Call-by-Value in a Basic Logic for Interaction

Ulrich Schöpp

Ludwig-Maximilians-Universität München

Abstract. In game semantics and related approaches to programming language semantics, programs are modelled by interaction dialogues. Such models have recently been used by a number of authors for the design of compilation methods, in particular for applications where resource control is important. The work in this area has focused on call-by-name languages. In this paper we study the compilation of call-by-value into a first-order low-level language by means of an interpretation in a semantic interactive model. We refine the methods developed for call-by-name languages to allow an efficient treatment of call-by-value. We introduce an intermediate language that is based on the structure of an interactive computation model and that can be seen as a fragment of Linear Logic. The main result is that Plotkin's call-by-value CPS-translation and its soundness proof can be refined to target this intermediate language. This refined CPS-translation amounts to a direct compilation of the source language into a first-order language.

1 Introduction

The compilation of programming languages to machine code is usually considered as a series of translation steps between intermediate languages of varying expressiveness. While managing the details of low level intermediate languages is sometimes considered an implementation problem, there are many good reasons for studying the logical principles underlying the low level details of compilation, e.g. for the formal verification of compilers and optimisers, the design of intermediate languages, or the analysis of machine code behaviour and resource usage.

The study of the logical structure of low-level computation has been fruitful in recent work on the compilation of programming languages with strong resource constraints. A number of authors, e.g. [10,11,7,9], have used semantic models related to game semantics to design compilation methods with various resource usage guarantees. Game semantics explains higher-order computation by interaction dialogues and can be used to organise low-level programs into semantic models. The idea is to think of low-level programs as implementations of game semantic strategies, so that interaction dialogues appear as traces of low-level programs. Following this idea, one can construct models that have interesting structure, enough to interpret higher-order languages, but also suitable for fine-grained control of resources. For example, the structure has been shown to allow good control of stack space usage in work on using of higher-order languages for hardware synthesis [10] and for programming in logarithmic space [7].

It is reasonable to ask if such semantically-motivated compilation methods are not useful for analysing and organising compilation even if one does not want to impose resource restrictions. Control of stack space usage, for example, is important in compilation and having logical and semantic principles to account for it should be useful.

One way of assessing this question is to capture the structure of semantically-motivated compilation methods in terms of higher intermediate languages and to study their utility for general compilation. In our case, this entails to study the structure of interactive models built from a low-level language, to define intermediate languages for working with this structure and to assess questions such as: Can existing languages be compiled by translation to such an intermediate language? Would we obtain efficient compilation methods and how would they relate to existing methods? Most importantly, would we gain anything from moving to such a more structured intermediate language? Can we identify logical principles for the intermediate language that allows us to reason about low-level programs, such as for proving compiler correctness or resource bounds?

For the compilation of call-by-name languages, such as PCF, there is growing evidence that these questions have a positive answer. The above-mentioned work on resource-aware compilation considers call-by-name languages that can be seen as fragments of PCF with various resource constraints. By relaxing constraints on resources, it is possible to extend this work to cover all of PCF. It turns out that one obtains efficient compilation methods that are related to standard techniques in the compilation of programming languages, such as CPS-translation and defunctionalization [23]. The translation from PCF to the interactive model can be seen as a thunkifying translation [13] into an intermediate language, such as the one described in [24], and correctness follows immediately from equations in the intermediate language.

In this paper we consider the case of call-by-value source languages, which has received much less attention so far. We present a basic intermediate language close to Tensorial Logic [17] that captures just the structure needed to handle the translation of a call-by-value source language. What we obtain is a translation from source language to low-level language that fully specifies all details of the translation, including closure representation. It is compositional, allows for separate compilation and specifies abstractly the interfaces of compiled modules. We show that standard techniques for equational reasoning in the polymorphic λ-calculus can be transported to the intermediate language, which allows for a simple proof of correctness for the translation from call-by-value source language to low-level language.

The translation is a refined call-by-value CPS-translation. Existing work has shown that interactive models naturally support call-by-name languages, so it seems reasonable to try to reduce the case for call-by-value languages to that for call-by-name languages. Indeed, a standard CPS-translation from call-by-value to call-by-name leads to programs that implement call-by-value programs in a natural way. However, the translation fails to satisfy the well-known requirement that call-by-value translations should be *safe for space* [26], which means that the value of a variable should be discarded after the last use in its scope. It appears that the structure identified so far for call-by-name languages does not give us enough low-level control to satisfy such requirements.

