Reduction Strategies for Program Extraction *

Maribel Fernández and Ian Mackie
Department of Computer Science, King's College London,
Strand, WC2R 2LS London, U.K. {ian,maribel}@dcs.kcl.ac.uk

and

Paula Severi
Departimento di Informatica, Università di Torino,
Corso Svizzera 185, 10149 Torino, Italy. severi@di.unito.it

and

Nora Szasz
In.Co., Facultad de Ingeniería,
Herrera y Reissig 525, Montevideo, Uruguay. nora@fing.edu.uy

Abstract

We introduce Pure Type Systems with Pairs generalising earlier work on program extraction in Typed Lambda Calculus. We model the process of program extraction in these systems by means of a reduction relation called σ-reduction, and give strategies for $\beta\sigma$-reduction which will be useful for an implementation of a proof assistant. More precisely, we give an algorithm to compute the σ-normal form of a term in a Pure Type System with Pairs, and show that this defines a projection from Pure Type Systems with Pairs to standard Pure Type Systems. This result shows that σ-reduction is an operational description of program extraction that is independent of the particular Typed Lambda Calculus specified as a Pure Type System. For β-reduction, we define weak and strong reduction strategies using Interaction Nets, generalising well-known efficient strategies for the λ-calculus to the general setting of Pure Type Systems.
Keywords: Specifications, program extraction, λ-calculus, pure type systems, interaction nets.

Resumen

Presentamos Sistemas de Tipos Puros con Pares, generalizando trabajos previos sobre extracción de programas en Cálculo Lambda Tipado. Modelizamos el proceso de extracción de programas en estos sistemas a través de una relación de reducción σ, y definimos estrategias de reducción $\beta\sigma$ que servirán para implementar un asistente de prueba. Más precisamente, damos un algoritmo de normalización σ para Sistemas de Tipos Puros con Pares, y mostramos que define una proyección de los Sistemas de Tipos Puros con Pares a los Sistemas de Tipos Puros. Este resultado muestra que la reducción σ es una descripción operacional de la extracción de programas que es independiente de Cálculo Lambda Tipado particular especificado en el Sistema de Tipos Puros. Para la β-reducción, definimos estrategias débiles y fuertes usando redes de interacción, generalizando de esta forma las estrategias eficientes del cálculo λ al marco general de los Sistemas de Tipos Puros.
Palabras claves: Especificaciones, extracción de programas, cálculo λ, sistemas de tipos puros, redes de interacción.

*This work has been partially supported by the ECOS-Sud program of cooperation between France and Uruguay. The third author is also partially supported by IST-2001-322222 MIKADO; IST-2001-33477 DART.

1 Introduction

A specification of a program, such as *for every finite list of natural numbers there is a sorted permutation*, is in general of the form $\forall x.\exists y.P(x,y)$. In Typed Lambda Calculus, this is expressed as a type $\Pi x{:}A.\Sigma y{:}B.P(x,y)$. The idea of program extraction is to extract from an inhabitant t of such a type a function $f : A \to B$ such that $P(x, f(x))$ holds for all x.

In the last two decades several approaches in Typed Lambda Calculus to program extraction have been studied [7, 24, 6, 30, 31, 27, 19, 5]. Taking as starting point the Theory of Specifications introduced in [27, 28], where the process of extracting a program from a proof of a specification is modelled by means of a reduction relation $\to_\sigma$, in [8] we show how the existing approaches to program extraction in Typed Lambda Calculus can all be modelled as Pure Type Systems extended with pairs and σ-reduction ($\mathrm{PTS}_{\beta\sigma}$). In this paper we go one step further: we give an algorithm of program extraction for $\mathrm{PTS}_{\beta\sigma}$ and prove a general Program Extraction Theorem, which apply to several calculi like the Theory of Specifications based on Martin Löf's Type Theory [27], the Theory of Specifications based on the Calculus of Constructions [28, 12] and the method of extracting programs based on realisability interpretations by C. Paulin [23].

Moreover we define strategies of $\beta\sigma$-reduction which will be useful for implementing $\mathrm{PTS}_{\beta\sigma}$ as a proof assistant. For the type checking algorithm and the evaluation of programs and proofs, we need to compute the $\beta\sigma$-normal form of a term. Since β-reduction should be restricted to σ-normal forms to ensure Subject Reduction and Strong Normalisation [26, 12], we first compute the σ-normal form and then the β-weak head normal form or the β-normal form if necessary. This leads us to consider two independent modules for $\beta\sigma$-reduction:

1. Module Sigma. This module computes the σ-normal form of a term in a $\mathrm{PTS}_{\beta\sigma}$, generalising a normalisation strategy defined in [27]. The result is typeable in a Pure Type System with only β-reduction (PTS_β or simply PTS): we prove that σ-normalisation is a projection mapping from $\mathrm{PTS}_{\beta\sigma}$ to PTS_β.

2. Module Beta. The second module deals with β-reduction for the standard Pure Type Systems with only β-reduction. We develop reduction strategies using Interaction Nets [15] in the general setting of Pure Type Systems. Interaction Nets have been successfully used to implement the (untyped or simply typed) λ-calculus, where efficient strategies for computing normal forms of terms, including the optimal reduction strategy, can be achieved [20, 14]. It is therefore natural to apply these techniques to proof assistants based on Pure Type Systems. We adapt the YALE encoding of the λ-calculus [20] to Pure Type Systems. In order to prove the correctness of our normalisation strategy we give a presentation of Pure Type Systems using a syntax with explicit substitutions and explicit resource management (copying and erasure). The reduction rules for this new presentation define a weakly normalising strategy in the spirit of closed reduction [10], which is free from α-conversion and still permits certain reductions to take place under abstractions and propagations of substitutions under abstractions. To obtain a strong reduction strategy computing full normal forms, as is required to test β-equality in the context of Pure Type Systems, we use the technique of normalisation by evaluation [4, 20].

Related Work on Program Extraction. The Theory of Specifications of [27] and the one of [28] can be both described as $\mathrm{PTS}_{\beta\sigma}$. Moreover the Verification Calculus of [27] and the one of [28] are both described as Pure Type Systems with only β-reduction. Therefore, the program extraction theorems proved in [27, 28] that state that the σ-normal form is a mapping from the Theory of Specifications to the Verification Calculus are particular cases of our Program Extraction Theorem for $\mathrm{PTS}_{\beta\sigma}$.

In Coq the extraction procedure is based on an external function that uses realisability interpretations [24, 23]. The system in [23] does not have the σ-conversion rule, instead the process of program extraction is described by two external functions $\mathcal{E}$ and $\mathcal{R}$. The result computed by these functions coincide with the normal form of the σ-reduction.

In [9] realisability is expressed by means of a judgement relation. Though this judgement internalises realisability in the system, the type of a realizer is still defined by an external function. A rule called "type extraction" which states that any realizer is typeable has to be added to obtain a system where provability and Kreisel's modified realisability coincide. This rule connects the judgement of realisability with the external function. As a consequence of our general Program Extraction Theorem, using Pairs the type extraction rule is derivable.

In LEGO [18, 31] a specification mechanism is implemented based on a pair of a type and a predicate over it [6, 22, 17]. Specifications of programs are described by an existential type in ECC [16], and first and second order deliverables are defined. In our approach, the notion of specification is more general, and there is no need to distinguish between first and second order deliverables.

In $\lambda\omega_L$ [25] specifications are always pairs which get manipulated at the meta-level. The logic $\lambda\omega_L$ can be coded as a $\mathrm{PTS}_{\beta\sigma}$. In $\lambda\omega_L$ the derivation rules are given in pairs – called coupled derivation rules – in order to construct both components simultaneously. For each constructor a couple of derivation rules is given: one for the program-part and one for the proof-part. The notion of specification and coupled derivation rules of [25] can be made explicit in the syntax using our notion of pair (for more detail see [8]).

Related Work on Interaction Nets. The results described in this paper are related to previous work on the use of Interaction Nets as a tool to implement efficient strategies in the λ-calculus [20, 14].

The closed reduction strategy of evaluation for the λ-calculus was defined in [10] using conditional reduction rules on λ-terms with an extended syntax where substitutions, erasing and copying are explicit. In order to internalise the conditions in the rewrite rules, a syntax with director strings was introduced in [11] and generalised in [29] to achieve strong reduction.

YALE [20] implements closed reduction for PCF using interaction nets. In this paper, we extend the YALE encoding in order to deal with terms in Pure Type Systems, and define the corresponding closed reduction strategies (weak and strong).

Organisation. Section 2 gives the basic definitions and notations for Pure Type Systems that we will use in the rest of the paper. We define Pure Type Systems with Pairs and $\beta\sigma$-reduction in Section 3, and give an example of program extraction in Section 4. In Section 5 we give a normalisation strategy for σ-reduction, and prove the Program Extraction Theorem. In Section 6 we define closed reduction on the set of pseudoterms for Pure Type Systems extended with explicit substitutions, copy and erasing; we show the implementation of the closed reduction strategy for β-normalisation using interaction nets in the appendix. We conclude the paper in Section 7.

2 Background

We assume the reader to be familiar with the notion of Pure Type Systems (PTS_β or simply PTS for short), and introduce the notation that we will use.

Definition 2.1. We say that $\mathsf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ is *a specification of a Pure Type System* if $\mathcal{S}$ is a set of elements called *sorts*, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of pairs called *axioms* and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a set of triples called *rules for the* Π*-constructor.*

We write $(k_1, k_2) \in \mathcal{R}$ when $(k_1, k_2, k_2) \in \mathcal{R}$.

Definition 2.2. The set $\mathcal{T}$ of pseudoterms is defined as the least set that contains a set $\mathcal{V}$ of variables, the set $\mathcal{S}$ of sorts, abstractions $\lambda x{:}U.u$, applications $(u\,v)$ and products $\Pi x{:}U.V$.

If x does not occur in V we write $U \to V$ instead of $\Pi x{:}U.V$.

Definition 2.3. A Pure Type System with specification $\mathsf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ is denoted $\lambda(\mathsf{S})$, and defined by the rules shown in Figure 1. Typeable pseudoterms are called terms.

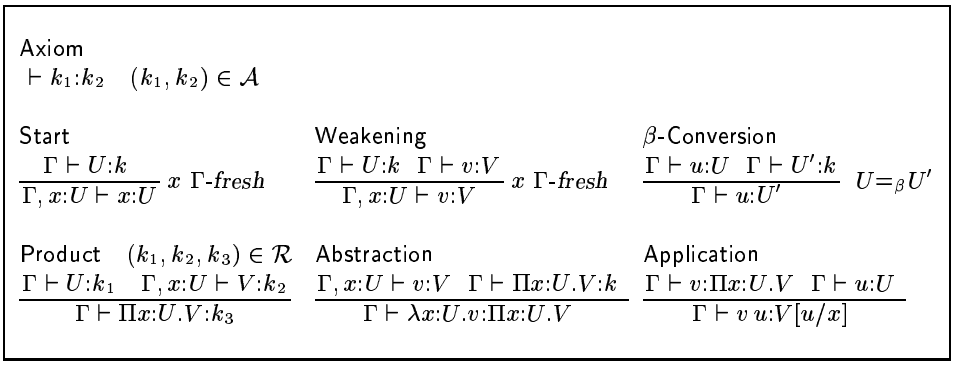


Figure 1: Typing rules for Pure Type Systems

3 Pure Type Systems with Pairs

In this section we define the notion of Pure Type Systems with Pairs ($\mathrm{PTS}_{\beta\sigma}$) and give some examples.

Definition 3.1. The set $\mathcal{T}_P$ of pseudoterms with pairs is defined by adding pairs $\langle u\,,v\rangle$ to the set $\mathcal{T}$ (see Definition 2.2). We use the same notation of pairs for types and objects, i.e. we write $\Sigma x{:}A.P =^{\mathsf{def}} \langle A\,,\lambda x{:}A.P\rangle$.

