From Call-by-Value to Interaction by Typed Closure Conversion

Ulrich Schöpp

LMU Munich, Germany
Ulrich.Schoepp@ifi.lmu.de

Abstract. We study the efficient implementation of call-by-value using the structure of interactive computation models. This structure has been useful in applications to resource-bounded compilation, but much of the existing work in this area has focused on call-by-name programming languages. This paper works towards the goal of a simple, efficient treatment of call-by-value languages. In previous work we have studied CPS-translation as an approach to implementing call-by-value and have observed that it needs to be refined in order to achieve efficient space usage. In this paper we give an alternative presentation of the refined translation, which is close to existing methods of typed closure conversion. We show that a simple correctness proof following Benton and Hur is possible for this formulation. Moreover, we extend previous work to cover full recursion in the source language.

1 Introduction

Recent advances in compiler verification, resource-bounded compilation, and resource usage certification underline the value of a good understanding of the properties of low-level computation. Certifying correctness and resource usage of compiled low-level programs makes it important to have a good understanding of their structure. While individually low-level programs have little structure, interesting structure appears when one considers them collectively and studies how they can be constructed and composed. For example for separate compilation and low-level program linking, perhaps of programs written in different programming languages, one is interested in the possible ways of composing of low-level programs. For resource-bounded compilation, one is interested in the resource usage of compiled programs, e.g. with respect to memory usage or execution time. Resource usage is a property of low-level programs and it is desirable to obtain a compositional understanding of such resource usage properties.

One line of investigation to identify useful mathematical structure of low-level computation consists of applying ideas from interaction semantics to low-level computation. Ideas from interaction semantics have been applied independently in a number of contexts, mainly for applications where particular control of low-level aspects is required. Examples include the interactive implementation of functional programs [12], abstract machines [4], hardware synthesis from functional programs [7], functional programming with logarithmic space [3], distributed programming [6], quantum programming languages [8], to mention just a few. While much of the work in this area was concerned with call-by-name programming languages, the success of the approach motivates an

increasing interest in applying these ideas to call-by-value computation. In addition to early approaches [5], call-by-value has received more attention recently [10,11,17].

In this paper we study a decomposition of call-by-value into the structure found in interactive models. This continues the work reported in [18] and [17], which investigates an approach of factoring the translation from a call-by-value source language to a first-order low-level language through a higher-order low-level calculus INT. This calculus is based on the structure of interactive computation models and captures structure of low-level computation in terms of a higher-order λ-calculus. Higher-order functions capture concepts of low-level code modules, their interfaces and their composition, see [18] for details. Factoring the translation from call-by-value to a first-order low-level language through this higher-order calculus allows one to use standard high-level proof techniques from semantics to show correctness of the translation from call-by-value to the first-order low-level language [17].

Here we develop the work reported in [17] on a decomposition of call-by-value computation that is suitable for targeting a calculus of interactive computation, such as INT. The aim of this paper is first to simplify the translation from [17], to formulate it in a direct style, close to other approaches to typed closure conversion [13,1], and second to generalise the correctness argument to include recursion in the source language.

The first contribution of this paper is to simplify the translation from [17], which is defined as a CPS-translation. It is observed in [17] that while it is possible to use a standard call-by-value CPS-translation to translate a call-by-value source language to INT, this would lead to an implementation of call-by-value with inefficient space usage. To address this, we propose in [17] a refined CPS-translation that makes the environment of all values explicit, so that unneeded values may be discarded explicitly when they are not needed anymore. The idea is to use a continuation "monad" of the form

$$(\forall\alpha.\, X(\alpha) \multimap \bot^{\alpha}) \multimap \bot^{\beta} \quad , \tag{1}$$

where $\multimap$ can be thought of as a function type that formalises module abstraction and composition (similar to functors in ML) and where $\bot^{\alpha}$ denotes a standard strict function space $\alpha \to \bot$. Terms of the above type represent source programs as follows. The type β should be thought of as a first-order type of the environments of the source program, e.g. the tuples of the values of its free variables, and the type α is a first-order type whose values can encode the possible values of the source term. To compute the source program, one throws its environment into $\bot^{\beta}$. This forces the computation of the program value, whose code will be returned to us by being thrown into $\bot^{\alpha}$. That α is universally quantified means that we must accept any possible encoding of the source value. In order to nevertheless make use of such an encoding, we are also provided with an argument of type $X(\alpha)$. This gives us an interface for making use of the value of type α we were give as the code for a source value. In the case of functions, $X(\alpha)$ will provide a way to apply any function given to us in the form of a code in α.

In this paper we simplify the translation by formulating it in a direct way. Instead of the above type we use a type of the form

$$\exists\alpha.\, X(\alpha) \otimes (\beta \to \alpha) \quad , \tag{2}$$

where $\otimes$ can be understood as a pairing operation on modules. This translation can be explained just as above; it represents the above way of computing source values more

directly. The simplification makes it easier to define and hopefully also to understand the translation. The result is a translation that is quite close to existing methods of typed closure conversion [13,1], which allows us to connect such techniques to work on the applications of interaction semantics [7,10].

We note that, when we use INT as a target for the translation, then the reformulation has no effect on the resulting first-order low-level programs. In INT (suitably extended with existential types) the types $(\forall\alpha.\, X(\alpha) \multimap \bot^{\alpha}) \multimap \bot^{\beta}$ and $\exists\alpha.X(\alpha) \times (\beta \to \alpha)$ lead to first-order low-level programs with the same interface. The translations are such that the programs turn out to be the same too. This establishes a formal connection between a CPS-translation and a typed closure conversion. In this paper, we use the term closure conversion to mean a conversion that makes the representation and manipulation of function closures fully explicit.

The second main contribution of this paper is to extend the correctness proof of the translation to cover recursion in the source language. To do so we follow the approach of Benton and Hur [2]. The resulting correctness proof uses a form of $\top\top$-lifting and turns out to be pleasingly simple.

While the translation of call-by-value presented in this paper is intended to target languages such as INT, it is not necessary to explain the particular details of INT. We present the translation more generally as a translation from a call-by-value source language to a variant of Idealized Algol [14]. Indeed, INT can be seen as variant of Idealized Algol with type annotations that make low-level computation details explicit. The point of this paper is to present the translation and its soundness proof. While the low-level details of the implementation of the target Algol-like language are interesting for understanding the overall implementation of call-by-value, here we focus just on the step to such an Algol-like language.

As a result we obtain a simple typed translation of call-by-value to the structure found in interaction semantics. The translation is quite similar to existing typed closure conversion approaches, which makes the results from this work available in the interactive context. While the translation presented in this paper is close to existing methods of closure conversion, it is not exactly the same, and it is not immediate to re-target the existing ones directly to Algol-like target languages with a call-by-name semantics.

By targeting Algol-like languages, we hope to be able to apply the results on call-by-value to resource-bounded situations. In particular, the Geometry of Synthesis [7] and the LOGSPACE-fragment of INT [3] are instances of Algol-like languages that are based on interaction semantics and that are interesting to study in further work.

An important feature of the translation in this paper is its compositionality. Open terms of the source language are translated to target programs with a well-specified interface. By composing the translation with an interpretation in INT, one obtains a well-defined interface of the low-level code arising from open source terms. This may be used to define a low-level calling convention for call-by-value.

The interfaces are formulated so that the choice of closure representation remains abstract. It is possible to translate parts of the same source terms using different closure representations and combine the translated programs to obtain a correct whole program. We believe that this is useful for studying separate compilation and linking of programs that are written in different languages.

$$\textsc{var}\ \frac{}{x{:}X \vdash x\colon X} \qquad \textsc{weak}\ \frac{\Gamma \vdash t\colon Y}{\Gamma, \Delta \vdash t\colon Y} \qquad \textsc{exch}\ \frac{\Gamma, y{:}Y, x{:}X, \Delta \vdash t\colon Z}{\Gamma, x{:}X, y{:}Y, \Delta \vdash t\colon Z}$$

$$\textsc{contr}\ \frac{\Gamma, x{:}X, y{:}X, \Delta \vdash t\colon Y}{\Gamma, z{:}X, \Delta \vdash t[z/x, z/y]\colon Y} \qquad \textsc{zero}\ \frac{}{\Gamma \vdash \mathtt{zero}\colon \mathbb{N}} \qquad \textsc{succ}\ \frac{\Gamma \vdash t\colon \mathbb{N}}{\Gamma \vdash \mathtt{succ}(t)\colon \mathbb{N}}$$

$$\textsc{abs}\ \frac{\Gamma, x{:}X \vdash t\colon Y}{\Gamma \vdash \mathtt{fn}\ x \Rightarrow t\colon X \to Y} \qquad \textsc{app}\ \frac{\Gamma \vdash s\colon X \to Y \qquad \Delta \vdash t\colon X}{\Gamma, \Delta \vdash s\ t\colon Y}$$

$$\textsc{if}\ \frac{\Gamma \vdash s\colon \mathbb{N} \qquad \Delta_1 \vdash t_1\colon \mathbb{N} \qquad \Delta_2 \vdash t_2\colon \mathbb{N}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathtt{if0}\ s\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2\colon \mathbb{N}} \qquad \textsc{rec}\ \frac{\Gamma, f{:}X \to Y, x{:}X \vdash t\colon Y}{\Gamma \vdash \mathtt{fun}\ f\ x \Rightarrow t\colon X \to Y}$$

Fig. 1. Source Typing Rules

2 Source Language

Our source language is a variant of call-by-value PCF. In this section we begin by fixing the details of this language. Its types and terms are defined as follows.

$$\begin{aligned} X, Y &::= \mathbb{N} \mid X \to Y \\ s, t &::= x \mid \mathtt{fn}\ x \Rightarrow t \mid \mathtt{fun}\ f\ x \Rightarrow t \mid s\ t \\ &\quad \mid \mathtt{zero} \mid \mathtt{succ}(s) \mid \mathtt{if0}\ s\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 \end{aligned}$$

The term $\mathtt{fun}\ f\ x \Rightarrow t$ represents a recursive function definition with the intended operational semantics $(\mathtt{fun}\ f\ x \Rightarrow t)\ v \longrightarrow t[(\mathtt{fun}\ f\ x \Rightarrow t)/f, v/x]$. We omit type annotations on abstractions, as we will work with typing derivations, so that such annotations are not important for our purposes.

The typing rules for the source language appear in Fig. 1. We make the structural rules explicit, as this makes it clear where variable duplication is needed later in the translation. The typing rule for `if-then-else` is restricted to base types. This simplifies the technical development and allows us to focus on the issues pertaining to recursion. We believe that the results in this paper can be extended to an unrestricted instance of this rule. For the time being, one may reduce case-distinction on function types to base-type case-distinction in the source language, e.g. replacing $\mathtt{if0}\ x\ \mathtt{then}\ (\mathtt{fn}\ y \Rightarrow t_1)\ \mathtt{else}\ (\mathtt{fn}\ y \Rightarrow t_2)$ with $\mathtt{fn}\ y \Rightarrow \mathtt{if0}\ x\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2$.

The aim in this paper is to give a correct implementation of this source language. The proof of correctness proceeds by an argument close to that of Benton and Hur [2]. It shows that the implementation correctly implements the standard denotational semantics of the source language [19]. The semantics of types is given by $[\![\mathbb{N}]\!] = \mathbb{N}$ and $[\![X \to Y]\!] = [\![X]\!] \Rightarrow [\![Y]\!]_\perp$, where $\mathbb{N}$ is the discrete cpo of the natural numbers, where $(-)_\perp$ denotes the lifting operation, and where $D \Rightarrow E$ is the cpo of continuous functions from cpo D to cpo E. We write $\lfloor - \rfloor\colon D \to D_\perp$ for the injection of a cpo D into its lifting. The interpretation of types is extended to contexts in the standard way, i.e. if Γ is $x_1{:}X_1, \ldots, x_n{:}X_n$, then $[\![\Gamma]\!]$ is $[\![X_1]\!] \times \cdots \times [\![X_n]\!]$.

For each derivation Π of $\Gamma \vdash t\colon X$ in the source language, one defines an interpretation $[\![\Pi]\!] \in [\![\Gamma]\!] \Rightarrow [\![X]\!]_\perp$ in the usual way [19].

3 Target Language

We translate the source language to a target language that is close in spirit to Idealized Algol [14]. As outlined in the Introduction, this choice is motivated by the development of the calculus INT [18,16], which can be seen as an Algol-like language with a type system that allows for the control of low-level aspects of computation. By translating the source language to INT, one obtains a fully specified translation to a simple first-order low-level language. Here, we concentrate on the translation from the source language to Algol-like languages, as definition of the translation and its correctness does not depend on the particular details of INT. This allows us to concentrate on the essential issues of the translation.

The core of the target language has the following types.

$$\begin{aligned} \text{Value Types } A, B &::= \alpha \mid \mathtt{void} \mid \mathtt{unit} \mid \mathtt{nat} \mid A \times B \\ \text{Types } X, Y &::= \mathtt{exp}(A, B) \mid X \to Y \mid X \times Y \mid \forall\alpha.\, X \mid \exists\alpha.\, X \end{aligned}$$

The type $\mathtt{exp}(A, B)$ is meant to contain (strict) functions that map values of type A to values of type B. This is a slight generalisation of Idealized Algol, which only has a type $\mathtt{exp}(A)$ corresponding to $\mathtt{exp}(\mathtt{unit}, A)$.

At first it may appear that this target language is of similar expressiveness as the source language. However, note first that in the type $\mathtt{exp}(A, B)$, both A and B are restricted to be value types. This means that call-by-value functions are not available directly in this language. Of course, it is possible to represent call-by-value computation by means of a call-by-value CPS-translation, for example. However, we have observed in [17] that such a CPS-translation would generally not result in a space-efficient implementation of the source language.