Let us outline the issue concretely, using as an example the language INTML [7], which is a call-by-name language developed for compilation with LOGSPACE guarantees. In INTML stack memory management information is made explicit in the type system. Function types have the form $A \cdot X \multimap Y$, where A is a type of values that the function needs to preserve on the stack when it makes a call to its argument. When the function makes a call to its argument, a value of type A is put on the stack and is kept there until

the call returns, whereupon the value of type A is removed from the stack for further use. Thus, values are removed from the stack only when a call returns.

That values are removed from the stack only upon function return is problematic for programming in continuation passing style. Continuations are functions that are invoked, but that typically never return. If one considers CPS-translated call-by-value programs, then this means that no value will ever be discarded in the course of computation. Indeed, if we translate the term $\mathsf{let}\ x{=}5\ \mathsf{in}\ \mathsf{let}\ y{=}x+1\ \mathsf{in}\ \mathsf{let}\ z{=}y+4\ \mathsf{in}\ z+3$ to INTML using Plotkin's CPS-translation (see Section 4.1), then the resulting term can be given type $(\mathsf{nat}\times\mathsf{nat}\times\mathsf{nat})\cdot(\mathsf{unit}\cdot[\mathsf{nat}]\multimap[0])\multimap[0]$. The type already shows that the continuation will be called with three natural numbers on the stack, even though only a single number, namely 13, should be needed. Indeed, the stack will contain the triple $\langle 6,10,13\rangle$ of all the intermediate values for x, y and z. The question is therefore how we can give a translation that deallocates values that are not needed anymore.

In this paper we show how to refine the CPS-translation to target an intermediate language derived from an interactive model in a way that addresses this issue of managing values and their deallocation. It turns out that this issue is orthogonal to duplication in the source language. In order to focus on value management, we shall therefore first focus on the linear case. To allow duplication it will be enough to allow duplication in the intermediated language, see Section 5.

In Section 3 we first define a basic linear intermediate language. This intermediate language allows us to formulate the call-by-value translation for a linear source language as a refinement of Plotkin's call-by-value CPS-translation [19] in Section 4. In contrast to standard approaches of compiling with continuations, such as [4], where CPS-translation targets a higher-order intermediate language that still requires closure conversion, the refined CPS-translation fully specifies a translation from source language to the first-order low-level language. The translation appears to be related to defunctionalizing compilation methods [5,6], see also [23]. We believe that it is also related to the call-by-value games of [14,3], in particular [16] seems relevant. Notice, however, that these games do not make explicit which values must be stored for how long. The translation in this paper makes this and other low-level details, such as closure conversion, explicit. In Section 4 we show it nevertheless allows the soundness of the translation to be proved by an argument close to Plotkin's original soundness proof. In Section 5 we then explain how to lift the linearity restriction in order to translate a simply-typed source language.

2 Low-Level Programs

We start by fixing the low-level language, which is essentially a goto language. It is typed and works with values of the following first-order types.

$$\text{Value Types}\quad A,B ::= \alpha \mid \mathsf{nat} \mid \mathsf{unit} \mid A\times B \mid 0 \mid A+B$$
$$\text{Values}\quad v,w ::= x \mid n \mid \langle\rangle \mid \langle v,w\rangle \mid \mathsf{inl}(v) \mid \mathsf{inr}(v)$$

Low-level programs are built from *blocks* of the form $f(x\colon A)\,\{\,b\,\}$, where f is the block label, x is its formal parameter and b is the body, formed according to the grammar below. Therein, v ranges over values, g over labels and op over primitive operation constants.

$$b ::= \mathsf{let}\ x{=}op(v)\ \mathsf{in}\ b \mid \mathsf{let}\ \langle x,y\rangle{=}v\ \mathsf{in}\ b \mid \mathsf{case}\ v\ \mathsf{of}\ \mathsf{inl}(x)\Rightarrow b_1;\mathsf{inr}(y)\Rightarrow b_2 \mid g(v)$$

The set of constants that takes arguments of type A and returns values of type B is defined by a set $\mathit{Prim}(A, B)$. In this paper, we assume $\mathit{Prim}(\mathsf{nat}, \mathsf{unit}) = \{\mathsf{print}\}$, $\mathit{Prim}(\mathsf{nat} \times \mathsf{nat}, \mathsf{int}) = \{\mathsf{add}, \mathsf{times}, \mathsf{div}\}$, $\mathit{Prim}(\mathsf{nat} \times \mathsf{nat}, \mathsf{unit} + \mathsf{unit}) = \{\mathsf{eq}, \mathsf{lt}\}$ and that all other *Prim*-sets are empty.

