
Higher-Order and Symbolic Computation,, 12, 125–170 (1999)
© 1999 Kluwer Academic Publishers. Manufactured in The Netherlands.

CPS Translations and Applications: The Cube and Beyond

GILLES BARTHE gilles@di.uminho.pt
Departamento de Informática, Universidade do Minho, 4709 Braga Codex, Portugal

JOHN HATCLIFF hatcliff@cis.ksu.edu
Department of Computing and Information Sciences, Kansas State University, Manhattan, KS, USA, 66506

MORTEN HEINE B. SØRENSEN rambo@diku.dk
DIKU, Universitetsparken 1, DK-2100 Copenhagen, Denmark

Abstract. Continuation passing style (CPS) translations of typed λ-calculi have numerous applications. However, the range of these applications has been confined by the fact that CPS translations are known for *non-dependent* type systems only, thus excluding well-known systems like the calculus of constructions (CC) and the logical frameworks (LF). This paper presents techniques for CPS translating systems with dependent types, with an emphasis on pure type-theoretical applications.

In the *first part* of the paper we review several lines of work in which the need for CPS translations of dependent type systems has arisen, and discuss the difficulties involved with CPS translating such systems. One way of overcoming these difficulties is to work with so-called *domain-free* type systems. Thus, instead of Barendregt's λ-cube we shall consider the *domain-free λ-cube*, and instead of traditional pure type systems, we shall consider *domain-free pure type systems*.

We therefore begin the *second part* by reviewing the domain-free λ-cube, which includes domain-free versions of CC and LF, and then present CPS translations for all the systems of the domain-free λ-cube. We also introduce Direct Style (DS) (i.e., inverse CPS) translations for all the systems of the domain-free λ-cube; such DS translations, which have been used in a number of applications, were previously formulated for untyped and simply-typed languages only.

In the *third part* we review domain-free pure type systems and generalize the CPS translations of the domain-free λ-cube to a large class of domain-free pure type systems which includes most of the systems that appear in the literature, including those of the domain-free λ-cube. Many translations that appear in the literature arise as special cases of ours.

In the *fourth part* of the paper we present two approaches to CPS translations of traditional pure type systems. The first, indirect, technique lifts the CPS translation of domain-free pure type systems to the analogous class of traditional pure type systems by using results that relate derivations in domain-free and traditional pure type systems. The second, direct, approach translates derivations, requiring a certain order on derivations to be well-founded. Both techniques yield translations for most of the systems that appear in the literature, including those of Barendregt's λ-cube.

Keywords: continuation-passing style, pure type systems, dependent types, the lambda cube

1. Introduction

Continuation passing style (CPS) translations of typed λ-calculi have appeared throughout the literature since the work of Meyer and Wand [46]. Applications, too numerous to be

listed exhaustively here, include compilation [1, 25], transformation [14, 49], and analysis [58, 59, 65] of typed programming languages, construction of semantics definitions for languages with jumps [56, 61], exceptions, and concurrency primitives [26], embedding of classical logics in intuitionistic logics [31, 48], techniques to infer strong normalization from weak normalization in typed λ-calculi [66, 74], and the construction of looping combinators in inconsistent pure type systems [15]. Related Direct Style (DS) translations [17, 19, 58] have also been used in both theoretical [57] and implementation-oriented applications [60].

The range of these applications has been confined thus far by the fact that CPS translations are known for *non-dependent* type systems only. Indeed, the most general class of systems with known CPS translation seems to be the non-dependent logical pure type systems, studied by Coquand and Herbelin [15] and later by Werner [73]. While this class contains a number of well-known systems, like simply typed λ-calculus and Girard's System F (second-order λ-calculus) and System F^ω (higher-order λ-calculus), it also excludes some well-known dependent systems, e.g., the calculus of constructions (CC) and the logical frameworks (LF). As for DS translations, they have only been defined for untyped and simply-typed languages.

Below we sketch three of the above mentioned applications of CPS translations in more detail: intermediate languages for compiling and partial evaluation, inferring strong normalization from weak normalization in typed λ-calculi and embedding classical logics in intuitionistic logics. With these examples we hope to convince the reader that, indeed, it is desirable to extend existing results concerning CPS translations to richer type systems, including systems with dependent types. The last two subsections discuss the difficulties involved with this extension and how we propose to solve them in this paper.

1.1. Intermediate languages for compiling

CPS or languages with properties similar to CPS (such as A-normal forms [25] or monadic normal forms [39]—we refer to these as *CPS-based languages*) are often used as intermediate languages when compiling and partial evaluating functional languages [1, 21, 38, 40, 43, 50, 60, 68]. These applications take advantage of the fact that, in CPS, all intermediate values are named and that contexts are represented explicitly. For example, intermediate value naming is used to aid register allocation in compiling (roughly, each name corresponds to a register) and to prevent computations from being discarded or reordered in partial evaluation. Explicit context representation is helpful for optimizing tail-calls in compiling and for performing binding-time improvements in partial evaluation.

Moreover, current trends in compilation and partial evaluation are emphasizing the importance of *typed* CPS-based intermediate languages. These intermediate languages are particularly relevant for compilation of languages like ML and Haskell that have sophisticated polymorphic type systems. Recently, Meijer and Peyton Jones [45] suggested using the λ-cube as a basis for typed intermediate languages. This strategy builds on a well-established practice of using System F for compilation (see e.g., [35, 37, 47, 51, 71, 64]). The λ-cube and more general pure type systems seem well-suited for intermediate languages because of the following features:

- *compactness*: since terms and types are conflated in one syntactic category, the same functions can be used for their manipulation, and
- *parametricity*: the intermediate language is robust with respect to changes to the type system of the source language—extensions to the source type system often amount to changing only the specification of the pure type system.

Direct-style (DS) translations have been used for performing partial evaluation, and for designing intermediate languages for compilation and partial evaluation. Danvy and Lawall [17, 19] used CPS and DS translations to enable partial evaluation of functional programs with control operators without requiring partial evaluators to treat control operators. This proceeds in three steps: (1) remove control operators by CPS translating, (2) partially evaluate, (3) apply a DS translation to obtain a specialized program in direct-style. They apply this technique to e.g., co-routines written using Scheme's *call/cc* control operator.

Sabry and Felleisen [58] and Flanagan et al. [25] illustrated how a DS translation could be used to derive an appropriate reduction calculus for an intermediate language called A-normal forms. Sabry and Wadler [60] and the authors [8] use DS translations to establish correctness properties of CPS-based translations. Lawall and Thiemann use a DS translation to prove the correctness of a partial evaluation strategy for specializing functional programs with computational effects [43].

Given the wide applicability of CPS and DS translations for compilation and partial evaluation as well as the recent emphasis on typed CPS-based intermediate languages, it is worthwhile to investigate how they can be scaled up to languages with type systems that can be described using the λ-cube and the framework of pure type systems. Various researchers are advocating programming languages with dependent types, and an investigation of CPS seems especially relevant in this context. For example, Augustsson [2] has developed a version of Haskell with dependent types called Cayenne, and Xi and Pfenning have proposed an extension of ML that uses dependent types to eliminate array bound checking [75]. The lack of CPS/DS translations for dependently-typed systems means that none of the CPS/DS techniques discussed above can be applied immediately to such systems.

1.2. Strong normalization from weak normalization

Informally, a term in some calculus is weakly normalizing if it has a reduction sequence ending in a normal form, i.e., in a term to which no further reductions apply. A term is strongly normalizing if all reduction sequences from the term eventually end in normal forms; that is, if the term has no infinite reductions.

The classical proof of strong normalization for β-reduction in simply typed λ-calculus is due to Tait [69]. It was generalized to second-order typed λ-calculus by Girard [30], and subsequently simplified by Tait [70]. The technique is very flexible and has been generalized to a variety of λ-calculi.

For some notions of reduction in some typed λ-calculi there is a technique to prove weak normalization that is simpler than the Tait-Girard technique to prove strong normalization. For instance, Turing [27] proves weak normalization for β-reduction in simply typed λ-calculus by giving an explicit measure which decreases in every step of a certain β-reduction

sequence. Prawitz [53] independently uses the same technique to prove weak normalization for reduction of natural deduction derivations in predicate logic.

Since weak normalization is sometimes easier to establish than strong normalization, it is natural to develop techniques to infer the latter from the former. Indeed, several such techniques have been presented, most of which infer strong normalization of one notion of reduction from weak normalization of a *more complicated* notion of reduction, see [66] for references. However Sørensen [66] and Xi [74] recently developed techniques which infer strong normalization of β-reduction in a typed λ-calculus from weak normalization of the *same* notion of reduction, i.e. β-reduction.

These techniques provide some hope for a positive answer to a conjecture, presented by Barendregt at *Typed Lambda-Calculus and Applications*, *Edinburgh 1995*, stating that for every pure type system weak normalization of β-reduction implies strong normalization of β-reduction. The conjecture has also been mentioned by Geuvers [28], and, in a less concrete form, by Klop.

To explain these recent techniques by Sørensen and Xi we consider simply typed λ-calculus à la Curry. Let Λ denote the set of *untyped λ-terms*, as defined by the abstract syntax:

$$\Lambda \ni M ::= x \mid \lambda x.M \mid M_1 M_2$$

where x ranges over a denumerable set of variables. The set of *simple types* $\mathcal{T}$ is defined by the abstract syntax:

$$\mathcal{T} \ni \tau ::= \alpha \mid \tau_1 \to \tau_2$$

where α ranges over a denumerable set of type variables. The set of *contexts* is the set of all

$$\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$$

where $\tau_1, \ldots, \tau_n \in \mathcal{T}$, $x_1, \ldots, x_n$ are variables, and where $x_i \not\equiv x_j$ for $i \neq j$. We write $x : \tau$ for $\{x : \tau\}$ and Γ, Γ' for $\Gamma \cup \Gamma'$ where it is assumed that for no variable x is there σ and τ such that both $x : \sigma \in \Gamma$ and $x : \tau \in \Gamma'$. The relation $\vdash$ on triples (Γ, M, σ), where Γ is a context, M is a λ-term, and σ is a simple type, is defined by:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \sigma \vdash P : \tau}{\Gamma \vdash \lambda x.P : \sigma \to \tau} \qquad \frac{\Gamma \vdash P : \sigma \to \tau \quad \Gamma \vdash Q : \sigma}{\Gamma \vdash PQ : \tau}$$

A λ-term M is *typable* if there are Γ and σ such that $\Gamma \vdash M : \sigma$.

As usual *β-reduction* $\to_\beta$ on Λ is the smallest compatible relation such that

$$(\lambda x.P)Q \to_\beta P\{x := Q\}$$

where $P\{x := Q\}$ denotes substitution of Q for all the free occurrences of x in P.[1] A *β-normal form* is a λ-term M such that there is no other λ-term N with $M \to_\beta N$. A

$$\begin{array}{lcl} [x] & = & \lambda k.x\ k \\ [\lambda x.M] & = & \lambda k.k\ (\lambda x.[M]) \\ [M_1\ M_2] & = & \lambda k.[M_1]\ (\lambda m.m\ [M_2]\ k) \end{array}$$

Figure 1. Plotkin's call-by-name CPS translation for untyped λ-terms.

$$\begin{array}{lcl} \langle x \rangle & = & \lambda k.x\ k \\ \langle \lambda x.M \rangle & = & \lambda k.k\ (\lambda x.\lambda h.y\ (\langle M \rangle\ h)\ x) \\ \langle M_1\ M_2 \rangle & = & \lambda k.\langle M_1 \rangle\ (\lambda m.m\ \langle M_2 \rangle\ k) \end{array}$$

Figure 2. Modified call-by-name CPS translation for untyped λ-terms.

λ-term M is *β-weakly normalizing* if there exists a β-reduction sequence $M \to_\beta \cdots \to_\beta N$ ending in a β-normal form N, and *β-strongly normalizing* if *all* β-reduction sequences from M end in β-normal forms (that is, if there are no infinite β-reduction sequences from M).[2] We shall say that simply typed λ-calculus is *weakly* and *strongly normalizing* if all typable terms are weakly and strongly normalizing, respectively. The latter property trivially implies the former, but the converse is not obvious.

Recall Plotkin's [52] CPS translation $[\bullet]: \Lambda \to \Lambda$ defined in figure 1. The main idea of the techniques by Sørensen and Xi is to modify Plotkin's translation by changing the clause for abstractions so as to arrive at the translation $\langle \bullet \rangle : \Lambda \to \Lambda$ defined in figure 2. The free variable y on the right hand side must be so construed as to be fresh every time the clause containing it is invoked. Thus, we do not have

$$\langle \lambda x.\lambda z.x \rangle \equiv \lambda k.k(\lambda x.\lambda h.y(\lambda l.l(\lambda z.\lambda m.y((\lambda k.xk)m)z)h)x)$$

but rather

$$\langle \lambda x.\lambda z.x \rangle \equiv \lambda k.k(\lambda x.\lambda h.y_1(\lambda l.l(\lambda z.\lambda m.y_2((\lambda k.xk)m)z)h)x)$$

For every abstraction $\lambda x.P$ in $\langle M \rangle$, x occurs free in P. Therefore, by the Conservation Theorem for λI (see [3]), weak normalization of $\langle M \rangle$ implies strong normalization of $\langle M \rangle$. The intuition behind this result is that the only way a term can be weakly normalizing and at the same time fail to be strongly normalizing is by having, roughly, a subterm with an infinite reduction and the capability to erase—via an abstraction $\lambda x.P$ where x is not free in P—this subterm.

One can also show that strong normalization of $\langle M \rangle$ implies strong normalization of M. Thus, if $\langle M \rangle$ is weakly normalizing, M is strongly normalizing.[3]

Incidentally, the modified translation does not map a term to something which is β-equivalent (or $\beta\eta$-equivalent) with the original term, but the translation is still very useful for the intended application.

Now let $\perp$ be a fixed type variable, and $\neg\sigma \equiv \sigma \to \perp$. The map $\langle \bullet \rangle : \mathcal{T} \to \mathcal{T}$ defined in figure 3 is the call-by-name analogue of Meyer and Wand's [45] CPS translation of types. As usual, $\langle \Gamma \rangle = \{x : \langle \sigma \rangle \mid x : \sigma \in \Gamma\}$. One can now show that if $\Gamma \vdash M : \sigma$ then $\langle \Gamma \rangle, \Delta \vdash \langle M \rangle : \langle \sigma \rangle$ for a certain Δ; that is, the translation maps simply typable terms

$$\begin{array}{lcl} \langle\sigma\rangle & = & \neg\neg\langle\sigma\rangle' \\ \\ \langle\alpha\rangle' & = & \alpha \\ \langle\sigma \to \tau\rangle' & = & \langle\sigma\rangle \to \langle\tau\rangle \end{array}$$

Figure 3. Call-by-name CPS translation of types.

to simply typable terms. It follows that if all typable terms are weakly normalizing then all typable terms are in fact strongly normalizing. Indeed, suppose all typable terms are weakly normalizing. Given any typable term M, also $\langle M\rangle$ is typable, hence $\langle M\rangle$ is weakly normalizing, and therefore M is strongly normalizing.

A main problem in extending the technique to other typed λ-calculi is to extend the CPS translation. The paper [66] presents the technique for versions of second- and higher-order λ-calculus, as well as for systems with subtypes and recursive types. Later, the authors [7] generalize the technique to a class of non-dependent pure type systems. Thus far the technique has not been applied to dependent systems, partly because CPS translations for such systems are not known, and partly because of certain other difficulties which are beyond the scope of this paper.

1.3. Classical pure type systems

The *Curry-Howard isomorphism* [16, 42] states a correspondence between constructive logics and typed λ-calculi, and reflects an old idea that proofs in constructive logics are certain functions and objects. The isomorphism has evolved with the invention of numerous typed λ-calculi and corresponding natural deduction logics—see [4, 28].

Until the late 1980s, the Curry-Howard isomorphism was concerned exclusively with constructive logics. At that time Griffin [31] discovered that Felleisen's [22, 24] control operator $\mathcal{C}$ could be meaningfully added to the simply typed λ-calculus by typing $\mathcal{C}$ with the double negation rule. The reduction rules for $\mathcal{C}$ are related to well-known reductions on classical proofs [53, 62, 67]. Moreover, Griffin discovered that well-known double-negation embeddings of classical logic in intuitionistic logic, due to Kolmogorov and others, correspond to CPS-translations.[4]

For concreteness we consider again simple types; here we assume that $\bot$ is a fixed type variable. Let Λ_μ denote the set of untyped λ-terms extended with a simple control operator μ:

$$\Lambda_\mu \ni M ::= x \mid \lambda x.M \mid M_1 M_2 \mid \mu x.M$$

where x ranges over a denumerable set of variables. The relation $\vdash$ on triples (Γ, M, σ), where Γ is a context, $M \in \Lambda_\mu$, and σ is a simple type, is defined by:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \sigma \vdash P : \tau}{\Gamma \vdash \lambda x.P : \sigma \to \tau}$$

$$\frac{\Gamma \vdash P : \sigma \to \tau \quad \Gamma \vdash Q : \sigma}{\Gamma \vdash P\,Q : \tau} \qquad \frac{\Gamma, x : \neg\sigma \vdash M : \bot}{\Gamma \vdash \mu x.M : \sigma}$$

An $M \in \Lambda_\mu$ is *typable* if there are Γ and σ such that $\Gamma \vdash M : \sigma$.