In the target language, the terms of type $\mathtt{exp}(A, B)$ are our basic notion of computation. We use the higher-order structure only to compose such computations and to organise them, much like in [18]. The reader should think of a term of type $\mathtt{exp}(A, B) \to \mathtt{exp}(C, D)$ less as a function and more as a program that expects to be connected with a module that contains a function from A to B and that then provides a function from C to D. In OCaml notation this would roughly amount to functor of the following type:

```
functor (X: sig val t: A -> B end) -> sig val t: C -> D end
```

Indeed, one could use a language with ML-style modules as a target language in this paper. Notice, in particular, that the universal and existential quantification in the target language are restricted to range over value types only. However, it seems that a lambda-calculus is more convenient as a language for defining and composing modules [15,18].

The terms of the target language are standard for all but the type $\mathtt{exp}(A, B)$. For this type it is also possible to follow the standard approach for Idealized Algol and define the terms of this type using suitable constants. In Idealized Algol one finds constants like

$\mathtt{succ}\colon \mathtt{exp}(\mathtt{nat}) \to \mathtt{exp}(\mathtt{nat})$ and a similar approach can be taken for $\mathtt{exp}(A, B)$. A possible choice of constants for this type could include constants such as the following:

$$\begin{aligned}\mathtt{proj}&\colon \mathtt{exp}(A \times B, B)\\ \mathtt{pair}&\colon \mathtt{exp}(C, A) \times \mathtt{exp}(C, B) \to \mathtt{exp}(C, A \times B)\\ \mathtt{seq}&\colon \mathtt{exp}(C, A) \times \mathtt{exp}(C \times A, B) \to \mathtt{exp}(C, B)\end{aligned}$$

However, working with such constants is somewhat awkward syntactically.

We therefore define a slightly extended target calculus that allows a simpler, more direct, definition of terms of type $\mathtt{exp}(A, B)$ as in [18]. It also contains the type $\mathtt{exp}(A)$, which is familiar from Idealized Algol and corresponds to $\mathtt{exp}(\mathtt{unit}, A)$.

$$\begin{aligned}\text{Value Types } A, B &::= \alpha \mid \mathtt{void} \mid \mathtt{unit} \mid \mathtt{nat} \mid A \times B\\ \text{Types } X, Y &::= \mathtt{exp}(A) \mid \mathtt{exp}(A, B) \mid X \to Y \mid X \times Y \mid \forall\alpha.\, X \mid \exists\alpha.\, X\end{aligned}$$

We use the abbreviations $I := \mathtt{exp}(\mathtt{void}, \mathtt{void})$ for a vacuous type with a single inhabitant $\star := (\mathtt{fn}\, v{:}\mathtt{void} \Rightarrow \mathtt{return}\, v)$. (Note that I is not isomorphic to $\mathtt{exp}(\mathtt{unit}, \mathtt{unit})$, which has two inhabitants, the function returning $\langle\rangle$ and the non-terminating function.)

The terms are given by the grammar below, in which n ranges over natural numbers.

$$\begin{aligned}\text{Values } v, w &::= x \mid \langle\rangle \mid \langle v, w\rangle \mid n\\ \text{Terms } s, t &::= x \mid \mathtt{return}\, v \mid \mathtt{let}\, x = s\, \mathtt{in}\, t \mid \mathtt{if0}\, v\, s\, t \mid \mathtt{fn}\, x{:}A \Rightarrow t \mid t(v)\\ &\quad \mid \lambda x{:}X.\, t \mid s\, t \mid \langle s, t\rangle \mid \mathtt{let}\, \langle x, y\rangle = s\, \mathtt{in}\, t\\ &\quad \mid \Lambda\alpha.t \mid t\, A \mid \mathtt{pack}_X(A, t) \mid \mathtt{let}\, \mathtt{pack}(\alpha, x) = s\, \mathtt{in}\, t\\ &\quad \mid \mathtt{fix}^n_X \mid \mathtt{fix}_X\end{aligned}$$

We write $\langle t_1, t_2, \ldots, t_k\rangle$ as an abbreviation for $\langle \ldots \langle t_1, t_2\rangle \ldots, t_k\rangle$, i.e. pairs associate to the left. We will often omit type annotations, such as the X in $\mathtt{pack}_X(A, t)$, for readability.

The terms include constructs that allow one to view the elements of type $\mathtt{exp}(A, B)$ as functions. Terms of this type can be defined using an abstraction $\mathtt{fn}\, x{:}A \Rightarrow t$. In this term the body t must have type $\mathtt{exp}(B)$, which is why we have included types of the form $\mathtt{exp}(B)$ in the language. Elements of type $\mathtt{exp}(B)$ can be defined using $\mathtt{return}\, v$, and the term $\mathtt{let}\, x = t_1\, \mathtt{in}\, t_2$ can be used to sequence computations $t_1\colon \mathtt{exp}(B_1)$ and $t_2\colon \mathtt{exp}(B_2)$. For a term t of type $\mathtt{exp}(A, B)$, there is an application $t(v)$. It is restricted to values as arguments, which corresponds to the view of $\mathtt{exp}(A, B)$ as strict functions from values of type A to values of type B.

The typing rules for the target calculus are given in Figs. 2 and 3. The typing judgements have the form $\Gamma \vdash t\colon A$ for value types A and $\Gamma \mid \Delta \vdash t\colon X$ for types X, where Γ is a value context of the form $x_1{:}A_1, \ldots, x_n{:}A_n$ (the A_i range over value types) and Δ is a context $y_1{:}Y_1, \ldots, y_n{:}Y_n$ (the Y_i range over types). We identify these contexts up to reordering of variable declarations.

The value context is there to allow the formulation of the term constructs for the types $\mathtt{exp}(A)$ and $\mathtt{exp}(A, B)$. If one were to prefer a formulation of the calculus with constants for $\mathtt{exp}(A, B)$, as mentioned above, then the value context Γ could be replaced with a context of the form $x_1{:}\mathtt{exp}(A_1), \ldots, x_n{:}\mathtt{exp}(A_n)$. We prefer the version with two contexts, as it clarifies which terms can be assumed to denote values.

$$\frac{}{\Gamma,\, x{:}A \vdash x\colon A} \qquad \frac{}{\Gamma \vdash \langle\rangle\colon \mathtt{unit}} \qquad \frac{}{\Gamma \vdash n\colon \mathbb{N}}$$

$$\frac{\Gamma \vdash v\colon A \qquad \Gamma \vdash w\colon B}{\Gamma \vdash \langle v, w\rangle\colon A \times B} \qquad \frac{\Gamma \vdash v\colon A \times B \qquad \Gamma,\, x{:}A,\, y{:}B \vdash w\colon C}{\Gamma \vdash \mathtt{let}\ \langle x, y\rangle = v\ \mathtt{in}\ w\colon C}$$

Fig. 2. Target Language: Terms of Value Types

3.1 Equational Theory

We assume a standard equational theory for the target language. Equations are given in context, i.e. they are given by a judgement of the form $\Gamma \mid \Delta \vdash s = t\colon X$. For brevity, we state equations without typing contexts, but with the understanding that the equations are subject to suitable typing constraints.

$$\begin{aligned}
\mathtt{let}\ \langle x, y\rangle = \langle s, t\rangle\ \mathtt{in}\ u &= u[s/x, t/y] && (\times\beta)\\
\mathtt{let}\ \langle x, y\rangle = s\ \mathtt{in}\ t[\langle x, y\rangle/z] &= t[s/z] \quad \text{if } x, y \text{ not free in } t && (\times\eta)\\
(\lambda x{:}X.\, s)\ t &= s[t/x] && (\to\beta)\\
\lambda x{:}X.\, (t\ x) &= t \quad \text{if } x \text{ not free in } t && (\to\eta)\\
(\Lambda\alpha.\, t)\ A &= t[A/\alpha] && (\forall\beta)\\
\Lambda\alpha.\, t\ \alpha &= t \quad \text{if } \alpha \text{ not free in } t && (\forall\eta)\\
\mathtt{let}\ \mathtt{pack}(\alpha, x) = \mathtt{pack}(A, s)\ \mathtt{in}\ t &= t[A/\alpha, s/x] && (\exists\beta)\\
\mathtt{let}\ \mathtt{pack}(\alpha, x) = s\ \mathtt{in}\ t[\mathtt{pack}(\alpha, x)/y] &= t[s/y] \quad \text{if } \alpha, x \text{ not free in } t && (\exists\eta)\\
\mathtt{fix}_X\ t &= t\ (\mathtt{fix}_X\ t) && (\mathtt{fix}\beta)\\
\mathtt{fix}_X^{i+1}\ t &= t\ (\mathtt{fix}_X^{i}\ t) && (\mathtt{fix}^i\beta)
\end{aligned}$$

Equation $(\times\eta)$ is formulated with $t[\langle x, y\rangle/z]$ instead of just $\langle x, y\rangle$, in order to make commuting conversions derivable.

For the terms for the types $\mathtt{exp}(A)$ and $\mathtt{exp}(A, B)$ and for value types, we assume the following equations.

$$\begin{aligned}
(\mathtt{fn}\ x{:}A \Rightarrow t)(v) &= t[v/x] && (\mathtt{exp}\beta)\\
\mathtt{fn}\ x{:}A \Rightarrow (t(x)) &= t \quad \text{if } x \text{ not free in } t && (\mathtt{exp}\eta)\\
\mathtt{let}\ x = \mathtt{return}\ v\ \mathtt{in}\ t &= t[v/x] && (\mathtt{let1})\\
\mathtt{let}\ x = (\mathtt{let}\ y = s\ \mathtt{in}\ t)\ \mathtt{in}\ u &= \mathtt{let}\ y = s\ \mathtt{in}\ \mathtt{let}\ x = t\ \mathtt{in}\ u && (\mathtt{let2})\\
\mathtt{succ}(n) &= n + 1 && (\mathtt{succ})\\
\mathtt{if0}\ 0\ \mathtt{then}\ v\ \mathtt{else}\ w &= v && (\mathtt{if1})\\
\mathtt{if0}\ n\ \mathtt{then}\ v\ \mathtt{else}\ w &= w \quad \text{if } n > 0 && (\mathtt{if2})
\end{aligned}$$

(The equations for $\times$ also hold for terms of value types; we do not repeat them here.) Finally, there are rules for reflexivity, symmetry, transitivity and there are congruence rules that allow one to apply these equations in any context.

$$\frac{}{\Gamma \mid \Delta,\, x{:}X \vdash x\colon X}$$

$$\frac{\Gamma \vdash v\colon A}{\Gamma \mid \Delta \vdash \mathtt{return}\, v\colon \mathtt{exp}(A)} \qquad \frac{\Gamma \mid \Delta \vdash s\colon \mathtt{exp}(A) \qquad \Gamma,\, x{:}A \mid \Delta \vdash t\colon \mathtt{exp}(B)}{\Gamma \mid \Delta \vdash \mathtt{let}\, x = s\, \mathtt{in}\, t\colon \mathtt{exp}(B)}$$

$$\frac{\Gamma \vdash v\colon \mathbb{N} \qquad \Gamma \mid \Delta \vdash s\colon \mathtt{exp}(A) \qquad \Gamma \mid \Delta \vdash t\colon \mathtt{exp}(A)}{\Gamma \mid \Delta \vdash \mathtt{if0}\, v\, \mathtt{then}\, s\, \mathtt{else}\, t\colon \mathtt{exp}(A)}$$

$$\frac{\Gamma,\, x{:}A \mid \Delta \vdash t\colon \mathtt{exp}(B)}{\Gamma \mid \Delta \vdash \mathtt{fn}\, x{:}A \Rightarrow t\colon \mathtt{exp}(A, B)} \qquad \frac{\Gamma \mid \Delta \vdash t\colon \mathtt{exp}(A, B) \qquad \Gamma \vdash v\colon A}{\Gamma \mid \Delta \vdash t(v)\colon \mathtt{exp}(B)}$$

$$\frac{\Gamma \mid \Delta,\, x{:}X \vdash t\colon Y}{\Gamma \mid \Delta \vdash \lambda x{:}X.\, t\colon X \to Y} \qquad \frac{\Gamma \mid \Delta \vdash s\colon X \to Y \qquad \Gamma \mid \Delta \vdash t\colon X}{\Gamma \mid \Delta \vdash s\, t\colon Y}$$

$$\frac{\Gamma \mid \Delta \vdash s\colon X \qquad \Gamma \mid \Delta \vdash t\colon Y}{\Gamma \mid \Delta \vdash \langle s, t\rangle\colon X \times Y} \qquad \frac{\Gamma \mid \Delta \vdash s\colon X \times Y \qquad \Gamma \mid \Delta,\, x{:}X,\, y{:}Y \vdash t\colon Z}{\Gamma \mid \Delta \vdash \mathtt{let}\, \langle x, y\rangle = s\, \mathtt{in}\, t\colon Z}$$

$$\frac{\Gamma \mid \Delta \vdash t\colon X}{\Gamma \mid \Delta \vdash \Lambda\alpha.\, t\colon \forall\alpha.\, X}\ (*) \qquad \frac{\Gamma \mid \Delta \vdash t\colon \forall\alpha.\, X}{\Gamma \mid \Delta \vdash t\ A\colon X[A/\alpha]}$$

$$\frac{\Gamma \mid \Delta \vdash t\colon X[A/\alpha]}{\Gamma \mid \Delta \vdash \mathtt{pack}_{\exists\alpha.\, X}(A, t)\colon \exists\alpha.\, X} \qquad \frac{\Gamma \mid \Delta \vdash s\colon \exists\alpha.\, X \qquad \Gamma \mid \Delta,\, x{:}X \vdash t\colon Z}{\Gamma \mid \Delta \vdash \mathtt{let}\, \mathtt{pack}(\alpha, x) = s\, \mathtt{in}\, t\colon Z}\ (*)$$

$$\frac{}{\Gamma \mid \Delta \vdash \mathtt{fix}_X\colon (X \to X) \to X} \qquad \frac{}{\Gamma \mid \Delta \vdash \mathtt{fix}^i_X\colon (X \to X) \to X}$$

$(*)$ — α not free in Γ, Δ

Fig. 3. Target Language: Terms

4 Translation

We are now ready to define the translation from source to target language. It is compositional and identifies an interface in the target language for the translation of each subterm of a source term.