We assume the set of variables is split in three pairwise disjoint sets: *data-variables*, *prop-variables* and *spec-variables*. We denote data-variables by x_d, y_d, prop-variables by x_p, y_p and spec-variables by x_s, y_s. Similarly the set of sorts is split in three pairwise disjoint sets: *data-sorts*, *prop-sorts* and *spec-sorts*.

We partition the set $\mathcal{T}_P$ into three sets of data, prop, and spec-pseudoterms for data types, propositions and specifications respectively. This partition can be made by using the notion of *heart*:

Definition 3.2. The *heart of a pseudoterm* is defined as follows:
$\mathsf{heart}(u) = u$ if u is either a variable, a sort or a pair.
$\mathsf{heart}(\Pi x{:}U.V) = \mathsf{heart}(\lambda x{:}U.V) = \mathsf{heart}(V\ U) = \mathsf{heart}(V)$.

Definition 3.3. Let $U \in \mathcal{T}_P$.

1. We say that U is a *data-pseudoterm* if $\mathsf{heart}(U)$ is a data-variable or a data-sort. We denote data-pseudoterms by A, B, a, b ....
2. We say that U is a *prop-pseudoterm* if $\mathsf{heart}(U)$ is a prop-variable or a prop-sort. We denote prop-pseudoterms by P, Q, $p, q, \ldots$.
3. We say that U is a *spec-pseudoterm* if $\mathsf{heart}(U)$ is a spec-variable, a spec-sort or a pair. We denote spec-pseudoterms by S, T, $s, t, \ldots$.

We use the metavariables U, V, u, v ... for an arbitrary pseudoterm.

Example 3.4. Assume Nat and Eq are a data-variable and a prop-variable respectively [1].

1. Data-pseudoterms are used to express data types and programs as:

$$\Pi x{:}\mathsf{Nat}.\mathsf{Nat} = \mathsf{Nat} \to \mathsf{Nat}$$
$$\lambda x{:}\mathsf{Nat}.x$$

[1] In Pure Type Systems constants are treated as variables.

2. Prop-pseudoterms are used to express propositions and proofs as:

$$\Pi x{:}\mathsf{Nat}.(\mathsf{Eq}\ x\ x)$$
$$\lambda x{:}\mathsf{Nat}.\mathsf{refl}_{\mathsf{Nat}}\ x$$

where $\mathsf{refl}_{\mathsf{Nat}}\ x$ is a proof of $\mathsf{Eq}\ x\ x$.

3. Spec-pseudoterms are used to express specifications (and implementations) of programs as:

$$Spec_{\mathsf{pred}} = \Pi n{:}\mathsf{Nat}.\Sigma m{:}\mathsf{Nat}.n > 0 \to \mathsf{Eq}\ n\ (\mathsf{suc}\, m)$$

We recall that $\Sigma x{:}A.P =^{\mathsf{def}} \langle A\,, \lambda x{:}A.P\rangle$, and $A \to B =^{\mathsf{def}} \Pi x{:}A.B$ when x is not free in B. In first-order logic $Spec_{\mathsf{pred}}$ would be written as $\forall n.\exists m.(n > 0) \to (n = \mathsf{suc}\, m)$. This specification states that every natural number greater than zero has a predecessor.

Definition 3.5. A *specification* $\mathsf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{R}')$ *for a Pure Type System extended with Pairs* consists of a set $\mathcal{S}$ of *sorts*, a set $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ of *axioms*, a set $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ of *rules for the* Π*-constructor* and a set $\mathcal{R}' \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ of *rules for the Pair constructor.*

We consider only specifications that satisfy the following conditions:

1. If $(k_1, k_2) \in \mathcal{A}$ then the sorts k_1, k_2 are both data-sorts or both prop-sorts or both spec-sorts.
2. If $(k_1, k_2), (k_1, k_2') \in \mathcal{A}$ then $k_2 = k_2'$.
3. If $(k_1, k_2, k_3) \in \mathcal{R}$ then $k_2 = k_3$. We write $(k_1, k_2) \in \mathcal{R}$ instead of $(k_1, k_2, k_2) \in \mathcal{R}$.
4. If $(k_1, k_2, k_3) \in \mathcal{R}'$ then k_1 is a data-sort, k_2 is a prop-sort and k_3 is a spec-sort.
5. If $(k_1, k_2, k_3) \in \mathcal{R}$ and $(k_1', k_2', k_3) \in \mathcal{R}'$ then $k_1 = k_1'$ and $k_2 = k_2'$.
6. For all spec-sorts k, there exists k_1, k_2 such that $(k_1, k_2, k) \in \mathcal{R}'$.

We make the following conventions for the syntax: [2]

1. In $\lambda x{:}U.V$ and $\Pi x{:}U.V$, x and U are both data-pseudoterms, both prop-pseudoterms, or both spec-pseudoterms. The same restriction applies to any typing context Γ, i.e. if $x{:}U \in \Gamma$ then both x and U are data-pseudoterms, both are prop-pseudoterms or both are spec-pseudoterms.
2. In $\langle u\,, v\rangle$, we assume that u is a data-pseudoterm and v is a prop-pseudoterm.

In Figure 2 we show the reduction rules defining $\to_\sigma$. This reduction gives an operational semantics to program extraction when we use pairs to express our specifications of programs. We assume that for each spec-variable x_s there is one and only one associated pair $\langle x_d\,, x_p\rangle$. This is expressed by the rule Splitting (note that x_s is a variable of the PTS, but in the rewrite system with only σ-reduction it is treated as a constant).

Example 3.6. By applying the first distributivity rule, we have that:

$$\begin{array}{rcl} Spec_{\mathsf{pred}} & = & \Pi n{:}\mathsf{Nat}.\Sigma m{:}\mathsf{Nat}.n > 0 \to \mathsf{Eq}\ n\ (\mathsf{suc}\, m) \\ & \to_\sigma & \Sigma f{:}\mathsf{Nat} \to \mathsf{Nat}.n > 0 \to \mathsf{Eq}\ n\ (\mathsf{suc}\,(f\ n)) \end{array}$$

Note that this rule is obtained by replacing the reduction by the implication in the Axiom of Choice.

Because of the correspondence between spec-variables and pairs of data- and prop-variables, the following notion of *completeness* is needed to restrict substitution of spec-variables x_s. We also need a modified definition of *freshness* with respect to a typing context.

[2] See [27, 28] where these restrictions are imposed by a grammar.

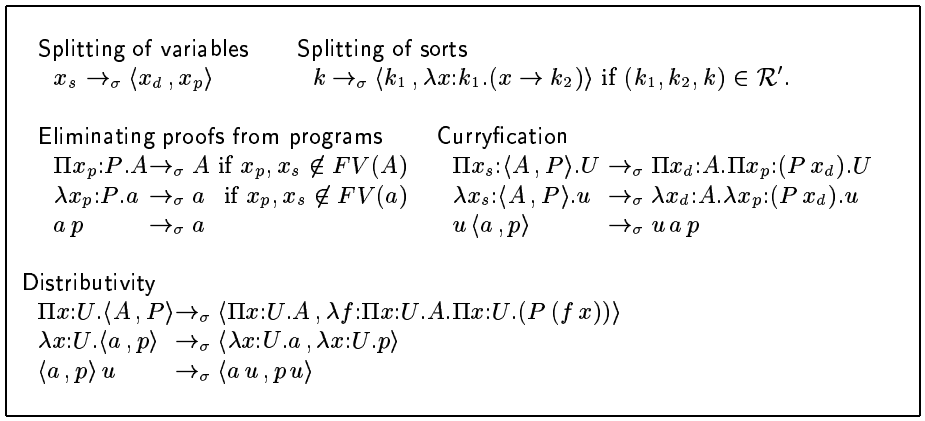


Figure 2: Definition of σ-reduction

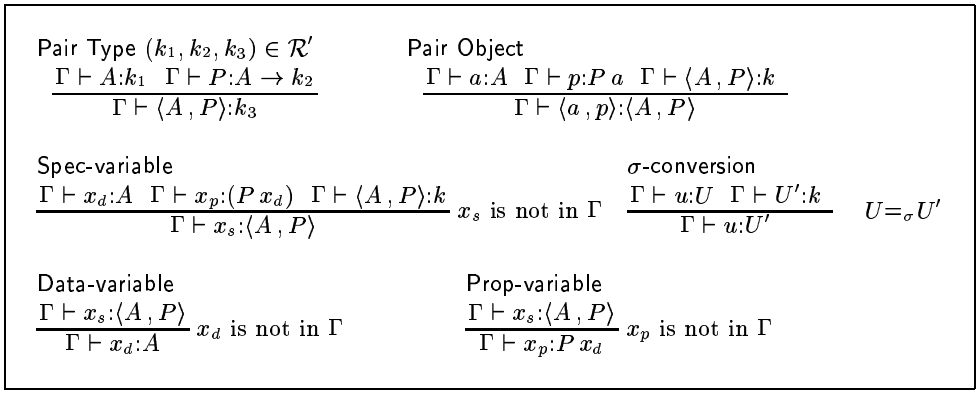


Figure 3: Typing rules for Pairs

- The variable x_d (resp. x_p) is *complete* for a term u if $x_s \notin FV(u)$. The variable x_d (resp. x_p) is *fresh* for a context Γ (Γ-fresh for short) if x_s and x_d (resp. x_p) do not occur in Γ.
- The variable x_s is *complete* for a term u if $x_d, x_p \notin FV(u)$. It is *fresh* for a context Γ if x_s, x_d and x_p do not occur in Γ.

We make the following conventions for substitutions and β-reduction:

1. Whenever a substitution $u[v/x]$ is performed we require that the variable x is complete for u.
2. β-reduction is restricted to σ-normal forms [3].

Definition 3.7. Let $\mathsf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{R}')$ where $\mathcal{S}$ is a set of sorts, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of axioms, $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a set of rules for the product and $\mathcal{R}' \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a set of rules for the pair constructor. The notion of *Pure Type System extended with Pairs*, abbreviated as $\mathrm{PTS}_{\beta\sigma}$ or $\lambda^\sigma(\mathsf{S})$, is inductively defined by adding the rules in Figure 3 to the rules of Pure Type Systems.

Due to the addition of the σ-conversion rule, Pairs become stronger than the strong-Σ [16]. Pairs have at least the power of strong-Σ since it is possible to code the first and second projections (it is not necessary to add them as primitives). The projections π_1 and π_2 can be coded for each

[3] See [26] for examples that show that if β-reduction is not restricted to σ-normal forms then Subject Reduction and Strong Normalisation for β-reduction do not hold.

type of the form $\langle A\,,P\rangle$ as follows: $\pi_1 =^{\mathsf{def}} \lambda x_s{:}\langle A\,,P\rangle.x_d$ and $\pi_2 =^{\mathsf{def}} \lambda x_s{:}\langle A\,,P\rangle.x_p$. Moreover, Pairs have an additional property which makes program extraction always possible. This additional property corresponds to the internalization of the notion of realizability: the proof of any specification computationally equal to a basic specification $\langle A\,,P\rangle$ is computationally equal to a pair $\langle a\,,p\rangle$ with $a{:}A$ and $p{:}Pa$. Then, program extraction from an inhabitant of a specification consists in just taking the first component of the pair. For instance, an inhabitant of a specification $\Pi x{:}A.\langle B\,,\lambda y:B.(P\,x\,y)\rangle$ reduces to $\langle f\,,q\rangle$ where $f{:}A\to B$ is the extracted program and q is the proof of its correctness $\Pi x{:}A.(P\,x\,(f\,x))$.

We now give examples of Pure Type System with Pairs.

Example 3.8.