There are several reduction rules one can adopt for Λ_μ; by $\rightarrow_{\beta\mu}$ we denote the smallest compatible relation such that

$$(\lambda x.P)Q \rightarrow_{\beta\mu} P\{x := Q\}$$
$$(\mu x.P)Q \rightarrow_{\beta\mu} \mu y.P\{x := \lambda z.y(zQ)\}$$

where again $P\{x := Q\}$ denotes substitution of Q for all the free occurrences of x in P; this is the core of the system studied in [55]. A $\beta\mu$-normal form is a λ-term M such that there is no other λ-term N with $M \rightarrow_{\beta\mu} N$. A λ-terms M is *$\beta\mu$-weakly normalizing* if there exists a $\beta\mu$-reduction sequence $M \rightarrow_{\beta\mu} \cdots \rightarrow_{\beta\mu} N$ ending in a $\beta\mu$-normal form N, and *$\beta\mu$-strongly normalizing* if *all* $\beta\mu$-reduction sequences from M end in $\beta\mu$-normal forms (that is, if there are no infinite $\beta\mu$-reduction sequences from the M). The above λ-calculus is *weakly* and *strongly normalizing* if all typable terms are $\beta\mu$-weakly and $\beta\mu$-strongly normalizing, respectively.

What we have above may be construed as a natural deduction presentation of classical propositional logic where proofs are decorated with terms (these can, in turn, be viewed as a linear representation of the proofs), whereas the formulation of the simply typed λ-calculus in the preceding subsection corresponds to minimal propositional logic.

As mentioned above, CPS translations of control operators correspond to embeddings of classical logics. Indeed, consider the extension of Plotkin's call-by-name CPS-translation in figure 4, which is a call-by-name variant of Griffin's translation. One can then show the *embedding property*, i.e. that $\Gamma \vdash M : \sigma$ implies $\langle\Gamma\rangle \vdash \langle M\rangle : \langle\sigma\rangle$. This shows that logical consistency of minimal, propositional logic implies consistency of classical propositional logic. Indeed, assume classical propositional logic were inconsistent, i.e. $\vdash M : \bot$ for some M. Then also $\vdash \langle M\rangle : \neg\neg\bot$, and then $\vdash \langle M\rangle\lambda x.x : \bot$. Thus, the assumption implies that minimal, propositional logic is inconsistent too.[5]

The embedding property can also be used to relate normalization in classical logic with normalization in minimal logic and to extract computational contents of classical proofs.

Recently, several authors have considered control operators in the context of rich type disciplines. Werner [73] considers non-dependent logical pure type systems extended with a variant of the μ-operator, and shows that the extended systems is strongly normalizing if the same holds for the underlying pure type system—using CPS translations from the extended systems to the underlying pure type systems. The authors [6] study a more general notion of classical pure type system, whereby most logical pure type systems, including those with dependent types, give rise to a corresponding classical pure type system.

$$\begin{array}{lcl}
\langle x\rangle & = & \lambda k.x\ k \\
\langle \lambda x.M\rangle & = & \lambda k.k\ (\lambda x.\langle M\rangle) \\
\langle M_1\ M_2\rangle & = & \lambda k.\langle M_1\rangle\ (\lambda m.m\ \langle M_2\rangle\ k) \\
\langle \mu x.M\rangle & = & \lambda k.(\langle M\rangle\{x := \lambda h.h\ (\lambda j.\lambda i.j\ k)\})\ \lambda z.z
\end{array}$$

Figure 4. Call-by-name CPS translation of Λ_μ.

In order to study embeddings of classical pure type systems into pure type systems, in general, the need arises for more general CPS-translations, and preliminary versions of such translations in fact appeared in [6].

1.4. Difficulties with dependent types

Recall Plotkin's translation in figure 1. The translation is defined by *induction over the structure* (*or alternatively*, *the size*) *of terms*. This method of definition scales up to richer languages that are either untyped [18, 58] or typed using a "Curry-style" type system [35, 39], i.e. a type system in which the terms are the untyped λ-terms possibly extended with other forms such as conditionals, or control operators as in the language Λ_μ.

However, as mentioned in the preceding two subsections, we are interested in extending CPS translations to pure type systems (in particular, dependent pure type systems) and these are "Church-style" type systems in that they make use of *domain-full* abstractions $\lambda x:\sigma.\, M$ where a tag σ is attached to indicate the domain of the argument x. For such systems the induction over terms does not generally work. In a nut-shell, the translation of an abstraction $\lambda x:\sigma.\, M$ has form

$$\langle \lambda x:\sigma.\, M\rangle = \lambda k:\tau.\, k(\lambda x:\langle\sigma\rangle.\, \langle M\rangle) \qquad (*)$$

and the problem is: what should τ be? It turns out that if we want the analogue of the embedding property to hold, then τ should be $\neg\langle\rho\rangle$ where ρ is the type of $\lambda x:\sigma.M$. Thus, we need to take the type of a term into account when transforming the term.

The conventional solution is to define the CPS translation by induction over the structure (or over the height) of *derivations* $\Gamma \vdash M : A$. For example, this is the approach taken by Harper and Lillibridge when CPS translating $\lambda\omega$ [34]. With such definitions, one generally desires a *coherence property:* given two different derivations D_1 and D_2 of $\Gamma \vdash M : \sigma$, the translations $\langle D_1\rangle$ and $\langle D_2\rangle$ are *equivalent* in some sense. Resonable notions of equivalence include syntactic identity as well as weaker notions such as β-convertibility. Coherence is a crucial property upon which proofs of other properties are built (e.g., properties showing how the CPS translation interacts with substitution, reduction, and conversion, as well as properties showing that the translation preserves typing).

For dependent systems this approach leads to difficulties. The problematic aspect of such systems is that *types may contain terms*. More specifically such systems operate with types of form $\Pi x:\sigma.\tau$. Informally, a term M of this type is such that for all N of type σ, MN has type $\tau\{x := N\}$, as expressed by the *application* rule

$$\frac{\Gamma \vdash M : \Pi\alpha:\sigma.\tau \;\&\; \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau\{\alpha := N\}}$$

For instance, one may think of M as a term whose type is arrays of any length where for each number N, the type of MN is arrays of length N. Here N is a term, and it occurs in the type. In most dependent systems the terms that can arise inside types are λ-terms (not

some distinct category of numbers) and one then adopts the *conversion* rule

$$\frac{\Gamma \vdash M : \sigma \, (\alpha \Gamma \vdash M : s) \;\&\; \sigma =_\beta \sigma'}{\Gamma \vdash M : \sigma'}$$

When naively defining the CPS translation of dependent systems by induction over derivations, one runs into difficulty with both the conversion rule (which involves β-conversion) and the application rule (which involves substitution). The definition of the translation itself involves convertability and substitution. Thus, one cannot proceed by first addressing coherence and *then* proceeding to substitution, and conversion, etc.; one is forced to tackle the associated properties simultaneously.

As a point of comparison, Harper and Lillibridge [34] avoid both the problems associated with the conversion and application rules. When translating two different derivations D_1 and D_2 of the same judgment $\Gamma \vdash M : \sigma$, use of the conversion rule in either D_1 or D_2 may cause the terms $\langle D_1 \rangle$ and $\langle D_2 \rangle$ to have different domain tags on abstractions. Instead of considering a general theory of equality based on β-convertability (which is what we desire), Harper and Lillibridge consider a notion of object equivalence based on an operationally-flavored form of reduction (e.g., call-by-name standard reduction) that is insensitive to domain tags. This insensitivity implies that differences in domain tags do not affect the equivalence of terms [34, p. 214]. The substitution in the application rule is neither an issue since $\lambda\omega$ (the system considered by Harper and Lillibridge) is a non-dependent system, and thus the substitution $\tau\{x := N\}$ degenerates into τ.

Instead of defining CPS translations by induction on derivations, one can simply add to the clause (*) the side condition that τ be $\neg\langle\rho\rangle$ where ρ is the type of $\lambda x : \sigma . M$. For non-dependent systems this works perfectly well because there is a stratification of levels into terms and types, where the latter do not depend on the former. This is the route taken by Harper and Lillibridge, and also by Coquand and Herbelin [15] in their CPS translation of non-dependent logical pure type systems. In dependent systems, however, the categories of terms and types will be mutually dependent, so in this case the translation $\langle\bullet\rangle$ on terms and types will be mutually recursive, and the definition with side conditions is not well-founded, since ρ may contain M as a subterm.

The preceding considerations show that defining CPS translations for dependent Church-style type systems involves a number of difficulties. This is true, in particular, for dependent pure type systems.[6]

1.5. This paper

A simple approach to CPS translations of classes of type systems that include dependent systems is to consider *domain-free* pure type systems [12], i.e. a variant of pure type systems with abstractions of the form $\lambda x . M$, instead of the traditional pure type systems—in fact, this idea was one of the motivations for introducing domain-free pure type systems. This idea is carried out in Sections 2–5.

We begin by reviewing the domain-free λ-cube (Section 2), which includes domain-free versions of CC and LF, and then present CPS translations of the systems of the domain-free

λ-cube (Section 3). Then we introduce DS translations for all the systems of the domain-free λ-cube (Section 4) and relate these translations to the corresponding CPS translations. We then briefly review the general notion of domain-free pure type system and generalize the above CPS translations to a certain class of domain-free pure type systems (Section 5) which includes all the systems of the domain-free λ-cube as well as domain-free versions of many the non-dependent logical systems of Coquand and Herbelin. Many translations that appear in the literature arise as special cases of our translation.

We also present two approaches to CPS translations of traditional pure type systems: an indirect and a direct method. This is carried out in Section 6. Both techniques work for most of the systems that occur in the literature. The first, indirect, technique factorises the translation for pure type systems through the translation for domain-free pure type systems. The second, direct, technique translates traditional pseudo-terms of pure type systems, but relies on a non-standard order to be well-founded. These results justify defining the CPS translation on domain-free pure type systems instead of on traditional pure type systems.

The paper is an extended and elaborated version of [5].

2. The domain-free λ-cube

This section is a brief introduction to the domain-free λ-cube. The first subsection is devoted to the definition of the systems involved. For readers with no previous knowledge of λ-calculi presented in this style, the second subsection includes a number of examples; readers familiar with [4] may skip the latter subsection.

2.1. Definition of the domain-free λ-cube

For the clarity of exposition, we depart from the original presentation of the domain-free λ-cube [12] and make explicit the distinction between objects, constructors and kinds.[7]

Definition 1 (*The domain-free λ-cube*).

1. Let V^* and $V^\Box$ be denumerable, disjoint sets of variables, ranged over by $x, y, \ldots$ and $\alpha, \beta, \ldots$, respectively. Define the syntactic classes *Obj*[DFCUBE], *Constr*[DFCUBE], *Kind*[DFCUBE] of domain-free pseudo-objects, pseudo-constructors, and pseudo-kinds, respectively, as follows:

$$\begin{aligned} \mathit{Obj}[\mathsf{DFCUBE}] \ni O &::= x \mid \lambda x.O \mid OO \mid \lambda\alpha.O \mid OC \\ \mathit{Constr}[\mathsf{DFCUBE}] \ni C &::= \alpha \mid \lambda x.C \mid CO \mid \lambda\alpha.C \mid CC \mid \Pi x:C.C \mid \Pi\alpha:K.C \\ \mathit{Kind}[\mathsf{DFCUBE}] \ni K &::= \Pi x:C.K \mid \Pi\alpha:K.K \mid * \end{aligned}$$

We use $A, B, \ldots$ to denote arbitrary pseudo-objects, -constructors, or -kinds, and s, s' to range over the set $\{\Box, *\}$ of so-called *sorts*. We assume the reader is familiar with the notions of free and bound variables, and the related conventions—see [3]. The symbol $\equiv$ denotes syntactic equality modulo renaming of bound variables. We write $C \to C'$ as

an abbreviation of $\Pi\alpha : C.C'$, where α is not free in C'. We also write $\lambda x_1, \ldots, x_n.M$ for $\lambda x_1, \ldots, \lambda x_n.M$.

2. *Substitution* $\bullet\{\bullet := \bullet\}$ is defined as the usual capture-free operation, with the proviso that x and N belong to the same class in $M\{x := N\}$.
3. The set *Contexts*[DFCUBE] of *domain-free pseudo-contexts* is defined by the abstract syntax:

   $$\mathit{Contexts}[\mathsf{DFCUBE}] \ni \Gamma ::= \cdot \mid \Gamma, x : A$$

   We write $\mathrm{dom}(x_1 : A_1, \ldots, x_n : A_n) = \{x_1, \ldots, x_n\}$ and use Γ, Δ, etc. to denote pseudo-contexts. If Γ is $x_1 : A_1, \ldots, x_n : A_n$ we also write $x_i : A_i \in \Gamma$ for each $i \in \{1, \ldots, n\}$.
4. *Domain-free* $\underline{\beta}$*-reduction* $\to_{\underline{\beta}}$ on

   $$\mathit{Obj}[\mathsf{DFCUBE}] \cup \mathit{Constr}[\mathsf{DFCUBE}] \cup \mathit{Kind}[\mathsf{DFCUBE}]$$

   is defined as the smallest compatible relation such that

   $$(\lambda x.A)B \to_{\underline{\beta}} A\{x := B\}$$

   for all x and B belonging to the same class. $\underline{\beta}$*-equality* $=_{\underline{\beta}}$ is the reflexive, transitive, symmetric closure of $\to_{\underline{\beta}}$. The relation $\to_{\underline{\beta}}$ is extended to contexts by:

   $$A \to_{\underline{\beta}} B \Rightarrow \Gamma, x : A, \Delta \to_{\underline{\beta}} \Gamma, x : B, \Delta$$

   and $=_{\underline{\beta}}$ on contexts is the reflexive, transitive, symmetric closure of $\to_{\underline{\beta}}$.
5. For $S \in \{\to, 2, P, \underline{\omega}, P2, 2\underline{\omega}, P\underline{\omega}, P2\underline{\omega}\}$ define the relation $\Vdash_S$ by the rules of figure 5, where a side condition, e.g. (2), indicates that the rule in question only be included when S contains the corresponding symbol, e.g. 2 (except that rules marked $(\to)$ are included in all systems). If $\Gamma \Vdash_S A : B$ then Γ, A and B are *legal*. We sometimes write $\Vdash M : A$ instead of $\cdot \Vdash M : A$.

According to the eight relations $\Vdash_S$ defined above, we speak of the eight domain-free λ-calculi $\underline{\lambda}S$ depicted in figure 6, collectively known as the domain-free λ-cube.[8] We use the abbreviations $\omega = 2\underline{\omega}$, $C = P\omega$.

Intuitively, one may view the systems of the domain-free λ-cube as the domain-free counterpart of some well-known typed λ-calculi, as depicted in the table below—see also the next subsection.[9]

$\to$	Simply typed λ-calculus
2	System F (second-order λ-calculus)
P	LF, Automath
$P2$	A system considered in [44]
$\underline{\omega}$	Polyrec
ω	System F^ω (higher-order λ-calculus)
$P\underline{\omega}$	Martin-Löf's type theory with one universe
$P\omega = C$	Calculus of constructions

$$(S)\ \frac{\Gamma \vdash A : s \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x{:}A \vdash x : A} \qquad (W)\ \frac{\Gamma \vdash A{:}B \quad \Gamma \vdash C : s \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x{:}C \vdash A : B}$$

$$(\beta)\ \frac{\Gamma \vdash A{:}B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'} \qquad (A)\ \ \cdot \vdash * : \Box$$

$$(\to)\ \frac{\Gamma, x{:}C \vdash O : C'}{\Gamma \vdash \lambda x.O : \Pi x{:}C.\, C'} \qquad (\to)\ \frac{\Gamma \vdash O : (\Pi x{:}C.\, C') \quad \Gamma \vdash O' : C}{\Gamma \vdash O\, O' : C'\{x := O'\}}$$

$$(2)\ \frac{\Gamma, \alpha{:}K \vdash O : C'}{\Gamma \vdash \lambda \alpha.O : \Pi \alpha{:}K.\, C'} \qquad (2)\ \frac{\Gamma \vdash O : (\Pi \alpha{:}K.\, C') \quad \Gamma \vdash C : K}{\Gamma \vdash O\, C : C'\{\alpha := C\}}$$

$$(P)\ \frac{\Gamma, x{:}C \vdash C' : K}{\Gamma \vdash \lambda x.C' : \Pi x{:}C.\, K} \qquad (P)\ \frac{\Gamma \vdash C' : (\Pi x{:}C.\, K) \quad \Gamma \vdash O' : C}{\Gamma \vdash C'\, O' : K\{x := O'\}}$$

$$(\underline{\omega})\ \frac{\Gamma, \alpha{:}K \vdash C' : K'}{\Gamma \vdash \lambda \alpha.C' : \Pi \alpha{:}K.\, K'} \qquad (\underline{\omega})\ \frac{\Gamma \vdash C : (\Pi \alpha{:}K.\, K') \quad \Gamma \vdash C' : K}{\Gamma \vdash C\, C' : K'\{\alpha := C'\}}$$

$$(\to)\ \frac{\Gamma, x{:}C \vdash C' : *}{\Gamma \vdash (\Pi x{:}C.\, C') : *} \qquad (2)\ \frac{\Gamma, \alpha{:}K \vdash C' : *}{\Gamma \vdash (\Pi \alpha{:}K.\, C') : *}$$

$$(P)\ \frac{\Gamma, x{:}C \vdash K : \Box}{\Gamma \vdash (\Pi x{:}C.\, K) : \Box} \qquad (\underline{\omega})\ \frac{\Gamma, \alpha{:}K \vdash K' : \Box}{\Gamma \vdash (\Pi \alpha{:}K.\, K') : \Box}$$

Figure 5. Rules for the domain-free λ-cube.

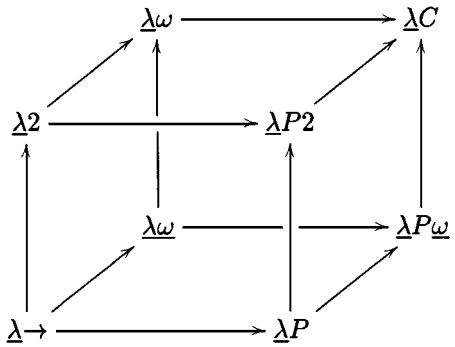

Figure 6. The domain-free λ-cube.

2.2. The systems at work

In this subsection we try to provide some intuitions behind the systems of the domain-free λ-cube, and illustrate the strength of each of the systems.