For each source type X, we define a family $\mathcal{C}[\![X]\!]_A$ of value types and a family $\mathcal{I}[\![X]\!]_A$ of types, both families being indexed by a value type A.

$$\begin{aligned} \mathcal{C}[\![\mathbb{N}]\!]_A &:= \mathtt{nat} & \mathcal{C}[\![X \to Y]\!]_A &:= A \\ \mathcal{I}[\![\mathbb{N}]\!]_A &:= I & \mathcal{I}[\![X \to Y]\!]_A &:= \forall\beta.\, \mathcal{I}[\![X]\!]_\beta \to \mathcal{M}[\![Y]\!]_{A \times \mathcal{C}[\![X]\!]_\beta} \end{aligned}$$

where

$$\mathcal{M}[\![X]\!]_A := \exists\beta.\, \mathcal{I}[\![X]\!]_\beta \times \mathtt{exp}(A, \mathcal{C}[\![X]\!]_\beta)$$

The intention is that a source language value of type X is represented by a target language value $c\colon \mathcal{C}[\![X]\!]_A$ and a term $a\colon \mathcal{I}[\![X]\!]_A$, where A may be any value type. The type $\mathcal{M}[\![X]\!]_A$ corresponds to (2) in the Introduction.

For the type of natural numbers, the choice of $\mathcal{C}[\![\mathbb{N}]\!]_A$ is such that we can simply choose c to be the represented number. The term a is not important for natural numbers, as all relevant information is already in c. Therefore, we choose the type $\mathcal{I}[\![\mathbb{N}]\!]_A$ to be vacuous.

Values of function type $X \to Y$, are represented by a value $c\colon A$ and a term a of type $\forall\beta.\, \mathcal{I}[\![X]\!]_\beta \to \mathcal{M}[\![Y]\!]_{A\times\mathcal{C}[\![X]\!]_\beta}$, where A may be an arbitrary value type. The idea is that c is a code value for the function, which may be encoded in any way using an arbitrary value type A. Of course, if we allow an arbitrary encoding, then we must explain how the code c can be used to apply the function. This is what the term a does. Suppose $c_x\colon \mathcal{C}[\![X]\!]_{A_X}$ and $a_x\colon \mathcal{I}[\![X]\!]_{A_X}$ represent a value of type X. Then a allows us to obtain a code for the result of the application of the function encoded by c and a to the argument encoded by c_x and a_x. Applying a to A_x and a_x amounts to connecting the module a_x to a. We obtain a term $a\ A_x\ a_x$ of type $\exists\alpha.\, \mathcal{I}[\![Y]\!]_\alpha \times \mathtt{exp}(A \times \mathcal{C}[\![X]\!]_{A_x}, \mathcal{C}[\![Y]\!]_\alpha)$. The function in the second component takes a pair of type $A \times \mathcal{C}[\![X]\!]_{A_x}$ consisting of the code for the function and the code for the argument. It returns a code for the result of the function application. Since the type α is existentially quantified, the type that is used to represent the result of the function application is abstract. The first component $\mathcal{I}[\![Y]\!]_\alpha$ allows us to make use of values encoded using α.

This application procedure is captured by the following application term.

$$\begin{aligned}
&\mathtt{app}((A,a,c),(A_x,a_x,c_x))\colon \exists\alpha.\, \mathcal{I}[\![Y]\!]_\alpha \times \mathtt{exp}(\mathtt{unit}, \mathcal{C}[\![Y]\!]_\alpha)\\
&\mathtt{app}((A,a,c),(A_x,a_x,c_x)) \ :=\ \mathtt{let}\ \mathtt{pack}(\alpha_y, \langle a_y, e_y\rangle) = a\ A_x\ a_x\ \mathtt{in}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{pack}(\alpha_y, \langle a_y, \mathtt{fn}\ \langle\rangle \Rightarrow e_y(\langle c, c_x\rangle)\rangle)
\end{aligned}$$

Here, and in the rest of the paper, we allow ourselves some syntactic sugar for pattern matching in `fn`-function and to avoid nested `let`-terms.

The translation from source to target language is defined by induction on the typing derivation. To define it, we extend the definition of $\mathcal{C}[\![-]\!]$ to source contexts by:

$$\mathcal{C}[\![\text{empty}]\!] := \mathtt{unit} \qquad\qquad \mathcal{C}[\![\Gamma,\, x{:}X]\!]_{\vec{B},A} := \mathcal{C}[\![\Gamma]\!]_{\vec{B}} \times \mathcal{C}[\![X]\!]_A$$

A derivation Π of a source sequent $\Gamma \vdash t\colon Y$, where Γ is $x_1{:}X_1, \ldots, x_k{:}X_k$, is then translated to a closed target term $(\!|\Pi|\!)$ of type

$$\forall\alpha_1 \ldots \alpha_k.\, \mathcal{I}[\![X_1]\!]_{\alpha_1} \to \cdots \to \mathcal{I}[\![X_k]\!]_{\alpha_k} \to \mathcal{M}[\![Y]\!]_{\mathcal{C}[\![\Gamma]\!]_{\vec{\alpha}}} \ .$$

The definition of $(\!|\Pi|\!)$ is given by induction on the derivation Π of $\Gamma \vdash t\colon X$.

– Case: Π is

$$\textsc{var}\ \frac{}{x{:}X \vdash x\colon X}$$

$$(\!|\Pi|\!) \ :=\ \Lambda\alpha.\, \lambda a\colon \mathcal{I}[\![X]\!]_\alpha.\, \mathtt{pack}(\alpha, \langle a, \mathtt{fn}\ \langle\rangle, x\rangle \Rightarrow \mathtt{return}\ x\rangle)$$

– Case: Π ends with rule (WEAK), whose premise is derived by Π_t.

$$\textsc{weak}\ \frac{\Gamma \vdash t\colon Y}{\Gamma, \Delta \vdash t\colon Y}$$

$$\begin{aligned}(\!|\Pi|\!) \ :=\ \Lambda\vec{\alpha}\vec{\beta}.\, \lambda\vec{a}\vec{b}.\, &\texttt{let pack}(A_t, \langle a_t, e_t\rangle) = (\!|\Pi_t|\!)\ \vec{\alpha}\ \vec{a}\ \texttt{in}\\ &\texttt{pack}(A_t, \langle a_t, \texttt{fn}\ (\vec{x}, \vec{y}) \Rightarrow e_t(\vec{x})\rangle)\end{aligned}$$

Here we use an informal pattern matching notation $(\vec{x}, \vec{y})$ that matches $\vec{x}$ against the code values for Γ and $\vec{y}$ against the values for Δ.
We omit the similar cases for the other structural rules.

– Case: Π ends with rule (ABS), whose premise is derived by Π_t.

$$\textsc{abs}\ \frac{\Gamma,\ x{:}X \vdash t\colon Y}{\Gamma \vdash \texttt{fn}\ x \Rightarrow t\colon X \to Y}$$

$$(\!|\Pi|\!) \ :=\ \Lambda\vec{\alpha}.\, \lambda\vec{x}.\, \texttt{pack}(\mathcal{C}[\![\Gamma]\!]_{\vec{\alpha}}, \langle(\Lambda\beta.\, \lambda x{:}\mathcal{I}[\![X]\!]_{\beta}.\, (\!|\Pi_t|\!)\ \vec{\alpha}\ \beta\ \vec{x}\ x),\ \texttt{fn}\ c \Rightarrow \texttt{return}\ c\rangle)$$

In this definition we choose the type $\mathcal{C}[\![\Gamma]\!]_{\vec{\alpha}}$ for the code type to represent the function defined by the abstraction. The function $\texttt{fn}\ c \Rightarrow \texttt{return}\ c$ gets as argument the tuple of the values of the variables in Γ and returns the code unchanged as the code for the function.
Notice that the term $(\!|\Pi_t|\!)\ \vec{\alpha}\ \beta\ \vec{x}\ x$ has type $\mathcal{M}[\![Y]\!]_{\mathcal{C}[\![\Gamma,\, x:X]\!]_{\vec{\alpha},\beta}}$, which is the same as $\mathcal{M}[\![Y]\!]_{\mathcal{C}[\![\Gamma]\!]_{\vec{\alpha}} \times \mathcal{C}[\![X]\!]_{\beta}}$. By our choice of $\mathcal{C}[\![\Gamma]\!]_{\vec{\alpha}}$ as the code type, the pair $\mathcal{C}[\![\Gamma]\!]_{\vec{\alpha}} \times \mathcal{C}[\![X]\!]_{\beta}$ can be understood as the pair of the function code and its argument, as is required for the translation of functions.
While we have chosen the tuple of the free variables for function codes here, other choices are possible, and can be made differently on a case-by-case basis.

– Case: Π ends with rule (APP), whose premises are derived by Π_s and Π_t.

$$\textsc{app}\ \frac{\Gamma \vdash s\colon X \to Y \qquad \Delta \vdash t\colon X}{\Gamma,\ \Delta \vdash s\ t\colon Y}$$

$$\begin{aligned}(\!|\Pi|\!) \ :=\ \Lambda\vec{\alpha}\vec{\beta}.\, \lambda\vec{x}.\, \lambda\vec{y}.\, &\texttt{let pack}(\varphi, \langle f, e_f\rangle) = (\!|\Pi_s|\!)\ \vec{\alpha}\ \vec{x}\ \texttt{in}\\ &\texttt{let pack}(\xi, \langle x, e_x\rangle) = (\!|\Pi_t|\!)\ \vec{\beta}\ \vec{y}\ \texttt{in}\\ &\texttt{let pack}(\rho, \langle y, a\rangle) = f\ \xi\ x\ \texttt{in}\\ &\texttt{pack}(\rho, \langle y, e\rangle)\end{aligned}$$

where e is $\texttt{fn}\ (\vec{x}, \vec{y}) \Rightarrow\ \texttt{let}\ v_f = e_f(\vec{x})\ \texttt{in let}\ v_x = e_x(\vec{y})\ \texttt{in}\ a(\langle v_f, v_x\rangle)$.

– Case: Π ends with rule (ZERO).

$$\textsc{zero}\ \frac{}{\Gamma \vdash \texttt{zero}\colon \mathbb{N}}$$

$$(\!|\Pi|\!) \ :=\ \Lambda\vec{\alpha}.\, \lambda\vec{x}.\, \texttt{pack}(\texttt{unit}, \langle\star, \texttt{fn}\ g \Rightarrow \texttt{return}\ 0\rangle)$$

– Case: Π ends with rule (SUCC), whose premise is derived by Π_t.

$$\textsc{succ}\ \frac{\Gamma \vdash t\colon \mathbb{N}}{\Gamma \vdash \texttt{succ}(t)\colon \mathbb{N}}$$

$$(\!|\Pi|\!) \;:=\; \Lambda\vec{\alpha}.\,\lambda\vec{x}.\,\mathtt{let\ pack}(\alpha_t, \langle a_t, e_t\rangle) = (\!|\Pi_t|\!)\ \vec{\alpha}\ \vec{x}\ \mathtt{in}$$
$$\mathtt{pack}(\alpha_t, \langle a_t, \mathtt{fn}\ g \Rightarrow \mathtt{let}\ y = e_t(g)\ \mathtt{in\ return\ succ}(y)\rangle)$$

– Case: Π ends with rule (IF) whose premises are derived by Π_s, Π_{t_1} and Π_{t_2}.

$$\text{IF}\ \frac{\Gamma \vdash s\colon \mathbb{N} \qquad \Delta_1 \vdash t_1\colon \mathbb{N} \qquad \Delta_2 \vdash t_2\colon \mathbb{N}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathtt{if0}\ s\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2\colon \mathbb{N}}$$

$$\begin{aligned}(\!|\Pi|\!) \;:=\; \Lambda\vec{\alpha}\vec{\beta}\vec{\gamma}.\,\lambda\vec{x}\vec{y}\vec{z}.\,&\mathtt{let\ pack}(\alpha_s, \langle a_s, e_s\rangle) = (\!|\Pi_s|\!)\ \vec{\alpha}\ \vec{x}\ \mathtt{in}\\ &\mathtt{let\ pack}(\alpha_1, \langle a_1, e_1\rangle) = (\!|\Pi_{t_1}|\!)\ \vec{\beta}\ \vec{y}\ \mathtt{in}\\ &\mathtt{let\ pack}(\alpha_2, \langle a_2, e_2\rangle) = (\!|\Pi_{t_2}|\!)\ \vec{\gamma}\ \vec{z}\ \mathtt{in}\\ &\mathtt{pack}(\mathtt{unit}, \langle\star, e\rangle)\end{aligned}$$

where e is $(\mathtt{fn}\ (\vec{x}, \vec{y}, \vec{z}) \Rightarrow \mathtt{let}\ v = e_s\ \vec{x}\ \mathtt{in\ if0}\ v\ \mathtt{then}\ e_1(\vec{y})\ \mathtt{else}\ e_2(\vec{z}))$.

– Case: Π ends with rule (REC), whose premise is derived by Π_t.

$$\text{REC}\ \frac{\Gamma,\ f{:}X \to Y,\ x{:}X \vdash t\colon Y}{\Gamma \vdash \mathtt{fun}\ f\ x \Rightarrow t\colon X \to Y}$$

$$\Lambda\vec{\alpha}.\,\lambda\vec{x}.\,\mathtt{pack}(\mathcal{C}[\![\Gamma]\!], \langle \mathtt{fix}\ \mathit{step}, \mathtt{fn}\ g \Rightarrow \mathtt{return}\ g\rangle)$$

where

$$\begin{aligned}\mathit{step} \;:=\; \lambda f.\,\Lambda\beta.\,\lambda x.\,&\mathtt{let\ pack}(\rho, \langle a, e\rangle) = (\!|\Pi_t|\!)\ \vec{\alpha}\ \mathcal{C}[\![\Gamma]\!]_{\vec{\alpha}}\ \beta\ \vec{x}\ f\ x\ \mathtt{in}\\ &\mathtt{pack}(\rho, \langle a, \mathtt{fn}\ \langle c, x\rangle \Rightarrow e(\langle\langle c, c\rangle, x\rangle)\rangle)\ .\end{aligned}$$

Here, c is the tuple of the values in the context Γ. This tuple is used as the code for the recursive function f.