1. This example is a variant of λP.

Specification A

$\mathcal{S}$	$\star_{\mathsf{d}}\quad \Box_{\mathsf{d}}$ data-sorts
	$\star_{\mathsf{p}}\quad \Box_{\mathsf{p}}$ prop-sorts
	$\star_{\mathsf{s}}\quad \Box_{\mathsf{s}}$ spec-sorts
$\mathcal{A}$	$(\star_{\mathsf{d}},\Box_{\mathsf{d}})\quad (\star_{\mathsf{p}},\Box_{\mathsf{p}})\quad (\star_{\mathsf{s}},\Box_{\mathsf{s}})$
$\mathcal{R}$	$(\star_{\mathsf{u}},\star_{\mathsf{v}})\quad (\star_{\mathsf{u}},\Box_{\mathsf{u}})$ for $\mathsf{u},\mathsf{v}\in\{\mathsf{d},\mathsf{p},\mathsf{s}\}$
	$(\star_{\mathsf{d}},\Box_{\mathsf{p}})$
$\mathcal{R}'$	$(\star_{\mathsf{d}},\star_{\mathsf{p}},\star_{\mathsf{s}})$

The Theory of Specifications presented in [27] is embedded in $\lambda^{\sigma}(\mathsf{A})$.

2. This is a variant of the Calculus of Constructions with an infinite hierarchy of universes.

Specification B

$\mathcal{S}$	$\star_{\mathsf{d}}{}^{\mathsf{i}}\quad \star_{\mathsf{p}}{}^{\mathsf{i}}\quad \star_{\mathsf{s}}{}^{\mathsf{i}}$ for $\mathsf{i}\in N$
$\mathcal{A}$	$(\star_{\mathsf{d}}{}^{\mathsf{i}},\star_{\mathsf{d}}{}^{\mathsf{i}+1})\quad (\star_{\mathsf{p}}{}^{\mathsf{i}},\star_{\mathsf{p}}{}^{\mathsf{i}+1})\quad (\star_{\mathsf{s}}{}^{\mathsf{i}},\star_{\mathsf{s}}{}^{\mathsf{i}+1})$ for $\mathsf{i}\in N$
$\mathcal{R}$	$(\star_{\mathsf{u}}^{\mathsf{i}},\star_{\mathsf{v}}^{\mathsf{j}})$ for $\mathsf{i}\le\mathsf{j}$ or $\mathsf{j}=0$ and $\mathsf{u},\mathsf{v}\in\{\mathsf{d},\mathsf{p},\mathsf{s}\}$
$\mathcal{R}'$	$(\star_{\mathsf{d}}{}^{\mathsf{i}},\star_{\mathsf{p}}{}^{\mathsf{i}},\star_{\mathsf{s}}{}^{\mathsf{i}})$ for $\mathsf{i}\in N$

The Theory of Specifications presented in [28] is embedded in $\lambda^{\sigma}(\mathsf{B})$.

3. This example is a variant of the Calculus of Constructions.

Specification C

$\mathcal{S}$	$\star_{\mathsf{d}}\quad \star_{\mathsf{p}}\quad \star_{\mathsf{s}}\quad \Box_{\mathsf{d}}\quad \Box_{\mathsf{p}}\quad \Box_{\mathsf{s}}$
$\mathcal{A}$	$(\star_{\mathsf{d}},\Box_{\mathsf{d}})\quad (\star_{\mathsf{p}},\Box_{\mathsf{p}})\quad (\star_{\mathsf{s}},\Box_{\mathsf{s}})$
$\mathcal{R}$	(k_1,k_2) for $k_1,k_2\in\mathcal{S}$
$\mathcal{R}'$	$(\star_{\mathsf{d}},\star_{\mathsf{p}},\star_{\mathsf{s}})$

The system in [23] can be embedded in $\lambda^{\sigma}(\mathsf{C})$. Also the system $\lambda\omega_L$ of [25] can be coded in the Pure Type System with Pairs $\lambda^{\sigma}(\mathsf{C})$.

4 An Example of Program Extraction

In order to develop this example of program extraction, we have first to illustrate how σ-reduction is extended to deal with natural numbers [4]. For this, we consider the Pure Type System with Pairs $\lambda^{\sigma}(\mathsf{A})$ defined by the specification A of Example 3.8, part 1. The inductive data type Nat with constructors zero and suc has the following elimination rule:

[4] In [28] we have done this extension for the Calculus of Inductive Constructions.

$$\text{(natrec)}\ \frac{\begin{array}{l}\Gamma \vdash n : \mathsf{Nat} \\ \Gamma, x{:}\mathsf{Nat} \vdash U : k \text{ if } k \in \{\star_{\mathsf{d}}, \star_{\mathsf{p}}, \star_{\mathsf{s}}\} \\ \Gamma \vdash u_1 : U[n/x] \\ \Gamma \vdash u_2 : \Pi m{:}\mathsf{Nat}.(U[m/x] \to U[\mathsf{suc}\, m/x])\end{array}}{\Gamma \vdash (\mathsf{natrec}\ u_1\ u_2\ n) : U[n/x]}$$

For the sake of exposition we omitted the first argument of natrec that corresponds to the type U. The reduction for natrec is defined as follows:

$$\begin{array}{lcl}(\mathsf{natrec}\ u_1\ u_2\ 0) & \to_\iota & u_1 \\ (\mathsf{natrec}\ u_1\ u_2\ \mathsf{suc}\ m) & \to_\iota & (u_2\, m\, (\mathsf{natrec}\ u_1\ u_2\ m))\end{array}$$

In order to obtain the correct reduction for this operator when U is a specification, we must extend the definition of σ-reduction to include the following distributivity law. In that rule we will use the following abbreviations:

$\hat{P} = \lambda n{:}Nat.(P\, n\, (\mathsf{natrec}\ a_1\ a_2\ n))$,

$\hat{p_2} = \lambda n{:}Nat.\lambda q{:}(\hat{P}\, n).(p_2\, n\, (\mathsf{natrec}\ a_1\ a_2\ n)\, q)$

(Distributivity of natrec over pairs)
$(\mathsf{natrec}\ \langle a_1\, , p_1\rangle\ \langle a_2\, , p_2\rangle\ n) \quad \to_\sigma \quad \langle (\mathsf{natrec}\ a_1\ a_2\ n)\, , (\mathsf{natrec}\ p_1\ \hat{p_2}\ n)\rangle$

Note that the predicate $\hat{P}$ is in fact the predicate P applied to the first component of the pair.

We are now ready to give an example of program extraction. We consider the specification stating that every natural number not equal to zero has a predecessor:

$$Spec_{\mathsf{pred}} = \Pi n{:}\mathsf{Nat}.\Sigma m{:}\mathsf{Nat}.n > 0 \to \mathsf{Eq}\ n\ (\mathsf{suc}\, m)$$

We use the following abbreviations and assumptions:

- $A = \lambda n{:}\mathsf{Nat}.\mathsf{Nat}$,
- $P = \lambda n{:}\mathsf{Nat}.\lambda m{:}\mathsf{Nat}.n > 0 \to \mathsf{Eq}\ n\ (\mathsf{suc}\, m)$,
- $U = \lambda n{:}\mathsf{Nat}.\Sigma m{:}\mathsf{Nat}.(P\, n\, m)$,
- there are terms p_0, p_m such that $\vdash p_0 : P\, 0\, 0$ and $m : \mathsf{Nat} \vdash p_m : P\, (\mathsf{suc}\, m)\, m$,
- $q = \lambda m{:}\mathsf{Nat}.\lambda x{:}(U\, m).p_m$.

The following term is an inhabitant of the type $Spec_{\mathsf{pred}}$:

$$\begin{array}{lll}s = \lambda n : \mathsf{Nat}.\, (\mathsf{natrec} & \langle 0\, , p_0\rangle & \\ & (\lambda m{:}\mathsf{Nat}.\lambda x{:}(U\, m).\langle m\, , p_m\rangle) & \\ & n)\end{array}$$

Using σ-reduction relation, we can reduce s in two stages. In the first stage, s is reduced to a pair consisting of a program and its correctness proof:

$$\begin{array}{ll}s \twoheadrightarrow_\sigma & \langle \lambda n{:}\mathsf{Nat}.\mathsf{natrec}\ 0\ (\lambda m{:}\mathsf{Nat}.\lambda x_d{:}\mathsf{Nat}.\lambda x_p{:}(P\, m\, x_d).m)\, n, \\ & \lambda n{:}\mathsf{Nat}.\mathsf{natrec}\ p_0\, q\, n\rangle\end{array}$$

The first component of this pair contains the abstraction $\lambda x_p{:}(P\, m\, x_d).m$ which is expecting a proof of $(P\, m\, x_d)$ and yields a natural number m. This abstraction can be removed using one of the rules that erase logical parts from a program in the definition of σ-reduction:

$$\begin{array}{l}\lambda n{:}\mathsf{Nat}.\mathsf{natrec}\ 0\ (\lambda m{:}\mathsf{Nat}.\lambda x_d{:}\mathsf{Nat}.\lambda x_p{:}(P\, m\, x_d).m)\, n \\ \to_\sigma \text{ (by erasure, also } \to_\sigma) \\ \lambda n{:}\mathsf{Nat}.\mathsf{natrec}\ 0\ (\lambda m{:}\mathsf{Nat}.\lambda x_d{:}\mathsf{Nat}.m)\, n =_{\mathsf{def}} \mathsf{pred}\end{array}$$

The last term pred is the extracted program.

5 A Strategy for σ-normalisation

In this section we describe a strategy for σ-normalisation. More precisely, we define a function to compute the σ-normal form of a term in a $\mathrm{PTS}_{\beta\sigma}$ (see Definition 13.1.1 of reduction strategy in [2]), and prove that the resulting term is typeable in a PTS_{β}. For this, we adapt the function that computes σ-normal forms in the Theory of Specifications of [27] to the setting of Pure Type Systems with Pairs.

Definition 5.1. We define the function $\mathsf{nf}_\sigma : \mathcal{T}_P \to \mathcal{T}_P$ by induction on pseudoterms:

Sorts.

$\mathsf{nf}_\sigma(k) = k$ if k is a data or prop-sort.

$\mathsf{nf}_\sigma(k) = \langle k_1 \,, \lambda x{:}k_1.x \to k_2 \rangle$ if $(k_1, k_2, k) \in \mathcal{R}'$.

Variables.

$\mathsf{nf}_\sigma(x) = \langle x_d \,, x_p \rangle$ if x is associated to the pair $\langle x_d \,, x_p \rangle$.
$\mathsf{nf}_\sigma(x) = x$ otherwise.

Products.

1. $\mathsf{nf}_\sigma(\Pi x{:}P.A) \;= \mathsf{nf}_\sigma(A)$
2. Let $\mathsf{nf}_\sigma(x) \;= \langle x_d \,, x_p \rangle$ and $\mathsf{nf}_\sigma(S) \;= \langle A \,, P \rangle$. Then,
   $\mathsf{nf}_\sigma(\Pi x{:}S.B) \;=\; \Pi x_d{:}A.\mathsf{nf}_\sigma(B)$
3. Let $\mathsf{nf}_\sigma(x) \;= \langle x_d \,, x_p \rangle$ and $\mathsf{nf}_\sigma(S) \;= \langle A \,, P \rangle$. Then,
   $\mathsf{nf}_\sigma(\Pi x{:}S.Q) \;=\; \Pi x_d{:}A.\Pi x_p{:}(P\, x_d).\mathsf{nf}_\sigma(Q)$
4. Let $\mathsf{nf}_\sigma(T) \;= \langle B \,, Q \rangle$ and $\mathsf{nf}_\sigma(A) = A'$. Then,
   $\mathsf{nf}_\sigma(\Pi x{:}A.T) \;= \langle \Pi x{:}A'.B \,, \lambda f{:}\Pi x{:}A'.B.\Pi x{:}A'.Q\,(f\, x) \rangle$
5. Let $\mathsf{nf}_\sigma(T) \;= \langle B \,, Q \rangle$. Then,
   $\mathsf{nf}_\sigma(\Pi x{:}P.T) \;= \langle B \,, \lambda z{:}B.\Pi x{:}\mathsf{nf}_\sigma(P).(Q\, z) \rangle$
6. Let $\mathsf{nf}_\sigma(S) \;= \langle A \,, P \rangle$, $\mathsf{nf}_\sigma(T) \;= \langle B \,, Q \rangle$ and $\mathsf{nf}_\sigma(x) \;= \langle x_d \,, x_p \rangle$. Then,
   $\mathsf{nf}_\sigma(\Pi x{:}S.T) \;= \langle \Pi x_d{:}A.B \,, \lambda f{:}\Pi x_d{:}A.B.\Pi x_d{:}A.\Pi x_p{:}(P\, x_d).Q\,(f\, x_d) \rangle$
7. Otherwise,
   $\mathsf{nf}_\sigma(\Pi x{:}U.V) \;=\; \Pi x{:}\mathsf{nf}_\sigma(U).\mathsf{nf}_\sigma(V)$

Abstractions.

