet \qquad \|\underline{B}_1 \,\&\, \underline{B}_2\| \stackrel{\text{def}}{=} (\|\underline{B}_1\|^\bullet \times \|\underline{B}_2\|^\bullet, \alpha_{\underline{B}_1 \& \underline{B}_2})$$

$$c : \mathbb{C}, x : \mathbb{C} \times \langle\!\langle A \rangle\!\rangle \vdash \alpha_{FA}(c, x) \stackrel{\text{def}}{=} \langle c \boxplus \pi_1(x), \pi_2(x) \rangle : \mathbb{C} \times \langle\!\langle A \rangle\!\rangle$$
$$c : \mathbb{C}, f : \langle\!\langle A \rangle\!\rangle \to \|\underline{B}\|^\bullet \vdash \alpha_{A \to \underline{B}}(c, f) \stackrel{\text{def}}{=} \lambda a.\ \alpha_{\underline{B}}(c, f(a)) : \langle\!\langle A \rangle\!\rangle \to \|\underline{B}\|^\bullet$$
$$c : \mathbb{C}, p : \|\underline{B}_1\|^\bullet \times \|\underline{B}_2\|^\bullet \vdash \alpha_{\underline{B}_1 \& \underline{B}_2}(c, p) \stackrel{\text{def}}{=} \left\langle \alpha_{\underline{B}_1}(c, \pi_1(p)), \alpha_{\underline{B}_2}(c, \pi_2(p)) \right\rangle : \|\underline{B}_1\|^\bullet \times \|\underline{B}_2\|^\bullet$$

Note: whenever $\Gamma \vdash E : \|FA\|$, we let $\begin{cases} \Gamma \vdash E_c \stackrel{\text{def}}{=} \pi_1(E) : \mathbb{C} \\ \Gamma \vdash E_p \stackrel{\text{def}}{=} \pi_2(E) : \langle\!\langle A \rangle\!\rangle \end{cases}$

$$\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{v}} x \searrow x : A}\ (\text{var}) \qquad \frac{}{\Gamma \vdash_{\mathsf{v}} \widetilde{n} \searrow \underline{n} : \mathsf{nat}} \qquad \frac{\Gamma, x : U\underline{B} \vdash_{\mathsf{c}} M \searrow P : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{fix}\ x.\ M \searrow \mathsf{fix}\ x.\ P : \underline{B}}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} V_1 \searrow W_1 : A_1 \quad \Gamma \vdash_{\mathsf{v}} V_2 \searrow W_2 : A_2}{\Gamma \vdash_{\mathsf{v}} (V_1, V_2) \searrow \langle W_1, W_2 \rangle : A_1 \times A_2} \qquad \frac{\Gamma \vdash_{\mathsf{v}} V \searrow P : A_1 \times A_2 \quad \Gamma, x_1 : A_1, x_2 : A_2 \vdash_{\mathsf{c}} N \searrow Q : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{split}(V; (x_1, x_2).\ N) \searrow Q[\pi_1(P)/x_1, \pi_2(P)/x_2] : \underline{B}}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} V \searrow P : A}{\Gamma \vdash_{\mathsf{c}} \mathsf{return}\ V \searrow \langle \mathbf{0}, P \rangle : FA} \qquad \frac{\Gamma \vdash_{\mathsf{c}} M \searrow Q : FA \quad \Gamma, x : A \vdash_{\mathsf{c}} N \searrow R : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{bind}\ x \Leftarrow M\ \mathsf{in}\ N \searrow Q_c +_{\underline{B}} R[Q_p/x] : \underline{B}}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{c}} M \searrow P : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \lambda x.\ M \searrow \lambda x.\ P : A \to \underline{B}} \qquad \frac{\Gamma \vdash_{\mathsf{c}} M \searrow P : A \to \underline{B} \quad \Gamma \vdash_{\mathsf{v}} N \searrow Q : A}{\Gamma \vdash_{\mathsf{c}} M\,N \searrow P\,Q : \underline{B}}$$

$$\frac{\Gamma \vdash_{\mathsf{c}} M \searrow P : \underline{B}_1 \quad \Gamma \vdash_{\mathsf{c}} N \searrow Q : \underline{B}_2}{\Gamma \vdash_{\mathsf{c}} \langle M, N \rangle \searrow \langle P, Q \rangle : \underline{B}_1 \,\&\, \underline{B}_2} \qquad \frac{\Gamma \vdash_{\mathsf{c}} M \searrow P : \underline{B}_1 \,\&\, \underline{B}_2}{\Gamma \vdash_{\mathsf{c}} \pi_i(M) \searrow \pi_i(P) : \underline{B}_i}$$

$$\frac{\Gamma \vdash_{\mathsf{c}} M \searrow P : \underline{B}}{\Gamma \vdash_{\mathsf{v}} \mathsf{thunk}\ M \searrow P : U\underline{B}} \qquad \frac{\Gamma \vdash_{\mathsf{v}} V \searrow P : U\underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{force}\ V \searrow P : \underline{B}}$$

$$\frac{\Gamma \vdash_{\mathsf{c}} M \searrow P : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{charge}.\ M \searrow \mathbf{1} +_{\underline{B}} P : \underline{B}} \qquad \frac{\Gamma \vdash_{\mathsf{v}} V \searrow W : \mathsf{nat} \quad \Gamma \vdash_{\mathsf{c}} M \searrow P : \underline{B} \quad \Gamma \vdash_{\mathsf{c}} N \searrow Q : \underline{B}}{\Gamma \vdash_{\mathsf{v}} \mathsf{ifz}(V;\ M;\ N) \searrow \mathsf{if}\ W\ \mathsf{then}\ P\ \mathsf{else}\ Q : \underline{B}}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} V \searrow M : \mathsf{nat} \quad \Gamma \vdash_{\mathsf{v}} W \searrow N : \mathsf{nat} \quad \Gamma, z : \mathsf{nat} \vdash_{\mathsf{c}} P \searrow Q : \underline{B} \quad \mathbf{op}^+ \in \{+, *\}}{\Gamma \vdash_{\mathsf{c}} \mathsf{calc}\ z \Leftarrow V\ \mathbf{op}^+\ W\ \mathsf{in}\ P \searrow Q[M\ \mathbf{op}^+\ N/z] : \underline{B}}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} V \searrow M : \mathsf{nat} \quad \Gamma, z : \mathsf{nat} \vdash_{\mathsf{c}} P \searrow Q : \underline{B} \quad \mathbf{op}^- \in \{-, \div\}}{\Gamma \vdash_{\mathsf{c}} \mathsf{calc}\ z \Leftarrow V\ \mathbf{op}^-\ \widetilde{n}\ \mathsf{in}\ P \searrow Q[M\ \mathbf{op}^-\ \widetilde{n}/z] : \underline{B}}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} V \searrow M : \mathsf{nat} \quad \Gamma \vdash_{\mathsf{v}} W \searrow N : \mathsf{nat} \quad \Gamma, z : \mathsf{nat} \vdash_{\mathsf{c}} P \searrow Q : \underline{B} \quad \mathbf{op}^- \in \{-, \div\} \quad N \neq \widetilde{n}}{\Gamma \vdash_{\mathsf{c}} \mathsf{calc}\ z \Leftarrow V\ \mathbf{op}^-\ W\ \mathsf{in}\ P \searrow Q[M/z] : \underline{B}}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} V \searrow M : \mathsf{nat} \quad \Gamma \vdash_{\mathsf{v}} W \searrow N : \mathsf{nat} \quad \Gamma, z : \mathsf{nat} \vdash_{\mathsf{c}} P \searrow Q : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{calc}\ z \Leftarrow V\ \mathsf{mod}\ W\ \mathsf{in}\ P \searrow Q[N - \underline{1}/z] : \underline{B}}$$