A *program* p is given by a set of block definitions together with two distinguished labels entry_p and exit_p. The program must be such that there are no two block definitions with the same label and that there is no definition of the exit label. We write short $(x \mapsto v)$ for the program with a single block $\mathit{entry}(x\colon A)\,\{\,\mathit{exit}(v)\,\}$ and use informal pattern matching notation, such as writing $(\langle x, y\rangle \mapsto v)$ for $(z \mapsto \mathsf{let}\ \langle x, y\rangle{=}z\ \mathsf{in}\ v)$. We write Ω_A for the nonterminating program $\mathit{entry}(x\colon A)\,\{\,\mathit{entry}(x)\,\}$.

Programs are assumed to be typed in the canonical way. We write $p\colon A \to B$ if p is a program whose entry and exit labels accept values of type A and B respectively.

Operational Semantics. The operational semantics of a program p is given by a relation $b_1 \xrightarrow{o}_p b_2$, which expresses that body term b_1 reduces to body term b_2 while outputting the sequence of closed values o using the print-operation. We write ε for the empty sequence and $o_1 o_2$ for concatenation of o_1 and o_2.

This relation is defined to be the smallest relation such that $b_1 \xrightarrow{o_1}_p b_2 \xrightarrow{o_2}_p b_3$ implies $b_1 \xrightarrow{o_1 o_2}_p b_3$, such that $f(v) \xrightarrow{\varepsilon}_p b[v/x]$ if p contains a block definition $f(x\colon A)\,\{\,b\,\}$, and such that the following hold.

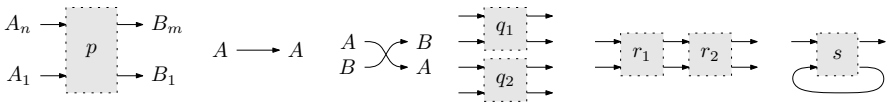
$$\mathsf{let}\ \langle x, y\rangle{=}\langle v, w\rangle\ \mathsf{in}\ b \xrightarrow{\varepsilon}_p b[v/x, w/y] \qquad \mathsf{case}\ \mathsf{inl}(v)\ \mathsf{of}\ \mathsf{inl}(y) \Rightarrow b; \ldots \xrightarrow{\varepsilon}_p b[v/y]$$

$$\mathsf{let}\ x{=}\mathsf{print}(v)\ \mathsf{in}\ b \xrightarrow{v}_p b[\mathsf{unit}/x] \qquad \mathsf{case}\ \mathsf{inr}(v)\ \mathsf{of}\ \ldots; \mathsf{inr}(y) \Rightarrow b \xrightarrow{\varepsilon}_p b[v/y]$$

$$\mathsf{let}\ x{=}\mathsf{add}(\langle m, n\rangle)\ \mathsf{in}\ b \xrightarrow{\varepsilon}_p b[m + n/x] \qquad \text{(similar cases for mul, div, eq and lt)}$$

Notation. We consider two programs $p, q\colon A \to B$ equal, written $p = q$, if they have the same observable effects and return the same values: Whenever $\mathit{entry}_p(v) \xrightarrow{o}_p b$ then $\mathit{entry}_q(v) \xrightarrow{o}_q b'$ for some b', whenever $\mathit{entry}_p(v) \xrightarrow{o}_p \mathit{exit}_p(w)$ then $\mathit{entry}_q(v) \xrightarrow{o}_q \mathit{exit}_q(w)$, and the same two conditions with the roles of p and q exchanged.

We use standard graphical notation [25] for working with low-level programs. A program $p\colon A_1 + \cdots + A_n \to B_1 + \cdots + B_m$ is depicted like on the left below. Shown next to it are the identity program $\mathit{id}_A\colon A \to A$ and the swapping program $\mathit{swap}_{A,B} : B + A \to A + B$. Programs can be composed by vertical and horizontal composition and by taking loops. For two programs $q_1\colon A \to B$ and $q_2\colon C \to D$, their vertical composition is the sum $q_1 + q_2\colon A + C \to B + D$, which is the program obtained by renaming all labels in q_1 and q_2 so that no label appears in both programs and by adding new definitions $\mathit{entry}(x\colon A + C)\,\{\,\mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}(x_1) \Rightarrow \mathit{entry}_{q_1}(x_1); \mathsf{inr}(x_2) \Rightarrow \mathit{entry}_{q_2}(x_2)\,\}$, $\mathit{exit}_{q_1}(x\colon B)\,\{\,\mathit{exit}(\mathsf{inl}(x))\,\}$ and $\mathit{exit}_{q_2}(x\colon D)\,\{\,\mathit{exit}(\mathsf{inr}(x))\,\}$, where entry and exit are the fresh entry and exit labels of $q_1 + q_2$. The horizontal composition of $r_1\colon A \to B$ and $r_2\colon B \to C$ stands for the sequential composition $r_2 \circ r_1\colon A \to C$ and is defined similarly. Loops are defined by jumping from the exit to entry label.