1. $\mathsf{nf}_\sigma(\lambda x{:}P.a) \;= \mathsf{nf}_\sigma(a)$
2. Let $\mathsf{nf}_\sigma(S) \;= \langle A \,, P \rangle$ and $\mathsf{nf}_\sigma(x) \;= \langle x_d \,, x_p \rangle$. Then,
   $\mathsf{nf}_\sigma(\lambda x{:}S.a) \;= \lambda x_d{:}A.\mathsf{nf}_\sigma(a)$
3. Let $\mathsf{nf}_\sigma(S) \;= \langle A \,, P \rangle$ and $\mathsf{nf}_\sigma(x) \;= \langle x_d \,, x_p \rangle$. Then,
   $\mathsf{nf}_\sigma(\lambda x{:}S.p) \;= \lambda x_d{:}A.\lambda x_p{:}(P\, x_d).\mathsf{nf}_\sigma(p)$
4. Let $\mathsf{nf}_\sigma(s) \;= \langle a \,, p \rangle$. Then,
   $\mathsf{nf}_\sigma(\lambda x{:}A.s) \;= \langle \lambda x{:}\mathsf{nf}_\sigma(A)a. \,, \lambda x{:}\mathsf{nf}_\sigma(A).p \rangle$
5. Let $\mathsf{nf}_\sigma(s) \;= \langle a \,, p \rangle$ Then,
   $\mathsf{nf}_\sigma(\lambda x{:}P.s) \;= \langle a \,, \lambda x{:}\mathsf{nf}_\sigma(P).p \rangle$
6. Let $\mathsf{nf}_\sigma(s) \;= \langle a \,, p \rangle$, $\mathsf{nf}_\sigma(S) \;= \langle A \,, P \rangle$ and $\mathsf{nf}_\sigma(x) \;= \langle x_d \,, x_p \rangle$. Then,
   $\mathsf{nf}_\sigma(\lambda x{:}S.s) \;= \langle \lambda y_d{:}A.a \,, \lambda x_d{:}A.\lambda x_p{:}(P\, x_d).p \rangle$
7. Otherwise,
   $\mathsf{nf}_\sigma(\lambda x{:}U.u) \;= \lambda x{:}\mathsf{nf}_\sigma(U).\mathsf{nf}_\sigma(u)$

Applications.

1. $\mathsf{nf}_\sigma(a\,p) \;=\; \mathsf{nf}_\sigma(a)$
2. Let $\mathsf{nf}_\sigma(t) \;=\langle b\,,q\rangle$. Then,
   $\mathsf{nf}_\sigma(a\,t) \;=\; \mathsf{nf}_\sigma(a)\,b$
3. Let $\mathsf{nf}_\sigma(t) \;=\langle b\,,q\rangle$. Then,
   $\mathsf{nf}_\sigma(p\,t) \;=\; \mathsf{nf}_\sigma(p)\,b\,q$
4. Let $\mathsf{nf}_\sigma(s) \;=\langle a\,,p\rangle$ and $\mathsf{nf}_\sigma(b) = b'$ Then,
   $\mathsf{nf}_\sigma(s\,b) \;=\langle a\,b'\,,p\,b'\rangle$
5. Let $\mathsf{nf}_\sigma(s) \;=\langle a\,,p\rangle$. Then,
   $\mathsf{nf}_\sigma(s\,q) \;=\langle a\,,p\,\mathsf{nf}_\sigma(q)\rangle$
6. Let $\mathsf{nf}_\sigma(s) \;=\langle a\,,p\rangle$ and $\mathsf{nf}_\sigma(t) \;=\langle b\,,q\rangle$. Then,
   $\mathsf{nf}_\sigma(s\,t) \;=\langle a\,b\,,p\,b\,q\rangle$
7. Otherwise,
   $\mathsf{nf}_\sigma(u\,v) \;=\; \mathsf{nf}_\sigma(u)\,\mathsf{nf}_\sigma(v)$

Pairs.
$\mathsf{nf}_\sigma(\langle a\,,p\rangle) \;=\langle \mathsf{nf}_\sigma(a)\,,\mathsf{nf}_\sigma(p)\rangle$

Theorem 5.2. [Unicity of σ-normal forms] If $u{\to_\sigma}u'$ then $\mathsf{nf}_\sigma(u) = \mathsf{nf}_\sigma(u')$, and if u is a term in σ-normal form then $\mathsf{nf}_\sigma(u) = u$.

This theorem is proved by induction on the definition of $\to_\sigma$. As a consequence we obtain the correction of nf_σ:

Corollary 5.3. If $u{\to_\sigma}^* u'$ and u' is in σ-normal then $\mathsf{nf}_\sigma(u) = u'$.

Corollary 5.4. Let $u =_{\beta\sigma} u'$. Then $\mathsf{nf}_\sigma(u) =_\beta \mathsf{nf}_\sigma(u')$.

Proof. It follows from Theorem 5.2 and the fact that β-reduction is restricted to σ-normal forms.

Subject Reduction of σ-reduction is still an open problem, but we will show that if u is typeable in a $PTS_{\beta\sigma}$ then $\mathsf{nf}_\sigma(u)$ is typeable in a PTS_β. In [12] Subject Reduction and Strong Normalisation are proved for a particular $PTS_{\beta\sigma}$ (which corresponds to $\lambda^\sigma(\mathsf{B})$ in Example 3.8 part 2). The extension to general $PTS_{\beta\sigma}$ is a subject for future work.

Theorem 5.6. [Projection from $\mathcal{T}_P$ to $\mathcal{T}$]. Let $u \in \mathcal{T}_P$.

1. If u is a data-pseudoterm or a prop-pseudoterm then $\mathsf{nf}_\sigma(u) \in \mathcal{T}$.
2. If u is a spec-pseudoterm then $\mathsf{nf}_\sigma(u) = \langle a\,,p\rangle$ where $a, p \in \mathcal{T}$.

This is proved by induction on the structure of u.

Definition 5.7. The function nf_σ is extended to contexts:

$$\begin{array}{lll}
\mathsf{nf}_\sigma(()) & = () & \\
\mathsf{nf}_\sigma(\Gamma, x{:}S) & = \mathsf{nf}_\sigma(\Gamma), x_d{:}A, x_p{:}(P\,x_d) & \text{if } x \text{ is a spec-variable,} \\
 & & \text{and } \mathsf{nf}_\sigma(S){=}\langle A\,,P\rangle \\
\mathsf{nf}_\sigma(\Gamma, x{:}U) & = \mathsf{nf}_\sigma(\Gamma), x{:}\mathsf{nf}_\sigma(U) & \text{otherwise}
\end{array}$$

We now define the class of Pure Type Systems with Pairs for which Theorem 5.9 holds. The conditions given in this definition ensure that typeability is preserved by the σ-normal form.

Definition 5.8. We say that a specification $\mathsf{S}_2 = (\mathcal{S}_2, \mathcal{A}_2, \mathcal{R}_2)$ of a Pure Type System is extractable from a specification $\mathsf{S}_1 = (\mathcal{S}_1, \mathcal{A}_1, \mathcal{R}_1, \mathcal{R}'_1)$ of a Pure Type System with Pairs if the following conditions are satisfied:

1. Condition for the sorts. Let k be a data or a prop-sort. If $k \in \mathcal{S}_1$ then $k \in \mathcal{S}_2$.
2. Conditions for the axioms.
   (a) Let k_1, k_2 be data or prop-sorts. If $(k_1, k_2) \in \mathcal{A}_1$ then $(k_1, k_2) \in \mathcal{A}_2$.
   (b) Let k_1 and k_2 be spec-sorts.
   Suppose $\mathsf{nf}_\sigma(k_1) = \langle k_3 \, , \lambda x{:}k_3.x \to k_4 \rangle$ and $\mathsf{nf}_\sigma(k_2) = \langle k_5 \, , \lambda x{:}k_5.x \to k_6 \rangle$.
   If $(k_1, k_2) \in \mathcal{A}_1$ then $(k_3, k_5), (k_4, k_6), (k_6, k_7) \in \mathcal{A}_2$ and $(k_3, k_6), (k_5, k_7) \in \mathcal{R}_2$.
3. Conditions for the formation of products.
   (a) Let k_1 be a data-sort and k_2 is either a data-sort or a prop-sort. If $(k_1, k_2) \in \mathcal{R}_1$ then $(k_1, k_2) \in \mathcal{R}_2$.
   (b) Let k_1 and k_2 be prop-sorts. If $(k_1, k_2) \in \mathcal{R}_1$ then $(k_1, k_2) \in \mathcal{R}_2$.
   (c) Case k_1 is a spec-sort and k_2 is a data-sort. If $(k_1, k_2) \in \mathcal{R}_1$ then $(k_3, k_2) \in \mathcal{R}_2$ where $\mathsf{nf}_\sigma(k_1) = \langle k_3 \, , \lambda x{:}k_3.x \to k_4 \rangle$.
   (d) Case k_1 is a spec-sort and k_2 is a prop-sort. If $(k_1, k_2) \in \mathcal{R}_1$ then $(k_3, k_2), (k_4, k_2) \in \mathcal{R}_2$ where $\mathsf{nf}_\sigma(k_1) = \langle k_3 \, , \lambda x{:}k_3.x \to k_4 \rangle$.
   (e) Case k_1 is a data-sort and k_2 is a spec-sort. If $(k_1, k_2) \in \mathcal{R}_1$ then $(k_1, k_3), (k_1, k_4) \in \mathcal{R}_2$ where $\mathsf{nf}_\sigma(k_2) = \langle k_3 \, , \lambda x{:}k_3.x \to k_4 \rangle$.
   (f) Case k_1 is a prop-sort and k_2 is a spec-sort. If $(k_1, k_2) \in \mathcal{R}_1$ then $(k_1, k_4) \in \mathcal{R}_2$ where $\mathsf{nf}_\sigma(k_2) = \langle k_3 \, , \lambda x{:}k_3.x \to k_4 \rangle$.
   (g) Case k_1 and k_2 are spec-sorts. If $(k_1, k_2) \in \mathcal{R}_1$ then $(k_3, k_5), (k_3, k_6), (k_4, k_6) \in \mathcal{R}_2$ where $\mathsf{nf}_\sigma(k_1) = \langle k_3 \, , \lambda x{:}k_3.x \to k_4 \rangle$ and $\mathsf{nf}_\sigma(k_2) = \langle k_5 \, , \lambda x{:}k_5.x \to k_6 \rangle$.

The following theorem states that the σ-normal form is a projection map from $\mathrm{PTS}_{\beta\sigma}$ into PTS_β.