1. The simplest system $\underline{\lambda}\!\!\rightarrow$ consists of just the four rules (S), (W), (β), (A), along with the three ($\rightarrow$) rules. This system is very similar to the simply typed λ-calculus à la Curry introduced in Section 1.2.

   Both systems contain a *start* rule for introducing variables; in Section 1.2 this is the left-most rule, and in the present subsection it is the (S) rule. However, the two rules do not work in exactly the same way. In the formulation in Section 1.2, contexts are *sets*, and one can "look up" any member of the context is the start rule. In contrast, in the present subsection contexts are *ordered sequences* in which only the right-most member can be looked up in the start rule. However, using the weakening rule (W) one can look up variables further to the left in the context formulated by sequences.

   The side condition $x \notin \mathrm{dom}(\Gamma)$ in the present subsection is mirrored in Section 1.2 by the stipulation that Γ, Γ' only be defined when the domains of Γ and Γ' are disjoint.

   The side condition $\Gamma \Vdash A : s$ together with the axiom (A) and the product rule for ($\rightarrow$) (the one in the lower left corner) in the present subsection ensure that simple types, and only simple types, can occur in contexts. Since simple types have no redexes, it follows that the conversion rule (β) is not used in $\underline{\lambda}\!\!\rightarrow$, but it is used in dependent systems, e.g., $\underline{\lambda}P$—see below.

   One can show that whenever, $\Gamma, z : D, \Delta \Vdash A : B$ in the system $\underline{\lambda}\!\!\rightarrow$ where A, B, or D has form $\Pi x : C.C'$ then x is not free in C', i.e., $\Pi x : C.C' \equiv C \rightarrow C'$. It follows that the abstraction and application rule in this subsection are identical to those in Section 1.2.
2. The system $\underline{\lambda}2$ allows polymorphic functions to be defined. For example, one can define polymorphic composition

   $$\Vdash \lambda A, B, C,\ f, g, x\ .\ g\,(f\,x) : \Pi A, B, C : *\ .\ (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow A \rightarrow C$$

   or polymorphic projections, e.g.

   $$\Vdash \lambda A, B, x, y\ .\ x : \Pi A, B : *\ .\ A \rightarrow B \rightarrow A$$

   In logical terms, $\underline{\lambda}2$ corresponds to second-order propositional logic via the Curry-Howard isomorphism. In this context, the formalization of contexts as ordered sequences captures the usual side condition on the introduction rule for universal quantification.
3. The specification ω allows polymorphic and higher-order functions to be defined. For example, let

   $$\Gamma \equiv \mathsf{List} : * \rightarrow *, \mathsf{nil} : \Pi\alpha : *\ .\ \mathsf{List}\,\alpha, \mathsf{cons} : \Pi\alpha : *\ .\ (\alpha \rightarrow \mathsf{List}\,\alpha \rightarrow \mathsf{List}\,\alpha)$$

   $$\Gamma' \equiv \Gamma, \mathsf{MapList} : \Pi A, B : *\ .(A \rightarrow B) \rightarrow \mathsf{List}\ A \rightarrow \mathsf{List}\ B$$

One can derive a polymorphic function that makes every object into a one element list.

$$\Gamma \Vdash \lambda\alpha, a\,.\,\mathsf{cons}\;\alpha\;a(\mathsf{nil}\;\alpha) : \Pi\alpha : *.\,\alpha \to \mathsf{List}\;\alpha$$

or a polymorphic function which manipulates functions on lists:

$$\Gamma' \Vdash \lambda A, B, C,\ f, g, l\,.\,\mathsf{MapList}\;BCg\;(\mathsf{MapList}\;ABfl) :$$
$$\Pi A, B, C : *\,.\,(A \to B) \to (B \to C) \to \mathsf{List}\;A \to \mathsf{List}\;C$$

4. The specification P has the power of first-order predicate logic. The system $\underline{\lambda}P$, which is closely related to the type system of the proof-assistants Authomath, Alf and Elf, acts as a framework in which formal systems can be defined. The following example, taken from [33, Section 3.1], introduces first-order arithmetic as a context. There are two base types ι and o, which respectively correspond to arithmetic expressions and logical formulae. In addition, one introduces the usual operations on natural numbers and the usual logical connectives and quantifiers.

$$\begin{array}{lll}
\iota : *, & o : *, & \wedge : o \to o \to o,\\
0 : \iota, & = : \iota \to \iota \to o, & \vee : o \to o \to o,\\
s : \iota \to \iota, & < : \iota \to \iota \to o, & \supset : o \to o \to o,\\
+ : \iota \to \iota \to \iota, & \forall : (\iota \to o) \to o, & \neg : o \to o\\
\times : \iota \to \iota \to \iota, & \exists : (\iota \to o) \to o, &
\end{array}$$

This context provides enough structure for terms and formulae of first-order arithmetic to be encoded into the system. However, the context lacks enough structure to build or manipulate proofs. Following [33, Section 4.1], it is possible to extend the above context so that proofs may be encoded. The basic judgment from 'ϕ is a logical truth' is encoded in the context by adding a declaration

$$\mathsf{true} : o \to *$$

and the basic rules for logic are encoded in the context by adding suitable declarations. For example, the left introduction rule for disjunction is encoded by

$$\vee_l : \Pi x, y : o.\,(\mathsf{true}\;x) \to (\mathsf{true}(\vee xy))$$

Note how the type $o \to *$ and how the above declaration require the use of dependent types.

5. The specification $P2$ combines dependent types and polymorphism. In logical terms, $\underline{\lambda}P2$ has the power of second-order predicate logic with equality. Indeed, one may use polymorphism (sometimes known as impredicative quantification) to encode the usual connectives and quantifiers. The encoding is given below, where we let $A, B, P, T \in V^{\Box}$ and $x, y \in V^*$:

Operator	Definition	Type
Implication	$\to \equiv \lambda A, B.(A \to B)$	$* \to * \to *$
Universal quantifier	$\forall \equiv \lambda T, P.(\Pi x : T.Px)$	$\Pi T : * . (T \to *) \to *$
Truth	$\top \equiv \Pi x : *.x \to x$	$*$
Falsehood	$\bot \equiv \Pi x : *.x$	$*$
Conjunction	$\wedge \equiv \lambda A, B.(\Pi x : *.A \to B \to x)$	$* \to * \to *$
Disjunction	$\vee \equiv \lambda A, B(\Pi x : *. (A \to x) \to (B \to x) \to x)$	$* \to * \to *$
Negation	$\neg \equiv \lambda A.(A \to \bot)$	$* \to *$
Existential quantifier	$\exists \equiv \lambda A, P(\Pi p : *.(\Pi x : T.((Px) \to p) \to p)$	$\Pi T : * . (T \to *) \to *$
(Leibniz) equality	$= \equiv \lambda T, x, y.\Pi P : T \to *.(Px) \to (Py)$	$\Pi T : * . T \to T \to *$

Besides, one can define λ-terms that correspond to the standard natural deduction rules. For example, we have

$$\Vdash \lambda A, B, p\,.\, pA(\lambda x, y.x) : \Pi A, B : *.(\wedge AB) \to A$$

Note a fundamental difference between $\underline{\lambda}P$ and $\underline{\lambda}P2$: in $\underline{\lambda}P$, predicate logic is defined in a context, while in $\underline{\lambda}P2$, logic is present *in* the system and is not defined in a context.

6. The system $\underline{\lambda}P\omega = \underline{\lambda}C$ has the power of higher-order predicate logic. In $\underline{\lambda}C$, one can define, e.g. a polymorphic function that forms the conjunction of two predicates:

$$\Vdash \lambda A, P, Q, x.\wedge (Px)(Qx) : \Pi A : *.(A \to *) \to (A \to *) \to A \to *$$

where $\wedge$ is the impredicative conjunction.

3. CPS translation for the domain-free λ-cube

In this section we present CPS translations for all the systems of the domain-free λ-cube. This is carried out in the first subsection. In the second subsection we present optimizing CPS translations.

3.1. CPS translation for the λ-cube

When one moves from $\underline{\lambda}\to$ to systems with richer type structure, it is not immediately clear which syntactic categories should be converted by the CPS translation. For example, in the domain-free λ-cube, one might imagine a *non-pervasive* CPS translation where only objects are converted (i.e., only abstractions at the object level are passed continuations), or a *pervasive* translation where both objects and constructors are converted (i.e., all abstractions are passed continuations).

Our presentation, which is based on a non-pervasive translation, is motivated by particular applications. In classical pure type systems [6], the control operator corresponding to the

reductio ad absurdum rule only appears at the object level. Thus, one only needs to convert objects when embedding classical pure type systems into traditional pure type systems. Similarly, in Coquand and Herbelin's [15] method for showing the existence of looping combinators and in compilation and partial evaluation applications, one only needs to convert objects to obtain the associated benefits of CPS.

Definition 2 (*CPS translation*). Define the CPS translation functions $\mathcal{C}\langle\!|\cdot|\!\rangle$ and $\mathcal{C}\langle\cdot\rangle$ for the domain-free λ-cube by the clauses in figure 7.

As noted earlier, the translation of objects is based on Plotkin's original call-by-name translation of the untyped λ-calculus (see figure 1). Continuation-passing only occurs at the object level. Thus, the translation of constructors and kinds is straightforward. $\neg\mathcal{C}$ abbreviates $\mathcal{C} \to \bot$ where $\bot \in V^{\Box}$ is a distinguished (but arbitrary) type variable (of *answers*).

The top level translation $\mathcal{C}\langle\!|\cdot|\!\rangle$ for objects and kinds simply call $\mathcal{C}\langle\cdot\rangle$. For constructors, a double negation is added at the outer level. For assumptions, the answer variable $\bot$ is added.

The following theorem establishes the correctness of the translation.

Theorem 1 (Correctness of CPS translation).

$$\Gamma \Vdash A : B \Rightarrow \mathcal{C}\langle\!|\Gamma|\!\rangle \Vdash \mathcal{C}\langle A\rangle : \mathcal{C}\langle\!|B|\!\rangle$$

Proof sketch: We proceed in four steps:

(i) show that for all $O \in \mathit{Obj}[\mathsf{DFCUBE}]$, $\mathcal{C}\langle O\rangle \equiv \lambda k.O'$ for some O' and therefore $\lambda k.\mathcal{C}\langle O\rangle k \to_{\underline{\beta}} \mathcal{C}\langle O\rangle$;

(ii) prove by induction on the structure of $A \in \mathit{Terms}[\mathsf{DFCUBE}]$ that

  (a) $\mathcal{C}\langle A\rangle\{x := \mathcal{C}\langle O\rangle\} \twoheadrightarrow_{\underline{\beta}} \mathcal{C}\langle A\{x := O\}\rangle$ and
  (b) $\mathcal{C}\langle A\rangle\{\alpha := \mathcal{C}\langle C\rangle\} \equiv \mathcal{C}\langle A\{\alpha := C\}\rangle$;

(iii) prove by induction on the structure of $A \in \mathit{Terms}[\mathsf{DFCUBE}]$ that $A \twoheadrightarrow_{\underline{\beta}} B$ implies $\mathcal{C}\langle A\rangle \twoheadrightarrow_{\underline{\beta}} \mathcal{C}\langle B\rangle$;

(iv) prove $\Gamma \Vdash A : B \Rightarrow \mathcal{C}\langle\!|\Gamma|\!\rangle \Vdash \mathcal{C}\langle A\rangle : \mathcal{C}\langle\!|B|\!\rangle$ by induction on the structure of derivations.

□

The following example illustrates the CPS translation applied to the context and term that makes every object into a one element list (see Section 2.2).

$$\begin{aligned}\Gamma_{cps} \equiv\ & \mathsf{List} : * \to *,\\ & \mathsf{nil} : \neg\neg(\Pi\alpha : *.\neg\neg(\mathsf{List}\,\alpha)),\\ & \mathsf{cons} : \neg\neg(\Pi\alpha : *.\neg\neg(\neg\neg\alpha \to \neg\neg(\neg\neg(\mathsf{List}\,\alpha) \to \neg\neg(\mathsf{List}\,\alpha))))\end{aligned}$$

Objects

$$\begin{aligned}
\mathcal{C}\langle x\rangle &= \lambda k.x\ k\\
\mathcal{C}\langle \lambda x.O\rangle &= \lambda k.k\ (\lambda x.\mathcal{C}\langle O\rangle)\\
\mathcal{C}\langle O\ O'\rangle &= \lambda k.\mathcal{C}\langle O\rangle\ (\lambda y.y\ \mathcal{C}\langle O'\rangle\ k)\\
\mathcal{C}\langle \lambda \alpha.O\rangle &= \lambda k.k\ (\lambda \alpha.\mathcal{C}\langle O\rangle)\\
\mathcal{C}\langle O\ C\rangle &= \lambda k.\mathcal{C}\langle O\rangle\ (\lambda y.y\ \mathcal{C}\langle C\rangle\ k)
\end{aligned}$$

Constructors

$$\begin{aligned}
\mathcal{C}\langle \alpha\rangle &= \alpha\\
\mathcal{C}\langle \lambda x.C'\rangle &= \lambda x.\mathcal{C}\langle C'\rangle\\
\mathcal{C}\langle C\ O\rangle &= \mathcal{C}\langle C\rangle\ \mathcal{C}\langle O\rangle\\
\mathcal{C}\langle \lambda \alpha.C\rangle &= \lambda \alpha.\mathcal{C}\langle C\rangle\\
\mathcal{C}\langle C\ C'\rangle &= \mathcal{C}\langle C\rangle\ \mathcal{C}\langle C'\rangle\\
\mathcal{C}\langle \Pi x{:}C.\ C'\rangle &= \Pi x{:}\neg\neg\mathcal{C}\langle C\rangle.\ \neg\neg\mathcal{C}\langle C'\rangle\\
\mathcal{C}\langle \Pi \alpha{:}K.\ C\rangle &= \Pi \alpha{:}\mathcal{C}\langle K\rangle.\ \neg\neg\mathcal{C}\langle C\rangle
\end{aligned}$$

Kinds

$$\begin{aligned}
\mathcal{C}\langle *\rangle &= *\\
\mathcal{C}\langle \Pi x{:}C.\ K\rangle &= \Pi x{:}\neg\neg\mathcal{C}\langle C\rangle.\ \mathcal{C}\langle K\rangle\\
\mathcal{C}\langle \Pi \alpha{:}K.\ K'\rangle &= \Pi \alpha{:}\mathcal{C}\langle K\rangle.\ \mathcal{C}\langle K'\rangle
\end{aligned}$$

Contexts

$$\begin{aligned}
\mathcal{C}\langle \cdot\rangle &= \cdot\\
\mathcal{C}\langle \Gamma, x{:}C\rangle &= \mathcal{C}\langle \Gamma\rangle, x{:}\neg\neg\mathcal{C}\langle C\rangle\\
\mathcal{C}\langle \Gamma, \alpha{:}K\rangle &= \mathcal{C}\langle \Gamma\rangle, \alpha{:}\mathcal{C}\langle K\rangle
\end{aligned}$$

Top-level translation

$$\begin{aligned}
\mathcal{C}\langle\!\langle O\rangle\!\rangle &= \mathcal{C}\langle O\rangle\\
\mathcal{C}\langle\!\langle C\rangle\!\rangle &= \neg\neg\mathcal{C}\langle C\rangle\\
\mathcal{C}\langle\!\langle K\rangle\!\rangle &= \mathcal{C}\langle K\rangle\\
\mathcal{C}\langle\!\langle \Box\rangle\!\rangle &= \Box\\
\mathcal{C}\langle\!\langle \Gamma\rangle\!\rangle &= \bot{:}*, \mathcal{C}\langle \Gamma\rangle
\end{aligned}$$

Figure 7. Call-by-name CPS translation for the domain-free λ-cube.

$$\Gamma_{cps} \Vdash \lambda k.k(\lambda \alpha.\lambda k.k(\lambda a.O_1)) : \neg\neg(\Pi \alpha : *.\neg\neg(\neg\neg\alpha \to \neg\neg(\mathsf{List}\ \alpha)))$$

where
$$\begin{aligned}
O_1 &\equiv \lambda k.O_2\ (\lambda y_1.y_1 O_3 k)\\
O_2 &\equiv \lambda k.O_4\ (\lambda y_2.y_2(\lambda k.ak)k)\\
O_3 &\equiv \lambda k.(\lambda k.\mathsf{nil}\ k)(\lambda y_3.y_3 \alpha k)\\
O_4 &\equiv \lambda k.(\lambda k.\mathsf{cons}\ k)(\lambda y_4.y_4 \alpha k)
\end{aligned}$$

Note that there is a higher degree of continuation-passing than one would obtain when CPS-ing a conventional call-by-value program for the following reasons.

- A call-by-name evaluation strategy is being encoded, and thus all identifiers are passed continuations (i.e., they are "CPS thunks").
- Operations such as cons and nil are given in curried form, and thus continuations are introduced at each function space.

Recent work on *monadic type systems* [11] provides a framework where one can distinguish between values and non-values in an extended type system. A CPS translation based on this richer type system allows a finer control over continuation introduction, and one can give a specification that avoids the overly general introduction of continuations in this particular example.

3.2. An optimizing CPS translation

When discussing terms in the image of the CPS translation, it is convenient to divide abstractions into two classes:

- *source abstractions*: these are CPS-translated versions of abstractions appearing in the argument of the CPS translation. In figure 7, abstractions of the form $(\lambda x \ldots)$ and $(\lambda \alpha \ldots)$ in the image of the translation are source abstractions.
- *administrative abstractions*: these are abstractions introduced by the translation to manipulate continuations—there are no corresponding abstractions in the argument of the translation. In figure 7, abstractions of the form $(\lambda k \ldots)$ and $(\lambda y \ldots)$ are administrative abstractions.

Reducing a CPS term involves contracting many *administrative redexes*—redexes involving administrative abstractions [52, p. 149]. We write $A \to_{adm} A'$ when A β-reduces to A' by contracting an administrative redex.