4.1 Examples

Let us spell out the translation of the following simple source term.

$$\vdash (\mathtt{fn}\ s \Rightarrow s\ \mathtt{zero})\ (\mathtt{fn}\ x \Rightarrow \mathtt{succ}(x))\colon \mathbb{N}$$

We spell out the non-trivial subterms:

– $(\!|s{:}\mathbb{N} \to \mathbb{N} \vdash s\ \mathtt{zero}\colon \mathbb{N}|\!)$ is defined to be

$$\begin{aligned}\Lambda\alpha.\,\lambda s\colon \mathcal{I}[\![\mathbb{N} \to \mathbb{N}]\!]_\alpha.\,&\mathtt{let\ pack}(\varphi, \langle f, e_f\rangle) = (\!|s{:}\mathbb{N} \to \mathbb{N} \vdash s\colon \mathbb{N} \to \mathbb{N}|\!)\ \alpha\ s\ \mathtt{in}\\ &\mathtt{let\ pack}(\xi, \langle x, e_x\rangle) = (\!|\vdash 0\colon \mathbb{N}|\!)\ \mathtt{in}\\ &\mathtt{let\ pack}(\rho, \langle y, a\rangle) = f\ \xi\ x\ \mathtt{in}\\ &\mathtt{pack}(\rho, \langle y, e\rangle)\end{aligned}$$

where e is $\mathtt{fn}\ \langle\langle\rangle, c_s\rangle \Rightarrow \mathtt{let}\ c_f = e_f(\langle\langle\rangle, c_s\rangle)\ \mathtt{in\ let}\ c_x = e_x(\langle\rangle)\ \mathtt{in}\ a(\langle c_f, c_x\rangle)$.
Using the equational theory, this term may be simplified as follows:

$$\begin{aligned}(\!|s{:}\mathbb{N} \to \mathbb{N} \vdash s\ \mathtt{zero}\colon \mathbb{N}|\!) \;=\; \Lambda\alpha.\,\lambda s\colon \mathcal{I}[\![\mathbb{N} \to \mathbb{N}]\!]_\alpha.\,&\mathtt{let\ pack}(\rho, \langle y, a\rangle) = s\ \mathtt{unit}\ \star\ \mathtt{in}\\ &\mathtt{pack}(\rho, \langle y, \mathtt{fn}\ \langle\langle\rangle, c_s\rangle \Rightarrow a(c_s, 0)\rangle)\end{aligned}$$

– $(\!|\, \vdash \texttt{fn}\ s \Rightarrow s\ \texttt{zero}\colon \mathbb{N} |\!)$ is then given by:

$$\texttt{pack}(\texttt{unit}, \langle (\!|s{:}\mathbb{N} \to \mathbb{N} \vdash s\ \texttt{zero}\colon \mathbb{N}|\!), \texttt{fn}\ g \Rightarrow \texttt{return}\ g\rangle)$$

– For the function argument, we get:

$$\begin{aligned}&(\!|\, \vdash \texttt{fn}\ x{:}\mathbb{N} \Rightarrow \texttt{succ}(x)\colon \mathbb{N} \to \mathbb{N}|\!)\\ &= \texttt{pack}(\texttt{unit}, \langle \Lambda\alpha.\, \lambda x\colon I.\, \texttt{pack}(\alpha, \langle x, \texttt{fn}\ \langle\rangle, c_x\rangle \Rightarrow \texttt{return}\ \texttt{succ}(c_x)\rangle),\\ &\qquad\qquad \texttt{fn}\ c \Rightarrow \texttt{return}\ c\rangle)\end{aligned}$$

– Putting these terms together the translation of the whole term we can be calculated and simplified as follows.

$$\begin{aligned}&(\!|\vdash (\texttt{fn}\ s \Rightarrow s\ \texttt{zero})\ (\texttt{fn}\ x \Rightarrow \texttt{succ}(x)\colon \mathbb{N}|\!)\\ &= \texttt{let}\ \texttt{pack}(\varphi, \langle f, e_f\rangle) = (\!|\, \vdash \texttt{fn}\ s \Rightarrow s\ \texttt{zero}|\!)\ \texttt{in}\\ &\quad \texttt{let}\ \texttt{pack}(\xi, \langle x, e_x\rangle) = (\!|\, \vdash \texttt{fn}\ x \Rightarrow \texttt{succ}(x)|\!)\ \texttt{in}\\ &\quad \texttt{let}\ \texttt{pack}(\rho, \langle y, a\rangle) = f\ \xi\ x\ \texttt{in}\\ &\quad \texttt{pack}(\rho, \langle y, \texttt{fn}\ \langle\rangle \Rightarrow \texttt{let}\ c_f = e_f(\langle\rangle)\ \texttt{in}\ \texttt{let}\ c_x = e_x(\langle\rangle)\ \texttt{in}\ a(\langle c_f, c_x\rangle)\rangle)\\ &= \texttt{let}\ \texttt{pack}(\xi, \langle x, e_x\rangle) = (\!|\, \vdash \texttt{fn}\ x \Rightarrow \texttt{succ}(x)|\!)\ \texttt{in}\\ &\quad \texttt{let}\ \texttt{pack}(\rho, \langle y, a\rangle) = (\!|s\ \texttt{zero}|\!)\ \xi\ x\ \texttt{in}\\ &\quad \texttt{pack}(\rho, \langle y, \texttt{fn}\ \langle\rangle \Rightarrow \texttt{let}\ c_x = e_x(\langle\rangle)\ \texttt{in}\ a(\langle\langle\rangle, c_x\rangle)\rangle)\\ &= \texttt{let}\ \texttt{pack}(\rho, \langle y, a\rangle) = (\!|s\ \texttt{zero}|\!)\ \texttt{unit}\ (\Lambda\alpha.\, \lambda x\colon I.\, \texttt{pack}(\alpha, \langle x, e\rangle))\ \texttt{in}\\ &\quad \texttt{pack}(\rho, \langle y, \texttt{fn}\ \langle\rangle \Rightarrow a(\langle\langle\rangle, \langle\rangle\rangle)\rangle) \text{ where } e \text{ is } \texttt{fn}\ \langle\langle\rangle, c_x\rangle \Rightarrow \texttt{return}\ \texttt{succ}(c_x)\\ &= \texttt{let}\ \texttt{pack}(\rho, \langle y, a\rangle) = \texttt{pack}(\texttt{unit}, \langle\star, \texttt{fn}\ \langle\langle\rangle, c_f\rangle \Rightarrow 1\rangle)\ \texttt{in}\\ &\quad \texttt{pack}(\rho, \langle y, \texttt{fn}\ \langle\rangle \Rightarrow a(\langle\langle\rangle, \langle\rangle\rangle)\rangle)\\ &= \texttt{pack}(\texttt{unit}, \langle\star, \texttt{fn}\ \langle\rangle \Rightarrow 1\rangle)\end{aligned}$$

While we have used the equational theory here to illustrate that the translation produces the right result, we emphasise that computation is not intended to be implemented by rewriting on target terms, but by a compilation of target programs to low-level programs. Rewriting may be used for optimisation, of course.

To illustrate that the translation correctly implements the call-by-value evaluation strategy, consider a function that takes a function as an argument and applies it more than once:

$$\texttt{fn}\ s \Rightarrow \texttt{if0}\ (s\ \texttt{zero})\ \texttt{then}\ (s\ \texttt{zero})\ \texttt{else}\ \texttt{zero}$$

The translation of the body of this function can be calculated as

$$\begin{aligned}&(\!|s{:}\mathbb{N} \to \mathbb{N} \vdash \texttt{if0}\ (s\ \texttt{zero})\ \texttt{then}\ (s\ \texttt{zero})\ \texttt{else}\ \texttt{zero}\colon \mathbb{N}|\!)\\ &= \Lambda\alpha.\, \lambda s\colon \mathcal{I}[\![\mathbb{N} \to \mathbb{N}]\!]_\alpha.\, \texttt{let}\ \texttt{pack}(\rho, \langle y, a\rangle) = s\ \texttt{unit}\ \star\ \texttt{in}\\ &\qquad\qquad \texttt{pack}(\texttt{unit}, \langle\star, \texttt{fn}\ \langle\langle\rangle, c_s\rangle\rangle \Rightarrow e\rangle)\ ,\end{aligned}$$

where e is $\texttt{let}\ x = a(c_s, \texttt{zero})\ \texttt{in}\ \texttt{if0}\ x\ \texttt{then}\ a(c_s, \texttt{zero})\ \texttt{else}\ \texttt{return}\ 0$. Thus, the function value c_s is computed only once, but the function itself is invoked twice, as would be expected from an implementation of call-by-value.

5 Correctness

In this section we show the correctness of the translation. The main result is that a closed source term of base type $\mathbb{N}$ evaluates to a value n if and only if the translated term for this term returns the code value n.

The basic idea for the correctness proof is to formalise the intuition when a target term of type $\mathcal{M}[\![X]\!]_{\mathtt{unit}}$ represents a source term of type X, as outlined in Section 4 above. This naturally leads one to considering a realisability argument using logical relations. In the simply-typed case, soundness can be shown using a straightforward argument. With recursion, however, we must account for the effect of non-termination. This naturally leads to an argument using $^{\top\top}$-lifting. It turns out that an argument very similar to that of Benton and Hur [2] suffices to treat our translation.

5.1 Lower Bound

First we show that translated terms do not return wrong results.

In Section 4 we have outlined how $c\colon \mathcal{C}[\![X]\!]_A$ and $a\colon \mathcal{I}[\![X]\!]_A$ represent a source value of type X. Here we make precise in which sense triples (A, a, e) represent source values (or rather their domain-theoretic denotation). We extend this definition to computations by means of a form of $^{\top\top}$-lifting. To define it, we need some notation.

Define a type of continuations for type X as follows.

$$\mathcal{K}[\![X]\!] \;:=\; \forall\alpha.\, \mathcal{I}[\![X]\!]_\alpha \to \mathtt{exp}(\mathcal{C}[\![X]\!]_\alpha, \mathtt{unit})$$

A continuation $k\colon \mathcal{K}[\![X]\!]$ can be composed with any term of type $p\colon \mathcal{M}[\![X]\!]_{\mathtt{unit}}$ to give us a term $p \bullet k\colon \mathtt{exp}(\mathtt{unit})$.

$$p \bullet k \;:=\; \mathtt{let}\ \mathtt{pack}(A, \langle a, e\rangle) = p\ \mathtt{in}\ \mathtt{let}\ c = e(\langle\rangle)\ \mathtt{in}\ (k\ A\ a)(c)$$

We say that a closed term $t\colon \mathtt{exp}(\mathtt{unit})$ *terminates* if the equation $t = \mathtt{return}\ \langle\rangle$ is derivable and that it *diverges* otherwise.

$$\begin{aligned}
\downarrow^X V \;&:=\; \{k\colon \mathcal{K}[\![X]\!] \mid \forall (A, a, c) \in V.\ k\ A\ a\ c \text{ diverges}\} \\
\Uparrow^X K \;&:=\; \{p\colon \mathcal{M}[\![X]\!]_{\mathtt{unit}} \mid \forall k \in K.\ p \bullet k \text{ diverges }\}
\end{aligned}$$

The reader should think of $\Uparrow^X\downarrow^X V$ as the set of all terms that produce only values – encoded as a triple (A, a, c) – that are indistinguishable from one in V. It would be possible to make an intensional definition to this effect, but the extensional definition using $\Uparrow^X$ and $\downarrow^X$ appears to be more natural and general.

For any source type X, we define a relation

$$(A, a, c) \leq^X d$$

where A is a base type and $\vdash a\colon \mathcal{I}[\![X]\!]_A$ and $\vdash c\colon \mathcal{C}[\![X]\!]_A$ and $d \in [\![X]\!]$ (recall that $[\![X]\!]$ is the denotational interpretation of X). This relation expresses that the triple (A, a, c) implements only behaviour that is also found in d. It is defined by induction on the type X:

– Base type: $v \leq^{\mathbb{N}} n$ if and only if v is of the form $(A, \star, n)$ for some A.
– Function type: $v \leq^{X \to Y} f$ if and only if:

$$\forall w, x.\ \ w \leq^X x \implies \mathtt{app}(v, w) \in \Uparrow^Y \downarrow^Y \{r \mid \exists d.\, f(x) = \lfloor d \rfloor \wedge r \leq^Y d\}.$$

This approximation relation $\leq^X$ is extended to the interpretation of typing sequents. For any source context $\Gamma = x_1{:}X_1, \ldots, x_n{:}X_n$, the relation $p \leq^{\Gamma, X} d$ relates a term

$$p\colon \forall \alpha_1 \ldots \alpha_k.\, \mathcal{I}[\![X_1]\!]_{\alpha_1} \to \cdots \to \mathcal{I}[\![X_k]\!]_{\alpha_k} \to \mathcal{M}[\![Y]\!]_{\mathtt{unit} \times \mathcal{C}[\![X_1]\!]_{\alpha_1} \times \cdots \times \mathcal{C}[\![X_k]\!]_{\alpha_k}}$$

to a domain element $d \in [\![X_1]\!] \times \cdots \times [\![X_n]\!] \Rightarrow [\![X]\!]_\perp$. It is defined such that $p \leq^{\Gamma, Y} f$ holds if and only if: Whenever $v_i \leq^{X_i} x_i$ for $i = 1, \ldots, n$, then

$$\mathtt{inst}(p, v_1, \ldots, v_n) \in \Uparrow^Y \downarrow^Y \{v \mid \exists d.\, f(x_1, \ldots, x_n) = \lfloor d \rfloor \wedge v \leq^Y d\}.$$

Here, `inst` is defined by:

$$\begin{aligned}
&\mathtt{inst}(p, (A_1, a_1, c_1), \ldots, (A_n, a_n, c_n)) \\
&\quad := \mathtt{let}\ \mathtt{pack}(\alpha_y, \langle a_y, e_y \rangle) = p\ A_1 \ldots A_n\ a_1 \ldots a_n\ \mathtt{in} \\
&\qquad \mathtt{pack}(\alpha_y, a_y, \mathtt{fn}\ \langle\rangle \Rightarrow e_y\ \langle \langle\rangle, c_1, \ldots, c_n \rangle)
\end{aligned}$$

Lemma 1. *If $v \leq^X d$ and $d \sqsubseteq e$, then $v \leq^X e$.*

The proof is a straightforward induction on the type X.