Theorem 5.9. [Program Extraction Theorem] Let $\mathsf{S}_2 = (\mathcal{S}_2, \mathcal{A}_2, \mathcal{R}_2)$ be a specification extractable from $\mathsf{S}_1 = (\mathcal{S}_1, \mathcal{A}_1, \mathcal{R}_1, \mathcal{R}_1')$. Assume $\Gamma \vdash u{:}U$ in $\lambda^\sigma(\mathsf{S}_1)$.

1. If u is a data or prop-pseudoterm then, $\mathsf{nf}_\sigma(\Gamma) \vdash \mathsf{nf}_\sigma(u){:}\mathsf{nf}_\sigma(U)$ in $\lambda(\mathsf{S}_2)$.
2. If u is a spec-pseudoterm and $\Gamma \vdash U{:}k$ then $\mathsf{nf}_\sigma(\Gamma) \vdash a{:}A$ and $\mathsf{nf}_\sigma(\Gamma) \vdash p{:}(P\,a)$ in $\lambda(\mathsf{S}_2)$ where $\mathsf{nf}_\sigma(U) = \langle A \, , P \rangle$ and $\mathsf{nf}_\sigma(u) = \langle a \, , p \rangle$.
3. If u is a spec-pseudoterm and $U = k$ then $\mathsf{nf}_\sigma(\Gamma) \vdash A{:}k_1$ and $\mathsf{nf}_\sigma(\Gamma) \vdash P{:}A \to k_2$ in $\lambda(\mathsf{S}_2)$ where $\mathsf{nf}_\sigma(u) = \langle A \, , P \rangle$ and $\mathsf{nf}_\sigma(k) = \langle k_1 \, , \lambda x{:}k_1.x \to k_2 \rangle$.

Proof. This is proved by induction on the derivation. For the case of the product rule, we use the conditions for the formation of products given in Definition 5.8. For the case of the $\beta\sigma$-conversion rule, we use Corollary 5.4.

As a consequence of this theorem and the embedding of specific Typed Lambda Calculi into Pure Type Systems with Pairs, we obtain several program extraction theorems for particular systems.

Example 5.11.

1. We consider the specification $\mathsf{A}_1 = \mathsf{A}$ of Example 3.8 and A_2 defined as:

Specification A_2

$\mathcal{S}_2$	$\star_\mathsf{d} \quad \star_\mathsf{p} \; \Box_\mathsf{d} \quad \Box_\mathsf{p} \quad \triangle_\mathsf{p}$
$\mathcal{A}_2$	$(\star_\mathsf{d}, \Box_\mathsf{d}) \quad (\star_\mathsf{p}, \Box_\mathsf{p})$
$\mathcal{R}_2$	$(\star_\mathsf{u}, \star_\mathsf{v}) \quad (\star_\mathsf{u}, \Box_\mathsf{u}) \quad (\star_\mathsf{d}, \Box_\mathsf{p}) \quad (\Box_\mathsf{d}, \triangle_\mathsf{p})$
	if $\mathsf{u} = \mathsf{d}$ then $\mathsf{v} \in \{\mathsf{d}, \mathsf{p}\}$, if $\mathsf{u} = \mathsf{p}$ then $\mathsf{v} = \mathsf{p}$

We also have to associate a pair to each spec-sort:

Spec-sort	Pair
$\star_{\mathsf{s}}$	$\langle \star_{\mathsf{d}} , \lambda x{:}\star_{\mathsf{d}}.x \to \star_{\mathsf{p}} \rangle$
$\Box_{\mathsf{s}}$	$\langle \Box_{\mathsf{d}} , \lambda x{:}\, \Box_{\mathsf{d}} \,.x \to \Box_{\mathsf{p}} \rangle$

The Theory of Specifications and the Verification Calculus presented in [27] are embedded in $\lambda^{\sigma}(\mathsf{A}_1)$ and $\lambda(\mathsf{A}_2)$ respectively. Since the specification A_2 is extractable from A_1, Theorem 5.9 applies. The theorem in [27] that states that nf_{σ} is a projection from the Theory of Specifications to the Verification Calculus is a particular case of Theorem 5.9.

2. We consider the specification $\mathsf{B}_1 = \mathsf{B}$ of Example 3.8 and B_2 defined as:

Specification B_2

$\mathcal{S}_2$	$\star_{\mathsf{d}}{}^{\mathsf{i}} \quad \star_{\mathsf{p}}{}^{\mathsf{i}} \quad$ for $\mathsf{i} \in N$
$\mathcal{A}_2$	$(\star_{\mathsf{d}}{}^{\mathsf{i}}, \star_{\mathsf{d}}{}^{\mathsf{i}+1}) \quad (\star_{\mathsf{p}}{}^{\mathsf{i}}, \star_{\mathsf{p}}{}^{\mathsf{i}+1}) \quad$ for $\mathsf{i} \in N$
$\mathcal{R}_2$	$(\star^{\mathsf{i}}_{\mathsf{u}}, \star^{\mathsf{j}}_{\mathsf{v}}) \quad$ for $\mathsf{i} \leq \mathsf{j}$ or $\mathsf{j} = 0$ and
	if $\mathsf{u} = \mathsf{d}$ then $\mathsf{v} \in \{\mathsf{d}, \mathsf{p}\}$, if $\mathsf{u} = \mathsf{p}$ then $\mathsf{v} = \mathsf{p}$

The spec-sort $\star_{\mathsf{s}}{}^{\mathsf{i}}$ is associated to the pair $\langle \star_{\mathsf{d}}{}^{\mathsf{i}} , \lambda x{:}\star_{\mathsf{d}}{}^{\mathsf{i}}.x \to \star_{\mathsf{p}}{}^{\mathsf{i}} \rangle$.

Note that B_2 is obtained from B_1 by removing the spec-sorts $\star_{\mathsf{s}}{}^{\mathsf{i}}$ and the product rules $(\star_{\mathsf{p}}{}^{\mathsf{i}}, \star_{\mathsf{d}}{}^{\mathsf{j}})$.

The Theory of Specifications and the Verification Calculus presented in [28] are embedded in $\lambda^{\sigma}(\mathsf{B}_1)$ and $\lambda(\mathsf{B}_2)$ respectively. The specification B_2 is extractable from B_1 and Theorem 5.9 applies. The theorem in [28] that states that nf_{σ} is a projection from the Theory of Specifications to the Verification Calculus is a particular case of Theorem 5.9.

3. We consider the specification $\mathsf{C}_1 = \mathsf{C}$ of Example 3.8 and C_2 defined as:

Specification C_2

$\mathcal{S}_2$	$\star_{\mathsf{d}} \quad \star_{\mathsf{p}} \; \Box_{\mathsf{d}} \quad \Box_{\mathsf{p}} \quad \triangle_{\mathsf{p}}$
$\mathcal{A}_2$	$(\star_{\mathsf{d}}, \Box_{\mathsf{d}}) \quad (\star_{\mathsf{p}}, \Box_{\mathsf{p}})$
$\mathcal{R}_2$	$(\star_{\mathsf{u}}, \star_{\mathsf{v}}) \quad (\star_{\mathsf{u}}, \Box_{\mathsf{v}}) \quad (\Box_{\mathsf{u}}, \star_{\mathsf{v}}) \quad (\Box_{\mathsf{u}}, \Box_{\mathsf{v}}) \quad (\Box_{\mathsf{d}}, \triangle_{\mathsf{p}})$
	if $\mathsf{u} = \mathsf{d}$ then $\mathsf{v} \in \{\mathsf{d}, \mathsf{p}\}$, if $\mathsf{u} = \mathsf{p}$ then $\mathsf{v} = \mathsf{p}$

The spec-sorts are associated to the following pairs:

spec-sort	associated pair
$\star_{\mathsf{s}}$	$\langle \star_{\mathsf{d}} , \lambda x{:}\star_{\mathsf{d}}.x \to \star_{\mathsf{p}} \rangle$
$\Box_{\mathsf{s}}$	$\langle \Box_{\mathsf{d}} , \lambda x{:}\, \Box_{\mathsf{d}} \,.x \to \Box_{\mathsf{p}} \rangle$

The system in [23] can be embedded in $\lambda^{\sigma}(\mathsf{C}_1)$. The specification C_2 is extractable from C_1 and Theorem 5.9 applies. The theorem of program extraction in [23] is, then, a particular case of Theorem 5.9.

Another consequence of the Theorem of Program Extraction is related to consistency. Though our pairs are stronger than strong Σ, consistency is preserved because we have divided the set of pseudo-terms, sorts and types into three with a special category of pseudo-terms, sorts and types for specifications.

Corollary 5.12. Let $\mathsf{S}_2 = (\mathcal{S}_2, \mathcal{A}_2, \mathcal{R}_2)$ be a specification extractable from $\mathsf{S}_1 = (\mathcal{S}_1, \mathcal{A}_1, \mathcal{R}_1, \mathcal{R}_1')$. If $\lambda(\mathsf{S}_2)$ is consistent (not all propositions are inhabited) then so is $\lambda^{\sigma}(S_2)$.

6 Strategies for β-reduction

In this section we study strategies for β-reduction, and give an algorithm to test β-conversion in strongly normalising Pure Type Systems. We will adapt the interaction net encoding developed for the λ-calculus in YALE [20] to the general framework of Pure Type Systems. For this, we first introduce an intermediate rewrite system which is obtained by adding explicit substitutions, duplication and erasing to the set $\mathcal{T}$ of pseudoterms and including new reduction rules to manipulate these constructors.

6.1 Explicit Substitutions and Resource Management

The λ-calculus lacks, in addition to an explicit definition of substitution, explicit information about sharing and evaluation orders. It is well-known that the order in which substitutions are performed can have dramatic consequences on the efficiency of the β-reduction process. To ensure that we have a tight control over the way substitutions are performed, we will make explicit not only the substitutions, but also the copying and erasing phases of substitution.

We use a notation inspired by various calculi derived from linear logic (see [20, 10]). We extend the set $\mathcal{T}$ of pseudoterms for Pure Type Systems to include explicit substitutions, denoted $u\{v/x\}$, and add explicit syntactical constructs for duplication, denoted $C_x^{y,z}(u)$, and erasing, denoted $E_x(u)$.

The process of pushing a substitution through a λ-abstraction is a delicate operation since free variables may become bound, thus renaming may be necessary (or "shifting" operations if we use the De Bruijn notation as in the $\lambda\sigma$-calculus [1]), which can be regarded as an expensive syntactical overhead on the calculus. In order to avoid this overhead, we will use the closed reduction strategy (see [20, 10]), where only closed substitutions are propagated under abstractions.

Another source of inefficiency in an implementation of β-reduction is the duplication of terms containing actual or potential redexes. Copying terms with free variables implies that redexes that might be created later during reduction will be duplicated. For this reason, we will only copy terms that have no free variables, and moreover, when they are in normal form.

We will define a strategy for β-reduction in Pure Type Systems based on these ideas. The strategy is weak (it reduces terms to weak-head normal form), but we shall show an extension that can achieve strong reduction (i.e. reduction to full normal form).

6.1.1 Syntax

As usual, we first define the syntax. The explicit copy and erasing constructs will allow us to define linearised pseudoterms, where variables are used linearly (each free variable occurs at most once in a term).

Definition 6.1.

1. The set $\mathcal{T}_{\mathsf{x}}$ of pseudoterms with explicit substitutions and resource management is defined by adding substitutions $u\{v/x\}$, erasure $E_x(u)$ and duplication $C_x^{y,z}(u)$ to the set $\mathcal{T}$ of Definition 2.2.

2. The set $\mathcal{T}_L$ of linearised pseudoterms is defined as the subset of $\mathcal{T}_{\mathsf{x}}$ that satisfies the variable constraints shown in Fig. 4 (this table also defines the set of free variables of each linearised pseudoterm). Note that the copy and substitution constructs are binders, as well as abstraction and product.