Fig. 11. Recurrence extraction for CBPV

$$\widetilde{n} \lesssim^{\mathrm{val}}_{\mathrm{nat}} E \stackrel{\mathrm{def}}{\equiv} \underline{n} \leqslant E \qquad M \lesssim^{\mathrm{c}}_{FA} E \stackrel{\mathrm{def}}{\equiv} E_c\downarrow \implies \exists n, V.\ \begin{cases} M \Downarrow^n \text{return } V \\ \widehat{n} \leqslant E_c \\ V \lesssim^{\mathrm{val}}_{A} E_p \end{cases}$$

$$(V_1, V_2) \lesssim^{\mathrm{val}}_{A_1 \times A_2} E \stackrel{\mathrm{def}}{\equiv} \begin{cases} V_1 \lesssim^{\mathrm{val}}_{A_1} \pi_1(E) \\ V_2 \lesssim^{\mathrm{val}}_{A_2} \pi_2(E) \end{cases} \qquad M \lesssim^{\mathrm{c}}_{A \to \underline{B}} E \stackrel{\mathrm{def}}{\equiv} \forall (N \lesssim^{\mathrm{val}}_{A} X).\ M\,N \lesssim^{\mathrm{c}}_{\underline{B}} E\,X$$

$$\text{thunk } M \lesssim^{\mathrm{val}}_{U\underline{B}} E \stackrel{\mathrm{def}}{\equiv} M \lesssim^{\mathrm{c}}_{\underline{B}} E \qquad M \lesssim^{\mathrm{c}}_{\underline{B}_1 \& \underline{B}_2} E \stackrel{\mathrm{def}}{\equiv} \begin{cases} \pi_1(M) \lesssim^{\mathrm{c}}_{\underline{B}_1} \pi_1(E) \\ \pi_2(M) \lesssim^{\mathrm{c}}_{\underline{B}_2} \pi_2(E) \end{cases}$$

Fig. 12. Bounding relations for CBPV

$M : \underline{B}$ is bounded above by E." The value relation is similar to the CBV value bounding relation, with the clause for U-types corresponding to the switch from value bounding to expression bounding in function types. The computation relation is similar to the CBN bounding relation, with the clause for F-types corresponding to the CBV expression relation.

We then establish that this relation holds at the diagonal, i.e. that

Theorem 6.2 (Bounding theorem).
(1) If $\cdot \vdash V : A$ *then* $V \lesssim^{\mathrm{val}}_{A} \langle\!\langle V \rangle\!\rangle$.
(2) If $\cdot \vdash M : \underline{B}$ *then* $M \lesssim^{\mathrm{c}}_{\underline{B}} \|M\|$.

While technical, the proof of this theorem is unsurprising, and may be found in an extended version of this paper. Amongst other things, it requires an auxiliary notion of *costless weak head reduction*. It is denoted by $M \longrightarrow_0 N$, and encodes all weak head reductions, plus a 'commuting conversion' that makes negative eliminators commute with the unit cost construct, e.g. $\pi_i(\mathsf{charge}.\, M) \longrightarrow_0 \mathsf{charge}.\, \pi_i(M)$. The main lemmas used are:

Lemma 6.3 (Algebra Monotonicity). $E \leqslant E' : \mathbb{C} \implies E +_{\underline{B}} M \leqslant E' +_{\underline{B}} M$

Lemma 6.4 (Bound weakening).
(1) $V \lesssim^{\mathrm{val}}_{A} E \ \wedge\ E \leqslant E' : \langle\!\langle A \rangle\!\rangle \implies V \lesssim^{\mathrm{val}}_{A} E'$
(2) $M \lesssim^{\mathrm{c}}_{\underline{B}} E \ \wedge\ E \leqslant E' : \left\|\underline{B}\right\| \implies M \lesssim^{\mathrm{c}}_{\underline{B}} E'$

Lemma 6.5 (Head expansion/reduction).
(1) $M' \lesssim^{\mathrm{c}}_{\underline{B}} E \ \wedge\ M \longrightarrow_0 M' \implies M \lesssim^{\mathrm{c}}_{\underline{B}} E$
(2) $M \lesssim^{\mathrm{c}}_{\underline{B}} E \ \wedge\ M \longrightarrow_0 M' \implies M' \lesssim^{\mathrm{c}}_{\underline{B}} E$

Lemma 6.6 (Unit charge). $M \lesssim^{\mathrm{c}}_{\underline{B}} E \implies \mathit{charge}.\, M \lesssim^{\mathrm{c}}_{\underline{B}} \mathbf{1} +_{\underline{B}} E$

Theorem 6.7 (Compactness). $\mathcal{E}[\mathit{fix}\, x.\ M]\downarrow \implies \exists n \geq 0.\ \mathcal{E}[\mathit{fix}_n\, x.\ M]\downarrow$

Lemma 6.8 (Bounds for Recursion).
(1) (Infinity) $E\uparrow \implies M \lesssim^{\mathrm{c}}_{\underline{B}} E$.
(2) (Infinity-Algebra) $L\uparrow \implies M \lesssim^{\mathrm{c}}_{\underline{B}} L +_{\underline{B}} E$.
(3) (Fixed Point Induction) $\left(\forall n \geq 0.\ M \lesssim^{\mathrm{c}}_{\underline{B}} \mathit{fix}_n\, x.\ E\right) \implies M \lesssim^{\mathrm{c}}_{\underline{B}} \mathit{fix}\, x.\ E$

Compactness is known to hold for PCF. It can be obtained through a standard denotational semantics—see e.g. Pitts [1994, §10.9]—or through syntactic methods—see [Pitts 1997, §4].

A (PCF type, CBN)	$A^\dagger$ (CBPV+ computation type)
nat	$F(\text{nat})$
$A_1 \times A_2$	$A^\dagger \,\&\, B^\dagger$
$A \to B$	$U(A^\dagger) \to B^\dagger$

$x_1 : A_1, \ldots, x_n : A_n \vdash M : A$	$x_1 : U(A_1^\dagger), \ldots, x_n : U(A_n^\dagger) \vdash_{\mathsf{c}} M^\dagger : A^\dagger$
x	force x
$\underline{n}$	return $\widetilde{n}$
$M \,\mathbf{op}\, N$	bind $m \Leftarrow M^\dagger$ in bind $n \Leftarrow N^\dagger$ in calc $v \Leftarrow m \,\mathbf{op}\, n$ in return v
if N then P else Q	bind $n \Leftarrow N^\dagger$ in ifz$(n;\ P^\dagger;\ Q^\dagger)$
$\langle M, N\rangle$	$\langle \mathsf{charge}.\, M^\dagger, \mathsf{charge}.\, N^\dagger\rangle$
$\pi_i(M)$	$\pi_i \left(M^\dagger\right)$
$\lambda x.\, M$	$\lambda x.\, \left(\mathsf{charge}.\, M^\dagger\right)$
MN	$M^\dagger(\text{thunk}\, N^\dagger)$
fix $x.\, M$	fix $x.\, (\mathsf{charge}.\, M^\dagger)$

Fig. 13. Call-by-name PCF to CBPV translation

7 EMBEDDING PCF INTO CBPV

The only piece of the puzzle that remains is to show how both CBN and CBV PCF can be embedded in CBPV in a cost-preserving way. At a high level, we follow [Levy 2003, §2.7] in defining a term translation, but insert charge expressions wherever the operational semantics of CBV or CBN PCF incurs a cost. We then prove that this translation preserves and reflects the operational semantics, including the cost.

7.1 CBN

In order to embed CBN into CBPV, we must recall that

(1) the only *observable* type of CBN PCF is nat, and
(2) the variables of CBN terms represent *thunks*, as we may substitute them with arbitrary terms.

These two facts directly lead us to a translation of any CBN type A to a CBPV *computation* type $A^\dagger$. As nat is observable, it is mapped to $F\,(\text{nat})$ (it returns a value). Products are compositionally mapped to *negative* products. Finally, as variables stand for thunks, we must thunk the domain when translating the function type.

The translation of both types and terms is defined in Figure 13. Notice that computation steps where explicit evaluation to canonical form is necessary—e.g. for testing whether a number is zero—are punctuated by the appearance of the bind $x \Leftarrow (-)$ in $(-)$ construct. Furthermore, notice that a variable represents a thunk, so it must be forced. Finally, notice the appearance of charge. $(-)$ whenever the operational semantics of PCF would impose a unit charge.

Extending $(-)^\dagger$ to contexts pointwise, we have that

Proposition 7.1. *If* $\Gamma \vdash M : A$ *in call-by-name PCF then* $\Gamma^\dagger \vdash_c M^\dagger : A^\dagger$.