Most practical applications of CPS (such as compiling and partial evaluation) use an optimizing version of a particular CPS translation that produces terms with fewer administrative redexes [18, 26, 52, 58, 68]. In this section, we present an optimizing version of the translation in figure 7 that yields terms in *administrative normal-form* (i.e., the terms contain no administrative redexes). We need to consider the CPS translation of objects only, since continuations are introduced only in this category.

Definition 3 (*CPS translation*). Define the optimizing CPS translation functions $\mathcal{C}^+[\![\cdot]\!]$ and $\mathcal{C}^+\langle\cdot\rangle$ for the domain-free λ-cube by the clauses in figure 8.

Figure 8 only presents the translation on objects; translation of the remaining syntactic categories is the same as in figure 7. $O : \mathcal{K}$ represents the translation of an object where $\mathcal{K}$ is a continuation corresponding to the evaluation context in which O appeared. The translation proceeds recursively over O—transforming it into a continuation with no administrative redexes. Depending on the form of O, the resulting continuation is either, passed as an argument to the translation of O (if O is a variable), or applied to the translation of O (if O is an abstraction).

$$
\begin{aligned}
\mathcal{C}^+\langle O\rangle &= \lambda k.(O : k)\\
\\
x : \mathcal{K} &= x\,\mathcal{K}\\
(\lambda x.O) : k &= k\,(\lambda x.\mathcal{C}^+\langle O\rangle)\\
(\lambda \alpha.O) : k &= k\,(\lambda \alpha.\mathcal{C}^+\langle O\rangle)\\
(\lambda x.O) : (\lambda y.y\,O'\,\mathcal{K}) &= (\lambda x.\mathcal{C}^+\langle O\rangle)\,O'\,\mathcal{K}\\
(\lambda \alpha.O) : (\lambda y.y\,C\,\mathcal{K}) &= (\lambda \alpha.\mathcal{C}^+\langle O\rangle)\,C\,\mathcal{K}\\
O\,O' : \mathcal{K} &= O : (\lambda y.y\,\mathcal{C}^+\langle O'\rangle\,\mathcal{K})\\
O\,C : \mathcal{K} &= O : (\lambda y.y\,\mathcal{C}^+\langle C\rangle\,\mathcal{K})
\end{aligned}
$$

Figure 8. Optimizing CPS translation for the domain-free λ-cube (excerpts).

Lemma 1 (Optimizing CPS translation for the λ-cube). *For all $A \in$ Terms*[CUBE] *such that $\Gamma \Vdash A : B$,*

- $\mathcal{C}\langle A\rangle \twoheadrightarrow_{adm} \mathcal{C}^+\langle A\rangle$
- $\mathcal{C}^+\langle A\rangle$ *is in administrative normal-form.*

The following example illustrates the optimizing CPS translation applied to the term that makes every object into a one element list (the context and type are the same as for the example for the non-optimizing translation).

$$\Gamma_{cps} \Vdash \lambda k.k(\lambda\alpha.\lambda k.k(\lambda a.O_1)) : \neg\neg(\Pi\alpha : *\,.\neg\neg(\neg\neg\alpha \to \neg\neg(\mathsf{List}\,\alpha)))$$

where
$$
\begin{aligned}
O_1 &\equiv \lambda k.O_2\\
O_2 &\equiv \mathsf{cons}(\lambda y_4.y_4\,\alpha(\lambda y_2.y_2(\lambda k.a\,k)(\lambda y_1.y_1\,O_3 k)))\\
O_3 &\equiv \lambda k.\mathsf{nil}(\lambda y_3.y_3 \alpha k)
\end{aligned}
$$

In contrast to the analogous term produced by the non-optimizing translation, the term above has no administrative redexes.

4. DS translation for the domain-free λ-cube

In this section, we present a DS translation for the domain-free λ-cube that maps continuation-passing terms back to direct-style terms. Since the rules of the λ-cube involve β-conversion, the DS translation must be able to handle not only terms in the image of the CPS translation, but terms β-convertible with terms in the image of the CPS translation. Since systems of the λ-cube are Church-Rosser, it is sufficient to reason about the language of legal terms in the image of CPS translation closed under β-reduction. The first subsection below presents CPS pseudo-terms, and then gives rules for deriving legal CPS terms. The DS translation is then defined in the following subsection. The last subsection addresses the relation between the CPS and DS translations.

4.1. *Language of CPS terms*

We define the DS translation of the language CPS of CPS terms.

Definition 4 (CPS *pseudo-terms and contexts*). Define the pseudo-terms and context by the grammar in figure 9.

Objects are divided into the following five categories.

- *computations*: these are pseudo-objects to which continuations will be passed. The types of these objects will be double-negated at the top level.
- *values*: these are the source abstractions defined above. The types of these objects will not be double-negated at the top-level.

Objects

$$\begin{array}{lll} \mathcal{M} \in \mathit{Computations}[\mathsf{CPS}] & & \mathcal{A} \in \mathit{Answers}[\mathsf{CPS}] \\ \mathcal{M} ::= x \mid \lambda k.\mathcal{A} \mid \mathcal{V}\,\mathcal{N} & & \mathcal{A} ::= \mathcal{K}\,\mathcal{V} \mid \mathcal{M}\,\mathcal{K} \\ \\ \mathcal{V} \in \mathit{Values}[\mathsf{CPS}] & & \mathcal{K} \in \mathit{Continuations}[\mathsf{CPS}] \\ \mathcal{V} ::= \lambda x.\lambda k.\mathcal{A} \mid \lambda\alpha.\lambda k.\mathcal{A} & & \mathcal{K} ::= k \mid \lambda y.y\,\mathcal{N}\,\mathcal{K} \\ \\ \mathcal{N} \in \mathit{Arguments}[\mathsf{CPS}] & & \\ \mathcal{N} ::= \lambda k.\mathcal{A} \mid C & & \end{array}$$

Constructors and Kinds

$$\begin{array}{l} C \in \mathit{Constr}[\mathsf{CPS}] \\ C ::= \alpha \mid \lambda x.C_2 \mid \lambda\alpha.C_2 \mid C\,\mathcal{N} \mid \\ \qquad \Pi x{:}\neg\neg C_1.\,\neg\neg C_2 \mid \Pi\alpha{:}K.\,\neg\neg C \\ \\ K \in \mathit{Kind}[\mathsf{CPS}] \\ K ::= * \mid \Pi x{:}\neg\neg C.\,K \mid \Pi\alpha{:}K_1.\,K_2 \end{array}$$

Identifiers

$$\begin{array}{lll} x \in \mathit{Computation\text{-}ident}[\mathsf{CPS}] & & \subseteq V^* \\ y \in \mathit{Value\text{-}ident}[\mathsf{CPS}] & & \subseteq V^* \\ \alpha \in \mathit{Constructor\text{-}ident}[\mathsf{CPS}] & & \subseteq V^{\square} \\ k \in \mathit{Continuation\text{-}ident}[\mathsf{CPS}] = \{k\} & & \subseteq V^* \end{array}$$

Contexts

$$\begin{array}{l} \Gamma \in \mathit{Contexts}[\mathsf{CPS}] \\ \Gamma ::= \bot{:}* \mid \Gamma, x{:}\neg\neg C \mid \Gamma, \alpha{:}K \end{array}$$

Figure 9. CPS pseudo-terms.

The terms *computation* and *value* are inspired by presentations of CPS based on monads [39]. In a monadic framework, our computations have "computational types", and our values have "value types".

- *arguments*: these will be the arguments to source abstractions.
- *answers*: these are pseudo-objects that will have answer type $\perp$. An answer results from passing a value to continuation, or passing a continuation to a computation.
- *continuations*: These are either continuation identifiers, or abstractions that will be passed values.

It is convenient to divide identifiers of the CPS language into four disjoint sets:

- *computation identifiers*: bound by source abstractions that take computations as arguments;
- *value identifiers*: bound by administrative abstractions that take values as arguments—they are the formal parameters of continuations;
- *constructor identifiers*: bound by source abstractions that take constructors as arguments;
- *continuation identifiers*: bound by administrative abstractions that take continuations as arguments.

The first three sets of identifiers are countably infinite. Only one continuation identifier k is needed—a familiar property of CPS terms [17, 19, 58].

CPS contexts always include the answer variable $\perp$. CPS contexts only contain constructor and computation identifiers (identifiers bound by source abstractions). It is unnecessary for contexts to contain value identifiers y and continuation identifiers k (identifiers bound by administrative abstractions). These are handled as special cases when defining legal terms.

- Value identifiers are bound immediately after they are introduced, so that don't need to appear in the contexts (see the last rule in figure 10).
- Since only one continuation identifier is needed, and since no types can depend on it, we define special purpose judgments $\langle\Gamma\rangle\ \langle k : \neg C\rangle \Vdash_\phi A : B$ for $\phi \in \{ans, cnt\}$ that keep the continuation identifier separate from the rest of the context.

Definition 5 (CPS *legal terms and contexts*). Figures 10–12 present rules for deriving legal terms in each syntactic category of the CPS language.

In the judgments for answers and continuations in figures 10 and 11, continuation identifiers k are given special treatment in the context. In addition to the rules for specific categories, figure 12 gives generic weakening and conversion rules which apply to each syntactic category. Since all of the conversion rules have a similar form, we only show cases.

The following property states that legal terms in CPS are also legal terms in the domain-free cube.

Computations

$$\frac{\Gamma \vDash_{\text{con}} C : *}{\Gamma, x{:}\neg\neg C \vDash_{\text{com}} x : \neg\neg C} \quad \text{if } x \notin \Gamma \text{ and } x \in \textit{Computation-ident}[\mathsf{CPS}]$$

$$\frac{\langle\Gamma\rangle\,\langle k{:}\neg C\rangle \vDash_{\text{ans}} \mathcal{A} : \bot}{\Gamma \vDash_{\text{com}} \lambda k.\mathcal{A} : \neg\neg C}$$

$$\frac{\Gamma \vDash_{\text{val}} \mathcal{V} : \Pi\alpha{:}K_1.\,\neg\neg C_2 \qquad \Gamma \vDash_{\text{arg}} C_1 : K_1}{\Gamma \vDash_{\text{com}} \mathcal{V}\,C_1 : \neg\neg C_2\{\alpha := C_1\}}$$

$$\frac{\Gamma \vDash_{\text{val}} \mathcal{V} : \Pi x{:}\neg\neg C_1.\,\neg\neg C_2 \qquad \Gamma \vDash_{\text{arg}} (\lambda k.\mathcal{A}) : \neg\neg C_1}{\Gamma \vDash_{\text{com}} \mathcal{V}\,(\lambda k.\mathcal{A}) : \neg\neg C_2\{x := \lambda k.\mathcal{A}\}}$$

Values

$$\frac{\Gamma, x{:}\neg\neg C_1 \vDash_{\text{com}} \lambda k.\mathcal{A} : \neg\neg C_2 \qquad \Gamma \vDash_{\text{con}} \Pi x{:}\neg\neg C_1.\,\neg\neg C_2 : *}{\Gamma \vDash_{\text{val}} \lambda x.\lambda k.\mathcal{A} : \Pi x{:}\neg\neg C_1.\,\neg\neg C_2}$$

$$\frac{\Gamma, \alpha{:}K_1 \vDash_{\text{com}} \lambda k.\,\bot : \neg\neg C_2 \qquad \Gamma \vDash_{\text{con}} \Pi\alpha{:}K_1.\,\neg\neg C_2 : *}{\Gamma \vDash_{\text{val}} \lambda\alpha.\lambda k.\mathcal{A} : \Pi\alpha{:}K_1.\,\neg\neg C_2}$$

Arguments

$$\frac{\langle\Gamma\rangle\,\langle k{:}\neg C\rangle \vDash_{\text{ans}} \mathcal{A} : \bot}{\Gamma \vDash_{\text{arg}} \lambda k.\mathcal{A} : \neg\neg C} \qquad \frac{\Gamma \vDash_{\text{con}} C : K}{\Gamma \vDash_{\text{arg}} C : K}$$

Answers

$$\frac{\langle\Gamma\rangle\,\langle k{:}\neg C_1\rangle \vDash_{\text{cnt}} \mathcal{K} : \neg C_2 \qquad \Gamma \vDash_{\text{val}} \mathcal{V} : C_2}{\langle\Gamma\rangle\,\langle k{:}\neg C_1\rangle \vDash_{\text{ans}} \mathcal{K}\,\mathcal{V} : \bot}$$

$$\frac{\Gamma \vDash_{\text{com}} \mathcal{M} : \neg\neg C_2 \qquad \langle\Gamma\rangle\,\langle k{:}\neg C_1\rangle \vDash_{\text{cnt}} \mathcal{K} : \neg C_2}{\langle\Gamma\rangle\,\langle k{:}\neg C_1\rangle \vDash_{\text{ans}} \mathcal{K}\,\mathcal{V} : \bot}$$

Figure 10. CPS legal objects.

(*Continued on next page.*)

Continuations

$$\frac{\Gamma \Vdash_{\mathrm{con}} C : *}{\langle\Gamma\rangle\,\langle k{:}\neg C\rangle \Vdash_{\mathrm{cnt}} k : \neg C}$$

$$\frac{\begin{array}{l}\Gamma \Vdash_{\mathrm{con}} \Pi x{:}\neg\neg C_1.\ \neg\neg C_2 : *\\ \Gamma \Vdash_{\mathrm{arg}} \lambda k.\mathcal{A} : \neg\neg C_1\\ \langle\Gamma\rangle\,\langle k{:}\neg C_0\rangle \Vdash_{\mathrm{cnt}} \mathcal{K} : \neg C_2\{x := (\lambda k.\mathcal{A})\}\end{array}}{\begin{array}{l}\langle\Gamma\rangle\,\langle k{:}\neg C_0\rangle \Vdash_{\mathrm{cnt}} (\lambda y.y\ (\lambda k.\mathcal{A})\ \mathcal{K}) : \neg\Pi x{:}\neg\neg C_1.\ \neg\neg C_2\\ \qquad\text{if } y \notin \Gamma \text{ and } y \in \textit{Computation-ident}[\mathsf{CPS}]\end{array}}$$

$$\frac{\begin{array}{l}\Gamma \Vdash_{\mathrm{con}} \Pi\alpha{:}K_1.\ \neg\neg C_2 : *\\ \Gamma \Vdash_{\mathrm{arg}} C_1 : K_1\\ \langle\Gamma\rangle\,\langle k{:}\neg C_0\rangle \Vdash_{\mathrm{cnt}} \mathcal{K} : \neg C_2\{\alpha := C_1\}\end{array}}{\begin{array}{l}\langle\Gamma\rangle\,\langle k{:}\neg C_0\rangle \Vdash_{\mathrm{cnt}} (\lambda y.y\ C_1\ \mathcal{K}) : \neg\Pi\alpha{:}K_1.\ \neg\neg C_2\\ \qquad\text{if } y \notin \Gamma \text{ and } y \in \textit{Computation-ident}[\mathsf{CPS}]\end{array}}$$

Figure 10. (*Continued*).

Property 1.

$$\Gamma \Vdash_\theta A : B \Rightarrow \Gamma \Vdash A : B \qquad \theta \in \{com, val, arg, con, knd\}$$
$$\langle\Gamma\rangle\langle k : \neg C\rangle \Vdash_\phi A : B \Rightarrow \Gamma, k : \neg C \Vdash A : B \qquad \phi \in \{ans, cnt\}$$

The following property states that the CPS language does indeed contain the image of the CPS translation.

Property 2.

$$\Gamma \Vdash O : C \Rightarrow \mathcal{C}(\!|\Gamma|\!) \Vdash_{com} \mathcal{C}\langle O\rangle : \mathcal{C}(\!|C|\!)$$
$$\Gamma \Vdash C : K \Rightarrow \mathcal{C}(\!|\Gamma|\!) \Vdash_{con} \mathcal{C}\langle C\rangle : \mathcal{C}(\!|K|\!)$$
$$\Gamma \Vdash K : \Box \Rightarrow \mathcal{C}(\!|\Gamma|\!) \Vdash_{knd} \mathcal{C}\langle K\rangle : \mathcal{C}(\!|\Box|\!)$$

The following property states that legal terms in the CPS language are closed under reduction.