A few remarks on the notation are in order. Abstraction $\lambda x{:}U.u$ enforces that the variable actually does occur at least once free in the term u. The construct $E_x(u)$ is used to make the variable x occur free explicitly. Application (uv) enforces that the free variables in u and v are disjoint, thus do not occur more than once. The construction $C_x^{y,z}(u)$ ensures that all variables have a unique name: If u has two occurrences of the variable x, then we rename one to y, the

Name	L-Pseudoterm	Variable Constraint	Free Variables
Sort	k	$-$	$-$
Variable	x	$-$	$\{x\}$
Abstraction	$\lambda x{:}U.u$	$x \in \mathsf{fv}(u), \mathsf{fv}(u) \cap \mathsf{fv}(U) = \emptyset,$	$(\mathsf{fv}(u) - \{x\}) \cup \mathsf{fv}(U)$
Product	$\Pi x{:}U.u$	$x \in \mathsf{fv}(u), \mathsf{fv}(u) \cap \mathsf{fv}(U) = \emptyset,$	$(\mathsf{fv}(u) - \{x\}) \cup \mathsf{fv}(U)$
Application	(uv)	$\mathsf{fv}(u) \cap \mathsf{fv}(v) = \emptyset$	$\mathsf{fv}(u) \cup \mathsf{fv}(v)$
Erase	$E_x(u)$	$x \notin \mathsf{fv}(u)$	$\mathsf{fv}(u) \cup \{x\}$
Copy	$C_x^{y,z}(u)$	$x \notin \mathsf{fv}(u), y \neq z, \{y,z\} \subseteq \mathsf{fv}(u)$	$(\mathsf{fv}(u) - \{y,z\}) \cup \{x\}$
Substitution	$u\{v/x\}$	$x \in \mathsf{fv}(u), (\mathsf{fv}(u) - \{x\}) \cap \mathsf{fv}(v) = \emptyset,$	$(\mathsf{fv}(u) - \{x\}) \cup \mathsf{fv}(v)$

Figure 4: Variable Constraints and Free Variables for Linearised Pseudoterms

other to z and then use the construct. If x occurs more than twice, then we can use this construct repeatedly.

There are obvious translations from the set $\mathcal{T}$ of pseudoterms for Pure Type Systems into the set $\mathcal{T}_L$ of linearised pseudoterms and vice versa which we will not elaborate. We will assume that the translation $\mathsf{lin} : \mathcal{T} \to \mathcal{T}_L$ introduces the erasing construct as soon as possible (just after the abstraction or product that binds the variable) whereas the copy construct is introduced as late as possible. The readback function $\mathsf{rb} : \mathcal{T} \to \mathcal{T}_L$ simply erases the erasing constructs, replaces $C_x^{y,z}(u)$ by $\mathsf{rb}(u)[x/y][x/z]$, and performs the substitutions. Linearised pseudoterms are an intermediate language for our translations to interaction nets (shown in the appendix), where the main bureaucratic issue of counting occurrences of variables in a term has already been taken into account.

6.1.2 Reduction Rules: Closed Reduction

The β-reduction rule generates a term with an explicit substitution, and we add the rules for the propagation of the substitution. The latter are inspired by the rules given in [10].

Definition 6.2. [β-reduction and Propagation of Substitutions] The following table defines a set of conditional rewrite rules which generate the relation $\to_{\beta\mathsf{x}}$ on the set of linearised pseudoterms.

Name	Reduction			Condition
Beta	$(\lambda x{:}U.u)\, v$	$\to_\beta$	$u\{v/x\}$	$\mathsf{fv}(\lambda x{:}U.u) = \emptyset$
Var	$x\{v/x\}$	$\to_\mathsf{x}$	v	$-$
App1	$(tu)\{v/x\}$	$\to_\mathsf{x}$	$(t\{v/x\})u$	$x \in \mathsf{fv}(t)$
App2	$(tu)\{v/x\}$	$\to_\mathsf{x}$	$t(u\{v/x\})$	$x \in \mathsf{fv}(u)$
Abs1	$(\lambda y{:}U.u)\{v/x\}$	$\to_\mathsf{x}$	$(\lambda y{:}U.u\{v/x\})$	$\mathsf{fv}(v) = \emptyset, x \in \mathsf{fv}(u)$
Abs2	$(\lambda y{:}U.u)\{v/x\}$	$\to_\mathsf{x}$	$(\lambda y{:}U\{v/x\}.u)$	$x \in \mathsf{fv}(U)$
Prod1	$(\Pi y{:}U.u)\{v/x\}$	$\to_\mathsf{x}$	$(\Pi y{:}U.u\{v/x\})$	$\mathsf{fv}(v) = \emptyset, x \in \mathsf{fv}(u)$
Prod2	$\Pi y{:}U.u\{v/x\}$	$\to_\mathsf{x}$	$(\Pi y{:}U\{v/x\}.u)$	$x \in \mathsf{fv}(U)$
Copy1	$(C_x^{y,z}(u))\{v/x\}$	$\to_\mathsf{x}$	$(u\{v/y\})\{v/z\}$	$\mathsf{fv}(v) = \emptyset$
Copy2	$(C_{x'}^{y,z}(u))\{v/x\}$	$\to_\mathsf{x}$	$C_{x'}^{y,z}(u\{v/x\})$	$-$
Erase1	$(E_x(u))\{v/x\}$	$\to_\mathsf{x}$	u	$\mathsf{fv}(v) = \emptyset$
Erase2	$(E_{x'}(u))\{v/x\}$	$\to_\mathsf{x}$	$E_{x'}(u\{v/x\})$	$-$
Comp	$(u\{w/y\})\{v/x\}$	$\to_\mathsf{x}$	$u\{w\{v/x\}/y\}$	$x \in \mathsf{fv}(w)$

Note that variables of Pure Type Systems are constants in the signature of the rewrite system, thus $x \neq x'$, etc.

The conditions on the reduction rules propagate the substitution to the subterms where it is needed, and ensure that α-conversion, which is a costly operation, is not needed. To avoid α-conversion, in *Abs1*, *Prod1*, and *Copy1* we require that the substitution is closed.

There are variants of the reduction rules which do not change the basic results of this paper. For instance, the *Copy1* rule can reduce the substituted term v to normal form before copying, thus avoiding duplicating redexes. This variant is used in the encoding to interaction nets (see the appendix).

In the rest of the paper we will only consider linearised pseudoterms that are in the image of the translation lin from $\mathcal{T}$ into $\mathcal{T}_L$. In the following theorem we show that the image of the translation is closed under reduction.

Theorem 6.3. Let u be a linearised pseudoterm in the image of a translation from $\mathcal{T}$ into $\mathcal{T}_L$. If $u \to_{\beta\mathsf{x}} v$ then

1. $\mathsf{fv}(u) = \mathsf{fv}(v)$.
2. v is also a linearised pseudoterm (i.e. it satisfies the variable constraints).

Proof.

1. Only closed terms are erased, and no new variables are created during reduction.
2. It is sufficient to prove that $\to_{\beta\mathsf{x}}$ preserves the variable constraints given in Def. 6.2. We distinguish cases:

   Beta: $(\lambda x{:}U.u)\, v \to_\beta u\{v/x\}$ if $\mathsf{fv}(\lambda x{:}U.u) = \emptyset$.
   By assumption, $x \in \mathsf{fv}(u)$, and clearly $(\mathsf{fv}(u) - \{x\}) \cap \mathsf{fv}(v) = \emptyset$ because $\mathsf{fv}((\lambda x{:}U.u)) = \emptyset$. Thus $u\{v/x\}$ satisfies the constraints.

   Var: $x\{v/x\} \to_x v$. Trivial.

   App1: $(tu)\{v/x\} \to_x (t\{v/x\})u$ if $x \in \mathsf{fv}(t)$.
   The term $t\{v/x\}$ is valid, since $x \in \mathsf{fv}(t)$ and $\mathsf{fv}(v) \cap (\mathsf{fv}(t) - \{x\}) = \emptyset$. By assumption $\mathsf{fv}(t) \cap \mathsf{fv}(u) = \emptyset$, and $\mathsf{fv}(v) \cap \mathsf{fv}(u) = \emptyset$ therefore $(t\{v/x\})u$ is valid.

   App2: Follows the same reasoning as the case for *App1*.

   Abs1: $(\lambda y{:}U.u)\{v/x\} \to_x (\lambda y{:}U.u\{v/x\})$ if $\mathsf{fv}(v) = \emptyset, x \in \mathsf{fv}(u)$.
   The term $u\{v/x\}$ is valid since $x \in \mathsf{fv}(u)$ and $\mathsf{fv}(v) = \emptyset$. Since $y \in \mathsf{fv}(u)$ then $(\lambda y{:}U.u\{v/x\})$ is valid.

   Abs2: $(\lambda y{:}U.u)\{v/x\} \to_x (\lambda y{:}U\{v/x\}.u)$ if $x \in \mathsf{fv}(U)$.
   The term $U\{v/x\}$ is valid since $x \in \mathsf{fv}(U)$ and $(\mathsf{fv}(U) - \{x\}) \cap \mathsf{fv}(v) = \emptyset$ because $(\lambda y{:}U.u)\{v/x\}$ is valid. The other constraints are trivially satisfied.

   Prod1 and *Prod2* are similar to *Abs1* and *Abs2*.

   Copy1: $(C_{x'}^{y,z}(u))\{v/x\} \to_x u\{v/y\}\{v/z\}$ if $\mathsf{fv}(v) = \emptyset$.
   The term $u\{v/y\}$ is valid since $y \in \mathsf{fv}(u)$ by assumption and $\mathsf{fv}(v) = \emptyset$. Similarly, $u\{v/y\}\{v/z\}$ is also valid because $z \in \mathsf{fv}(u\{v/y\})$.

   Copy2: $(C_{x'}^{y,z}(u))\{v/x\} \to_x C_{x'}^{y,z}(u\{v/x\})$.
   The term $u\{v/x\}$ is valid since $x \in \mathsf{fv}(u)$ by assumption. Now, $C_{x'}^{y,z}(u\{v/x\})$ is valid since y and z do not occur in v (the translation lin introduces fresh variables for each copy construct).

   Erase1: $(E_x(u))\{v/x\} \to_x u$. Trivial.

   Erase2: $(E_{x'}(u))\{v/x\} \to_x E_{x'}(u\{v/x\})$. Using the same reasoning as above, the term $u\{v/x\}$ is valid, and since $x' \notin \mathsf{fv}(u\{v/x\})$ the term $E_{x'}(u\{v/x\})$ satisfies the constraints.

   Comp: $(u\{w/y\})\{v/x\} \to_x u\{w\{v/x\}/y\}$ if $x \in \mathsf{fv}(w)$.
   Since $(u\{w/y\})$ satisfies the constraints, $y \in \mathsf{fv}(u)$. The other constraints are obtained by easy operations on sets.

It is clear that the reduction rules defining $\rightarrow_{\beta \mathsf{x}}$ perform β-reduction. It is also clear that, due to the conditions in the rules, we cannot simulate arbitrary β-reduction steps. However, we can guarantee that we can reach at least a weak head normal form, as the following theorem shows.

Theorem 6.5. [Adequacy for Reduction to Weak-Head-Normal-Form] If u is a closed term typeable in a strongly normalising Pure Type System, and $\mathsf{lin}(u) = \overline{u}$, then there is a reduction sequence $\overline{u} \rightarrow_{\beta \mathsf{x}}{}^* v$ such that $\mathsf{rb}(v)$ is a weak head normal form of u.

Proof. Although we cannot simulate arbitrary β-reduction steps using $\rightarrow_{\beta \mathsf{x}}$, the reduction $\rightarrow_{\beta \mathsf{x}}$ allows us to simulate Call-by-value and Call-by-name reduction, which is sufficient to obtain a weak-head-normal. To simulate Call-by-value and Call-by-name reduction to weak-head-normal form we use the techniques of [10].