To prove that the operational semantics are preserved and reflected under this translation we follow the technique of [Levy 2006, §7]. We inductively define a relation $\rightmapsto_{\text{cbn}}$ between terms of PCF and terms of CBPV. The relation includes one rule for each line of the table of Figure 13, e.g.

$$\frac{M \rightmapsto_{\text{cbn}} M' \quad N \rightmapsto_{\text{cbn}} N'}{M\,N \rightmapsto_{\text{cbn}} M'(\text{thunk}\, N')}$$

Moreover, the definition also includes the rule

$$\frac{M \rightrightarrows_{\mathrm{cbn}} M'}{M \rightrightarrows_{\mathrm{cbn}} \mathsf{force}\ \mathsf{thunk}\ M'}$$

which—for technical reasons—inserts enough occurrences of force thunk $(-)$. Evidently,

Proposition 7.2. $M \rightrightarrows_{\mathrm{cbn}} M^{\dagger}$

We can then show that

Theorem 7.3 (CBN bisimulation).
(1) If $M \downarrow^n V$ and $M \rightrightarrows_{\mathrm{cbn}} M'$, then $M' \Downarrow^n T'$ for some T' with $V \rightrightarrows_{\mathrm{cbn}} T'$.
(2) If $M' \Downarrow^n T'$ and $M \rightrightarrows_{\mathrm{cbn}} M'$, then $M \downarrow^n V$ for some V with $V \rightrightarrows_{\mathrm{cbn}} T'$.

Proof. We first show that $M \rightrightarrows_{\mathrm{cbn}} M'$ and $N \rightrightarrows_{\mathrm{cbn}} N'$ implies $M[N/x] \rightrightarrows_{\mathrm{cbn}} M'[\mathsf{thunk}\ N'/x]$, by an easy induction on $M \rightrightarrows_{\mathrm{cbn}} M'$. We then show the main claims. In the first case, we proceed by induction on $M \downarrow^n V$, followed by local inductions on $M \rightrightarrows_{\mathrm{cbn}} M'$. In the second case, we proceed by induction on $M' \Downarrow^n T'$, followed by local inductions on $M \rightrightarrows_{\mathrm{cbn}} M'$. □

Corollary 7.4 (CBN preservation).
(1) If $M \downarrow^n V$ then $M^{\dagger} \Downarrow^n T$ with $V \rightrightarrows_{\mathrm{cbn}} T$.
(2) If $M^{\dagger} \Downarrow^n T$ then $M \Downarrow^n V$ with $V \rightrightarrows_{\mathrm{cbn}} T$.

7.2 CBV

In order to embed CBV into CBPV, we must recall that

(1) in CBV, termination at every type is observable, and returns a *value*
(2) the variables of CBV terms represent *values*

In short: *all terms of CBV are returners*. Thus, we aim to translate each type CBV type A to a CBPV value type A^*, and each term $x : A \vdash M : B$ to a term $x : A^* \vdash_{\mathrm{c}} M^* : F(B^*)$ of CBPV.

Translating types is then fairly evident: nat is translated to itself, and products are compositionally mapped to *positive* products. Finally, $A \to B$ is mapped to $A^* \to F(B^*)$, which is then thunked in order to become a value type.

On the level of terms there are two translations: one takes CBV PCF terms to CBPV terms of $F(-)$ type, and the other one takes CBV PCF values to CBPV values. They are both defined in Figure 14. Notice once again that computation steps where explicit evaluation to canonical form is necessary—e.g. for function application, or construction of pairs—necessitate a use of the bind $x \Leftarrow (-)$ in $(-)$ construct, which forces evaluation. Furthermore, notice that a variable represents a value, so it must be returned. Finally, notice the appearance of charge. $(-)$ whenever the operational semantics of PCF would impose a unit charge.

Extending $(-)^*$ to contexts pointwise, we have that

Proposition 7.5.
(1) If $\Gamma \vdash V : A$ is a PCF value, then $\Gamma^* \vdash_v V^{\mathrm{val}} : A^*$.
(2) If $\Gamma \vdash M : A$ in call-by-value PCF, then $\Gamma^* \vdash_c M^* : F(A^*)$.

It is even easier than CBN to establish that the operational semantics of CBV are preserved and reflected under this translation.

Theorem 7.6 (CBV preservation).
(1) If $M \downarrow^n V$ then $M^* \Downarrow^n V^{\mathrm{val}}$.
(2) If $M^* \Downarrow^n$ return W then there exists V such that $M \downarrow^n V$ and $W \equiv_{\alpha} V^{\mathrm{val}}$.

A (PCF type, CBV)	A^* (CBPV+ value type)
nat	nat
$A_1 \times A_2$	$A^* \times B^*$
$A_1 \to A_2$	$U\,(A^* \to F(B^*))$

$x_1 : A_1, \ldots, x_n : A_n \vdash M : A$	$x_1 : A_1^*, \ldots, x_n : A_n^* \vdash_{\mathsf{c}} M^* : F(A^*)$
x	return x
$\underline{n}$	return $\widetilde{n}$
M **op** N	bind $m \Leftarrow M^*$ in bind $n \Leftarrow N^*$ in calc $v \Leftarrow m$ **op** n in return v
if N then P else Q	bind $n \Leftarrow N^*$ in ifz$(n;\ P^*;\ Q^*)$
$\langle M, N\rangle$	bind $x \Leftarrow M^*$ in bind $y \Leftarrow N^*$ in return (x, y)
$\pi_i(M)$	bind $p \Leftarrow M^*$ in charge. split$(p; (x_1, x_2).$ return $x_i)$
$\lambda x.\ M$	return thunk $\lambda x.\ M^*$
MN	bind $f \Leftarrow M^*$ in bind $x \Leftarrow N^*$ in charge. (force $f)x$
rec $f(x) = M$	return thunk (fix $f.\ \lambda x.\ M^*$)

$x_1 : A_1, \ldots, x_n : A_n \vdash V : A$	$x_1 : A_1^*, \ldots, x_n : A_n^* \vdash_{\mathsf{v}} V^{\mathrm{val}} : A^*$
x	x
$\underline{n}$	$\widetilde{n}$
$\lambda x.\ M$	thunk $\lambda x.\ M^*$
rec $f(x) = M$	thunk (fix $f.\ \lambda x.\ M^*$)

Fig. 14. Call-by-value PCF to CBPV translation

Proof. First, we need to establish two lemmas:

- $V^* \equiv_\alpha$ return V^{val} for a call-by-value PCF value V
- $(M[V/x])^* \equiv_\alpha M^*[V^{\mathrm{val}}/x]$

These follow straightforwardly by induction on values and terms respectively. The first claim then follows by induction on $M \downarrow^n V$, and the second by induction on $M^* \Downarrow^n$ return W. □

7.3 Completing the Circle

Combining the bounding theorem of Section 6 and these results, we can prove the main theorems of §4. For Theorem 4.2 we will make use of head reduction, viz. Lemma 6.5. We then define

$$M \sqsubseteq^{\mathrm{val}}_{\approx A} E \stackrel{\mathrm{def}}{\equiv} V^{\mathrm{val}} \lesssim^{\mathrm{val}}_{A^*} E \qquad\qquad M \sqsubseteq_{\approx A} E \stackrel{\mathrm{def}}{\equiv} M^* \lesssim^{\mathrm{c}}_{F(A^*)} E$$

and calculate that the desired conditions hold. We then do the same for Theorem 4.5 and

$$M \sqsubseteq_{\approx A} E \quad \stackrel{\mathrm{def}}{\equiv} \quad M^\dagger \lesssim^{\mathrm{c}}_{A^\dagger} E$$

8 DENOTATIONAL SEMANTICS FOR PCF WITH COSTS

This section is concerned with the development of a denotational semantics for PCFc. Having completed the step of recurrence extraction, we would now like to interpret our *syntactic recurrences*—expressed in the formal language of PCF—into *semantic recurrences*. These should be as close as possible to good old-fashioned recurrences found in textbooks on algorithms.

Because of the presence of recursion, this endeavour requires a modicum of *domain theory*. Standard references to domain theory include [Abramsky and Jung 1994; Stoltenberg-Hansen et al. 1994], whereas references on the denotational semantics of PCF in domains may be found in §2. We assume knowledge of the standard definition of *complete partial order (cpo)*, i.e. a partial order

$(D, \sqsubseteq)$ with least upper bounds of directed subsets, and of Scott-continuous functions. We also recall the notion of an ω-chain $(x_i)_{i\in\omega}$ in a cpo, i.e. an increasing sequence of elements $x_1 \sqsubseteq x_2 \sqsubseteq \dots$.