Lemma 2. *For any derivation Π of $\Gamma \vdash t\colon X$, we have $(\!|\Pi|\!) \leq^{\Gamma, X} [\![\Pi]\!]$.*

Proof. The proof goes by induction on the derivation Π. We continue by case distinction on the last rule in Π and show just the cases for application and recursion.

– Case (APP): Denote the derivations of the two premises by Π_s and Π_t.
Suppose Γ is $x_1{:}X_1, \ldots, x_n{:}X_n$ and Δ is $y_1{:}Y_1, \ldots, y_m{:}Y_m$. Assume $v_i \leq^{X_i} x_i$ and $w_j \leq^{Y_j} y_j$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. We have to show the relation $\mathtt{app}((\!|\Pi|\!), \vec{v}, \vec{w}) \leq^Y [\![s]\!](\vec{x}, [\![t]\!](\vec{y}))$.
By induction hypothesis, we get $\mathtt{app}((\!|\Pi_s|\!), \vec{v}) \in \Uparrow^{X \to Y} \downarrow^{X \to Y} \{u \mid u \leq^{X \to Y} [\![s]\!](\vec{x})\}$ and $\mathtt{app}((\!|\Pi_t|\!), \vec{w}) \in \Uparrow^X \downarrow^X \{r \mid v \leq^X [\![t]\!](\vec{y})\}$. To show $\mathtt{app}((\!|\Pi_s|\!), \vec{v}) \bullet k$ diverges, it therefore suffices to show that $k\ A_u\ a_u\ c_u$ diverges for all A_u, a_u and c_u with $(A_u, a_u, c_u) \leq^{X \to Y} [\![s]\!](\vec{x})$, and likewise for Π_t.
This means that to show that

$$\begin{aligned}
&\mathtt{let}\ \mathtt{pack}(A_u, \langle a_u, e_u \rangle) = \mathtt{inst}((\!|\Pi_s|\!), \vec{v})\ \mathtt{in} \\
&\mathtt{let}\ c_u = e_u()\ \mathtt{in} \\
&\mathtt{let}\ \mathtt{pack}(A_r, \langle a_r, e_r \rangle) = \mathtt{inst}((\!|\Pi_t|\!), \vec{w})\ \mathtt{in} \\
&\mathtt{let}\ c_r = e_r()\ \mathtt{in} \\
&\mathtt{app}((A_u, a_u, c_u), (A_r, a_r, c_r)) \bullet k
\end{aligned}$$

diverges, it suffices to show that $\mathtt{app}((A_u, a_u, c_u), (A_r, a_r, c_r)) \bullet k$ diverges for any $(A_u, a_u, c_u) \leq^{X \to Y} [\![s]\!](\vec{x})$ and any $(A_r, a_r, c_r) \leq^X [\![t]\!](\vec{y})$. The definition of $\leq^{X \to Y}$ gives us that this is true when k is in $\downarrow^Y \{y \mid y \leq^Y [\![s]\!](\vec{x}, [\![t]\!](\vec{y}))\}$.
But this means that to show $\mathtt{app}((\!|\Pi|\!), \vec{v}, \vec{w}) \in \Uparrow^Y \downarrow^Y \{y \mid y \leq^Y [\![s]\!](\vec{x}, [\![t]\!](\vec{y}))\}$, it suffices to show that $\mathtt{app}((\!|\Pi|\!), \vec{v}, \vec{w})$ is equal to the above program. But this follows by unfolding the definitions and direct equational reasoning.

– Case (REC): Write Π_t for the derivation of the premise of this rule. Suppose Γ is $x_1{:}X_1, \ldots, x_n{:}X_n$. Assume $(A_i, a_i, c_i) \leq^{X_i} x_i$ for $i = 1, \ldots, n$.
Let $f_0 := \bot$ and $f_i := [\![\Pi_t]\!](\vec{x}, f_{i-1})$ and $f := \bigsqcup_i f_i$.
We have to show that

$$\begin{aligned}&\texttt{let pack}(A, \langle a, e\rangle) = (\!|\Pi|\!)\ A_1\ \ldots\ A_n\ a_1\ \ldots\ a_n\ \texttt{in}\\ &\texttt{pack}(A, a, \texttt{fn}\ \langle\rangle \Rightarrow e(\langle\langle\rangle, c_1, \ldots, c_n\rangle))\end{aligned}$$

is in $\Uparrow^{X\to Y}\downarrow^{X\to Y}\{v \mid v \leq^{X\to Y} f\}$.
By definition of $(\!|\Pi|\!)$, the above program equals

$$\texttt{pack}(\mathcal{C}[\![\Gamma]\!], a_\Pi, \texttt{fn}\ \langle\rangle \Rightarrow (\langle\langle\rangle, c_1, \ldots, c_n\rangle)).$$

It therefore suffices to show $(\mathcal{C}[\![\Gamma]\!], a_\Pi, \langle\langle\rangle, c_1, \ldots, c_n\rangle) \leq^{X\to Y} f$. By definition, a_Π has the form $\texttt{fix}(step)$. Let $b_i := \texttt{fix}^i(step)$.
First we show $\forall i.\ (\mathcal{C}[\![\Gamma]\!], b_i, \langle\langle\rangle, c_1, \ldots, c_n\rangle) \leq^{X\to Y} f_i$ by a straightforward induction on i. This implies $\forall i.\ (\mathcal{C}[\![\Gamma]\!], b_i, \langle\langle\rangle, c_1, \ldots, c_n\rangle) \leq^{X\to Y} f$ by monotonicity.
From this we can conclude $(\mathcal{C}[\![\Gamma]\!], a_\Pi, \texttt{fn}\ \langle\rangle \Rightarrow \langle\langle\rangle, c_1, \ldots, c_n\rangle) \leq^{X\to Y} f$ as follows. Suppose $(A_x, a_x, c_x) \leq^X x$. We have to show that

$$\begin{aligned}&\texttt{let pack}(A_y, \langle a_y, e_y\rangle) = a_\Pi\ A_x\ a_x\ \texttt{in}\\ &\texttt{pack}(A_y, a_y, \texttt{fn}\ \langle\rangle \Rightarrow e_y(\langle\langle\rangle, c_1, \ldots, c_n, c_x\rangle))\end{aligned}$$

is in $\Uparrow^Y\downarrow^Y \{v \mid \exists d.\ f(x) = [d] \wedge v \leq^Y d\}$. For this, we have to show that

$$\begin{aligned}&\texttt{let pack}(A_y, \langle a_y, e_y\rangle) = a_\Pi\ A_x\ a_x\ \texttt{in}\\ &\texttt{let}\ c_y = e_y(\langle\langle\rangle, c_1, \ldots, c_n, c_x\rangle)\ \texttt{in}\\ &k\ A_y\ a_y\ c_y\end{aligned}$$

diverges, given that k diverges for all inputs v with $\exists d.\ f(x) = [d] \wedge v \leq^Y d$. But we know that, for any i, the program with b_i in place of a_Π diverges. Thus, if the program were to terminate, then it would already terminate with an approximation of `fix`. But this would also mean that, for some i, the program with b_i in place of a_Π were to terminate, leading to a contradiction.

5.2 Upper Bound

It now remains to show that the translation produces target terms that compute at least the information specified by the domain interpretation. To this end we define a relation $(A, a, c) \geq^X d$ expressing that the triple (A, a, c) implements at least the behaviour specified by d. The proof that the translation of each source program is $\geq$-related to the domain-theoretic interpretation of the program also follows [2].

As in [2], some care needs to be taken with recursion, in order to make the definition of $\geq$ closed under taking the least upper bound in its second argument. We therefore first define a relation $\geq_0$ and then define $\geq$ from it by closing under limits.

$$\underline{\downarrow^X} V = \{k\colon \mathcal{K}[\![X]\!] \mid \forall (A, a, c) \in V.\ k\ A\ a\ c \text{ terminates}\}$$
$$\overline{\Uparrow^X} K = \{p\colon \mathcal{M}[\![X]\!]_{\texttt{unit}} \mid \forall k \in K.\ \texttt{let}\ (A, \langle a, c\rangle) = p\ \texttt{in}\ k\ A\ a\ c \text{ terminates}\}$$

Define $\geq_0^X$ between triples (A, a, c) with $a\colon \mathcal{I}[\![X]\!]_A$ and $c\colon \mathcal{C}[\![X]\!]_A$, and elements of $[\![X]\!]$ to be the least relation with the following properties:

- Base type: $(A, \star, n) \geq_0^{\mathbb{N}} n$
- Function type: $v \geq_0^{X\to Y} f$ if and only if:

$$\forall w, x, d.\ w \geq_0^X x \ \wedge\ f(x) = \lfloor d \rfloor \implies \mathtt{app}(v, w) \in \overline{\Uparrow^Y}\underline{\downarrow^Y}\{v \mid v \geq_0^Y d\}$$

For a source context $\Gamma = x_1{:}X_1, \ldots, x_n{:}X_n$, we define $p \geq_0^{\Gamma,Y} f$ to mean: Whenever $w_i \geq_0^{X_i} x_i$ for $i = 1, \ldots, n$ and $f(x_1, \ldots, x_n) = \lfloor d \rfloor$, then also $\mathtt{inst}(p, w_1, \ldots, w_n) \in \overline{\Uparrow^Y}\underline{\downarrow^Y}\{v \mid v \geq_0^Y d\}$.

This definition is then closed under least upper bounds, as in [2]:

$$v \geq^X x \iff \text{for some } \omega\text{-chain } (x_i)_{i\geq 0}\colon\ x \sqsubseteq \bigsqcup_i x_i \ \wedge\ \forall i.\, v \geq_0^X x_i$$

$$v \geq^{\Gamma,X} x \iff \text{for some } \omega\text{-chain } (x_i)_{i\geq 0}\colon\ x \sqsubseteq \bigsqcup_i x_i \ \wedge\ \forall i.\, v \geq_0^{\Gamma,X} x_i$$

Lemma 3. *For any derivation Π of $\Gamma \vdash t\colon X$, we have $(\!|\Pi|\!) \geq^{\Gamma,X} [\![\Pi]\!]$.*

Proof. With the explicit closure under limits, the case for recursion is straightforward this time, but we have to check that all operations are compatible with closing under limits. For example, in the case of application we use continuity of function application, i.e. $\bigsqcup_i f_i(\bigsqcup_i x_i) = \bigsqcup_i f_i(x_i)$. Using this equality it suffices to show that $v \geq_0^{X\to Y} f_i$ and $w \geq_0^X x_i$ implies $\mathtt{app}(v, w) \geq_0^Y f_i(x_i)$ for all i, which follows using the same kind of reasoning as for the lower bound. □

Theorem 1. *Suppose Π derives $\vdash t\colon \mathbb{N}$. Define the target term $p\colon \mathtt{exp}(\mathtt{nat})$ to be $\mathtt{let}\ \mathtt{pack}(\alpha, \langle a, e\rangle) = (\!|\Pi|\!)\ \mathtt{in}\ e(\langle\rangle)$. Then the following are true.*

1. *If $[\![\Pi]\!] = \bot$, then p diverges.*
2. *If $[\![\Pi]\!] = \lfloor n \rfloor$, then $\vdash p = \mathtt{return}\, n\colon \mathbb{N}$.*

Proof. If $[\![\Pi]\!] = \bot$, then $(\!|\Pi|\!) \in \Uparrow^{\mathbb{N}}\downarrow^{\mathbb{N}}\emptyset$ by $(\!|\Pi|\!) \leq^{\mathtt{empty},\mathbb{N}} [\![\Pi]\!]$. This means that $(\!|\Pi|\!) \bullet k$ must diverge for any k, in particular also $\Lambda\alpha.\, \lambda x.\, \mathtt{fn}\ m \Rightarrow \mathtt{return}\ \langle\rangle$. Hence, $(\!|\Pi|\!)$ cannot be equal to $\mathtt{return}\ m$ for any m.

By $(\!|\Pi|\!) \geq^{\mathtt{empty},\mathbb{N}} [\![\Pi]\!]$ and the fact that $\mathbb{N}_\bot$ is a flat cpo, we obtain that $(\!|\Pi|\!)$ is in $\overline{\Uparrow^{\mathbb{N}}}\underline{\downarrow^{\mathbb{N}}}\{v \mid v \geq_0^{\mathbb{N}} n\}$. This means that $(\!|\Pi|\!) \bullet k$ must terminate for the continuation defined by the pseudo-code $\Lambda\alpha.\, \lambda x.\, \mathtt{fn}\ m \Rightarrow \mathtt{if}\ m = n\ \mathtt{then}\ \mathtt{return}\ \langle\rangle\ \mathtt{else}$ diverge. But this can only be the case if $(\!|\Pi|\!)$ returns n. □

6 Conclusion and Further Work

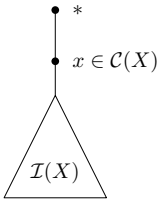
We have defined a simple typed closure conversion from a call-by-value source language to an Algol-like target language. The target language is modelled on the structure of interactive models of computation. Overall, we have there arrived at a simple

implementation of call-by-value in models of interactive computation, simplifying earlier work in this direction, in particular [17]. Moreover, the translation in this paper also covers recursion. The formulation should be general enough to account for effects other than non-termination as well.