Property 3.

$$\Gamma \Vdash_\theta A : B \text{ and } A \to_{\underline{\beta}} A' \Rightarrow \Gamma \Vdash_\theta A' : B \qquad \theta \in \{com, val, arg, con, knd\}$$
$$\langle\Gamma\rangle\langle k : \neg C\rangle \Vdash_\phi A : B \text{ and } A \to_{\underline{\beta}} A' \Rightarrow \langle\Gamma\rangle\langle k : \neg C\rangle \Vdash_\phi A' : B \qquad \phi \in \{ans, cnt\}$$

Constructors

$$\frac{\Gamma \vdash_{\mathrm{knd}} K : \Box}{\Gamma, \alpha{:}K \vdash_{\mathrm{con}} \alpha : K} \quad \text{if } \alpha \notin \Gamma \text{ and } \alpha \in \textit{Constructor-ident}[\mathsf{CPS}]$$

$$\frac{\Gamma, x{:}\neg\neg C_1 \vdash_{\mathrm{con}} C_2 : K_2 \qquad \Gamma \vdash_{\mathrm{knd}} \Pi x{:}\neg\neg C_1.\, K_2 : \Box}{\Gamma \vdash_{\mathrm{con}} \lambda x.C_2 : \Pi x{:}\neg\neg C_1.\, K_2}$$

$$\frac{\Gamma, \alpha{:}K_1 \vdash_{\mathrm{con}} C_2 : K_2 \qquad \Gamma \vdash_{\mathrm{knd}} \Pi\alpha{:}K_1.\, K_2 : \Box}{\Gamma \vdash_{\mathrm{con}} \lambda\alpha.C_2 : \Pi\alpha{:}K_2.\, K_2}$$

$$\frac{\Gamma \vdash_{\mathrm{con}} C_0 : \Pi x{:}\neg\neg C_1.\, K_2 \qquad \Gamma \vdash_{\mathrm{com}} \lambda k.\mathcal{A} : \neg\neg C_1}{\Gamma \vdash_{\mathrm{con}} C_0\ (\lambda k.\mathcal{A}) : K_2\{x := \lambda k.\mathcal{A}\}}$$

$$\frac{\Gamma \vdash_{\mathrm{con}} C_0 : \Pi\alpha{:}K_1.\, K_2 \qquad \Gamma \vdash_{\mathrm{com}} C_1 : K_1}{\Gamma \vdash_{\mathrm{con}} C_0\ C_1 : K_2\{\alpha := C_1\}}$$

$$\frac{\Gamma \vdash_{\mathrm{con}} C_1 : * \qquad \Gamma, x{:}\neg\neg C_1 \vdash_{\mathrm{con}} C_2 : *}{\Gamma \vdash_{\mathrm{con}} \Pi x{:}\neg\neg C_1.\, \neg\neg C_2 : *}$$

$$\frac{\Gamma \vdash_{\mathrm{con}} K_1 : \Box \qquad \Gamma, \alpha{:}K_1 \vdash_{\mathrm{con}} C_2 : *}{\Gamma \vdash_{\mathrm{con}} \Pi\alpha{:}K_1.\, \neg\neg C_2 : *}$$

Kinds

$$\bot : * \vdash_{\mathrm{knd}} * : \Box$$

$$\frac{\Gamma \vdash_{\mathrm{con}} C : * \qquad \Gamma, x{:}\neg\neg C \vdash_{\mathrm{knd}} K : \Box}{\Gamma \vdash_{\mathrm{knd}} \Pi x{:}\neg\neg C.\, K : \Box}$$

$$\frac{\Gamma \vdash_{\mathrm{knd}} K_1 : \Box \qquad \Gamma, \alpha{:}K_1 \vdash_{\mathrm{knd}} K_2 : \Box}{\Gamma \vdash_{\mathrm{knd}} \Pi\alpha{:}K_1.\, K_2 : \Box}$$

Figure 11. CPS legal constructors and kinds.

Weakening

$$\frac{\Gamma \Vdash_\theta A : B \qquad \Gamma \Vdash_{\mathrm{knd}} K : \Box}{\Gamma, \alpha{:}K \Vdash_\theta A : B}$$
if $\alpha \notin \Gamma$ and $\alpha \in$ *Constructor-ident*[CPS]

$$\frac{\langle\Gamma\rangle\,\langle k{:}\neg C_0\rangle \Vdash_\phi A : B \qquad \Gamma \Vdash_{\mathrm{knd}} K : \Box}{\langle\Gamma, \alpha{:}K\rangle\,\langle k{:}\neg C_0\rangle \Vdash_\phi A : B}$$
if $\alpha \notin \Gamma$ and $\alpha \in$ *Constructor-ident*[CPS]

$$\frac{\Gamma \Vdash_\theta A : B \qquad \Gamma \Vdash_{\mathrm{con}} C : *}{\Gamma, x{:}\neg\neg C \Vdash_\theta A : B}$$
if $x \notin \Gamma$ and $x \in$ *Computation-ident*[CPS]

$$\frac{\langle\Gamma\rangle\,\langle k{:}\neg C_0\rangle \Vdash_\phi A : B \qquad \Gamma \Vdash_{\mathrm{knd}} K : \Box}{\langle\Gamma, x{:}\neg\neg C\rangle\,\langle k{:}\neg C_0\rangle \Vdash_\phi A : B}$$
if $x \notin \Gamma$ and $x \in$ *Computation-ident*[CPS]

Conversion: (excerpts)

$$\frac{\Gamma \Vdash_{\mathrm{con}} C : K \qquad \Gamma \Vdash_{\mathrm{knd}} K' : \Box}{\Gamma \Vdash_{\mathrm{con}} C : K'} \text{ if } K =_{\underline{\beta}} K'$$

$$\frac{\Gamma \Vdash_{\mathrm{com}} \mathcal{M} : \neg\neg C \qquad \Gamma \Vdash_{\mathrm{con}} C' : *}{\Gamma \Vdash_{\mathrm{com}} \mathcal{M} : \neg\neg C'} \text{ if } C =_{\underline{\beta}} C'$$

$\theta \in \{com, val, arg, con, knd\}$
$\phi \in \{ans, cnt\}$

Figure 12. CPS weakening and conversion rules.

4.2. DS translation for the λ-cube

Figure 13 presents a DS translation from the domain-free CPS language to the domain-free λ-cube. For objects, the most interesting aspect is that continuations are translated to call-by-name *evaluation contexts* [23] defined by the following grammar.

$E \in$ *Evaluation-Contexts*[CUBE]
$E ::= [\cdot] \mid E\,O \mid E\,C$

Objects

$$\begin{aligned}
\mathcal{D}\langle x\rangle_{\mathrm{com}} &= x \\
\mathcal{D}\langle \lambda k.\mathcal{A}\rangle_{\mathrm{com}} &= \mathcal{D}\langle \mathcal{A}\rangle_{\mathrm{ans}} \\
\mathcal{D}\langle \mathcal{V}\,\mathcal{N}\rangle_{\mathrm{com}} &= \mathcal{D}\langle \mathcal{V}\rangle_{\mathrm{val}}\,\mathcal{D}\langle \mathcal{N}\rangle_{\mathrm{arg}} \\
\\
\mathcal{D}\langle \lambda x.\lambda k.\mathcal{A}\rangle_{\mathrm{val}} &= \lambda x.\mathcal{D}\langle \lambda k.\mathcal{A}\rangle_{\mathrm{com}} \\
\mathcal{D}\langle \lambda \alpha.\lambda k.\mathcal{A}\rangle_{\mathrm{val}} &= \lambda \alpha.\mathcal{D}\langle \lambda k.\mathcal{A}\rangle_{\mathrm{com}} \\
\\
\mathcal{D}\langle \lambda k.\mathcal{A}\rangle_{\mathrm{arg}} &= \mathcal{D}\langle \lambda k.\mathcal{A}\rangle_{\mathrm{com}} \\
\mathcal{D}\langle C\rangle_{\mathrm{arg}} &= \mathcal{D}\langle C\rangle_{\mathrm{con}} \\
\\
\mathcal{D}\langle \mathcal{K}\,\mathcal{V}\rangle_{\mathrm{ans}} &= \mathcal{D}\langle \mathcal{K}\rangle_{\mathrm{cnt}}[\mathcal{D}\langle \mathcal{V}\rangle_{\mathrm{val}}] \\
\mathcal{D}\langle \mathcal{M}\,\mathcal{K}\rangle_{\mathrm{ans}} &= \mathcal{D}\langle \mathcal{K}\rangle_{\mathrm{cnt}}[\mathcal{D}\langle \mathcal{M}\rangle_{\mathrm{com}}] \\
\\
\mathcal{D}\langle k\rangle_{\mathrm{cnt}} &= [\cdot] \\
\mathcal{D}\langle \lambda y.y\,\mathcal{N}\,\mathcal{K}\rangle_{\mathrm{cnt}} &= \mathcal{D}\langle \mathcal{K}\rangle_{\mathrm{cnt}}[[\cdot]\,\mathcal{D}\langle \mathcal{N}\rangle_{\mathrm{arg}}]
\end{aligned}$$

Constructors

$$\begin{aligned}
\mathcal{D}\langle \alpha\rangle_{\mathrm{con}} &= \alpha \\
\mathcal{D}\langle \lambda x.C_2\rangle_{\mathrm{con}} &= \lambda x.\mathcal{D}\langle C_2\rangle_{\mathrm{con}} \\
\mathcal{D}\langle C_0\ (\lambda k.\mathcal{A})\rangle_{\mathrm{con}} &= \mathcal{D}\langle C_0\rangle_{\mathrm{con}}\,\mathcal{D}\langle \lambda k.\mathcal{A}\rangle_{\mathrm{com}} \\
\mathcal{D}\langle \lambda \alpha.C\rangle_{\mathrm{con}} &= \lambda \alpha.\mathcal{D}\langle C\rangle_{\mathrm{con}} \\
\mathcal{D}\langle C_1\ C_2\rangle_{\mathrm{con}} &= \mathcal{D}\langle C_1\rangle_{\mathrm{con}}\,\mathcal{D}\langle C_2\rangle_{\mathrm{con}} \\
\mathcal{D}\langle \Pi x{:}\neg\neg C_1.\,\neg\neg C_2\rangle_{\mathrm{con}} &= \Pi x{:}\mathcal{D}\langle C_1\rangle_{\mathrm{con}}.\,\mathcal{D}\langle C_2\rangle_{\mathrm{con}} \\
\mathcal{D}\langle \Pi \alpha{:}K.\,\neg\neg C\rangle_{\mathrm{con}} &= \Pi \alpha{:}\mathcal{D}\langle K\rangle_{\mathrm{knd}}.\,\mathcal{D}\langle C\rangle_{\mathrm{con}}
\end{aligned}$$

Kinds

$$\begin{aligned}
\mathcal{D}\langle \Pi x{:}\neg\neg C.\,K\rangle_{\mathrm{knd}} &= \Pi x{:}\mathcal{D}\langle C\rangle_{\mathrm{con}}.\,\mathcal{D}\langle K\rangle_{\mathrm{knd}} \\
\mathcal{D}\langle \Pi \alpha{:}K_1.\,K_2\rangle_{\mathrm{knd}} &= \Pi \alpha{:}\mathcal{D}\langle K_1\rangle_{\mathrm{knd}}.\,\mathcal{D}\langle K_2\rangle_{\mathrm{knd}} \\
\mathcal{D}\langle *\rangle_{\mathrm{knd}} &= *
\end{aligned}$$

Contexts

$$\begin{aligned}
\mathcal{D}\langle \bot{:}*\rangle &= \cdot \\
\mathcal{D}\langle \Gamma, x{:}\neg\neg C\rangle &= \mathcal{D}\langle \Gamma\rangle, x{:}\mathcal{D}\langle C\rangle_{\mathrm{con}} \\
\mathcal{D}\langle \Gamma, \alpha{:}K\rangle &= \mathcal{D}\langle \Gamma\rangle, \alpha{:}\mathcal{D}\langle K\rangle_{\mathrm{knd}}
\end{aligned}$$

Top-level translation

$$\begin{aligned}
\mathcal{D}\langle\!| O |\!\rangle &= \mathcal{D}\langle O\rangle \\
\mathcal{D}\langle\!| \neg\neg C |\!\rangle &= \mathcal{D}\langle C\rangle \\
\mathcal{D}\langle\!| K |\!\rangle &= \mathcal{D}\langle K\rangle \\
\mathcal{D}\langle\!| \Box |\!\rangle &= \Box \\
\mathcal{D}\langle\!| \Gamma |\!\rangle &= \mathcal{D}\langle \Gamma\rangle
\end{aligned}$$

Figure 13. DS translation for the λ-cube.

The holes in the evaluation contexts are filled during the translation of the CPS *answer* syntactic category. The translation of constructors and kinds is straightforward—double negations are removed and translation continues on substructures.

The following example illustrates the interaction between the CPS and DS translation for the term $(\lambda x.O_1)O_2C_1$. An easy induction over the structure of terms shows that $\mathcal{D}$ is the inverse of $\mathcal{C}$.

The calculation below illustrates the interaction between the translations $\mathcal{C}$ and $\mathcal{D}$ for arbitrary terms O_1, O_2, and C_1.

$$
\begin{aligned}
&\mathcal{D}\langle\mathcal{C}\langle((\lambda x.O_1)O_2C_1)\rangle\rangle \\
&\quad =_{\underline{\beta}} \mathcal{D}\langle(\lambda k_2.(\lambda k_0.k_0(\lambda x.\mathcal{C}\langle O_1\rangle))(\lambda y_1.y_1\mathcal{C}\langle O_2\rangle(\lambda y_2.y_2\mathcal{C}\langle C_1\rangle k_2)))\rangle_{\mathrm{com}} \\
&\quad =_{\underline{\beta}} \mathcal{D}\langle((\lambda k_0.k_0(\lambda x.\mathcal{C}\langle O_1\rangle))(\lambda y_1.y_1\mathcal{C}\langle O_2\rangle(\lambda y_2.y_2\mathcal{C}\langle C_1\rangle k_2)))\rangle_{\mathrm{ans}} \\
&\quad =_{\underline{\beta}} \mathcal{D}\langle(\lambda y_1.y_1\mathcal{C}\langle O_2\rangle(\lambda y_2.y_2\mathcal{C}\langle C_1\rangle k_2))\rangle_{\mathrm{cnt}}[\mathcal{D}\langle(\lambda k_0.k_0(\lambda x.\mathcal{C}\langle O_1\rangle))\rangle_{\mathrm{com}}] \\
&\quad \ldots \\
&\quad =_{\underline{\beta}} \mathcal{D}\langle(\lambda y_1.y_1\mathcal{C}\langle O_2\rangle(\lambda y_2.y_2\mathcal{C}\langle C_1\rangle k_2))\rangle_{\mathrm{cnt}}[(\lambda x.O_1)] \\
&\quad =_{\underline{\beta}} \mathcal{D}\langle(\lambda y_2.y_2\mathcal{C}\langle C_1\rangle k_2)\rangle_{\mathrm{cnt}}[(\lambda x.O_1)O_2] \\
&\quad =_{\underline{\beta}} \mathcal{D}\langle k_2\rangle_{\mathrm{cnt}}[(\lambda x.O_1)O_2C_1] \\
&\quad =_{\underline{\beta}} (\lambda x.O_1)O_2C_1
\end{aligned}
$$

Note that in contrast to the CPS translation, the DS translation can be defined by induction over the structure of pseudo-terms even when mapping to domain-full systems. Since no abstractions (e.g., abstractions analogous to the administrative abstractions of the CPS translations) are introduced during the DS translation, all required domain tags can be constructed by translating tags appearing on abstractions in the argument of the translation.

Theorem 2 (Correctness of DS translation).

$$\Gamma \Vdash_\theta A : B \Rightarrow \mathcal{D}\langle\!|\Gamma|\!\rangle \Vdash \mathcal{D}\langle A\rangle : \mathcal{D}\langle\!|B|\!\rangle$$

for $\theta \in \{com, con, knd\}$.

Proof sketch: The proof follows the same outline as the proof for the CPS translation in Theorem 1. □

4.3. Interaction between CPS and DS translations

Since $\underline{\beta}$-conversion is the principal notion of equality in the cube, we now consider the interaction of the translations up to this notion of equality. The following theorem states an equational correspondence (as presented by Sabry and Felleisen [58]) between direct style terms and the CPS language.

Theorem 3. *Let* $\Gamma \Vdash A, A_1, A_2 : B$ *and* $\Sigma \Vdash_\theta P, P_1, P_2 : Q$ *for* $\theta \in \{com, con, knd\}$.
1. $A =_{\underline{\beta}} \mathcal{D}\langle \mathcal{C}\langle A\rangle\rangle$
2. $P =_{\underline{\beta}} \mathcal{C}\langle \mathcal{D}\langle P\rangle\rangle$
3. $A_1 =_{\underline{\beta}} A_2$ *iff* $\mathcal{C}\langle A_1\rangle =_{\underline{\beta}} \mathcal{C}\langle A_2\rangle$
4. $P_1 =_{\underline{\beta}} P_2$ *iff* $\mathcal{D}\langle P_1\rangle =_{\underline{\beta}} \mathcal{D}\langle P_2\rangle$

Using the optimizing CPS translation, the equational correspondence in the above theorem can be strengthened to a *reduction correspondence* [8]. This is noteworthy since the applications we have in mind (e.g., compilation, partial evaluation, inferring strong normalization from weak normalization) often require relating DS and CPS reductions.

Theorem 4 (Reduction correspondence for $\mathcal{C}^+$ and $\mathcal{D}$). *Let* $\Gamma \Vdash A, A_1, A_2 : B$ *and* $\Sigma \Vdash_\theta P, P_1, P_2 : Q$ *for* $\theta \in \{com, con, knd\}$.
1. $A \equiv \mathcal{D}\langle \mathcal{C}^+\langle A\rangle\rangle$
2. $P \twoheadrightarrow_{\underline{\beta}} \mathcal{C}\langle \mathcal{D}\langle P\rangle\rangle$
3. $A_1 \rightarrow_{\underline{\beta}} A_2$ *implies* $\mathcal{C}\langle A_1\rangle \twoheadrightarrow_{\underline{\beta}} \mathcal{C}\langle A_2\rangle$
4. $P_1 \rightarrow_{\underline{\beta}} P_2$ *implies* $\mathcal{D}\langle P_1\rangle \twoheadrightarrow_{\underline{\beta}} \mathcal{D}\langle P_2\rangle$

Proof sketch: The proof is very similar to the proof of the analogous theorem for the untyped setting given in detail by the authors in [10]. □

5. CPS translations for domain-free pure type systems

The definition of the domain-free λ-cube in figure 5 contains a certain amount of redundancy. Indeed, the four product rules (the four bottom-most rules) are very similar; in fact, they may be viewed as instances of a general scheme. Something similar can be said of the abstraction rules (to the left in the figure) and application rules (to the right in the figure). Indeed it is possible to compactify the presentation of the domain-free λ-cube by having a single rule for product, a single rule for abstraction and a single rule for application and by parameterizing their use according to the system considered. In this section we systematize this observation by reviewing the general notion of *domain-free pure type systems* from [12]. The development here is completely analogous to the classical development of pure type systems by Barendregt [4], Berardi [13] and Terlouw [72].

In the first subsection we introduce the notion of a domain-free pure type system. In the second subsection we show that the systems of the domain-free λ-cube can be viewed as domain-free pure type systems. The third subsection introduces the notion of logical specification, due to Coquand and Herbelin [15], which captures most of the common features of specifications for which CPS translations may be defined. In the fourth subsection, we establish a classification lemma, which provides the main technical tool for the definition of the CPS translations. The translation itself is defined in the fifth and last subsection.