To obtain the full normal form of a term, we must force blocked reductions. We use the standard technique of normalisation by evaluation [4] (see also [29]). The idea is to reduce a closed term to weak head normal form, then to distinguish the variable bound by the outermost binder in some way (to "freeze" it), so that we can still consider the term under the binder as closed, and recursively apply the same process to this subterm. There are several ways to distinguish those variables in the syntax, the easiest one is to treat them as constants.

In the appendix we give the interaction net implementation of β-reduction in Pure Type Systems using a closed reduction strategy.

7 Conclusions

We have shown that σ-reduction is normalising, providing an operational description of the process of program extraction which is independent of the typed lambda calculus we choose. Subject reduction of σ in the general framework of $PTS_{\beta\sigma}$ is still an open problem.

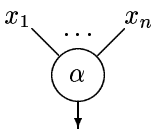
An obvious conclusion from this paper is that any program that can be extracted from a proof of its specification using for instance LEGO [6] or Coq [23] can also be extracted using σ-reduction. It will be interesting to find an example of program extraction that can be handled by σ-reduction but cannot be done in LEGO or Coq for instance.

There are two implementations of closed reduction available for the λ-calculus, one using interaction nets and the other one using director strings. The benchmarks obtained so far show that closed reduction is more efficient than standard evaluation strategies for the λ-calculus, and its low overheads make it even more efficient than optimal reduction in many cases (see [20, 29] for experimental results). It remains to test this strategy in an actual implementation of a proof assistant.

A Appendix: Interaction Nets for Pure Type Systems

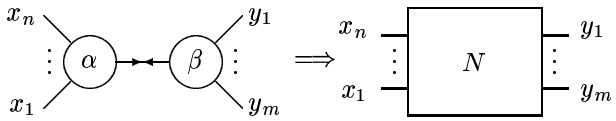
A.1 Interaction Nets

Interaction nets, introduced by Lafont [15], are graph rewriting systems derived from the multiplicative proof nets of linear logic [13]. The graphical syntax of interaction nets is user-defined: an interaction net system is specified by a set Σ of symbols, and a set IR of interaction rules. Each symbol $\alpha \in \Sigma$ has an associated (fixed) *arity*. An occurrence of a symbol $\alpha \in \Sigma$ will be called an *agent*. If the arity of α is n, then the agent has $n+1$ *ports*: a distinguished one called the *principal port* depicted by an arrow, and n *auxiliary ports* corresponding to the arity of the symbol. We will say that the agent has $n+1$ *free* ports, depicted:



A *net* N built on Σ is a graph (not necessarily connected) with agents at the vertices. The edges of the net connect agents together at the ports such that there is only one edge at every port (edges may connect two ports of the same agent). A net may also have edges with free extremes, called *wires*, and their extremes are called ports by analogy. The *interface* of a net is its set of free ports. We denote by $\mathcal{N}(\Sigma)$ the set of interaction nets built with agents in Σ (we omit Σ when it is clear from the context).

A pair of agents $(\alpha, \beta) \in \Sigma^2$ connected together on their principal ports is called an *active pair*; the interaction net analogue of a redex. An *interaction rule* $((\alpha, \beta) \Longrightarrow N) \in \mathsf{IR}$ replaces an occurrence of the active pair (α, β) by the net N. The following diagram illustrates the idea, where N is any net built from Σ.



Rules have to satisfy two very strong conditions: the interface of the active pair must be equal to the interface of the right-hand side, and at most one rule can be defined for each active pair. These conditions imply that interactions are always binary, local, and strongly confluent. For this reason, interactions can take place in any order in a net, even in parallel.

We do not require that there is a rule for each pair of agents, but if we create a net with an active pair for which there is no rewrite rule, then we have a deadlock. The interaction net system that we present in this paper will be deadlock-free in this sense. An interaction net is in normal form when there are no active pairs.

The fact that the interface of the active pair is preserved in an interaction step (this linear aspect of rewriting is inherited from linear logic) implies that we must explicitly erase or copy when modelling a non-linear interaction. For this, standard agents are used: δ (duplicator) and ϵ (eraser). We refer to [15] for a detailed presentation and examples of interaction nets.

A.2 Encoding PTS Terms as Nets

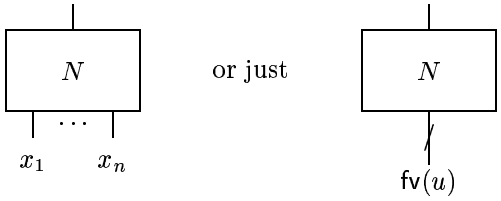
YALE [20] is an interaction net evaluator for PCF which uses agents *app* and λ to represent application and abstraction. As in every interaction net implementation of the λ-calculus, the key part of the encoding deals with the representation of the body of an abstraction (the encoding of a box), and the copying and erasing of abstractions. This is achieved in YALE by using b agents to represent the limits of the box (i.e. the list of free variables of the abstraction). More precisely, there is one b agent for each free variable in the abstraction and the agent v represents the empty list.

We now show how we can implement β-normalisation and the β-equality test of Pure Type Systems, by giving an encoding as an interaction net. We will use agents @, λ and π to represent application, abstraction and product. Variables are encoded by edges and we will use YALE's encoding of boxes for λ and π. More precisely, we give a translation function

$$\mathsf{net} : \mathcal{T}_L \to \mathcal{N}$$

from the linearised terms of Pure Type Systems into interaction nets. The readback function which maps a net in normal form into a term will also be called rb and its definition is straightforward and omitted.

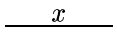
A linearised pseudoterm u with $\mathsf{fv}(u) = \{x_1, \ldots, x_n\}$ will be translated as a net $\mathsf{net}(u) = N$ with the root edge at the top, and n free edges corresponding to the free variables, which we draw as either



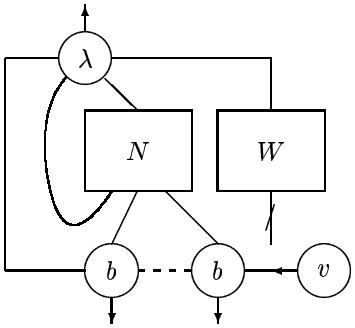
We will drop the labelling on the edges since they are derived directly from the term, and the order is preserved.

Constants. Constants k are coded by introducing a new unary agent for each constant. The only port is the principal port:

Variable. If u is a variable, then $\mathsf{net}(u)$ is simply an edge.



Abstraction. Let $\mathsf{net}(u) = N$ and $\mathsf{net}(U) = W$, then $\mathsf{net}(\lambda x{:}U.u)$ is given by the following net, where we have introduced three different kinds of agent. First, an agent λ of arity 4, which corresponds to abstraction. The remaining two kinds of agents represent a list of the free variables of the term. We use the agent b, one for each free variable, and an agent v which represents the end of this list.

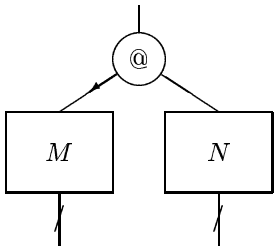


The key idea is that the coding contains a pointer to all the free variables of the abstraction; the body of the abstraction is encapsulated in a box structure.

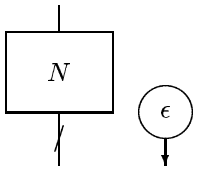
We have assumed, without loss of generality, that the (unique) occurrence of the variable x is in the left-most position of N.

Product. The translation is the same as for Abstraction, using an agent π.

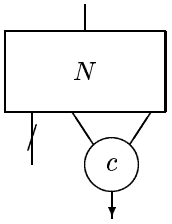
Application. Let $\mathsf{net}(u) = M$, and $\mathsf{net}(v) = N$, then $\mathsf{net}(uv)$ is given by the following net, where we have introduced an agent @ of arity 2 which corresponds to an application agent in the usual graph representations of the λ-calculus.



Erasing. Let $\mathsf{net}(u) = N$, then $\mathsf{net}(E_x(u))$ is given by the following net using a new agent ϵ, of arity 0.

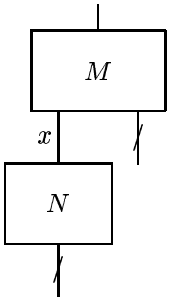


Duplication. Let $\mathsf{net}(u) = N$, then $\mathsf{net}(C_x^{y,z}(u))$ is given by the following net using a new agent c, of arity 2.



We have assumed, without loss of generality, that the (unique) occurrences of the variables y, z are in the right-most positions of N.

Substitution. Let $\mathsf{net}(u) = M$, and $\mathsf{net}(v) = N$, then $\mathsf{net}(u\{v/x\})$ is given by the following net, where we simply connect the free edge x from the net M to the net N.

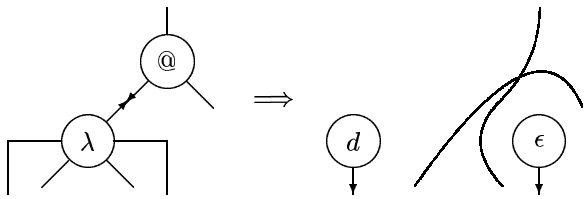


A.3 Dynamics

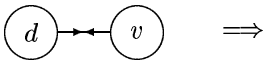
We now give the interaction rules implementing β-reduction. We begin by giving the *weak* system, then show how we can implement strong reduction. We show how each rewrite rule defining the reduction relation $\rightarrow_{\beta\mathsf{x}}$ (see Definition 6.2) is implemented in the interaction system, which is sufficient to show that the encoding is correct.

First, we note that several of the reduction rules in Definition 6.2 are implicit. The translation of the rule $x\{v/x\} \rightarrow_{\mathsf{x}} v$ is implicit in this system of interaction, since $\mathsf{net}(x\{v/x\}) = \mathsf{net}(v)$, thus no interaction rules are required. The rules for pushing a substitution through an application: $(tu)\{v/x\} \rightarrow_{\mathsf{x}} (t\{v/x\})u$ and $(tu)\{v/x\} \rightarrow_{\mathsf{x}} t(u\{v/x\})$ are also implicit, as well as the rules *Abs2, Prod2, Copy2* and *Erase2*. The last rule $(u\{w/x\})\{v/x\} \rightarrow_{\mathsf{x}} u\{w\{v/x\}/y\}$ is equally implicit (the translation of both sides of the rules yield the same interaction net). We therefore only need to encode the remaining 5 rewrite rules.

The first interaction rule is the linear part of β-reduction. This rule connects the body of the abstraction to the root, and the argument to the variable occurrences. A new agent d is introduced which serves to erase the box structure used in the translation, which will be explained below.



The second rule shows that if the function was closed (i.e. the abstraction had an empty list of free variables) in the above β-reduction, then the d agent introduced simply erases the v agent which marks the empty list.

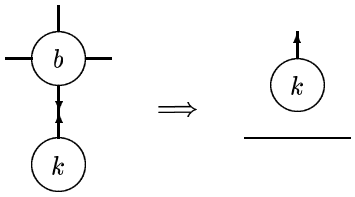


Thus if the abstraction is closed, then clearly we have $\mathsf{net}((\lambda x{:}U.u)v) \Longrightarrow^* \mathsf{net}(u\{v/x\})$, thus correctly implementing the first rule for $\rightarrow_{\beta\mathsf{x}}$.