While the translation in this paper is similar to other typed closure conversion methods [13,1], it does not appear to be exactly the same. In addition to corresponding to the CPS-translation from [17], we believe that the translation here is also canonical in the sense that it is an implementation of call-by-value game semantics [9]. This was observed in discussions with Nikos Tzevelekos.

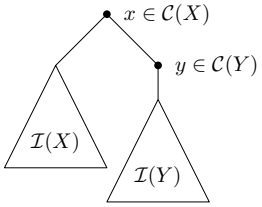
Let us consider the definition of arenas in call-by-value games [9]. Arenas are forests of labelled trees (with additional information) that explain which sequences of moves are allowed as plays. For a source type X, define the labelled forest $\mathcal{M}(X)$ as shown below, where $\mathcal{C}(\mathbb{N})$ is the set of natural numbers and $\mathcal{C}(X \to Y) = \{*\}$.

$\mathcal{M}(X)$ is forest of all trees of the form



$\mathcal{I}(\mathbb{N})$ is the empty forest

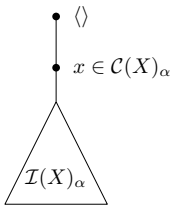
$\mathcal{I}(X \to Y)$ is forest of all trees of the form



In call-by-value games, the semantics of a closed PCF term of type X is explained in terms of plays on $\mathcal{M}(X)$. Such a play is a sequences of nodes $m_1, \ldots, m_n$ with the property (among others) that, for all $i > 1$ there exists $j < i$ such that m_j is the parent node of m_i in the forest $\mathcal{M}(X)$.

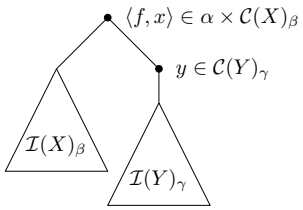
Now, consider an implementation of the target language in this paper which records the trace of values that terms of type $\mathtt{exp}(A, B)$ receive as arguments and that they return. The types $\mathcal{I}[\![X]\!]_A$ and $\mathcal{M}[\![X]\!]_A$ specify an ordering on the traces that can appear at these types. Consider $\mathcal{I}[\![(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}]\!]_A$, i.e. $\forall\beta_1.\,(\forall\beta_2.\,I \times \mathtt{exp}(\beta_1 \times \mathtt{nat}, \mathtt{nat})) \to \exists\gamma.\,I \times \mathtt{exp}(A \times \beta_1, \mathtt{nat})$. A trace of a term of this type must start with a value of type $A \times \beta_1$ (invoking the function). This can be followed by a message of type $\beta_1 \times \mathtt{nat}$ (invoking the argument function) and then the corresponding return value of type $\mathtt{nat}$. Such a call to the argument function can be repeated until the final value of type $\mathtt{nat}$ is returned. If we spell out the constraints on the traces in terms of forests as above, then we get the following forests, in which type variables range over arbitrary types.

$\mathcal{M}(X)_{\mathtt{unit}}$ is forest of all trees of the form



$\mathcal{I}(\mathbb{N})_\alpha$ is the empty forest

$\mathcal{I}(X \to Y)_\alpha$ is forest of all trees of the form



The trees from the game semantic arenas are obtained from these trees by removing all values whose type is a type variable. In game semantic plays, this removed information is not needed, as it can be recovered from the history. This makes it reasonable to expect that the translation in this paper can be seen as an efficient, *history-free* implementation of call-by-value game semantics. We intend to make the relation precise in further work.

References

1. Ahmed, A., Blume, M.: Typed closure conversion preserves observational equivalence. In: International conference on Functional programming, ICFP 2008. (2008) 157–168
2. Benton, N., Hur, C.: Biorthogonality, step-indexing and compiler correctness. In: International Conference on Functional programming, ICFP 2009. (2009) 97–108
3. Dal Lago, U., Schöpp, U.: Functional programming in sublinear space. In Gordon, A.D., ed.: European Symposium on Programming, ESOP 2010. (2010) 205–225
4. Danos, V., Herbelin, H., Regnier, L.: Game semantics and abstract machines. In: Logic in Computer Science, LICS 1996, IEEE (1996) 394–405
5. Fernández, M., Mackie, I.: Call-by-value lambda-graph rewriting without rewriting. In: International Conference on Graph Transformation, ICGT 2002. (2002) 75–89
6. Fredriksson, O., Ghica, D.R.: Seamless distributed computing from the geometry of interaction. In Palamidessi, C., Ryan, M.D., eds.: Trustworthy Global Computing, TGC 2012. Volume 8191 of Lecture Notes in Computer Science., Springer (2012) 34–48
7. Ghica, D.R.: Geometry of synthesis: a structured approach to VLSI design. In Hofmann, M., Felleisen, M., eds.: Principles of Programming Languages, POPL 2007, ACM (2007) 363–375
8. Hasuo, I., Hoshino, N.: Semantics of higher-order quantum computation via geometry of interaction. In: Logic in Computer Science, LICS 2011, IEEE (2011) 237–246
9. Honda, K., Yoshida, N.: Game-theoretic analysis of call-by-value computation. Theor. Comput. Sci. **221**(1-2) (1999) 393–456
10. Hoshino, N., Muroya, K., Hasuo, I.: Memoryful geometry of interaction: from coalgebraic components to algebraic effects. In: Computer Science Logic – Logic in Computer Science, CSL-LICS 2014, ACM (2014)
11. Lago, U.D., Faggian, C., Hasuo, I., Yoshimizu, A.: The geometry of synchronization. In: Computer Science Logic – Logic in Computer Science, CSL-LICS 2014. (2014) 35:1–35:10
12. Mackie, I.: The geometry of interaction machine. In Cytron, R.K., Lee, P., eds.: Principles of Programming Languages, POPL 1995, ACM (1995) 198–208
13. Minamide, Y., Morrisett, J.G., Harper, R.: Typed closure conversion. In Boehm, H., Jr., G.L.S., eds.: Principles of Programming Languages, POPL 1996, ACM (1996) 271–283
14. Reynolds, J.C.: The essence of ALGOL. In O'Hearn, P.W., Tennent, R.D., eds.: ALGOL-like Languages, Volume 1. Birkhauser Boston Inc., Cambridge, MA, USA (1997) 67–88
15. Rossberg, A., Russo, C.V., Dreyer, D.: F-ing modules. J. Funct. Program. **24**(5) (2014) 529–607
16. Schöpp, U.: Computation-by-Interaction for Structuring Low-Level Computation. Habilitation thesis, Ludwig-Maximilians-Universität München (2015)
17. Schöpp, U.: Call-by-value in a basic logic for interaction. In Garrigue, J., ed.: Asian Symposium on Programming Languages and Systems, APLAS 2014. Volume 8858 of Lecture Notes in Computer Science., Springer (2014) 428–448
18. Schöpp, U.: Organising low-level programs using higher types. In: Principles and Practice of Declarative Programming, PPDP 2014, New York, NY, ACM (2014) to appear.
19. Winskel, G.: The formal semantics of programming languages - an introduction. Foundation of computing series. MIT Press (1993)

A Detailed Proofs

Lemma 2. *For any derivation Π of $\Gamma \vdash t\colon X$, we have $(\!|\Pi|\!) \leq^{\Gamma,X} [\![\Pi]\!]$.*

Proof. The proof goes by induction on the derivation Π. We continue by case distinction on the last rule in Π.

– Case (VAR):

$$\textsc{var}\ \frac{}{x{:}X \vdash x\colon X}$$

Suppose $(A_X, a_x, c_x) \leq^X x$. We have to show that

$$\begin{aligned}&\texttt{let pack}(\alpha_y, \langle a_y, c_y\rangle) = (\!|\Pi|\!)\ A_x\ a_x\ \texttt{in}\\ &\texttt{pack}(\alpha_y, \langle a_y, \texttt{fn}\ \langle\rangle \Rightarrow e_y(\langle\langle\rangle, c_x\rangle)\rangle)\end{aligned}$$

is in $\Uparrow^X\!\downarrow^X\{v \mid v \leq^X x\}$. The term is easily seen to be equal to

$$\texttt{pack}(\alpha_x, \langle a_x, \texttt{fn}\ \langle\rangle \Rightarrow c_x\rangle)\ .$$

If we write p for this term, then $p \bullet k$ is therefore equal to $k\ A_x\ a_x\ c_x$, from which the assertion follows.

– We omit the structural rules, which all follow directly from the induction hypothesis.

– Case (ZERO): This case is similar to that for (VAR).

– Case (SUCC):

$$\textsc{succ}\ \frac{\Gamma \vdash t\colon \mathbb{N}}{\Gamma \vdash \texttt{succ}(t)\colon \mathbb{N}}$$

Denote the derivation of the premise by Π_t. Suppose Γ is $x_1{:}X_1, \ldots, x_n{:}X_n$. Assume $(A_i, a_i, c_i) \leq^{X_i} x_i$ for $i = 1, \ldots, n$. Let p be the following term.

$$\begin{aligned}&\texttt{let pack}(\alpha_y, \langle a_y, e_y\rangle) = (\!|\Pi|\!)\ A_1\ \ldots\ A_n\ \texttt{in}\\ &\texttt{pack}(\alpha_y, a_y, \texttt{fn}\ \langle\rangle \Rightarrow e_y(\langle\langle\rangle, c_1, \ldots, c_n\rangle))\end{aligned}$$

We have to show $p \in \Uparrow^{\mathbb{N}}\!\downarrow^{\mathbb{N}}\{v \mid v \leq^{\mathbb{N}} [\![t]\!](\vec{x}) + 1\}$. Suppose k is such that $k\ A\ a\ ([\![t]\!](\vec{x}) + 1)$ diverges for any A and a. We have to show that that $p \bullet k$ diverges. Define k' to be

$$\Lambda\alpha.\, \lambda a\colon \mathcal{I}[\![\mathbb{N}]\!]_\alpha,\ \lambda c.\, \texttt{let}\ c' = \texttt{succ}(c)\ \texttt{in}\ k\ \alpha\ a\ c'$$

By definition, $k\ A\ a\ [\![t]\!](\vec{x})$ must diverge. By assumption we know that the program p' defined by

$$\begin{aligned}&\texttt{let pack}(\alpha_y, \langle a_y, e_y\rangle) = (\!|\Pi_t|\!)\ A_1\ \ldots\ A_n\ \texttt{in}\\ &\texttt{pack}(\alpha_y, a_y, \texttt{fn}\ \langle\rangle \Rightarrow e_y(\langle\langle\rangle, c_1, \ldots, c_n\rangle))\end{aligned}$$

is in $\Uparrow^{\mathbb{N}}\downarrow^{\mathbb{N}}\{v \mid v \leq^{\mathbb{N}} [\![t]\!](\vec{x})\}$. Hence, $p' \bullet k'$ diverges. But we note that $p \bullet k$ simplifies to:

$$\begin{aligned}&\texttt{let pack}(\alpha_y, \langle a_y, e_y\rangle) = (\!|\Pi|\!)\ A_1\ \ldots\ A_n\ \texttt{in}\\&\texttt{let } c = e_y(\langle\langle\rangle, c_1, \ldots, c_n\rangle)\ \texttt{in}\\&k\ \alpha_y\ a_y\ c\end{aligned}$$

By expanding $(\!|\Pi|\!)$, this equals:

$$\begin{aligned}&\texttt{let pack}(\alpha_y, \langle a_y, e_y\rangle) = (\!|\Pi_t|\!)\ A_1\ \ldots\ A_n\ \texttt{in}\\&\texttt{let } c = \texttt{let } y = e_y(\langle\langle\rangle, c_1, \ldots, c_n\rangle)\ \texttt{in succ}(y)\ \texttt{in}\\&k\ \alpha_y\ a_y\ c\end{aligned}$$

This equals:

$$\begin{aligned}&\texttt{let pack}(\alpha_y, \langle a_y, e_y\rangle) = (\!|\Pi_t|\!)\ A_1\ \ldots\ A_n\ \texttt{in}\\&\texttt{let } y = e_y(\langle\langle\rangle, c_1, \ldots, c_n\rangle)\ \texttt{in}\\&\texttt{let } c = \texttt{succ}(y)\ \texttt{in}\\&k\ \alpha_y\ a_y\ c\end{aligned}$$

But this is the same as $p' \bullet k'$, which shows that $p \bullet k$ diverges, as was required to show.