5.1. Specifications and domain-free pure type systems

Parametricity is achieved through the notion of *specification*, upon which the framework is parameterized. Specifications are abstract structures expressing dependencies between type universes, or sorts.

Definition 6 (*Specifications*). A *specification* is a triple $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ where

1. $\mathcal{S}$ is a set of *sorts*;
2. $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of *axioms*;
3. $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a set of *rules*.

As usual, we use (s_1, s_2) to denote rules of the form (s_1, s_2, s_2).

Most concrete specifications in the literature have form (s_1, s_2); as we shall see, this means that products $\Pi x : A.B$ live in the same universe as B. The role of specifications will be clearer at the end of this subsection. Throughout this subsection, we assume a fixed specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$.

In our presentation, we require each variable to have an underlying sort. This requirement, which is reminiscent of our presentation of the domain-free λ-cube, simplifies the definition of the CPS translation.

Definition 7 (*Sorted variables*). We let $V_{\mathbf{S}}$ (or, for brevity, just V leaving the dependence on **S** implicit) denote a denumerable set of variables, partitioned into countably infinite sets V^s for all $s \in \mathcal{S}$; that is,

- $V = \bigcup_{s \in \mathcal{S}} V^s$;
- $V^s \cap V^{s'} = \emptyset$ for $s \neq s'$;
- V^s is countably infinite for each $s \in \mathcal{S}$.

While we maintain the use of sorted variables, we shall not define syntactic categories by mutual recursion, because such an approach is overly complicated for arbitrary specifications. Instead, we revert to the traditional formulation of domain-free pure type systems, in which there is a single syntactic category of expressions. Note however that, in most cases, we shall be able to retrieve disjoint syntactic categories, akin to those of the domain-free λ-cube, via a a classification lemma.

Definition 8 (*Domain-free pseudo-terms*). The set *Terms*[DFPTS] of *domain-free pseudo-terms* (over **S**) is given by the abstract syntax below, where x ranges over V and s ranges over $\mathcal{S}$.

$$\mathit{Terms}[\mathsf{DFPTS}] \ni A, B ::= x \mid s \mid AB \mid \lambda x.A \mid \Pi x : A.B$$

We use $a, b, A, B, M, N \ldots$ to denote domain-free pseudo-terms, $x, y, z, \ldots$ to denote variables, and $s, s', \ldots$ to denote sorts.

The computational behavior of domain-free pure type systems is the expected one.

Definition 9 (*Domain-free reduction*). The notion of *$\underline{\beta}$-reduction* $\to_{\underline{\beta}}$ is defined as the smallest compatible relation

$$(\lambda x.M)N \to_{\underline{\beta}} M\{x := N\}$$

where $\bullet\{\bullet := \bullet\}$ denotes the obvious substitution operator.

We now turn to the typing relation of domain-free pure type systems.

Definition 10 (*Domain-free pure type system*).

1. The set *Contexts*[DFPTS] of *domain-free pseudo-contexts* is given by the abstract syntax below, where x ranges over V and A ranges over *Terms*[DFPTS].

   $$\mathit{Contexts}[\mathsf{DFPTS}] \ni \Gamma ::= \cdot \mid \Gamma, x : A$$

   We use $\Gamma, \Delta, \ldots$ to denote domain-free contexts. For $\Gamma \equiv x_1 : A_1, \ldots, x_n : A_n$, we write $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ and say that $x_i : A_i \in \Gamma$ for each $i \in \{1, \ldots, n\}$.
2. The *domain-free derivability* relation $\Vdash$ on triples (Γ, M, A), where $\Gamma \in$ *Contexts*[DFPTS] and $M, A \in$ *Terms*[DFPTS], is defined in figure 14. If $\Gamma \Vdash A : B$ then Γ, A, and B are *legal*. We also say that the *judgment* $\Gamma \Vdash M : A$ is *derivable*, or *legal*.
3. The tuple $\underline{\lambda}\mathbf{S} = (\mathit{Terms}[\mathsf{DFPTS}], \mathit{Contexts}[\mathsf{DFPTS}], \rightarrow_{\underline{\beta}}, \Vdash)$ is the *domain-free pure type system* (DFPTS) induced by $\mathbf{S}$.

Domain-free pure type systems enjoy most of the properties of pure type systems, except *uniqueness of types* (indeed, a term such as $\lambda x.x$ may have more than one type in a single context, and therefore uniqueness of types fails). We refer the reader to [12] for a detailed study of the theory of domain-free pure type systems.

$$\textbf{(axiom)} \qquad \cdot \Vdash s_1 : s_2 \qquad \text{if } (s_1, s_2) \in \mathcal{A}$$

$$\textbf{(start)} \qquad \frac{\Gamma \Vdash A : s}{\Gamma, x : A \Vdash x : A} \qquad \text{if } x \notin \mathrm{dom}(\Gamma) \text{ and } x \in V^s$$

$$\textbf{(weakening)} \qquad \frac{\Gamma \Vdash A : B \quad \Gamma \Vdash C : s}{\Gamma, x : C \Vdash A : B} \qquad \text{if } x \notin \mathrm{dom}(\Gamma) \text{ and } x \in V^s$$

$$\textbf{(product)} \qquad \frac{\Gamma \Vdash A : s_1 \quad \Gamma, x : A \Vdash B : s_2}{\Gamma \Vdash (\Pi x{:}A.\, B) : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}$$

$$\textbf{(application)} \qquad \frac{\Gamma \Vdash F : (\Pi x{:}A.\, B) \quad \Gamma \Vdash a : A}{\Gamma \Vdash F\, a : B\{x := a\}}$$

$$\textbf{(abstraction)} \qquad \frac{\Gamma, x : A \Vdash b : B \quad \Gamma \Vdash (\Pi x{:}A.\, B) : s}{\Gamma \Vdash \lambda x.b : \Pi x{:}A.\, B}$$

$$\textbf{(conversion)} \qquad \frac{\Gamma \Vdash A : B \quad \Gamma \Vdash B' : s}{\Gamma \Vdash A : B'} \qquad \text{if } B =_{\underline{\beta}} B'$$

Figure 14. Domain-free pure type systems.

5.2. Domain-free pure type systems vs. the domain-free λ-cube

We now show that one can recover from the formalism of domain-free pure type systems the systems of the domain-free λ-cube by choosing appropriate sorts, axioms, and rules. In fact, let $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ where $\mathcal{S} = \{*, \Box\}$ and $\mathcal{A} = \{(*, \Box)\}$. Then for each system of the domain-free λ-cube we obtain a corresponding domain-free pure type system by taking the following sets of rules $\mathcal{R}$:

Cube system S	Rules of corresponding specification **S**			
$\rightarrow$	$(*, *)$			
2	$(*, *)$	$(\Box, *)$		
P	$(*, *)$		$(*, \Box)$	
$P2$	$(*, *)$	$(\Box, *)$	$(*, \Box)$	
$\underline{\omega}$	$(*, *)$			$(\Box, \Box)$
ω	$(*, *)$	$(\Box, *)$		$(\Box, \Box)$
$P\underline{\omega}$	$(*, *)$		$(*, \Box)$	$(\Box, \Box)$
$P\omega = C$	$(*, *)$	$(\Box, *)$	$(*, \Box)$	$(\Box, \Box)$

Let us elaborate this in some detail. The set

$$Obj[\mathsf{DFCUBE}] \cup Constr[\mathsf{DFCUBE}] \cup Kind[\mathsf{DFCUBE}] \cup \{\Box\}$$

is a subset of the set *Terms*[DFPTS]. One can show a *classification result* stating that if $\Gamma \Vdash A : B$ in the domain-free pure type system $\underline{\lambda}\mathbf{S}$, then

1. $(A, B) \in Obj[\mathsf{DFCUBE}] \times Constr[\mathsf{DFCUBE}]$, or
2. $(A, B) \in Constr[\mathsf{DFCUBE}] \times Kind[\mathsf{DFCUBE}]$, or
3. $(A, B) \in Kind[\mathsf{DFCUBE}] \times \{\Box\}$.

Thus, although the notions of objects and constructors, etc., are not defined *a priori* in a domain-free pure type system, such notions can be derived from the typing system itself by classification results of the above form. Moreover, one can the show that if

1. $(A, B) \in Obj[\mathsf{DFCUBE}] \times Constr[\mathsf{DFCUBE}]$, or
2. $(A, B) \in Constr[\mathsf{DFCUBE}] \times Kind[\mathsf{DFCUBE}]$, or
3. $(A, B) \in Kind[\mathsf{DFCUBE}] \times \{\Box\}$

then $\Gamma \Vdash A : B$ in the domain-free λ-cube system $\underline{\lambda}S$ iff $\Gamma \Vdash A : B$ in the domain-free pure type system $\underline{\lambda}\mathbf{S}$. This shows that a cube system and the corresponding domain-free pure type system derive exactly the same judgments.

The intuition behind this is that if one unfolds the inference rules of the domain-free pure type system for all rules $(s, s') \in \mathcal{R}$, then one obtains the same inference rules as are present in the cube system:

1. The rules (S) and (start), (W) and (weakening), and (β) and (conversion) are clearly identical. The rule (A) is identical to (axiom) since we have chosen $\mathcal{A} = \{(*, \Box)\}$.
2. The product rules ($\rightarrow$), (2), ($\underline{\omega}$), (P) correspond to the instances $(*, *)$, $(\Box, *)$, $(\Box, \Box)$, $(*, \Box)$ of the rule (product). Thus, the rules $\mathcal{R}$ determine which generalized function spaces may be formed and what is their nature.

   Note that in the domain-free λ-cube an assumption is missing in the product rules compared to the domain-free pure type system. For instance, in the product rule ($\rightarrow$) the assumption $\Gamma \Vdash C : *$ is missing. However, due to the disjointness of the syntactic classes in the cube, one can show that if $\Gamma, x : C \Vdash A : B$, then in fact $\Gamma \Vdash C : *$.

   In the general setting of domain-free pure type systems where we have no *a priori* distinction into categories, the assumption is necessary.
3. The application rules ($\rightarrow$), (2), ($\underline{\omega}$), (P) correspond to the rule (application) with different products.
4. The abstraction rules ($\rightarrow$), (2), ($\underline{\omega}$), (P) correspond to the instances of the rule (abstraction) with different products.

   Note that in the domain-free λ-cube an assumption is missing in the abstraction rules compared to the domain-free pure type system. For instance, in the abstraction rule ($\rightarrow$) the assumption $\Gamma \vdash \Pi x : C.C' : s$ is missing. In the domain-free system this assumption is used to prevent invocations of the abstraction rule for products that cannot be constructed according to the rules $\mathcal{R}$; however, in the cube we obtained the same effect by explicitly excluding some of the inference rules; for instance, the cube system $\lambda\rightarrow$ does not include rules labeled (2).

5.3. Logical specifications

The notion of logical specification, due to Coquand and Herbelin [15], captures most of the features of specifications for which it is possible to define CPS translations. Below we review the notion of logical specification and introduce the class of locally persistent specifications, for which CPS translations may be defined.

Definition 11.

1. A specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ is *functional* if $\mathcal{A}$ and $\mathcal{R}$ are partial functions, i.e. for all $s_1, s_2, s_2', s_3, s_3' \in \mathcal{S}$
   - $(s_1, s_2) \in \mathcal{A}$ & $(s_1, s_2') \in \mathcal{A} \Rightarrow s_2 = s_2'$;
   - $(s_1, s_2, s_3) \in \mathcal{R}$ & $(s_1, s_2, s_3') \in \mathcal{R} \Rightarrow s_3 = s_3'$.
2. A *logical specification* is a quadruple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathsf{Prop})$ where $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is a functional specification and $\mathsf{Prop} \in \mathcal{S}$ is a sort such that

   (A) there exists $s \in \mathcal{S}$ such that $(\mathsf{Prop}, s) \in \mathcal{A}$;
   (B) there is no $s \in \mathcal{S}$ such that $(s, \mathsf{Prop}) \in \mathcal{A}$;
   (C) $(\mathsf{Prop}, \mathsf{Prop}, \mathsf{Prop}) \in \mathcal{R}$.

3. A logical specification $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathsf{Prop})$ is *locally persistent* if for every $(s_1, s_2, s_3) \in \mathcal{R}$,

$$s_2 = \mathsf{Prop} \Leftrightarrow s_3 = \mathsf{Prop}$$

4. A logical specification $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathsf{Prop})$ is *non-dependent* if it is locally persistent and for every $(s_1, s_2, s_3) \in \mathcal{R}$,

$$s_1 = \mathsf{Prop} \Rightarrow s_2 = s_3 = \mathsf{Prop}$$

Examples of non-dependent logical specification include the specifications on the left-hand side of the λ-cube (i.e. $\rightarrow, 2, \underline{\omega}$ and ω), the specifications of the L-cubes [4, 13, 28], HOL, U^- and U [4]. The specifications on the right-hand side of the λ-cube (i.e. P, $P2$, $P\underline{\omega}$ and $P\omega$), are examples of dependent, locally persistent specifications. More generally, every logical specification with rules of the form (s_1, s_2) only is locally persistent. Finally, the specification $*$ [4] is not logical because $\mathsf{Prop} : \mathsf{Prop}$ is an axiom and hence the second requirement is violated.

5.4. Injective specifications and the classification lemma

The CPS translations of Section 3 rely on a classification of expressions into different syntactic categories; in fact, this classification is built into the system by considering *a priori* the classes of objects, constructors, and kinds. In Section 5.2, we showed that, when starting out from domain-free pure type systems, then—at least for a particular choice of $\mathcal{S}, \mathcal{A}, \mathcal{R}$—one can use the type system to divide the pseudo-terms into different syntactic categories. In order to scale up the CPS translation to domain-free pure type systems, we generalize this classification. For this purpose, we introduce the following standard notions.

Definition 12. A specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ is *injective* if it is functional and moreover, for every $s_1, s_1', s_2, s_2', s_3 \in \mathcal{S}$,

- $(s_1, s_2) \in \mathcal{A}$ & $(s_1', s_2) \in \mathcal{A} \Rightarrow s_1 \equiv s_1'$
- $(s_1, s_2, s_3) \in \mathcal{R}$ & $(s_1, s_2', s_3) \in \mathcal{R} \Rightarrow s_2 \equiv s_2'$

Most specifications appearing in the literature are injective. Indeed, every specification that only uses rules of the form (s_1, s_2) trivially satisfies the conditions on rules; thus such a specification is injective iff $\mathcal{A}$ is an injective, partial function. We invite the reader to verify that all the specifications considered in this paper and most of the specifications introduced in [4] are indeed injective.

For these specifications, one can formulate the following classification result, which is a direct consequence of [12].

Definition 13. Let $\mathbf{S}$ be a specification. Define

$$\mathsf{Prp} = \{M \in \mathit{Terms}[\mathsf{DFPTS}] \mid \Gamma \Vdash M : \mathsf{Prop} \text{ for some } \Gamma\}$$

$$\mathsf{Set} = \{M \in \mathit{Terms}[\mathsf{DFPTS}] \mid \Gamma \Vdash M : s \text{ for some } s \neq \mathsf{Prop}\}$$

$$\mathsf{Prf} = \{M \in \mathit{Terms}[\mathsf{DFPTS}] \mid \Gamma \Vdash M : A \ \&\ \Gamma \Vdash A : \mathsf{Prop} \text{ for some } \Gamma \text{ and } A\}$$

$$\mathsf{Elt} = \{M \in \mathit{Terms}[\mathsf{DFPTS}] \mid \Gamma \Vdash M : A \ \&\ \Gamma \Vdash A : s \text{ for some } \Gamma, A \text{ and } s \neq \mathsf{Prop}\}$$

We also use Type for $\mathsf{Prp} \cup \mathsf{Set}$.

In the sequel, we often assume types to be normalizing, i.e. $\mathsf{Type} \subseteq \mathrm{WN}_{\underline{\beta}}$. This assumption is needed to ensure preservation of sorts, i.e.

$$\Gamma \Vdash A : s \ \&\ \Gamma \Vdash A' : s' \ \&\ A =_{\underline{\beta}} A' \Rightarrow s = s'$$

which in turn is needed in the proof of classification.

Proposition 1 (Classification for DFPTSs). *If* **S** *is injective and* $\mathsf{Type} \subseteq \mathrm{WN}_{\underline{\beta}}$ *then* $\mathsf{Prp} \cap \mathsf{Set} = \emptyset$ *and* $\mathsf{Prf} \cap \mathsf{Elt} = \emptyset$.

As for the λ-cube, it is possible to strengthen the result by defining pairwise disjoint, decidable, syntactic classes of terms that contain the classes below. For the purpose of conciseness, we limit ourselves to this weak form of classification.

5.5. CPS translations for domain-free pure type systems

In this section we define a CPS translation for any injective, locally persistent logical specification in which all types are weakly normalizing. It is worth pointing that there is no technical hurdle to scale up the CPS translations. For the sake of brevity, we only present the basic translation. Other (e.g. optimized, direct style) translations are generalized in a similar fashion.

Definition 14. Let **S** be an injective, locally persistent logical specification with $\mathsf{Type} \subseteq \mathrm{WN}_{\underline{\beta}}$. The (domain-free) CPS translation is defined in figure 15.