8.1 Sized Domains

The relation usually modelled by the denotational semantics of PCF is that of *equality*: the semantics of PCF terms is *sound*, in that it is stable under evaluation. However, our main notion here is that of the axiomatic size order, and in order to model it we introduce the following kind of domain.

Definition 8.1. A *sized domain* (with joins) $(D, \sqsubseteq, \preceq, \vee, \bot, \mathbf{0}, \mathbf{1})$ consists of

- a set D,
- a partial order $\sqsubseteq$, which we call the *information order on* D, and
- a preorder $\preceq$, which we call the *size order on* D, and
- an operation $\vee : D \times D \to D$

such that

(1) $(D, \sqsubseteq)$ is a cpo with least element $\bot$,
(2) $(D, \preceq)$ has chosen binary *joins* (maximums/least upper bounds) given by $\vee : D \times D \to D$ and least element $\mathbf{0}$,
(3) $\vee : D \times D \to D$ is continuous with respect to $(D, \sqsubseteq)$,
(4) $x \sqsubseteq y$ implies $y \preceq x$, and
(5) if $(x_i)_{i\in\omega}$ is a ω-chain in D and $\forall i \in \omega.\ z \preceq x_i$, then $z \preceq \bigsqcup_{i\in\omega} x_i$.

We immediately know that

Proposition 8.2. *$\bot$ is the greatest element (up to iso) with respect to $(D, \preceq)$.*

Proof. For any $x \in D$ we have that $\bot \sqsubseteq x$, and hence $x \preceq \bot$ by axiom (4). Thus, $\bot$ is a greatest element in $(D, \preceq)$. However, as $(D, \preceq)$ is not a partial order, it is only unique up to isomorphism. □

The most interesting axioms in this definition are (4) and (5). The first one formalizes the idea that *a more defined bound is a better bound.* Consider the function $f = \{1 \mapsto 2, x \mapsto \bot\}$ on the flat domain of natural numbers. When interpreted as a recurrence, f gives us an upper bound for the input 1, but no information about any other input. In fact, Proposition 8.2 tells us that $\bot$ is the greatest element in the size order: it represents the *infinite value*. Thus, the recurrence f maps all elements other than 1 to the trivial upper bound of infinity. The recurrence $g = \{1 \mapsto 2, 2 \mapsto 2, x \mapsto \bot\}$ is more well-defined than f, as it is also defined (i.e. not $\bot$) at input 2: we have $f \sqsubseteq g$. As $\bot$ represents infinity, this is indeed a *tighter* bound, and axiom (4) stipulates that we should be able to treat it as such: we should have $g \preceq f$. Axiom (4) is closely related to rule (rat) of the axiomatic size order of Figure 6, and is indeed used to verify its soundness.

Axiom (5) expresses the idea that *a recursively defined recurrence is an upper bound if all its approximants are.* Recall that fixed points in PCF are interpreted as the least upper bound of a chain $\left(f^i(\bot)\right)_{i\in\omega}$ that is increasing in the information order $\sqsubseteq$. Intuitively, each element of this chain is more well-defined than its predecessor. Axiom (5) requires that if every extra step of definition $f^i(\bot)$ of a recurrence is indeed an upper bound for an element, then so must be the limit $\bigsqcup_{i\in\omega} x_i$. Axiom (5) is very close to the rule (cpind), and is used to verify its soundness.

We define the category **SizeDom** to consist of sized domains and Scott-continuous functions. The morphisms need not preserve any aspect of the size order; for example, we do not requiring *size monotonicity* in general, because only the PCFc monotone contexts C are required to be interpreted as size-monotonic functions. It is also easy to see that this category is cartesian closed, as it inherits

products and exponentials from the underlying category **Cpo** of complete partial orders and Scott-continuous functions: we only need to define the size order and the chosen joins—which are in both cases defined pointwise—and show that the axioms of Definition 8.1 follow.

8.2 Interpreting PCF with Costs into Sized Domains

Defining a semantic interpretation of PCFc into sized domains is reasonably straightforward. To each type A of PCF we associate a sized domain $[\![A]\!] = (D_A, \sqsubseteq_A, \preceq_A, \vee_A, \bot_A, 0_A, 1_A)$. We interpret both the natural numbers nat and the type $\mathbb{C}$ of costs by the *flat sized domain of natural numbers*, which is defined to be $\mathbb{N}_\bot \stackrel{\text{def}}{=} (\mathbb{N} \cup \{\bot\}, \sqsubseteq, \preceq, \vee, \bot, 0, 1)$ where $\bot \notin \mathbb{N}$. Its underlying set consists of the natural numbers augmented with a 'fresh' element $\bot$, which simultaneously models non-termination and infinity. Writing $\leq_{\mathbb{N}}$, for the usual order on natural numbers, we define

$$x \sqsubseteq y \stackrel{\text{def}}{\equiv} x = \bot \text{ or } x = y, \qquad x \preceq y \stackrel{\text{def}}{\equiv} y = \bot \text{ or } x \leq_{\mathbb{N}} y, \qquad x \vee y \stackrel{\text{def}}{=} \begin{cases} \bot \text{ if } x = \bot \text{ or } y = \bot \\ \max_{\mathbb{N}} \{x, y\} \text{ otherwise} \end{cases}$$

It is easy to see that the requisite axioms are satisfied. The rest of the types are interpreted by

$$[\![A_1 \times A_2]\!] \stackrel{\text{def}}{=} [\![A_1]\!] \times [\![A_2]\!] \qquad [\![A \to B]\!] \stackrel{\text{def}}{=} [\![A]\!] \to [\![B]\!]]$$

where $[X \to Y]$ denotes the set of Scott-continuous functions from a cpo $(X, \sqsubseteq_X)$ to a cpo $(Y, \sqsubseteq_Y)$, and where both the information and size orders are given pointwise.

To each judgment $\Gamma \vdash M : A$, where $\Gamma \equiv x_1 : A_1, \ldots, x_n : A_n$, we associate a function $[\![\Gamma \vdash M : A]\!] : [\![\Gamma]\!] \to [\![A]\!]$ where $[\![\Gamma]\!] \stackrel{\text{def}}{=} [\![A_1]\!] \times \cdots \times [\![A_n]\!]$. This definition is mostly standard—see e.g. [Streicher 2006, §3.2]. However, there are two unexpected elements, which arise because the PCFc monotone contexts C must be interpreted as size-monotone functions in order to interpret the syntactic size order axiom (ctx), and because we interpret nat with the usual $0 \leq 1 \leq 2 \ldots \leq \infty$ size order to facilitate reasoning about semantic recurrences. First, because if C then P else Q is a monotone context, we must "monotonize" the conditional. We do this by sending ∞ to ∞, and using the chosen joins to make the interpretation for ≥ 1 dominate[5] the answer for 0:

$$[\![\Gamma \vdash \text{if } N \text{ then } P \text{ else } Q : A]\!](\vec{d}) \stackrel{\text{def}}{=} \begin{cases} \bot_A & \text{if } [\![\Gamma \vdash N : \text{nat}]\!](\vec{d}) = \bot \\ [\![P]\!](\vec{d}) & \text{if } [\![\Gamma \vdash N : \text{nat}]\!](\vec{d}) = 0 \\ [\![P]\!](\vec{d}) \vee_A [\![Q]\!](\vec{d}) & \text{otherwise} \end{cases}$$

Second, because the operands of an arithmetic operation are monotone contexts, we must interpret the operations by a monotone upper bound of their actual numerical value. While addition and multiplication are already monotone, for subtraction, division and modulo, we use the following definitions, where $\mathbf{op}^- \in \{-, \div\}$:[6]

$$[\![\Gamma \vdash M \, \mathbf{op}^- \, \underline{n} : \text{nat}]\!](\vec{d}) \stackrel{\text{def}}{=} [\![\Gamma \vdash M : \text{nat}]\!](\vec{d}) \, \mathbf{op}^- \, n$$

$$[\![\Gamma \vdash M \, \mathbf{op}^- \, N : \text{nat}]\!](\vec{d}) \stackrel{\text{def}}{=} [\![\Gamma \vdash M : \text{nat}]\!](\vec{d})$$

$$[\![\Gamma \vdash M \bmod N : \text{nat}]\!](\vec{d}) \stackrel{\text{def}}{=} [\![\Gamma \vdash N : \text{nat}]\!](\vec{d}) - 1$$

[5] The standard definition of the conditional along with $\underline{0} \leqslant \underline{1}$ and (ctx) would imply $[\![P]\!] = [\![\text{if } \underline{0} \text{ then } P \text{ else } Q]\!] \leqslant [\![\text{if } \underline{1} \text{ then } P \text{ else } Q]\!] = [\![Q]\!]$ for arbitrary terms P and Q.