– Case (IF): analogously to case (SUCC).
– Case (ABS):

$$\textsc{abs}\ \frac{\Gamma,\ x{:}X \vdash t{:}\ Y}{\Gamma \vdash \texttt{fn}\ x{:}X \Rightarrow t{:}\ X \to Y}$$

Denote the derivation of the premise by Π_t. Suppose Γ is $x_1{:}X_1, \ldots, x_n{:}X_n$. Assume $(A_i, a_i, c_i) \leq^{X_i} x_i$ for $i = 1, \ldots, n$. Let p be the following term.

$$\texttt{pack}(\mathcal{C}[\![\Gamma]\!]_{\vec{\alpha}}, \langle \Lambda\beta.\, \lambda x{:}\mathcal{C}[\![X]\!]_\beta.\, (\!|\Pi_t|\!)\ \vec{A}\ \beta\ \vec{a}\ x,\ \texttt{fn}\ \langle\rangle \Rightarrow \texttt{return}\ \langle\langle\rangle, c_1, \ldots, c_n\rangle\rangle)$$

We have to show $p \in \Uparrow^{\mathbb{N}}\downarrow^{\mathbb{N}}\{v \mid v \leq^{X\to Y} \lambda x.\, [\![t]\!](\vec{x}, x)\}$.
Suppose $k \in \downarrow^{\mathbb{N}}\{v \mid v \leq^{X\to Y} \lambda x.\, [\![t]\!](\vec{x}, x)\}$. We have

$$p \bullet k = k\ \mathcal{C}[\![\Gamma]\!]_{\vec{\alpha}}\ (\Lambda\beta.\, \lambda x{:}\mathcal{C}[\![X]\!]_\beta.\, (\!|\Pi_t|\!)\ \vec{A}\ \beta\ \vec{a}\ x)\ \langle\langle\rangle, c_1, \ldots, c_n\rangle$$

by β-reduction and have to show that $p \bullet k$ diverges. By assumption on k, it suffices to show:

$$(\mathcal{C}[\![\Gamma]\!]_{\vec{\alpha}}, \Lambda\beta.\, \lambda x{:}\mathcal{C}[\![X]\!]_\beta.\, (\!|\Pi_t|\!)\ \vec{A}\ \beta\ \vec{a}\ x, \langle\langle\rangle, c_1, \ldots, c_n\rangle) \leq^{X\to Y} \lambda x.\, [\![t]\!](\vec{x}, x)$$

To this end, suppose $(A, a, c) \leq^X x$. We must show that

$$\begin{aligned}&\texttt{let pack}(\alpha, \langle a', e\rangle) = (\!|\Pi_t|\!)\ \vec{A}\ A\ \vec{a}\ a\ \texttt{in}\\&\texttt{pack}(\alpha, \langle a', \texttt{fn}\ \langle\rangle \Rightarrow e(\langle\langle\langle\rangle, c_1, \ldots, c_n\rangle, c\rangle))\end{aligned}$$

is in $\Uparrow^Y\downarrow^Y\{v \mid v \leq^Y [\![t]\!](\vec{x}, x)\}$. But this is exactly what the induction hypothesis gives us.

– Case (APP):

$$\textsc{app}\ \frac{\Gamma \vdash s\colon X \to Y \qquad \Delta \vdash t\colon X}{\Gamma, \Delta \vdash s\ t\colon Y}$$

Denote the derivations of the two premises by Π_s and Π_t.
Suppose Γ is $x_1{:}X_1, \ldots, x_n{:}X_n$ and Δ is $y_1{:}Y_1, \ldots, y_m{:}Y_m$. Assume $v_i \leq^{X_i} x_i$ and $w_j \leq^{Y_j} y_j$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. We have to show the relation $\mathtt{app}(\llparenthesis \Pi \rrparenthesis, \vec{v}, \vec{w}) \leq^{Y} [\![s]\!](\vec{x}, [\![t]\!](\vec{y}))$.
The induction hypothesis gives

$$\begin{aligned}
&\mathtt{inst}(\llparenthesis \Pi_s \rrparenthesis, \vec{v}) \in \Uparrow^{X\to Y}\downarrow^{X\to Y} \{u \mid u \leq^{X\to Y} [\![s]\!](\vec{x})\},\\
&\mathtt{inst}(\llparenthesis \Pi_t \rrparenthesis, \vec{w}) \in \Uparrow^{X}\downarrow^{X} \{r \mid v \leq^{X} [\![t]\!](\vec{y})\}.
\end{aligned}$$

To show that $\mathtt{inst}(\llparenthesis \Pi_s \rrparenthesis, \vec{v}) \bullet k$ diverges, it therefore suffices to show that $k\ A_u\ a_u\ c_u$ diverges for any $(A_u, a_u, c_u) \leq^{X\to Y} [\![s]\!](\vec{x})$, and likewise for Π_t.
This means that to show that

$$\begin{aligned}
&\mathtt{let}\ \mathtt{pack}(A_u, \langle a_u, e_u\rangle) = \mathtt{inst}(\llparenthesis \Pi_s \rrparenthesis, \vec{v})\ \mathtt{in}\\
&\mathtt{let}\ c_u = e_u()\ \mathtt{in}\\
&\mathtt{let}\ \mathtt{pack}(A_r, \langle a_r, e_r\rangle) = \mathtt{inst}(\llparenthesis \Pi_t \rrparenthesis, \vec{w})\ \mathtt{in}\\
&\mathtt{let}\ c_r = e_r()\ \mathtt{in}\\
&\mathtt{app}((A_u, a_u, c_u), (A_r, a_r, c_r)) \bullet k
\end{aligned}$$

diverges, it suffices to show that $k'\ A_u\ a_u\ c_u$ diverges for any $(A_u, a_u, c_u) \leq^{X\to Y} [\![s]\!](\vec{x})$, where

$$\begin{aligned}
k' := \Lambda A_u.\, \lambda a_u.\, \mathtt{fn}\ c_u \Rightarrow\ &\mathtt{let}\ \mathtt{pack}(A_r, \langle a_r, e_r\rangle)) = \mathtt{app}(\llparenthesis \Pi_t \rrparenthesis, \vec{w})\ \mathtt{in}\\
&\mathtt{let}\ c_r = e_r()\ \mathtt{in}\\
&\mathtt{app}((A_u, a_u, c_u), (A_r, a_r, c_r)) \bullet k\ \ .
\end{aligned}$$

But to show that $k'\ A_u\ a_u\ c_u$ diverges, it suffices to show that $k''\ A_r\ a_r\ c_r$ diverges for any $(A_r, a_r, c_r) \leq^{X} [\![t]\!](\vec{y})$, where

$$k'' := \Lambda A_r.\, \lambda a_r.\, \mathtt{fn}\ c_r \Rightarrow \mathtt{app}((A_u, a_u, c_u), (A_r, a_r, c_r)) \bullet k\ \ .$$

The definition of $\leq^{X\to Y}$ gives us that $\mathtt{app}((A_u, a_u, c_u), (A_r, a_r, c_r)) \bullet k$ diverges when k is in $\downarrow^{Y} \{y \mid y \leq^{Y} [\![s]\!](\vec{x}, [\![t]\!](\vec{y}))\}$.
But this means that to show $\mathtt{inst}(\llparenthesis \Pi \rrparenthesis, \vec{v}, \vec{w}) \in \Uparrow^{Y}\downarrow^{Y} \{y \mid y \leq^{Y} [\![s]\!](\vec{x}, [\![t]\!](\vec{y}))\}$, it suffices that $\mathtt{inst}(\llparenthesis \Pi \rrparenthesis, \vec{v}, \vec{w})$ is equal to the above program.
Denote the components of v_i and w_j by $(A_i, a_i, c_i) := v_i$ and $(B_j, b_j, d_j) := w_j$.

By definition, the above program expands to:

$$\begin{array}{l}
\texttt{let pack}(A_u, \langle a_u, e_u\rangle) = (\!|\Pi_s|\!)\ \vec{A}\ \vec{a}\ \texttt{in}\\
\texttt{let } c_u = e_u(\vec{c})\ \texttt{in}\\
\texttt{let pack}(A_r, \langle a_r, e_r\rangle) = (\!|\Pi_t|\!)\ \vec{B}\ \vec{b}\ \texttt{in}\\
\texttt{let } c_r = e_r(\vec{d})\ \texttt{in}\\
\texttt{let pack}(\alpha_y, \langle a_y, e_y\rangle) = a_c\ A_r\ a_r\ \texttt{in}\\
\texttt{let } c_y = e_y(\langle c, c_x\rangle)\ \texttt{in}\\
k\ \alpha_y\ a_y\ c_y
\end{array}$$

Also by definition we have

$$\begin{array}{rcl}
\texttt{inst}((\!|\Pi|\!), \vec{v}, \vec{w}) & = & \texttt{let pack}(\varphi, \langle f, e_f\rangle) = (\!|\Pi_s|\!)\ \vec{A}\ \vec{a}\ \texttt{in}\\
& & \texttt{let pack}(\xi, \langle x, e_x\rangle) = (\!|\Pi_t|\!)\ \vec{B}\ \vec{b}\ \texttt{in}\\
& & \texttt{let pack}(\rho, \langle y, a\rangle) = f\ \xi\ x\ \texttt{in}\\
& & \texttt{pack}(\rho, \langle y, e\rangle)
\end{array}$$

where e is $\texttt{fn}\ \langle\rangle\ \Rightarrow\ \texttt{let}\ v_f = e_f(\vec{c})\ \texttt{in let}\ v_x = e_x(\vec{d})\ \texttt{in}\ a(\langle v_f, v_x\rangle)$. Hence, $\texttt{inst}((\!|\Pi|\!), \vec{v}, \vec{w}) \bullet k'$ is

$$\begin{array}{l}
\texttt{let pack}(\varphi, \langle f, e_f\rangle) = (\!|\Pi_s|\!)\ \vec{A}\ \vec{a}\ \texttt{in}\\
\texttt{let pack}(\xi, \langle x, e_x\rangle) = (\!|\Pi_t|\!)\ \vec{B}\ \vec{b}\ \texttt{in}\\
\texttt{let pack}(\rho, \langle y, a\rangle) = f\ \xi\ x\ \texttt{in}\\
\texttt{let } c_y = (\texttt{let } v_f = e_f(\vec{c})\ \texttt{in let } v_x = e_x(\vec{d})\ \texttt{in}\ a(\langle v_f, v_x\rangle))\ \texttt{in}\\
k'\ \rho\ y\ c_y
\end{array}$$

The result follows by commuting conversion.

– Case (REC):

$$\textsc{rec}\ \frac{\Gamma,\ f{:}X \to Y,\ x{:}X \vdash t{:}\ Y}{\Gamma \vdash \texttt{fun}\ f\ x \Rightarrow t{:}\ X \to Y}$$

Write Π_t for the derivation of the premise of this rule.
Suppose Γ is $x_1{:}X_1, \ldots, x_n{:}X_n$. Assume $(A_i, a_i, c_i) \leq^{X_i} x_i$ for $i = 1, \ldots, n$.
Let $f_0 := \bot$ and $f_i := [\![\Pi_t]\!](\vec{x}, f_{i-1})$ and $f := \bigsqcup_i f_i$.
We have to show that

$$\begin{array}{l}
\texttt{let pack}(A, \langle a, e\rangle) = (\!|\Pi|\!)\ A_1\ \ldots\ A_n\ a_1\ \ldots\ a_n\ \texttt{in}\\
\texttt{pack}(A, a, \texttt{fn}\ \langle\rangle \Rightarrow e(\langle\langle\rangle, c_1, \ldots, c_n\rangle))
\end{array}$$

is in $\Uparrow^{X\to Y}\downarrow^{X\to Y} \{v \mid v \leq^{X\to Y} f\}$.
By definition of $(\!|\Pi|\!)$, the above program equals

$$\texttt{pack}(\mathcal{C}[\![\Gamma]\!], a_\Pi, \texttt{fn}\ \langle\rangle \Rightarrow (\langle\langle\rangle, c_1, \ldots, c_n\rangle)).$$

It therefore suffices to show

$$(\mathcal{C}[\![\Gamma]\!], a_\Pi, \langle\langle\rangle, c_1, \ldots, c_n\rangle) \leq^{X\to Y} f.$$

By definition, a_Π has the form $\texttt{fix}(step)$. Let $b_i := \texttt{fix}^i(step)$.
We will show

$$\forall i.\ (\mathcal{C}[\![\Gamma]\!], b_i, \langle\langle\rangle, c_1, \ldots, c_n\rangle) \leq^{X\to Y} f_i$$

by induction on i. This then implies $\forall i.\ (\mathcal{C}[\![\Gamma]\!], b_i, \langle\langle\rangle, c_1, \ldots, c_n\rangle) \leq^{X\to Y} f$ by monotonicity, as $f_i \sqsubseteq f$.
From this we can conclude $(\mathcal{C}[\![\Gamma]\!], a_\Pi, \texttt{fn}\ \langle\rangle \Rightarrow \langle\langle\rangle, c_1, \ldots, c_n\rangle) \leq^{X\to Y} f$ as follows. Suppose $(A_x, a_x, c_x) \leq^X x$. We have to show that

$$\begin{aligned}&\texttt{let pack}(A_y, \langle a_y, e_y\rangle) = a_\Pi\ A_x\ a_x\ \texttt{in}\\ &\texttt{pack}(A_y, a_y, \texttt{fn}\ \langle\rangle \Rightarrow e_y(\langle\langle\rangle, c_1, \ldots, c_n, c_x\rangle))\end{aligned}$$

is in $\Uparrow^Y\!\downarrow^Y \{v \mid \exists d.\ f(x) = [d] \wedge v \leq^Y d\}$. For this, we have to show that

$$\begin{aligned}&\texttt{let pack}(A_y, \langle a_y, e_y\rangle) = a_\Pi\ A_x\ a_x\ \texttt{in}\\ &\texttt{let}\ c_y = e_y(\langle\langle\rangle, c_1, \ldots, c_n, c_x\rangle)\ \texttt{in}\\ &k\ A_y\ a_y\ c_y\end{aligned}$$

diverges, given that k diverges for all inputs v with $\exists d.\ f(x) = [d] \wedge v \leq^Y d$. But we know that for such inputs, for any i, the program with b_i in place of a_Π diverges. Thus, suppose the program were to terminate. Then it would already terminate with an approximation of `fix`. But this would also mean that, for some i, the program with b_i in place of a_Π were to terminate, leading to a contradiction.
It remains to show

$$(\mathcal{C}[\![\Gamma]\!], b_i, \langle\langle\rangle, c_1, \ldots, c_n\rangle) \leq^{X\to Y} f_i$$

by induction on i. Suppose $(A_x, a_x, c_x) \leq^X x$. We have to show that

$$\texttt{let}\ (A_y, a_y, c_y) = b_i\ A_x\ a_x\ \texttt{in pack}(A_y, a_y, \texttt{fn}\ () \Rightarrow c_y(\langle\langle\langle\rangle, c_1, \ldots, c_n\rangle, c_x\rangle))$$

is in $\Uparrow^Y\!\downarrow^Y \{v \mid \exists d.\ f_i(x) = [d] \wedge v \leq^Y d\}$. We have $b_i = \texttt{fix}^i\ (step)$ where

$$\begin{aligned}step\ =\ \lambda f.\, \Lambda\beta.\, \lambda x.\, &\texttt{let pack}(\rho, \langle a, e\rangle) = (\!|\Pi_t|\!)\ \vec{A}\ \mathcal{C}[\![\Gamma]\!]_{\vec{\alpha}}\ \beta\ \vec{x}\ f\ x\ \texttt{in}\\ &\texttt{pack}(\rho, \langle a, \texttt{fn}\ \langle c, x\rangle \Rightarrow e(\langle\langle c, c\rangle, x\rangle)\rangle)\ .\end{aligned}$$