Figure 15 provides a compact alternative to the CPS translation of the domain-free λ-cube by instantiating $\mathsf{Prop} = *$ and $\mathsf{Prf} = \mathit{Constr}[\mathsf{CUBE}]$. The two definitions are not exactly identical, as the former is not defined on all pseudo-terms, but this difference is of no importance: we are primarily interested in well-typed terms. Following the method of Theorem 1 and using Proposition 1, one proves:

Theorem 5.

$$\Gamma \Vdash A : B \Rightarrow \mathcal{C}\llbracket\Gamma\rrbracket \Vdash \mathcal{C}\langle A\rangle : \mathcal{C}\llbracket B\rrbracket$$

Note that the assumption of the specification being locally persistent is needed in the application rule.

$$
\begin{array}{lcll}
\mathcal{C}\langle x\rangle & = & \left\{\begin{array}{l}\lambda k.x\ k\\ x\end{array}\right. & \begin{array}{l}\text{if } x \in V^{\mathsf{Prop}}\\ \text{otherwise}\end{array}\\[2ex]
\mathcal{C}\langle s\rangle & = & s & \\[2ex]
\mathcal{C}\langle \lambda x.M\rangle & = & \left\{\begin{array}{l}\lambda k.k\ (\lambda x.\mathcal{C}\langle M\rangle)\\ \lambda x.\mathcal{C}\langle M\rangle\end{array}\right. & \begin{array}{l}\text{if } \lambda x.M \in \mathsf{Prf}\\ \text{otherwise}\end{array}\\[2ex]
\mathcal{C}\langle M\ M'\rangle & = & \left\{\begin{array}{l}\lambda k.\mathcal{C}\langle M\rangle\ (\lambda j.j\ \mathcal{C}\langle M'\rangle\ k)\\ \mathcal{C}\langle M\rangle\ \mathcal{C}\langle M'\rangle\end{array}\right. & \begin{array}{l}\text{if } M\ M' \in \mathsf{Prf}\\ \text{otherwise}\end{array}\\[2ex]
\mathcal{C}\langle \Pi x{:}A.\ B\rangle & = & \Pi x{:}\mathcal{C}\langle\!|A|\!\rangle.\ \mathcal{C}\langle\!|B|\!\rangle & \\[2ex]
\mathcal{C}\langle\!|M|\!\rangle & = & \left\{\begin{array}{l}\neg\neg\mathcal{C}\langle M\rangle\\ \mathcal{C}\langle M\rangle\end{array}\right. & \begin{array}{l}\text{if } M \in \mathsf{Prp}\\ \text{otherwise}\end{array}\\[2ex]
\mathcal{C}\langle\!|.|\!\rangle & = & \bot : \mathsf{Prop} & \\
\mathcal{C}\langle\!|\Gamma, x:A|\!\rangle & = & \mathcal{C}\langle\!|\Gamma|\!\rangle, x : \mathcal{C}\langle\!|A|\!\rangle &
\end{array}
$$

Figure 15. CPS translation for domain-free pure type systems.

One can envisage to generalize the CPS translation in several directions:

- one can consider even larger classes of logical specifications, e.g. by not requiring the specification to be injective. There is little incentive for such a translation as most specifications of interest are injective. However, one may use the domain-full CPS translation, as summarized in Section 6.4, and the relationship between domain-free pure type systems and domain-full pure type systems, as summarized in Section 6.3, to achieve the desired translation.
- one can consider a more general notion of logical specification, in which several universes of propositions are allowed. In fact, such a generalization is natural both from programming and logical perspectives. While we have not checked the details, we do not expect any difficulty in scaling up the CPS translation to a suitably defined extension of the class of locally persistent specifications.

6. CPS translations for (domain-full) pure type systems

Although domain-free pure type systems appear to be more appropriate for the purpose of defining CPS translation, there are strong reasons to study CPS translations in the more traditional setting of traditional pure type systems. The primary reason is of sociological nature: traditional pure type systems are better established than their domain-free counterpart and many of the existing CPS translations have been phrased in terms of domain-full systems. The second reason is of practical nature: in a number of applications, one is

interested in domain-full systems. Such applications range from looping combinators for inconsistent pure type systems to compilation of typed intermediate languages. In this section, we therefore embark on defining CPS translations for traditional pure type systems. Two methods are considered:

- the *direct method* which relies on a non-standard induction principle, inspired from earlier work by Dowek et al. [20], see also [9].
- the *indirect method* which relies on the close correspondence between domain-free and traditional pure type systems, see [12].

This section is organized as follows. In the first subsection, we briefly outline the main definitions for domain-full pure type systems. In the second subsection, we relate domain-full and domain-free pure type systems. In the third subsection, we exploit this relation to define CPS translations via the indirect method. In the fourth subsection, we define CPS translations via the direct method. In the fifth subsection, we summarize the different approaches to define CPS translations. For conciseness, we limit ourselves to the Plotkin's style, un-optimized, CPS translation. Other translations (optimized, direct style), can be treated likewise.

6.1. Domain-full pure type systems

Pure type systems are defined in essentially the same way as domain-free pure type systems. The only difference is that the former feature a domain-full λ-abstraction of the form $\lambda x : A.M$. Throughout the rest of this subsection, $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ denotes an arbitrary specification.

Definition 15 (*pure type systems*).

1. The set *Terms*[PTS] is given by the abstract syntax:

   $$\textit{Terms}[\mathsf{PTS}] \ni A, B ::= x \mid s \mid AB \mid \lambda x : A.B \mid \Pi x : A.B$$

   where $x \in V$ and $s \in \mathcal{S}$.
2. The set *Contexts*[PTS] is given by the abstract syntax:

   $$\textit{Contexts}[\mathsf{PTS}] \ni \Gamma ::= \cdot \mid \Gamma, x : A$$

3. *β-reduction* $\rightarrow_\beta$ is defined as the smallest compatible relation closed under the rule

   $$(\lambda x : A.M)N \rightarrow_\beta M\{x := N\}$$

4. The *pure type system derivability* relation $\vdash$ is given by the rules of figure 16.

We use the same notation, terminology, and conventions as were employed for domain-free pure type systems.

$$\text{(axiom)} \qquad \cdot \vdash s_1 : s_2 \qquad \text{if } (s_1, s_2) \in \mathcal{A}$$

$$\text{(start)} \qquad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \qquad \text{if } x \notin \mathrm{dom}(\Gamma) \text{ and } x \in V^s$$

$$\text{(weakening)} \qquad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \qquad \text{if } x \notin \mathrm{dom}(\Gamma) \text{ and } x \in V^s$$

$$\text{(product)} \qquad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.\, B) : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{R}$$

$$\text{(application)} \qquad \frac{\Gamma \vdash F : (\Pi x{:}A.\, B) \quad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B\{x := a\}}$$

$$\text{(abstraction)} \qquad \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x{:}A.\, B) : s}{\Gamma \vdash \lambda x{:}A.\, b : \Pi x{:}A.\, B}$$

$$\text{(conversion)} \qquad \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \qquad \text{if } B =_\beta B'$$

Figure 16. Pure type systems.

6.2. Domain-full pure type systems vs. domain-free pure type systems

In this subsection, we relate derivability in domain-full pure type systems to derivability in domain-free pure type systems. Most of the results in this subsection come from [12]. Throughout this subsection, we let **S** be a fixed specification.

It is trivial to define a map from domain-full pseudo-terms to domain-free pseudo-terms.

Definition 16 (*erasure*). The *erasure* map $|.| : \mathit{Terms}[\mathsf{PTS}] \to \mathit{Terms}[\mathsf{DFPTS}]$ is defined as follows:

$$\begin{aligned} |x| &= x \\ |s| &= s \\ |tu| &= |t||u| \\ |\lambda x : A.t| &= \lambda x.|t| \\ |\Pi x : A.B| &= \Pi x : |A|.|B| \end{aligned}$$

Erasure preserves typing.

Proposition 2. *If* $\Gamma \vdash M : A$ *then* $|\Gamma| \Vdash |M| : |A|$.

It is more difficult to define a decorating function that is inverse to erasure and that maps domain-free judgments to domain-full judgments.

Proposition 3. *Assume that* $\mathbf{S}$ *is functional and that* $\mathsf{Type} \subseteq \mathrm{WN}_\beta$. *If* $\Delta \Vdash E : F$ *then there exists* $\Gamma \in \mathit{Contexts}[\mathsf{PTS}]$ *and* $M, A \in \mathit{Terms}[\mathsf{PTS}]$ *s.t.* $\Gamma \vdash M : A$, $|\Gamma| \equiv \Delta$, $|M| \equiv E$ *and* $|A| \equiv F$.

The exact description of the decoration process, and the proof of the Proposition, are to be found in [12]. Below we briefly indicate how the decoration process works. Clearly the crucial case is that of the abstraction rule,

$$\frac{\Gamma, x : A \Vdash b : B \quad \Gamma \Vdash (\Pi x : A.B) : s}{\Gamma \Vdash \lambda x.b : \Pi x : A.B}$$

So assume that we have decorations of $\Gamma, x : A \Vdash b : B$ and $\Gamma \Vdash (\Pi x : A.B) : s$ from which we must construct a decoration of $\Gamma \Vdash \lambda x.b : \Pi x : A.B$. The only term to treat is $\lambda x.b$. We simply decorate it as $\lambda x : A'.b'$ where A' and b' are the respective decorations of A and b (in suitable contexts). Using the induction hypothesis and some auxiliary results, one then shows that typing is preserved.

For the sake of completeness, note that decoration needs to be context-dependent. Indeed, one cannot define a map $\mathsf{dec} : \mathit{Terms}[\mathsf{DFPTS}] \to \mathit{Terms}[\mathsf{PTS}]$ such that $|\mathsf{dec}(M)| = M$ for every $M \in \mathit{Terms}[\mathsf{DFPTS}]$ and

$$\Gamma \Vdash M : C \Rightarrow \mathsf{dec}(\Gamma) \Vdash \mathsf{dec}(M) : \mathsf{dec}(C)$$

where dec is extended in the obvious way to contexts.

Indeed, consider the term $\lambda x.x$. In $\underline{\lambda} \to$, we have

$$A : *, B : * \Vdash \lambda x.x : A \to A$$

and

$$A : *, B : * \Vdash \lambda x.x : B \to B$$

If there were a map dec with the above mentioned properties, it would satisfy

$$\mathsf{dec}(\lambda x.x) = \lambda x : C.x$$

with

$$A : *, B : * \vdash \lambda x : C.x : A \to A$$

and

$$A : *, B : * \vdash \lambda x : C.x : B \to B$$

The above judgments are only derivable in case $C = A = B$, which needs not be the case.

In contrast, there is no problem with context-dependent decoration as the above examples are decorated respectively into

$$A : *, B : * \vdash \lambda x : A.x : A \to A$$

and

$$A : *, B : * \vdash \lambda x : B.x : B \to B$$

6.3. The indirect method

In this subsection, we scale up to domain-full PTSs the domain-free CPS translation of Section 5.5. Below we let **S** be an injective, locally persistent logical specification such that $\mathsf{Type} \subseteq \mathrm{WN}_\beta$. (Note that $\mathsf{Type} \subseteq \mathrm{WN}_\beta$ implies $\mathsf{Type} \subseteq \mathrm{WN}_{\underline{\beta}}$).

The indirect method consists in factorizing the CPS translation of a domain-full pseudo-context Γ and a domain-full pseudo-term M in three steps:

1. erase the domains of the λ-abstractions so as to obtain a domain-free pseudo-context $|\Gamma|$ and a domain-free pseudo-term $|M|$;
2. translate $|\Gamma|$ into $\mathcal{C}\{|\Gamma|\}$ and $|M|$ into $\mathcal{C}\langle|M|\rangle$;
3. using $\mathcal{C}\{|\Gamma|\}$, decorate λ-abstractions in $\mathcal{C}\langle|M|\rangle$ so as to obtain a domain-full pseudo-term N.

In picture, the indirect method is represented as follows:

$$\begin{array}{ccc} \mathit{Terms}[\mathsf{PTS}] & & \mathit{Terms}[\mathsf{PTS}] \\ {\scriptstyle \text{erase}}\downarrow & & \uparrow{\scriptstyle \text{decorate}} \\ \mathit{Terms}[\mathsf{DFPTS}] & \xrightarrow[\text{CPS}]{} & \mathit{Terms}[\mathsf{DFPTS}] \end{array}$$

Symbolically, one can define

$$\mathcal{C}'_\Gamma\langle . \rangle : \mathit{Terms}[\mathsf{PTS}] \to \mathit{Terms}[\mathsf{PTS}]$$

as $\mathsf{dec}_{\mathcal{C}\{|\Gamma|\}}(\mathcal{C}\langle|M|\rangle)$, where $\mathsf{dec}_\Delta(M)$ is a decoration of M w.r.t. Δ, as given by Proposition 3.

6.4. The direct method

In this subsection, we present a family of CPS translations which preserve typing and act on domain-full pseudo-terms. More precisely, we define a family of (partial) maps $\mathcal{C}_\Gamma\langle . \rangle : \mathit{Terms}[\mathsf{PTS}] \to \mathit{Terms}[\mathsf{PTS}]$, where Γ is a pseudo-context. The definition, which proceeds by well-founded induction over pairs (Γ, M), where Γ is a pseudo-context and M is a pseudo-term, preserves typing and applies to most Pure Type Systems that appear

in the literature. For the sake of conciseness, we gloss over technical details, including the definition of the order $\prec$ (to be found in [9]) and limit ourselves to an informal description of the order.

Firstly, $\prec$ contains the subterm relation, defined on pairs (Γ, M) in the obvious way. However, this is not enough because the CPS translation cannot proceed by induction on the structure of pairs (Γ, M), where Γ is a pseudo-context and M is a pseudo-term. Take for example a context Γ and a variable x of sort $*$. It will be translated into $\lambda k : A.xk$ for some suitable A. If the CPS translation is to preserve typing, A must be related to the CPS translation of (Γ, B), where B is a type of x in Γ. So the order to be used in the definition of the translation should satisfy: if $\Gamma \vdash M : A$ then $(\Gamma, B) \prec (\Gamma, A)$ for some B such that $\Gamma \vdash M : B$. Following [9, 20], we take B to be the normal form of A. The resulting order $\prec$ is well-founded for most systems that appear in the literature.

Definition 17. $\mathcal{C}_\Gamma\langle . \rangle : \mathit{Legal\text{-}Terms}[\mathsf{PTS}] \to \mathit{Legal\text{-}Terms}[\mathsf{PTS}]$ is defined in figure 17.

The translation is well-defined for all locally persistent logical specifications for which $\prec$ is well-founded. For such specifications, we have:

Theorem 6.

$$\Gamma \vdash M : A \Rightarrow \mathcal{C}\llbracket\Gamma\rrbracket \vdash \mathcal{C}_\Gamma\langle M\rangle : \mathcal{C}_\Gamma\llbracket A\rrbracket$$

6.5. Assessment

In closing this section, it is perhaps worth noticing a number of variations in the CPS translations we have defined above for the domain-free cube (figure 7), for domain free PTSs (figure 15), and for domain-full PTSs (figure 17).

- Syntactic/type-based classification. The translation of terms and types are different, so one must have a means of distinguishing between the two notions. This can be done either syntactically by dividing the pseudo-terms into various categories (e.g. the categories of pseudo-objects, -constructors, and -kinds in the domain-free cube) or based on types by dividing the legal terms into various categories (e.g. the categories Prp, Set, Prf, Elt in domain-free PTSs).
- Context-dependent/-independent classification. The type-based classification can be done either independently of the context (as the categories Prp, etc. in domain-free PTSs) or depending on the context (by checking e.g. $\Gamma \vdash M : \mathsf{Prop}$ as in the translation of domain-full PTSs).
- Context-dependent/-independent translation. If the CPS translation uses a type-based context-dependent classification, then the translation must be parametrized by the context. Also, if the translation reconstructs types (as in the domain-full CPS-translation), then the translation must again be parametrized by the context.

The various translations are often related by results such as the following:

Proposition 4. *If* $\Gamma \vdash M : A$, *then* $|\mathcal{C}_\Gamma\langle M\rangle| \equiv \mathcal{C}\langle |M| \rangle$.

$$\mathcal{C}_\Gamma\langle x\rangle \;=\; \begin{cases} \lambda k{:}\neg\mathcal{C}^{\mathsf{nf}}_\Gamma\langle D\rangle.\, x\; k & \text{if } \Gamma \vdash x : D : \mathsf{Prop} \\ x & \text{otherwise} \end{cases}$$

$$\mathcal{C}_\Gamma\langle s\rangle \;= s$$

$$\mathcal{C}_\Gamma\langle \lambda x{:}A.\, M\rangle \;= \begin{cases} \lambda k{:}\neg\mathcal{C}^{\mathsf{nf}}_\Gamma\langle D\rangle.\, k\;(\lambda x{:}\mathcal{C}_\Gamma\langle A\rangle\;.\,\mathcal{C}_{(\Gamma,x:A)}\langle M\rangle\;) & \text{if } \Gamma \vdash \lambda x{:}A.\, M : D : \mathsf{Prop} \\ \lambda x{:}\mathcal{C}_\Gamma\langle\!|A|\!\rangle.\,\mathcal{C}_{(\Gamma,x:A)}\langle M\rangle & \text{otherwise} \end{cases}$$

$$\mathcal{C}_\Gamma\langle M\; M'\rangle \;= \begin{cases} \lambda k{:}\neg\mathcal{C}^{\mathsf{nf}}_\Gamma\langle D\rangle.\,\mathcal{C}_\Gamma\langle M\rangle\;\;(\lambda j{:}\mathcal{C}^{\mathsf{nf}}_\Gamma\langle D'\rangle.\, j\;\mathcal{C}_\Gamma\langle M'\rangle\;\;k) & \text{if } \Gamma \vdash M\; M' : D : \mathsf{Prop} \\ & \text{and } \Gamma \vdash M : D' \\ \mathcal{C}_\Gamma\langle M\rangle\;\;\mathcal{C}_\Gamma\langle M'\rangle & \text{otherwise} \end{cases}$$

$$\mathcal{C}_\Gamma\langle \Pi x{:}A.\, B\rangle \;= \Pi x{:}\mathcal{C}_\Gamma\langle\!|A|\!\rangle.\,\mathcal{C}_{(\Gamma,x:A)}\langle\!|B|\!\rangle$$

$$\mathcal{C}_\Gamma\langle\!|M|\!\rangle = \begin{cases} \neg\neg\mathcal{C}_\Gamma\langle M\rangle & \text{if } \Gamma \vdash M : \mathsf{Prop} \\ \mathcal{C}_\Gamma\langle M\rangle & \text{otherwise} \end{cases}$$

$$\mathcal{C}^{\mathsf{nf}}_\Gamma\langle M\rangle = \mathcal{C}_\Gamma\langle \mathsf{nf}\; M\rangle$$

$$\begin{aligned} \mathcal{C}\langle\!|\cdot|\!\rangle &= \bot : \mathsf{Prop} \\ \mathcal{C}\langle\!|\Gamma, x : A|\!\rangle &= \mathcal{C}\langle\!|\Gamma|\!\rangle, x : \mathcal{C}_\Gamma\langle\!|A|\!\rangle \end{aligned}$$

Figure 17. Domain-full CPS translation.