[6] We could in fact remove $\{-, \div\}$ with non-numerals and mod from PCFc, because they are never used as the target of the recurrence extraction in Section 6. In general, there is a choice about whether to perform this monotonization in recurrence extraction or in the semantic interpretation. The above recurrence extraction from CBPV to PCFc already does monotonization, eliminating all mods and $\{-, \div\}$ with non-numerals. However, we prefer to leave these operations in PCFc and interpret them similarly here to also support the other style of recurrence extraction into PCFc.

$$\begin{aligned}\text{rec exp}(n) &= \text{if } n \text{ then } 1 \text{ else let } z \leftarrow \text{exp}(n/2), y \leftarrow \text{if } n \bmod 2 \text{ then } 1 \text{ else } 2 \text{ in } z * z * y \\ \|\text{exp}\| &= \big\langle 0, \text{fix exp. } \lambda n.\text{if } n \text{ then } \langle \mathbf{0}, \underline{1}\rangle \\ &\qquad \text{else } \langle \widehat{3} \boxplus (\text{exp}(n/\underline{2}))_c, \\ &\qquad\quad (\text{exp}(n/\underline{2}))_p * (\text{exp}(n/\underline{2}))_p * (\text{if } \underline{1} \text{ then } \underline{1} \text{ else } \underline{2})\rangle\big\rangle\end{aligned}$$

Fig. 15. Divide-and-conquer exponentiation and its extracted recurrence

The 'new' clauses interpret $[\![\mathbf{0}]\!](\vec{d}) \stackrel{\text{def}}{=} 0$, $[\![\mathbf{1}]\!](\vec{d}) \stackrel{\text{def}}{=} 1$, and

$$[\![\Gamma \vdash M \boxplus N : \mathbb{C}]\!](\vec{d}) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } [\![M]\!](\vec{d}) = \bot \text{ or } [\![N]\!](\vec{d}) = \bot \\ [\![M]\!](\vec{d}) +_{\mathbb{N}} [\![N]\!](\vec{d}) & \text{otherwise} \end{cases}$$

We have that

Theorem 8.3 (Soundness). $\Gamma \vdash M \leqslant N : A \implies \forall \vec{d} \in [\![\Gamma]\!]\,.\ [\![M]\!](\vec{d}) \preceq_A [\![N]\!](\vec{d})$

Moreover, we can use standard logical relations techniques to show that this interpretation is *adequate* in an appropriately relaxed sense.

Theorem 8.4 (Adequacy).

(1) If $M : \mathsf{nat}$, then $[\![M]\!] = m$ implies $M \downarrow \underline{k}$ for some $k \leq_{\mathbb{N}} m$.

(2) If $M : \mathbb{C}$, then $[\![M]\!] = m$ implies $M \downarrow \widehat{k}$ for some $k \leq_{\mathbb{N}} m$.

We anticipate the existence of other models of PCFc in sized domains, which we expect to obtain by varying the orders on the interpretation of nat, or even $\times$ and $\to$. For example, there should be a more precise discrete model, where all size orders are interpreted by equality, all cost bounds are exact, and the potentials include the programs themselves.

9 EXAMPLE: EXPONENTIATION

Next, we show how the above technique works on an example of a non-structurally-recursive function, which is not supported by Danner et al. [2015]. Consider the recursive specification for "exponentiation-by-squaring":

$$2^n = \begin{cases} (2^{n/2})^2 & \text{if } n \text{ is even} \\ (2^{(n-1)/2})^2 * 2 & \text{otherwise} \end{cases}$$

We implement this in CBV PCF in Figure 15, using let expressions as syntactic sugar for β-expansions and recalling that we are performing floor division so that $n/2 = (n-1)/2$ when n is odd. We extract a recurrence in PCFc using the rules of Figure 7 and show the result in the same figure. In order to keep the term from becoming unwieldy, we assume a few additional size order axioms on PCFc terms beyond those of Figure 6, all of which are true in the semantics of sized domains (we write equality for the size order $\leqslant$ in both directions):

- the opposite direction of rules (assoc) and (zero), so that $(\boxplus, \mathbf{0})$ forms a monoid,
- the opposite direction of the β rules, as well as η equality rules for $\times$ and $\to$ types,
- the 'commuting conversion' $\pi_1(\text{if } N \text{ then } M_1 \text{ else } M_2) = \text{if } N \text{ then } \pi_1(M_1) \text{ else } \pi_1(M_2)$, and
- if $\underline{k}$ then N else $N = N$

It is $\|\mathsf{exp}\|_p$ that is the desired recurrence, and we observe that if $T(n) = \pi_1(\|\mathsf{exp}\|_p\, n)$, then

$$T(0) = 0 \qquad\qquad T(n) = 3 + T(n/2)$$

which is the expected recurrence from an informal analysis.

When n is even, the potential is an over-approximation of the actual value: we get $2{*}((\mathsf{exp}(n/2))_p)^2$ whether n is even or odd because the recurrence extraction sends n mod 2 to 1 in order to be monotone. We leave a more precise treatment of non-monotone operations to future work.

10 INDUCTIVE TYPES

Our previous work [Danner et al. 2015] only permitted call-by-value functional programs with strictly positive inductive datatypes and structural recursion. The present paper extends these techniques to general non-structural recursion, which often leads to significantly more concise and efficient code. It is thus interesting to examine the combination of this presentation with the datatypes supported by the previous techniques of Danner et al. [2015]. We believe that this combination is achievable. In this section we illustrate it by a careful treatment of the case of lists in CBV PCF, and we briefly discuss the general case. In future work we plan to extend our techniques to *general recursive types*, which subsume lazy/CBN PCF lists as well as other coinductive types, though we expect that defining the bounding relation will be somewhat more challenging.

Figure 16 introduces the necessary additions to the syntax of CBV PCF and CBPV, and also extends the cost-preserving embedding, recurrence extraction and bounding relation to the new constructs. Intuitively, we choose here to represent a list by its length. This is useful for analyzing many algorithms whose running time does not depend on the list elements; if the cost does depend on list elements, one should instead represent a list by its length and maximum element, or some other more precise information. We could perform this abstraction of lists as lengths in the semantics, as in Danner et al. [2015], but this would introduce quite a bit of notational overhead. For the sake of simplicity, we instead translate a list to its length in the recurrence extraction phase, defining the type of potentials of lists to be nat. This way we do not need to add lists to PCFc.

The only essentially non-trivial clause is the recurrence extraction for case $(V; M_{\mathsf{nil}}; (x, xs).\ M_{\mathsf{cons}})$. Intuitively, this should map to a PCFc zero test if W then P else Q, where V, M_{nil}, and M_{cons} extract to W, P, and Q respectively. However, Q has two free variables, x and xs, which must be substituted for. These correspond to the potentials of the head and tail of V. The answer for xs is immediate, as its potential is bounded above by $W - \underline{1}$. As we have taken the potential of a list to be its length, we do not have any information about the size of the list elements. Hence, we choose the potential of x to be ∞ (fix $x.\ x$). This gives an upper bound in all cases, and is useful when the cost of the function to be analyzed does not depend on the size of the list elements: in that case the ∞ substituted for x will drop out of the recurrence at some point, so we will still obtain a finite bound. To complete the development, we extend the proof of the bounding theorem by adding a straightforward case for each new term construct. To do so we need the size order axioms $M \leqslant (M + \underline{1}) - \underline{1}$, $\underline{0} \leqslant \underline{1}$ and fix $x.\ x \leqslant \mathcal{E}[\text{fix } x.\ x]$ for eliminative contexts $\mathcal{E}$, all of which are valid in the standard semantics.