- Case $i = 0$. By definition, $b_0\ A_x\ a_x$ can never be equal to any term other than one of the form $\texttt{fix}^0(\ldots)$, so the program in question always diverges, and the assertion is trivial.
- Case $i > 0$. By unfolding $\texttt{fix}^{i+1}$ once in $b_{i+1}\ A_x\ a_x$, we get

$$\begin{aligned}b_{i+1}\ A_x\ a_x = \texttt{let pack}(\rho, \langle a, e\rangle) = (\!|\Pi_t|\!)\ \vec{A}\ \mathcal{C}[\![\Gamma]\!]_{\vec{A}}\ \beta\ \vec{x}\ b_i\ x\ \texttt{in}\\ \texttt{pack}(\rho, \langle a, \texttt{fn}\ \langle c, c_x\rangle \Rightarrow e(\langle\langle c, c\rangle, c_x\rangle)\rangle)\end{aligned}$$

The inner induction hypothesis is $(\mathcal{C}[\![\Gamma]\!], b_i, \langle\langle\rangle, c_1, \ldots, c_n\rangle) \leq^{X\to Y} f_i$. The outer induction hypothesis is $(\!|\Pi_t|\!) \leq^{(\Gamma, f:X\to Y, x:X), X\to Y} [\![\Pi_t]\!]$. By plugging the former into the definition of the latter, we get a term in $\Uparrow^Y\downarrow^Y\{y \mid \exists d.\, f_{i+1}(x) = [d] \wedge y \leq^Y d\}$, which is equal to $\mathtt{app}((\mathcal{C}[\![\Gamma]\!], b_{i+1}, \langle\langle\rangle, c_1, \ldots, c_n\rangle), (A_x, a_x, c_x))$, by the above unfolding.

Lemma 3. *For any derivation Π of $\Gamma \vdash t\colon X$, we have $(\!|\Pi|\!) \geq^{\Gamma, X} [\![\Pi]\!]$.*

Proof. The proof goes by induction on the derivation Π. We continue by case distinction on the last rule in Π.

- Case (VAR):

$$\textsc{var}\ \frac{}{x{:}X \vdash x\colon X}$$

  Suppose $(A_X, a_x, c_x) \geq^X x$, i.e. for some ω-chain (x_i) whose limit is above x, we have $(A_X, a_x, c_x) \geq_0^X x_i$ for all i.
  It suffices to show $\mathtt{inst}(\!|\Pi|\!), (A_x, a_x, c_x))$ is in $\overline{\Uparrow^X}\underline{\downarrow^X}\{v \mid v \leq^X x\}$. The term is easily seen to be equal to $\mathtt{pack}(\alpha_x, \langle a_x, \mathtt{fn}\ \langle\rangle \Rightarrow c_x\rangle)$. If we write p for this term, then $p \bullet k$ is therefore equal to $k\ A_x\ a_x\ c_x$. The required property therefore follows directly from the assumption.
- We omit the structural rules, which all follow directly from the induction hypothesis.
- Cases (ZERO), Case (SUCC), Case (IF): Since $\mathbb{N}_\perp$ is a flat cpo, the result follows just as in the corresponding cases for $\leq$.
- Case (ABS):

$$\textsc{abs}\ \frac{\Gamma,\ x{:}X \vdash t\colon Y}{\Gamma \vdash \mathtt{fn}\ x{:}X \Rightarrow t\colon X \to Y}$$

  Denote the derivation of the premise by Π_t.
  The induction hypothesis gives us an ω-chain $(t_i)_{i\geq 0}$ with $[\![\Pi_t]\!] \sqsubseteq \bigsqcup t_i$. Let $f_i(\vec{x}) := \lambda x.\, t_i(\vec{x}, x)$. We have $[\![\Pi]\!] \sqsubseteq \bigsqcup f_i$ by the pointwise definition of $\sqsubseteq$ on functions.
  Suppose Γ is $x_1{:}X_1, \ldots, x_n{:}X_n$. Assume $(A_i, a_i, c_i) \geq_0^{X_i} x_i$ for $i = 1, \ldots, n$. Let p be the following term.

$$\mathtt{pack}(\mathcal{C}[\![\Gamma]\!]_{\vec{\alpha}}, \langle \Lambda\beta.\, \lambda x{:}\mathcal{C}[\![X]\!]_\beta.\, (\!|\Pi_t|\!)\ \vec{A}\ \beta\ \vec{a}\ x,\ \mathtt{fn}\ \langle\rangle \Rightarrow \mathtt{return}\ \langle\langle\rangle, c_1, \ldots, c_n\rangle\rangle)$$

  It suffices to show $p \in \overline{\Uparrow^{X\to Y}}\underline{\downarrow^{X\to Y}}\{v \mid v \geq_0^{X\to Y} f_i\}$ for all i.
  Suppose $k \in \underline{\downarrow^{X\to Y}}\{v \mid v \geq_0^{X\to Y} \lambda x.\, [\![t]\!](\vec{x}, x)\}$. We have

$$p \bullet k = k\ \mathcal{C}[\![\Gamma]\!]_{\vec{\alpha}}\ (\Lambda\beta.\, \lambda x{:}\mathcal{C}[\![X]\!]_\beta.\, (\!|\Pi_t|\!)\ \vec{A}\ \beta\ \vec{a}\ x)\ \langle\langle\rangle, c_1, \ldots, c_n\rangle$$

  by β-reduction and have to show that $p \bullet k$ terminates. By assumption on k, it suffices to show:

$$(\mathcal{C}[\![\Gamma]\!]_{\vec{\alpha}}, \Lambda\beta.\, \lambda x{:}\mathcal{C}[\![X]\!]_\beta.\, (\!|\Pi_t|\!)\ \vec{A}\ \beta\ \vec{a}\ x), \langle\langle\rangle, c_1, \ldots, c_n\rangle \geq_0^{X\to Y} f_i$$

To this end, suppose $(A, a, c) \geq_0^X x$. We must show that

$$\begin{aligned}&\texttt{let pack}(\alpha, \langle a', e\rangle) = (\!|\Pi_t|\!)\ \vec{A}\ A\ \vec{a}\ a\ \texttt{in}\\&\texttt{pack}(\alpha, \langle a', \texttt{fn}\ \langle\rangle \Rightarrow e(\langle\langle\rangle, c_1, \ldots, c_n, c\rangle))\end{aligned}$$

is in $\overline{\Uparrow^Y}\underline{\downarrow^Y}\{v \mid v \geq_0^Y t_i(\vec{x}, x)\}$. But this is just what the induction hypothesis gives for t_i.

– Case (APP):

$$\textsc{app}\ \frac{\Gamma \vdash s\colon X \to Y \qquad \Delta \vdash t\colon X}{\Gamma, \Delta \vdash s\ t\colon Y}$$

Denote the derivations of the two premises by Π_s and Π_t. Let $(s_k)_{k\geq 0}$ and $(t_k)_{k\geq 0}$ be ω-chains, whose limit is above $[\![\Pi_s]\!]$ and $[\![\Pi_t]\!]$ respectively.
Suppose Γ is $x_1{:}X_1, \ldots, x_n{:}X_n$ and Δ is $y_1{:}Y_1, \ldots, y_m{:}Y_m$. Assume $v_i \geq_0^{X_i} x_i$ and $w_j \geq_0^{Y_j} y_j$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. It suffices to show that $\texttt{inst}((\!|\Pi_s|\!), \vec{v}) \in \overline{\Uparrow^{X\to Y}}\underline{\downarrow^{X\to Y}}\{u \mid u \geq_0^{X\to Y} s_k(\vec{x})\}$ and $\texttt{inst}((\!|\Pi_t|\!), \vec{w}) \in \overline{\Uparrow^X}\underline{\downarrow^X}\{r \mid v \geq_0^X t_k(\vec{y})\}$ implies $\texttt{inst}((\!|\Pi|\!), \vec{v}, \vec{w}) \geq_0^Y s_k(\vec{x}, t_k(\vec{y}))$ for all k. This is shown like in the (APP)-case of Lemma 2.

– Case (REC):

$$\textsc{rec}\ \frac{\Gamma,\ f{:}X \to Y,\ x{:}X \vdash t\colon Y}{\Gamma \vdash \texttt{fun}\ f\ x \Rightarrow t\colon X \to Y}$$

Write Π_t for the derivation of the premise of this rule.
Suppose $(t_k)_{k\geq 0}$ is an ω-chain whose limit gets above $[\![\Pi_t]\!]$.
Let $f_{j,0} := \bot$ and $f_{j,i} := \lambda\vec{x}\, x.\, t_j(\vec{x}, f_{i-1}, x)$. Notice that $[\![\Pi]\!] \sqsubseteq \bigsqcup f_{i,i}$.
Suppose Γ is $x_1{:}X_1, \ldots, x_n{:}X_n$. Assume $(A_i, a_i, c_i) \geq_0^{X_i} x_i$ for $i = 1, \ldots, n$.
We show that

$$\begin{aligned}&\texttt{let pack}(A, \langle a, e\rangle) = (\!|\Pi|\!)\ A_1\ \ldots\ A_n\ a_1\ \ldots\ a_n\ \texttt{in}\\&\texttt{pack}(A, a, \texttt{fn}\ \langle\rangle \Rightarrow e(\langle\langle\rangle, c_1, \ldots, c_n\rangle))\end{aligned}$$

is in $\Uparrow^{X\to Y}\downarrow^{X\to Y}\{v \mid v \leq^{X\to Y} f_{j,i}\}$ for all i and j.
By definition of $(\!|\Pi|\!)$, the above program equals

$$\texttt{pack}(\mathcal{C}[\![\Gamma]\!], a_\Pi, \texttt{fn}\ \langle\rangle \Rightarrow e(\langle\langle\rangle, c_1, \ldots, c_n\rangle)).$$

It therefore suffices to show $r \geq_0^{X\to Y} f_{j,i}$ for $r := (\mathcal{C}[\![\Gamma]\!], a_\Pi, \langle\langle\rangle, c_1, \ldots, c_n\rangle)$.
Let j be arbitrary. We show $r \geq_0^{X\to Y} f_{j,i}$ by induction on i. Suppose $(A_x, a_x, c_x) \geq_0^X x$. We have to show that

$$\begin{aligned}&\texttt{let}\ (A_y, a_y, c_y) = a_\Pi\ A_x\ a_x\ \texttt{in}\\&\texttt{pack}(A_y, a_y, \texttt{fn}\ \langle\rangle \Rightarrow c_y(\langle\langle\langle\rangle, c_1, \ldots, c_n\rangle, c_x\rangle))\end{aligned}$$

is in $\overline{\Uparrow^Y}\underline{\downarrow^Y}\{v \mid v \geq_0^Y d\}$ whenever $f_{j,i}(\vec{x}, x) = \lfloor d \rfloor$.

- Case $i = 0$. Here we have $f_{j,i}(\vec{x}, x) = \bot$ by definition, so this case is vacuous.

- Case $i > 0$. By unfolding `fix` once in $a_\Pi\ A_x\ a_x$, we get

$$a_\Pi\ A_x\ a_x = \texttt{let pack}(\rho, \langle a, e\rangle) = (\!|\Pi_t|\!)\ \vec{A}\ \mathcal{C}[\![\Gamma]\!]_{\vec{A}}\ \beta\ \vec{x}\ a_\Pi\ x\ \texttt{in}$$
$$\texttt{pack}(\rho, \langle a, \texttt{fn}\ \langle c, c_x\rangle \Rightarrow e(\langle\langle c, c\rangle, c_x\rangle)\rangle)$$

The inner induction hypothesis is $(\mathcal{C}[\![\Gamma]\!]_{\vec{A}}, a_\Pi, \langle\langle\rangle, c_1, \ldots, c_n\rangle, c_x\rangle) \geq_0^{X\to Y} f_{j,i}$. The outer induction hypothesis is $(\!|\Pi_t|\!) \geq_0^{(\Gamma, f:X\to Y, x:X), X\to Y} t_k$, for all k. Suppose $f_{j,i}(\vec{x}, x) = \lfloor d \rfloor$. Plugging the inner in the outer induction hypothesis, we obtain that

$$\texttt{inst}((\!|\Pi_t|\!), (A_1, a_1, c_1), \ldots, (A_n, a_n, c_n), r, (A_x, a_x, c_x))$$

is in $\overline{\Uparrow^Y}\underline{\downarrow}^Y\{y \mid y \geq_0^Y d\}$ whenever $t_j(\vec{x}, f_{j,i}(\vec{x}), x) = \lfloor d \rfloor$. By the above unfolding this term equals

$$\begin{array}{l}\texttt{let}\ (A_y, a_y, c_y) = a_\Pi\ A_x\ a_x\ \texttt{in}\\ \texttt{pack}(A_y, a_y, \texttt{fn}\ \langle\rangle \Rightarrow c_y(\langle\langle\langle\rangle, c_1, \ldots, c_n\rangle, c_x\rangle))\ .\end{array}$$

Since $t_j(\vec{x}, f_{j,i}(\vec{x}), x) = f_{j,i+1}(\vec{x}, x)$, we have therefore shown that this term is in $\overline{\Uparrow^Y}\underline{\downarrow}^Y\{y \mid y \geq_0^Y d\}$ whenever $f_{j,i+1}(\vec{x}, x) = \lfloor d \rfloor$, which is what we had to show.