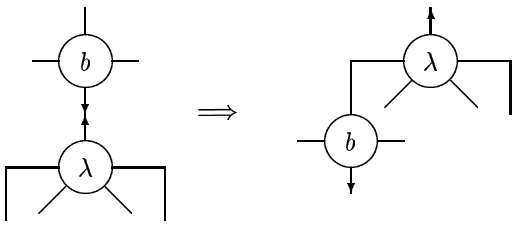
The next three rules are for the substitution of a closed value v through an abstraction:

$$(\lambda y{:}U.u)\{v/x\} \rightarrow_{\mathsf{x}} \lambda y{:}U.(u\{v/x\}).$$

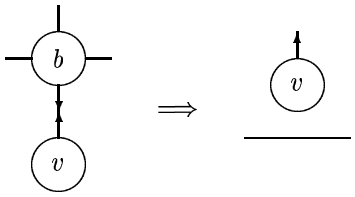
The first allows a constant k to be moved inside an abstraction in a single interaction:



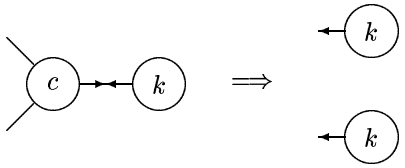
For abstraction and product values, we require two interactions to complete the substitution. We only show the rules for abstraction.



Since the abstraction that we are moving inside has no free variables, then the reduction successfully completes with the following interaction.

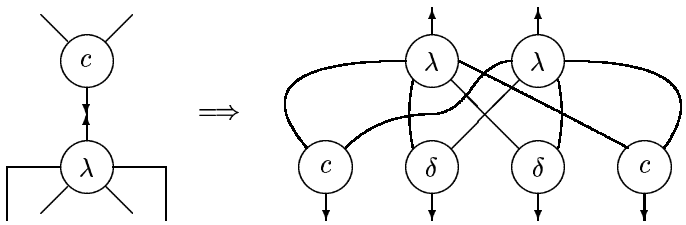


We have not shown the implementation of the rules *Copy1* and *Erase1*. It is clear that to obtain an efficient reduction we should reduce the substitution before copying it in *Copy1*. Interaction nets force this reduction since we cannot copy nets that have active pairs. The following interaction rules shows how a weak head normal form with no free variables can be duplicated (*Copy1*). The first rule shows that the copying agent can duplicate constants k:

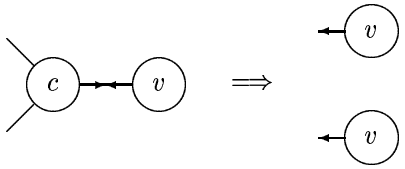


The next rule is when a copying agent meets an abstraction. We omit the rules for copying a product, which are the same replacing λ by π.

We copy the λ, and propagate δ agents inside the body of the abstraction. We also propagate the c agent along the list of free variables and the type.



If the term is closed, then the following reduction duplicates the end of list agent.

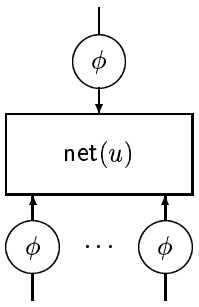


We omit the interaction rules for the agents δ and ϵ which are standard (we refer to [20] for details).

A.4 Checking β-equality

We have given the interaction rules for implementing β-reduction to weak head normal form. To test β-equality we will interleave the check for syntactical equality and the computation of the full normal form, as follows: If both terms reduce to a weak head normal form which has the same binder at the root (i.e. a λ or a π) then we need to check that the subterms are β-equivalent, but if the binders at the root are not the same, we already know they are not β-equivalent. In the worst case we need to compute the full normal form (for which we need strong reduction) to check syntactical equality. Note that the test for syntactical equality requires α-conversion when we use a λ-calculus with names, but it is trivial in interaction nets since we represent variables by edges.

To obtain the full normal form of a term we use the technique described in [20]. We introduce a new agent ϕ, and a set of interaction rules, which will be used to force reductions. It can be thought of as supplying dummy arguments to a term to allow reductions to complete (in a similar way that environment machines, for instance, can be supplied dummy arguments to force reduction to normal form). The translation $\mathsf{net}'(u)$ of the term u, to obtain full reduction, is now the following net, which we call the ϕ-closure of u:



Note that to test β-equality (as in the β-Conversion rule of a PTS) it is sufficient to consider PTS-terms without type information (i.e. we erase U in $(\lambda x{:}U.u)$ and $(\Pi x{:}U.u)$ and simplify the encoding in interaction nets accordingly).

In Figure 5 we show the interaction rules for the new agent ϕ. This new agent essentially behaves like an identity agent—it simply passes over all agents without effect, as shown by the first interaction rule. However, there are three special cases. The first is when ϕ interacts with itself, in this case both instances just cancel each other out, indicating that the forcing is complete. The final two rules are the most important: one converts the b agent into a b' agent which will be used to allow other reductions to complete, and the last rule converts the b' agent back to a b agent. The intuition is that ϕ is allowing blocked computations to complete: when no further substitutions can be done, the free variables of an abstraction are allowed to be duplicated and erased. Using these additional agents, we get strong reduction.

Theorem A.1. Let u be any (type-erased) term of a strongly normalising Pure Type System. If $u \to_\beta^* v$, for a β-normal form v, then there is a finite sequence of interactions $\mathsf{net}'(\mathsf{lin}(u)) \Longrightarrow^* N$ such that $\mathsf{rb}(N) = v$.

Proof. The system of interaction nets that we use corresponds directly to the encoding of cut elimination in linear logic given in [21]. The fact that an irreducible net in that system does not contain cuts implies that the readback of an irreducible net in our system is a term without β-redexes.

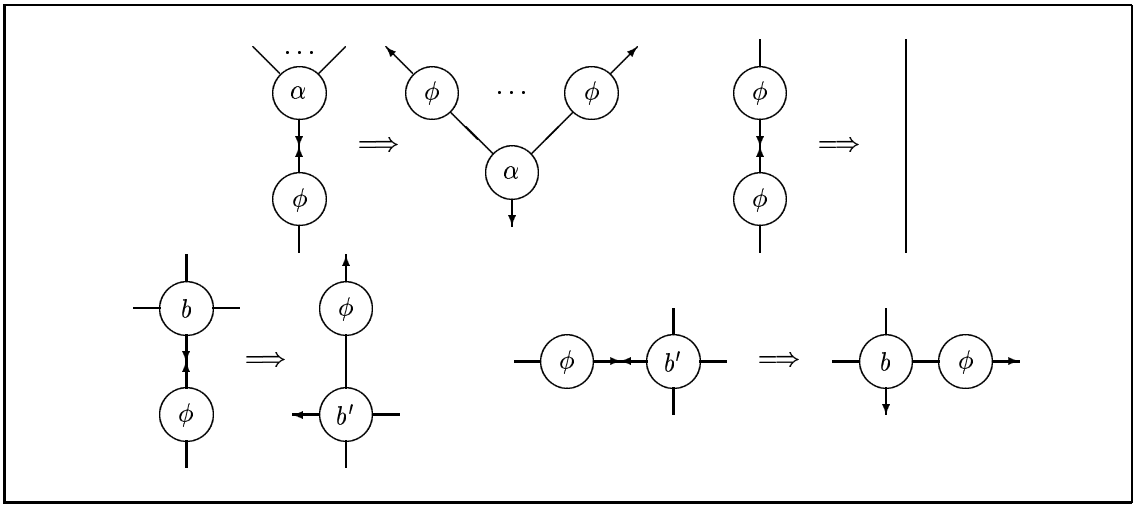


Figure 5: Forcing Reduction

References

[1] M. Abadi, L. Cardelli, P-L. Curien and J-J. Lévy. Explicit Substitutions. *Journal of Functional Programming*, 1:4, pages 375–416, 1991.

[2] H.P. Barendregt. The Lambda Calculus, Its Syntax and Semantics, Revised Edition. North-Holland Publishing Co., Amsterdam, 1984.

[3] Barras et al. The Coq Proof Assistant Reference Manual. Technical report, INRIA, 1999.

[4] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Logic in Computer Science (LICS'91)*, pages 203–211. IEEE Computer Society Press, 1991.

[5] S. Berghofer. Extraction in Simply-Typed Higher Order Logic. In *Proceedings of Types for Proofs and Programs*. Volume 2646 of LNCS.

[6] R. Burstall and J. McKinna. Deliverables: An approach to program development in the calculus of constructions. In *Proceedings of the First Workshop on Logical Frameworks*, 1990.

[7] J. L. Caldwell. Classical Propositional Decidability via Nuprl Proof Extraction. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOL'98), Australia*, pages 105–122. Springer, 1998.

[8] M. Fernández, I. Mackie, P. Severi, and N. Szasz. A uniform approach to program extraction: Pure Type Systems with ultra-Σ types. In *Proceedings of CLEI 2002*.

[9] T. Crolard. A type theory which is complete for Kreisel's modified realizability. *ENTCS*, 23(1), 1999.

[10] M. Fernández and I. Mackie. Closed Reduction in the λ-calculus. In *Proceedings of Computer Science Logic (CSL'99)*. Springer, 1999. LNCS 1683.

[11] M. Fernández and I. Mackie. Generalized Director Strings and Explicit Substitutions, In *Proceedings of the Fourth Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs (WESTAPP'01)*. Utrecht, 2001.

[12] M. Fernández and P. Severi. An operational approach to program extraction in the Calculus of Constructions. In *Proc. of the Int. Workshop on Logic Based Program Development and Transformation (LOPSTR'02)*. Springer, 2002. LNCS.

[13] J-Y. Girard. Linear Logic. *Theoretical Computer Science*,50:1, pages 1–102, 1987.

[14] G. Gonthier, M. Abadi and J-J. Lévy, The Geometry of Optimal Lambda Reduction, In *Proceedings of ACM Symposium Principles of Programming Languages*, pages 15–26. ACM Press, 1992.

[15] Y. Lafont. Interaction Nets. In *Proceedings of ACM Symposium Principles of Programming Languages*, pages 95–108. ACM Press, 1990.

[16] Z. Luo. ECC, an Extended Calculus of Constructions. In *Proceedings of LICS'89*, IEEE, pages 386–395. IEEE Computer Society Press, 1989.

[17] Z. Luo. Program specification and data refinement in type theory. *Mathematical Structures in Computer Science*, 3:333–363, 1993.

[18] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical report, University of Edinburgh, 1992.

[19] P. Letouzey. A New Extraction for Coq. In *Proceedings of Types for Proofs and Programs*. Volume 2646 of LNCS.

[20] I. Mackie. YALE: Yet Another Lambda Evaluator Based on Interaction Nets. In *Proceedings of the 3rd International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, 1998.

[21] I. Mackie. Interaction Nets for Linear Logic. Theoretical Computer Science, 247:1, pages 83–140, 2000.

[22] J.H. McKinna. *Deliverables: a categorical approach to program development in type theory*. PhD thesis, University of Edinburgh, 1992.

[23] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *16th ACM Symposium on Principles of Programming Languages*, Austin, January 1989.

[24] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* Thèse d'université, Paris 7, 1989.

[25] E. Poll. *A Programming Logic Based on Type Theory.* PhD thesis, Eindhoven University of Technology, 1994.

[26] F. van Raamsdonk and P. Severi. Eliminating proofs from programs. In *Proc. 3rd. International Workshop on Logical Frameworks and Meta-Languages*, volume 70.2 of *ENTCS*, 2002.

[27] P. Severi and N. Szasz. Studies of a theory of specifications with built-in program extraction. *Journal of Automated Reasoning*, 27(1):61–87, 2001.

[28] P. Severi and N. Szasz. Internal Program Extraction in the Inductive Calculus of Constructions. In *Proceedings of WAIT'02, 31st JAIIO*, 2002.

[29] F-R. Sinot, M. Fernández and I. Mackie. Efficient Reductions with Director Strings. In *Proceedings of the Int. Conf. Rewriting Techniques and Applications*, Valencia, 2003. LNCS.

[30] Coq Development Team. The Coq proof assistant reference manual version 7.1, 2001. URL: `http://pauillac.inria.fr/coq/doc/main.html`.

[31] LEGO Team. The LEGO library, version 1.3, 1998.
URL: `http://www.dcs.ed.ac.uk/home/lego/html/release.html`.