Though we have not written out all of the details, we expect that we can extend the above to allow potentials other than list length (e.g. length and maximum element), and to all strictly positive inductive types. Briefly, we need to add the corresponding inductive types to PCFc, add new size order axioms, extract case expressions as case expressions, and then modify the notion of sized domains to provide an interpretation of lists and other inductive types, as in Danner et al. [2015]. By doing so, the syntactic recurrence becomes a cost-annotated version of the original CBPV program that preserves a maximal amount of information about the inductive type values in the original program. Different sized domains can then be used to give different abstractions of size for inductive types, e.g. length for lists, height for (labelled) trees, or more complicated metrics,

CBV PCF

Types $A ::= \cdots \mid \mathsf{list}\ A$ **Canonical forms (CBV)** $V, W ::= \cdots \mid \mathsf{nil} \mid \mathsf{cons}(V, W)$

$$\frac{}{\Gamma \vdash \mathsf{nil} : \mathsf{list}\ A} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : \mathsf{list}\ A}{\Gamma \vdash \mathsf{cons}(M, N) : \mathsf{list}\ A} \qquad \frac{\Gamma \vdash N : \mathsf{list}\ A \quad \Gamma \vdash M_{\mathsf{nil}} : B \quad \Gamma, x : A, xs : \mathsf{list}\ A \vdash M_{\mathsf{cons}} : B}{\Gamma \vdash \mathsf{case}\ n\ \mathsf{of}\ \{\, \mathsf{nil} \mapsto M_{\mathsf{nil}} \mid \mathsf{cons}(x, xs) \mapsto M_{\mathsf{cons}} \,\} : B}$$

$$\frac{}{\mathsf{nil} \downarrow^{0} \mathsf{nil}} \qquad \frac{N \downarrow^{n} \mathsf{nil} \quad M_{\mathsf{nil}} \downarrow^{m} Z}{\mathsf{case}\ N\ \mathsf{of}\ \{\, \mathsf{nil} \mapsto M_{\mathsf{nil}} \mid \mathsf{cons}(x, xs) \mapsto M_{\mathsf{cons}} \,\} \downarrow^{n+m} Z}$$

$$\frac{M \downarrow^{m} V \quad N \downarrow^{n} W}{\mathsf{cons}(M, N) \downarrow^{m+n} \mathsf{cons}(V, W)} \qquad \frac{N \downarrow^{n} \mathsf{cons}(V, W) \quad M_{\mathsf{cons}}[V/x, W/xs] \downarrow^{m} Z}{\mathsf{case}\ N\ \mathsf{of}\ \{\, \mathsf{nil} \mapsto M_{\mathsf{nil}} \mid \mathsf{cons}(x, xs) \mapsto M_{\mathsf{cons}} \,\} \downarrow^{n+m} Z}$$

CBPV

Value types $A ::= \cdots \mid \mathsf{list}\ A$

$$\frac{}{\Gamma \vdash_{\mathsf{v}} \mathsf{nil} : \mathsf{list}\ A} \qquad \frac{\Gamma \vdash_{\mathsf{v}} M : A \quad \Gamma \vdash_{\mathsf{v}} N : \mathsf{list}\ A}{\Gamma \vdash_{\mathsf{v}} \mathsf{cons}(M, N) : \mathsf{list}\ A}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} N : \mathsf{list}\ A \quad \Gamma \vdash_{\mathsf{c}} M_{\mathsf{nil}} : \underline{B} \quad \Gamma, x : A, xs : \mathsf{list}\ A \vdash_{\mathsf{c}} M_{\mathsf{cons}} : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{case}\,(N; M_{\mathsf{nil}}; (x, xs).\ M_{\mathsf{cons}}) : \underline{B}}$$

$$\frac{M_{\mathsf{nil}} \Downarrow^{n} T}{\mathsf{case}\,(\mathsf{nil}; M_{\mathsf{nil}}; (x, xs).\ M_{\mathsf{cons}}) \Downarrow^{n} T} \qquad \frac{M_{\mathsf{cons}}[V/x, W/xs] \Downarrow^{n} T}{\mathsf{case}\,(\mathsf{cons}(V, W); M_{\mathsf{nil}}; (x, xs).\ M_{\mathsf{cons}}) \Downarrow^{n} T}$$

CBV PCF translation to CBPV

A (PCF type, CBV)	A^* (CBPV+ value type)
$\mathsf{list}\ A$	$\mathsf{list}\ A^*$

$x_1 : A_1, \ldots, x_n : A_n \vdash M : A$	$x_1 : A_1^*, \ldots, x_n : A_n^* \vdash_{\mathsf{c}} M^* : F(A^*)$
nil	$\mathsf{return}\ \mathsf{nil}$
$\mathsf{cons}(M, N)$	$\mathsf{bind}\ m \Leftarrow M^*\ \mathsf{in}\ \mathsf{bind}\ n \Leftarrow N^*\ \mathsf{in}\ \mathsf{return}\ \mathsf{cons}(m, n)$
$\mathsf{case}\ M\ \mathsf{of}\ \{\, \mathsf{nil} \mapsto M_{\mathsf{nil}} \mid \mathsf{cons}(x, xs) \mapsto M_{\mathsf{cons}} \,\}$	$\mathsf{bind}\ m \Leftarrow M^*\ \mathsf{in}\ \mathsf{case}\left(m; M_{\mathsf{nil}}^*; (x, xs).\ M_{\mathsf{cons}}^*\right)$

CBPV recurrence extraction

$$\langle\!\langle \mathsf{list}\ A \rangle\!\rangle \stackrel{\mathsf{def}}{=} \mathsf{nat} \qquad \frac{}{\Gamma \vdash_{\mathsf{v}} \mathsf{nil} \searrow \underline{0} : \mathsf{list}\ A} \qquad \frac{\Gamma \vdash_{\mathsf{v}} W \searrow P : \mathsf{list}\ A}{\Gamma \vdash_{\mathsf{v}} \mathsf{cons}(V, W) \searrow P + \underline{1} : \mathsf{list}\ A}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} V \searrow W : \mathsf{list}\ A \quad \Gamma \vdash_{\mathsf{c}} M_{\mathsf{nil}} \searrow P : \underline{B} \quad \Gamma, x : A, xs : \mathsf{list}\ A \vdash_{\mathsf{c}} M_{\mathsf{cons}} \searrow Q : \underline{B}}{\Gamma \vdash_{\mathsf{c}} \mathsf{case}\,(V; M_{\mathsf{nil}}; (x, xs).\ M_{\mathsf{cons}}) \searrow \mathsf{if}\ W\ \mathsf{then}\ P\ \mathsf{else}\ Q[\mathsf{fix}\ x.\ x/x, W - \underline{1}/xs] : \underline{B}}$$

$$\mathsf{nil} \lesssim^{\mathrm{val}}_{\mathsf{list}\ A} E \stackrel{\mathsf{def}}{\equiv} \underline{0} \leqslant E \qquad \mathsf{cons}(V, W) \lesssim^{\mathrm{val}}_{\mathsf{list}\ A} E \stackrel{\mathsf{def}}{\equiv} \underline{1} \leqslant E \wedge W \lesssim^{\mathrm{val}}_{\mathsf{list}\ A} E - \underline{1}$$

Fig. 16. Extensions for handling call-by-value lists

$$
\begin{array}{rcl}
\mathsf{rec}\ \mathtt{sort}(xs) & = & \\
\mathsf{case}\ xs\ \mathsf{of} & \mathsf{nil} \mapsto \mathsf{nil} & \\
& \mid \mathsf{cons}(y, ys) \mapsto \mathsf{case}\ ys\ \mathsf{of} & \mathsf{nil} \mapsto \mathsf{cons}(y, \mathsf{nil}) \\
& & \mid \mathsf{cons}(z, zs) \mapsto \mathsf{let}\ q = \mathtt{divide}(\mathsf{cons}(y, ys))\ \mathsf{in} \\
& & \qquad \mathtt{merge}(\mathtt{sort}(\pi_1\, q), \mathtt{sort}(\pi_2\, q)) \\
\|\mathtt{sort}\| & = & \langle 0, \mathsf{fix}\ \mathtt{sort}.\ \lambda xs.\mathsf{if}\ xs\ \mathsf{then}\ \langle 0, 0\rangle \\
& & \qquad \mathsf{else\ if}\ xs - 1\ \mathsf{then}\ \langle 0, 1\rangle\ \mathsf{else}\ F(xs-1)\rangle \\
F(xs') & = & (\widehat{2} \boxplus (\mathtt{divide}(xs'+1))_c) +_c E((\mathtt{divide}(xs'+1))_p) \\
E(q) & = & (\widehat{5} \boxplus (\mathtt{sort}(\pi_1\, q))_c \boxplus (\mathtt{sort}(\pi_2\, q))_c) +_c \mathtt{merge}\left\langle (\mathtt{sort}(\pi_1\, q))_p, (\mathtt{sort}(\pi_2\, q))_p \right\rangle
\end{array}
$$

Fig. 17. The merge sort function in CBV PCF and its extracted recurrence

such as one that records both the size of the tree along with the maximum label. These different abstractions in turn allow for more or less detailed cost analyses of the original programs.