Proof: By induction on the structure of derivations. □

7. Conclusion and directions for further work

In this paper, we have generalized some important CPS translations to classes of pure type systems and domain-free pure type systems. To our knowledge, our CPS translations are the first of their kind to handle systems of dependent types. Reflecting on this work, we believe that several research avenues deserve further attention; for the sake of clarity, we distinguish between pure CPS questions and applications-oriented questions.

7.1. Pure CPS questions

The most fundamental question left open in this paper is the role of normalization in the definition of CPS translations. While our translations require types to be normalizing,

one would hope to device CPS translations which do not rely on any such assumption. Unfortunately, we have been unable to come up with such translations so far.

Another important question is to unveil the limitations of CPS translations. While the translations apply to most Pure Type Systems that appear in the literature, they cannot be extended readily to Σ-types as the usual translation for pairs does not preserve typing. Similar difficulties appear with inductive types as the standard class of strictly positive inductive definitions is not closed under CPS translation. However, the class of positive inductive definitions is closed under CPS translation so it would be interesting to study CPS translations for typed λ-calculi with positive inductive types. These limitations of CPS translations are in fact deeply connected to inherent difficulties in extracting the computational content of impredicative classical predicate logic with the axiom of choice.

On a more positive side, one can try to build up on the results of this paper:

- one natural objective is to recast existing CPS translations in the framework of pure type systems and domain-free pure type systems. A priori the techniques introduced in the paper seem general enough for this purpose and we expect no difficulty there. In particular, it seems worth exploring Fischer-style CPS translations (where continuations are the first arguments to functions) for pure type systems. As Sabry and Felleisen [58] illustrate, the CPS terms produced by these translations have slightly different reduction properties than those produced by Plotkin-style translations. This may be of use in some applications;
- in order to enhance the generality of our approach, one can also envision a generic staging through a monadic metalanguage as presented by Hatcliff and Danvy [39]. A preliminary step in this direction is the introduction of monadic type systems [11], which generalize the simply typed metalanguage in the same way as the pure type systems generalize the simply typed λ-calculus.

7.2. Application-related questions

The applications sketched in the introduction give obvious directions for future work.

- Recent trends in compilation emphasize the use of typed intermediate languages with sophisticated type structures. It seems worth to apply our results in this line of work following the remarks in Section 1.1.
- Strong normalization from weak normalization: thus far the Barendregt-Geuvers-Klop conjecture remains open for systems of dependent types. Further investigation is needed to determine whether the tools developed in this paper can lead to a successful solution to the conjecture. In [7], the authors solve the conjecture for the so-called generalized non-dependent pure type systems by using a pervasive CPS translation. However, this work falls short of treating systems of dependent types;
- Classical pure type systems: one important objective is a suitable generalization of the Kreisel-Friedman theorem, which provides a foundation for extracting the computational content from classical proofs. Other applications related to consistency and strong normalization have been partially achieved in [6].

Acknowledgments

We would like to thank T. Coquand for useful discussions on the paper, and we are grateful to the referees for their comments and suggestions.

Part of this work was performed while the first author was working at CWI (Amsterdam, The Netherlands) and at Chalmers University (Göteborg, Sweden). The first author was partially supported by a European TMR Fellowship. The second author was partially supported by the United States National Science Foundation under grant CCR-9701418, and by the United States National Aeronautics and Space Administration (NASA) under award NAG 21209.

Notes

1. We assume that the reader is familiar with the notions of free and bound variables and the related conventions—see [3].
2. Below we often omit the prefix "β-".
3. The reader may well wonder what the intuition is behind this property and where the modification of Plotkin's translation comes from; these issues are explained at length in [66].
4. Griffin's discovery was followed by several lines of work on classical logic, control operators, and the Curry-Howard isomorphism—some initiated independently of his work. It is not possible here to explain the aims and achievements of the individual lines of work, but see e.g., [6] for more references.
5. Consistency of both classical and minimal propositional logic can of course be proved by the method of truth tables! However, the above method scales up to logics for which the truth table method does not apply.
6. The use of λ-abstractions with domain in pure type systems is motivated by history (most type systems have adopted such abstractions) as well as by practical considerations (domain-full abstractions are necessary for type-checking to be decidable).
7. This presentation is equivalent to the original one, as sketched in Section 5.2.
8. The $\underline{\lambda}$ is used to distinguish these systems from the corresponding systems λS of Barendregt's λ-cube, introduced later.
9. This claim implicitly relies on the view that pure type systems do indeed correspond to these well-known typed λ-calculi. This latter issue is studied in some detail in [28].

References

1. Appel, A. *Compiling with Continuations*. Cambridge University Press, 1992.
2. Augustsson, L. Cayenne: A programming language with dependent types. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*. Baltimore, Maryland, 1998, ACM Press, pp. 239–250.
3. Barendregt, H.P. *The Lambda Calculus—Its Syntax and Semantics*. North-Holland, 1984.
4. Barendregt, H.P. Lambda calculi with types. In *Handbook of Logic in Computer Science*, S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum (Eds.). Vol. 2, Oxford Science Publications, 1992, pp. 117–309.
5. Barthe, G., Hatcliff, J., and Sørensen, M.H. CPS-translation and applications: The cube and beyond. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations*, O. Danvy (Ed.), number NS-96-13 in BRICS Notes, 1996, pp. 4/1–4/31.
6. Barthe, G., Hatcliff, J., and Sørensen, M.H. A notion of classical pure type system. In *Proceedings of the Thirteenth Annual Conference on the Mathematical Foundations of Programming Semantics*, S. Brookes and M. Mislove (Eds.). Pittsburgh, Pennsylvania, March 1997. Electronic Notes in Theoretical Computer Science, vol. 6.

7. Barthe, G., Hatcliff, J., and Sørensen, M.H. *Weak Normalization Implies Strong Normalization in Generalized Non-Dependent Pure Type Systems*. March 1997, submitted for publication.
8. Barthe, G., Hatcliff, J., and Sørensen, M.H. Reflections on reflections. In *Proceedings of the Ninth International Symposium on Programming Languages, Implementations, Logics and Programs*, H. Glaser, P. Hartel, and H. Kuchen (Eds.). Southampton, United Kingdom, September 1997. Lecture Notes in Computer Science, vol. 1292, Springer-Verlag, pp. 241–258.
9. Barthe, G., Hatcliff, J., and Sørensen, M.H. *An Induction Principle for Pure Type Systems*. March 1998, submitted for publication.
10. Barthe, G., Hatcliff, J., and Sørensen, M.H. A taxonomy of CPS and DS translations. Technical Report TR 98-11, Department of Computing and Information Sciences, Kansas State University, 1988.
11. Barthe, G., Hatcliff, J., and Thiemann, P. Monadic type systems: Pure type systems for impure settings. In *Proceedings of the Second Workshop on Higher-Order Operational Techniques in Semantics (HOOTS II)*, A. Gordon, A. Pitts, and C. Talcott (Eds.). Stanford, California, December 1997. Electronic Notes in Theoretical Computer Science, vol. 10.
12. Barthe, G. and Sørensen, M.H. Domain-free pure type systems. In *Proceedings of Logical Foundations of Computer Science LFCS'97*, S. Adian and A. Nerode (Eds.), Yaroslav, Russia, July 1997. Lecture Notes in Computer Science, vol. 1234, Springer-Verlag, pp. 9–20.
13. Berardi, S. Type Dependence and Constructive Mathematics. Ph.D. Thesis. University of Torino, 1990.
14. Consel, C. and Danvy, O. For a better support of static data flow. In *Conference on Functional Programming and Computer Architecture*, J. Hughes (Ed.). 1991. Lecture Notes in Computer Science, vol. 523, Springer-Verlag, pp. 495–519.
15. Coquand, T. and Herbelin, H. A-translation and looping combinators in pure type systems. *Journal of Functional Programming*, **4**(1):77–88, 1994.
16. Curry, H.B. and Feys, R. *Combinatory Logic*. North-Holland, 1958.
17. Danvy, O. Back to direct style. *Science of Computer Programming*, **22**(3):183–195, 1994.
18. Danvy, O. and Filinski, A. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, **2**(4):361–391, 1992.
19. Danvy, O. and Lawall, J. Back to direct style II: First-class continuations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*. San Francisco, California, June 1992. ACM Press, LISP Pointers, V(1):299–310.
20. Dowek, G., Huet, G., and Werner, B. On the existence of long $\beta\eta$-normal forms in the cube. In *Informal Proceedings of TYPES'93*, H. Geuvers (Ed.). Nijmegen, The Netherlands, May 1993, pp. 115–130.
21. Dussart, D., Hughes, J., and Thiemann, P. Type specialisation for imperative languages. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*. Amsterdam, The Netherlands, June 1997. ACM Press, pp. 204–216.
22. Felleisen, M. The Calculi of λ_v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher Order Programming Languages. Ph.D. Thesis. Indiana University, 1987.
23. Felleisen, M. and Friedman, D. Control operators, the SECD machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, M. Wirsing (Ed.), North-Holland, 1986, pp. 193–217.
24. Felleisen, M., Friedman, D., Kohlbecker, E., and Duba, B. A syntactic theory of sequential control. *Theoretical Computer Science*, **52**(3):205–237, 1987.
25. Flanagan, C., Sabry, A., Duba, B., and Felleisen, M. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*. Albuquerque, New Mexico, June 1993, pp. 237–247, SIGPLAN Notices 28(6).
26. Friedman, D., Wand, M., and Haynes, C. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 1991.
27. Gandy, R.O. An early proof of normalization by A.M. Turing. In J.P. Seldin and J.R. Hindley, Academic Press Limited, 1980, pp. 453–455.
28. Geuvers, H. Logics and Type Systems. Ph.D. Thesis. University of Nijmegen, 1993.
29. Geuvers, H. and Nederhof, M.J. A modular proof of strong normalisation for the Calculus of Constructions. *Journal of Functional Programming*, **1**:155–189, 1991.
30. Girard, J.-Y. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thèse d'Etat. Université Paris VII, 1972.

31. Griffin, T.G. A formulae-as-types notion of control. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*. San Francisco, California, January 1990. ACM Press, pp. 47–58.
32. de Groote, P. The conservation theorem revisited. In *Typed Lambda Calculus and Applications*, M. Bezem and J.F. Groote (Eds.). Utrecht, The Netherlands, March 1993. Lecture Notes in Computer Science, vol. 664, Springer-Verlag, pp. 163–178.
33. Harper, R., Honsell, F., and Plotkin, G. A framework for defining logics. *Journal of the ACM*, **40**(1):143–184, 1993. A Preliminary Version Appeared in the Proceedings of the First IEEE Symposium on Logic in Computer Science, June 1987, pp. 194–204.
34. Harper, R. and Lillibridge, M. Explicit polymorphism and CPS conversion. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*. Charleston, South Carolina, January 1993, ACM Press, pp. 206–219.
35. Harper, R. and Lillibridge, M. Polymormphic type assignment and CPS conversion. *LISP and Symbolic Computation*, **6**:361–380, 1993.
36. Harper, R. and Mitchell, J.C. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, **15**(2):211–252, 1993.
37. Harper, R. and Morrisett, G. Compiling polymorphism using intensional type analysis. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*. San Francisco, California, January 1995, ACM Press, pp. 130–141.
38. Hatcliff, J. Foundations of partial evaluation of functional programs with computational effects. In *Symposium on Partial Evaluation*, O. Danvy, R. Glück, and P. Thiemann (Eds.). September 1998 ACM Computing Surveys, vol. 30.
39. Hatcliff, J. and Danvy, O. A generic account of continuation-passing styles. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*. Portland, Oregon, January 1994, ACM Press, pp. 458–471.
40. Hatcliff, J. and Danvy, O. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, **7**:507–541, 1997. Special issue devoted to selected papers from the *Workshop on Logic, Domains, and Programming Languages*. Darmstadt, Germany, May 1995.
41. Hindley, J.R. and Seldin, J.P. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.
42. Howard, W. The formulae-as-types notion of construction. In T.H.B. Curry: Essays on Combinatory Logic, Lambda, Calculus and Formalism, J.P. Seldin and J.R. Hindley, Academic Press Limited, 1980, pp. 479–490.
43. Lawall, J. and Thiemann, P. Sound specialization in the presence of computational effects. In *Proceedings of Theoretical Aspects of Computer Software*, M. Abadi and T. Ito (Eds.). Sendai, Japan, September 1997. Lecture Notes in Computer Science, vol. 1281, Springer-Verlag, pp. 165–190.
44. Longo, G. and Moggi, E. Constructive natural deduction and its 'ω-set' interpretation. *Mathematical Structures in Computer Science*, **1**(2):215–254, 1991.
45. Meijer, E. and Peyton Jones, S. Henk: A typed intermediate language. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation*, Amsterdam, The Netherlands, June 1997.
46. Meyer, A.R. and Wand, M. Continuation semantics in typed lambda-calculi (summary). In *Logics of Programs*, R. Parikh (Ed.). Lecture Notes in Computer Science, vol. 193, Springer-Verlag, 1985, pp. 219–224.
47. Minamide, Y., Morrisett, G., and Harper, R. Typed closure conversion. In *Conference Record of the Twenty-third Annual ACM Symposium on Principles of Programming Languages*. St. Petersburg, Florida, January 1996, ACM Press, pp. 271–283.
48. Murthy, C. Extracting Constructive Contents from Classical Proofs. Ph.D. Thesis, Cornell University, 1990.
49. Nielsen, K. and Sørensen, M.H. Call-by-name CPS-translation as a binding-time improvement. In *Static Analysis Symposium*, A. Mycroft (Ed.). Glasgow, Scotland, September 1995. Lecture Notes in Computer Science, Springer-Verlag, vol. 983, pp. 296–313.
50. Peyton Jones, S.L. *The Implementation of Functional Programming Languages*. Prentice Hall International. 1987.
51. Peyton Jones, S.L., Hall, C., Hammond, K., Partain, W., and Wadler, P. The Glasgow Haskell compiler: A technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, Keele, 1993.
52. Plotkin, G. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, **1**:125–159, 1975.

53. Prawitz, D. *Natural Deduction: A Proof Theoretical Study*. Almquist & Wiksell, 1965.
54. Prawitz, D. Ideas and results of proof theory. In *The 2nd Scandinavian Logical Symposium*, J.E. Fenstad (Ed.). North-Holland, 1970, pp. 235–307.
55. Rehof, N.J. and Sørensen, M.H. The λ_Δ calculus. In *Theoretical Aspects of Computer Software*, M. Hagiya and J. Mitchell (Eds.). Sendai, Japan, April 1994. Lecture Notes in Computer Science, vol. 789, Springer-Verlag, pp. 516–542.
56. Reynolds, J.C. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation* **11**(4):7–105, 1998. Reprinted from the proceedings of the 25th ACM National Conference, 1972.
57. Sabry, A. Note on Axiomatizing the Semantics of Control Operators. Technical Report CIS-TR-96-03. Department of Computer and Information Science, University of Oregon, 1996.
58. Sabry, A. and Felleisen, M. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, **6**:289–360, 1993.
59. Sabry, A. and Felleisen, M. Is continuation passing useful for data-flow analysis? In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994, pp. 1–12. SIGPLAN Notices 29(6).
60. Sabry, A. and Wadler, P. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, **19**(6):916–941, 1997. Earlier version in the proceedings of the 1996 International Conference on Functional Programming.
61. Schmidt, D.A. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.
62. Seldin, J.P. Normalization and excluded middle I. *Studia Logica*, **XLVIII**(2):193–217, 1989.
63. Seldin, J.P. and Hindley, J.R. (Eds.). *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press Limited, 1980.
64. Shao, Z. and Appel, A.W. A type-based compiler for Standard ML. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, La Jolla, California, June 1995, pp. 116–129, SIGPLAN Notices 30(6).
65. Shivers, O. Control-Flow Analysis of Higher-Order Languages. Ph.D. Thesis. Carnegie Mellon University, 1991.
66. Sørensen, M.H. Strong normalization from weak normalization in typed λ-calculi. *Information and Computation*, **133**(1):35–71, 1997.
67. Stålmarck, G. Normalization theorems for full first order classical natural deduction. *Journal of Symbolic Logic*, **56**(1):129–149, 1991.
68. Steele, G.L., Jr. Rabbit: A compiler for scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.
69. Tait, W.W. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, **32**(2):190–212, 1967.
70. Tait, W.W. A realizability interpretation of the theory of species. In *Logic Colloquium*, R. Parikh (Ed.). Lecture Notes in Mathematics, vol. 453, Springer-Verlag, 1975, pp. 240–251.
71. Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, Pennsylvania, May 1996, pp. 181–192. SIGPLAN Notices 31(5).
72. Terlouw, J. *Een Nadere Bewijstheoretische Analyse van GSTT's*. Manuscript (in Dutch), 1989.
73. Werner, B. *Continuations, Evaluation Styles and Types Systems*. Manuscript, 1992.
74. Xi, H. Weak and strong normalizations in typed λ-calculi. In *Proceedings of TLCA'97*, P. de Groote and J. Hindley (Eds.). Nancy, France, Lecture Notes in Computer Science, vol. 1210, Springer-Verlag, April 1997, pp. 390–404.
75. Xi, H. and Pfenning, F. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Languages Design and Implementation*, Montréal, Canada, June 1998, pp. 249–257. SIGPLAN Notices 33(5).