10.1 Merge Sort Example

We end with an example, viz. recurrence extraction for merge sort in CBV PCF. We assume the existence of two functions $\mathtt{divide} : \mathsf{list}\ A \to \mathsf{list}\ A \times \mathsf{list}\ A$ and $\mathtt{merge} : \mathsf{list}\ A \times \mathsf{list}\ A \to \mathsf{list}\ A$, which perform the usual tasks of splitting a list into two (nearly) equal-sized pieces and merging two sorted lists into a single sorted list. The translation into CBPV is extremely verbose, so we directly give the extracted recurrence in Figure 17. Corresponding to divide and merge are CBPV programs $\mathtt{divide} : U(\mathsf{list}\ A^* \to F(\mathsf{list}\ A^* \times \mathsf{list}\ A^*))$ and $\mathtt{merge} : U(\mathsf{list}\ A^* \times \mathsf{list}\ A^* \to F(\mathsf{list}\ A^*))$, and thus PCFc recurrences $\mathtt{divide} : \mathsf{nat} \to \mathbb{C} \times \mathsf{nat} \times \mathsf{nat}$ and $\mathtt{merge} : \mathsf{nat} \times \mathsf{nat} \to \mathbb{C} \times \mathsf{nat}$. The recurrence for divide expresses the cost of the CBV PCF divide and the lengths of the two returned lists in terms of the length of its input. The recurrence extraction is, as with exponentiation, a tedious unwinding of definitions which uses a few additional size order axioms that are valid in the standard semantics of §8.

We can then analyze the semantic recurrence—i.e. the denotation of $\|\mathtt{sort}\|$—and the two projections for potential and cost. For this discussion, we will write syntactic expressions but manipulate them as though they are the corresponding denotations. If $S(n) = \pi_2(\|\mathtt{sort}\|_p\, n)$ and $T(n) = \pi_1(\|\mathtt{sort}\|_c\, n)$, then the Bounding Theorem tells us that $S(n)$ and $T(n)$ are upper bounds on the length and cost of $\mathtt{sort}(xs)$ respectively, when xs has length n. We will assume that

$$\mathtt{divide}\ k = \langle ck, \lceil k/2 \rceil, \lfloor k/2 \rfloor \rangle \qquad \mathtt{merge}\ \langle k, \ell \rangle = \langle d(k+l), k+l \rangle$$

for some fixed constants c and d. This is one way to formalize the assumption that the complexity of sort does not depend on the potentials of the actual list elements: while merge depends on a comparison function, we are asserting that its cost and the length of the result depend only on the lengths of its arguments. Of course, this is exactly what we typically do for an informal analysis of merge sort on lists of constant-size elements (e.g. machine words), or on lists of arbitrary data under the assumption that the comparison function takes constant time. Moving the abstraction of lists to numbers from the recurrence extraction to the denotational semantics would permit an analysis that takes into account the complexity of the comparison function.

We can now write out a more 'traditional' recurrence for S:

$$S(0) = 0 \qquad S(1) = 1 \qquad \begin{aligned} S(n) &= (E((\texttt{divide}(n))_p))_p = (E(\langle \lceil n/2 \rceil, \lfloor n/2 \rfloor \rangle))_p \\ &= (\texttt{merge}\ \langle S(\lceil n/2 \rceil), S(\lfloor n/2 \rfloor) \rangle)_p \\ &= S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor) \end{aligned}$$

The standard proof tells us that $S(n) = n$. Next we write out a more traditional recurrence for $T(n)$ when n is a power of 2, which mimics the usual approach for analyzing such algorithms:

$$T(1) = 0 \qquad \begin{aligned} T(n) &= (2 + (\texttt{divide}(n))_c) + (E((\texttt{divide}(n))_p)_c \\ &= (2 + cn) + (E(\langle n/2, n/2 \rangle))_c \\ &= (2 + cn) + (5 + T(n/2) + T(n/2)) + (\texttt{merge}\ \langle S(n/2), S(n/2) \rangle)_c \\ &= 7 + cn + 2T(n/2) + (\texttt{merge}\ \langle n/2, n/2 \rangle)_c \\ &= 7 + (c + d)n + 2T(n/2). \end{aligned}$$

This is precisely the recurrence we expect. To proceed further with this example, we could define appropriate `divide` and `merge` functions and prove that the extracted recurrences satisfy the aforementioned assumptions.

11 RELATED WORK

There is a long history of techniques for extracting cost information from programs, probably starting with Wegbreit [1975], who computes closed bounds for Lisp programs. Much of the work has been done for imperative languages, as exemplified by the COSTA project for Java bytecode analysis [Albert et al. 2012, 2013] and SACO for parallel cost [Albert et al. 2018]. We cannot hope to review the entire field here, so concentrate on recent work on cost analysis for functional programs.

The idea that the potential of a function is itself a function is taken from Danner et al. [2015], who in turn adapt it from Danner and Royer [2007]. Danielsson [2008] focuses on amortized cost of lazy programs and makes use of what is essentially the writer monad $\mathbb{C} \times -$, though he requires the user to explicitly annotate programs with ticks (charges). The Resource Aware ML (RAML) project has achieved great results in automating the cost analysis of higher-order functional programs using automatic amortized resource analysis (AARA). This approach reduces the establishment of cost bounds on a program to type inference in a resource-aware type system, and that in turn is reduced to a linear programming problem. It has been used to verify the cost of significant portions of the OCaml libraries [Hoffmann et al. 2017] and for analyzing space usage in the presence of garbage collection [Niu and Hoffmann 2018], to name just a couple of recent achievements. However, the version of RAML that is current as of the time of this writing is restricted to deriving polynomial bounds, and hence it derives a quadratic bound for merge-sort; we simply derive a recurrence, and if the user of the system can prove a tighter bound on it than quadratic (e.g., by using the Master Theorem), more power to her. Another type-based approach is described by Avanzini and Dal Lago [2017], where a form of size types with index polymorphism is used. As with AARA, cost analysis comes down to type inference. This work also makes use of ticks (charges); an interesting aspect of it is that the clock itself becomes part of the program to which type inference is applied, and so cost comes down to an analysis of size. Somewhat earlier work takes a program transformation approach, defunctionalizing higher-order programs in a cost-preserving way, and then applying first-order analysis techniques to the result [Avanzini et al. 2015].

The accomplishments of these approaches in terms of automatically computing cost bounds is impressive, and not something we claim our approach as described here does. It is possible that after deploying our recurrence extraction technique, recurrence solvers such as OCRS [Kincaid et al. 2017] could be used, similarly to how RAML uses an off-the-shelf LP solver (of course,

since our recurrences are higher-order functions, this would presumably require some form of defunctionalization, but perhaps Benzinger's [2004] work would be applicable). The main contrast between our work and all of the above is that our work puts the standard informal approaches to cost analysis via recurrence extraction on firm mathematical footing, while the automatic techniques use different methods than what we teach students in introductory classes or what programmers do in their heads.

12 CONCLUSIONS AND FUTURE WORK

We have improved upon the results of Danner et al. [2015] by giving a uniform recurrence extraction method for a variety of source languages. Instead of formulating extraction directly for each source language, we embed them in an intermediate language—namely CBPV—and perform recurrence extraction for that language instead. Our method uniformly handles general recursion irrespective of the evaluation strategy of the source language. We have shown this strategy in action by showing how to extract recurrences from both call-by-value and call-by-name PCF programs.

The natural next step would be to increase the expressiveness of the source language types. For example, it would be very interesting to examine whether our strategy can be extended to cover *recursive types* [Pierce 2002, §20]. This would lead us to an extraction function for call-by-value and call-by-name versions of Plotkin's *Fixed Point Calculus (FPC)* [Plotkin 1985], [Gunter 1992, §7.4], thereby enabling the consideration of a large number of very expressive programming languages. On the other hand, such a programme would present significant technical challenges: amongst other things, it would require the solution of sized domain equations.

In either case, the combination of inductive or recursive data types and recursion would also aid us in obtaining a handle on the complexity of *coinductive data*. For example, in a call-by-value setting with inductive types and recursion we can express *streams* of natural numbers by the type $\mu X.\mathbf{1} \to (\mathbf{1} + \mathsf{nat} \times X)$. General recursion then allows us to define non-trivial streams and functions that compute on them. Recurrence extraction in this setting would guide us towards various notions of *stream complexity*. Moreover, this would be a special case of *functional reactive programming (FRP)* [Elliott and Hudak 1997], and extending recurrence extraction to FRP would naturally lead us to interesting notions of complexity for functional programs that intentionally do not terminate.

In another direction, deterministic programs with interesting average-case behavior, such as those implementing Quicksort, typically use general recursion. It should be the case that the notions of probabilistic recurrences [Karp 1994] used to informally analyze such algorithms can be formalized in terms of appropriate semantic models in which the size of an argument is interpreted as an appropriate random variable. Finally, formalizing amortized analysis is another direction of significant interest. The current approach analyzes composition by composing worst-case bounds, which do not always yield a tight result. The goal would not be the automated analysis of AARA, but a formalization of informal techniques such as Tarjan's [1985] banker's and physicist's methods.

ACKNOWLEDGMENTS

This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-16-1-0292. Any opinions, finding, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force. Additionally, this material is based upon work supported by the National Science Foundation under Grant Number 1618203.

REFERENCES

Samson Abramsky. 1990. The Lazy Lambda Calculus. In *Research Topics in Functional Programming*. Addison Wesley, 65–117. https://www.cs.ox.ac.uk/files/293/lazy.pdf

Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. 1996. Full Abstraction for PCF. *Information and Computation* 163 (1996), 409–470.

Samson Abramsky and Achim Jung. 1994. Domain Theory. In *Handbook of Logic in Computer Science*, Samson Abramsky, Dov M. Gabbay, and Thomas S. E. Maibaum (Eds.). Vol. 3. Oxford University Press, 1–168. https://www.cs.bham.ac.uk/~axj/pub/papers/handy1.pdf

Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. 2012. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science* 413, 1 (2012), 142–159. https://doi.org/10.1016/j.tcs.2011.07.009

Elvira Albert, Jesús Correas, Einar Broch Johnsen, Ka I. Pun, and Guillermo Román-Díez. 2018. Parallel Cost Analysis. *ACM Transactions on Computational Logic* 19, 4, Article 31 (Nov. 2018), 37 pages. https://doi.org/10.1145/3274278

Elvira Albert, Samir Genaim, and Abu Naser Masud. 2013. On the Inference of Resource Usage Upper and Lower Bounds. *ACM Transactions on Computational Logic* 14, 3 (2013), 22:1–22:35. https://doi.org/10.1145/2499937.2499943

Martin Avanzini and Ugo Dal Lago. 2017. Automating Sized-type Inference for Complexity Analysis. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 43 (2017), 29 pages. https://doi.org/10.1145/3110287

Martin Avanzini, Ugo Dal Lago, and Georg Moser. 2015. Analyzing the complexity of functional programs: higher-order meets first-order. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*, Kathleen Fisher and John Reppy (Eds.). 152–164. https://doi.org/10.1145/2784731.2784753

Ralph Benzinger. 2004. Automated higher-order complexity analysis. *Theoretical Computer Science* 318, 1-2 (2004), 79–103. https://doi.org/10.1016/j.tcs.2003.10.022

Richard Bird. 2014. *Thinking Functionally with Haskell*. Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9781316092415

Guy Blelloch and John Greiner. 1995. Parallelism in Sequential Functional Languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*. ACM, New York, NY, USA, 226–237. https://doi.org/10.1145/224164.224210

Nils Anders Danielsson. 2008. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, George Necula and Philip Wadler (Eds.). 133–144. https://doi.org/10.1145/1328438.1328457

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2015. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming - ICFP 2015*. ACM Press, New York, New York, USA, 140–151. https://doi.org/10.1145/2784731.2784749

Norman Danner, Jennifer Paykin, and James S. Royer. 2013. A static cost analysis for a higher-order language. In *Proceedings of the 7th workshop on Programming languages meets program verification - PLPV '13*. ACM Press, New York, New York, USA. https://doi.org/10.1145/2428116.2428123

Norman Danner and James S. Royer. 2007. Adventures in time and space. *Logical Methods in Computer Science* 3, 9 (2007), 1–53. https://doi.org/10.2168/LMCS-3(1:9)2007

Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN International Conference on Functional Programming - ICFP '97*. ACM Press, New York, New York, USA, 263–273. https://doi.org/10.1145/258948.258973

Martín Hötzel Escardó and Weng Kin Ho. 2009. Operational domain theory and topology of sequential programming languages. *Information and Computation* 207, 3 (2009), 411–437. https://doi.org/10.1016/j.ic.2008.12.003

Marcelo P. Fiore. 1996. *Axiomatic Domain Theory in Categories of Partial Maps*. Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9780511526565

Carl A. Gunter. 1992. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press.

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 359–373. https://doi.org/10.1145/3009837.3009842

Richard M. Karp. 1994. Probabilistic recurrence relations. *Journal of the Association for Computing Machinery* 41, 6 (1994), 1136–1150. https://doi.org/10.1145/195613.195632

Theodore Seok Kim. 2016. *Cost Semantics for Plotkin's PCF*. Master's Thesis. Wesleyan University. https://wesscholar.wesleyan.edu/etd_mas_theses/130/

Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. 2017. Non-linear Reasoning for Invariant Synthesis. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 54 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158142

Paul Blain Levy. 2003. *Call-by-Push-Value: A Functional-Imperative Synthesis*. Springer. https://doi.org/10.1007/978-94-007-0954-6

Paul Blain Levy. 2006. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation* 19 (2006), 377–414. https://doi.org/10.1007/s10990-006-0480-6

Robin Milner. 1977. Fully abstract models of typed λ-calculi. *Theoretical Computer Science* 4, 1 (1977), 1–22. https://doi.org/10.1016/0304-3975(77)90053-6

John C. Mitchell. 1996. *Foundations for programming languages.* The MIT Press.

Yue Niu and Jan Hoffmann. 2018. Automatic Space Bound Analysis for Functional Programs with Garbage Collection. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing)*, Gilles Barthe, Geoff Sutcliffe, and Margus Veanes (Eds.), Vol. 57. EasyChair, 543–563. https://doi.org/10.29007/xkwx

Benjamin C. Pierce. 2002. *Types and Programming Languages.* The MIT Press.

A. M. Pitts. 1994. *Some notes on inductive and co-inductive techniques in the semantics of functional programs.* Technical Report. BRICS. https://www.brics.dk/NS/94/5/BRICS-NS-94-5.pdf

A. M. Pitts. 1997. Operationally-Based Theories of Program Equivalence. In *Semantics and Logics of Computation*, A. M. Pitts and P. Dybjer (Eds.). Cambridge University Press, 241–298. https://www.cl.cam.ac.uk/~amp12/papers/opebtp/opebtp.pdf

Gordon Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *Foundations of Software Science and Computation Structures. FoSSaCS 2001*, Furio Honsell and M. Miculan (Eds.). Lecture Notes in Computer Science, Vol. 2030. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45315-6_1

Gordon D. Plotkin. 1977. LCF considered as a programming language. *Theoretical Computer Science* 5, 3 (1977), 223–255. https://doi.org/10.1016/0304-3975(77)90044-5

G. D. Plotkin. 1985. Lectures on predomains and partial functions. Notes for a course given at CSLI, Stanford University.

Jon G. Riecke. 1993. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science* 3 (1993), 387–415. https://doi.org/10.1017/S0960129500000293

V. Stoltenberg-Hansen, I. Lindstrom, and E. R. Griffor. 1994. *Mathematical Theory of Domains.* Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9781139166386

Thomas Streicher. 2006. *Domain-theoretic Foundations of Functional Programming.* World Scientific.

Robert Endre Tarjan. 1985. Amortized computational complexity. *SIAM J. Algebraic Discrete Methods* 6, 2 (1985), 306–318. https://doi.org/10.1137/0606031

Ben Wegbreit. 1975. Mechanical program analysis. *Communications of the Association for Computing Machinery* 18, 9 (1975), 528–539. https://doi.org/10.1145/361002.361016