The values of any closed type A can be encoded into natural numbers, that is one can define programs $\mathsf{encode}_A \colon A \to \mathsf{nat}$ and $\mathsf{decode}_A \colon \mathsf{nat} \to A$ such that $\mathsf{decode}_A \circ \mathsf{encode}_A = id_A$ holds. To simplify the examples, we assume that the encoding and decoding functions for nat are the identity. Given a value-type-with-hole $C[\cdot]$, i.e. a value type with zero or more occurrences of $\cdot$, we write $C[A]$ for the type obtained by replacing each $\cdot$ with A. The encoding and decoding programs can be lifted to $C[\mathsf{encode}_A] \colon C[A] \to C[\mathsf{nat}]$ and $C[\mathsf{decode}_A] \colon C[\mathsf{nat}] \to C[A]$ by induction on C.

3 A Basic Linear Intermediate Language

In this section we introduce a basic linear higher-order intermediate language LIN that can be used to organise low-level programs and to reason about them. It can be seen as a syntactic description of the mathematical structure obtained by applying the Int construction [15] to a term model of the low-level language, see [7]. The intermediate language recombines the ideas developed in [21,7,22] in a way that is suitable to handle also the translation of call-by-value. It was inspired by Melliès' Tensorial Logic [18,17]. We can give here only a concise definition of bare LIN; a richer intermediate language, defined not solely for the study of call-by-value, is described in detail in [24].

The types and terms of LIN are defined by the following grammars, in which A ranges over value types and α ranges over value type variables.

$$\text{Interface Types} \quad X, Y \ ::= \ A^\perp \ \mid \ X \multimap Y \ \mid \ \forall \alpha.\, X$$

These types may be thought of as the interfaces of interactive entities that are implemented in the low-level language. The type $A^\perp$ is the interface of low-level programs that accept inputs of type A and that never return anything. The type $X \multimap Y$ is the interface of low-level programs that when linked to a program implementing interface X become an implementation of interface Y. Finally, value type polymorphism $\forall \alpha.\, X$ makes it easier to write programs and to reason about them.

The terms of LIN are defined as follows, where p ranges over low-level programs.

$$s, t \ ::= \ x \ \mid \ p^* t \ \mid \ \langle s, t \rangle \ \mid \ \mathsf{let}\ \langle x, y \rangle = s\ \mathsf{in}\ t \ \mid \ \lambda x{:}X.\, t \ \mid \ s\ t \ \mid \ \Lambda \alpha.\, t \ \mid \ t\ A$$

The typing rules for LIN are given in Fig. 1. We explain the meaning of terms by defining a translation to low-level programs. Each type X represents an interface that consists of two value types X^- and X^+:

$$\begin{array}{lll} (A^\perp)^- = A & (X \multimap Y)^- = X^+ + Y^- & (\forall \alpha.\, X)^- = X^-[\mathsf{nat}/\alpha] \\ (A^\perp)^+ = 0 & (X \multimap Y)^+ = X^- + Y^+ & (\forall \alpha.\, X)^+ = X^+[\mathsf{nat}/\alpha] \end{array}$$

The idea is that an implementation of interface X is a program of type $X^- \to X^+$. For contexts we let $(-)^- = (-)^+ = 0$ and $(\Gamma, x \colon X)^- = \Gamma^- + X^-$ and $(\Gamma, x \colon X)^+ = \Gamma^+ + X^+$. A program of type $\Gamma^- \to \Gamma^+$ thus consists of an implementation of the interface of each variable in Γ.

A typing sequent $\Gamma \vdash t \colon X$ is translated as a low-level program $[\![\Gamma \vdash t \colon X]\!]$ of type $\overline{\Gamma^+} + \overline{X^-} \to \overline{\Gamma^-} + \overline{X^+}$, where we write $\overline{(-)}$ for the operation of substituting nat for all free type variables. The intention is that if one connects an implementation e of the interface of Γ as follows, then one obtains a program implementing interface X.

$$\text{AX}\ \frac{}{x\colon X \vdash x\colon X} \qquad \text{WEAK}\ \frac{\Gamma \vdash t\colon Y}{\Gamma,\, x\colon X \vdash t\colon Y} \qquad \text{EXCH}\ \frac{\Gamma,\, y\colon Y,\, x\colon X,\, \Delta \vdash t\colon Z}{\Gamma,\, x\colon X,\, y\colon Y,\, \Delta \vdash t\colon Z}$$

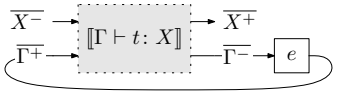
$$0\ \frac{}{\vdash \star\colon 0^\perp} \qquad \text{ACT}\ \frac{p\colon A \to B \qquad \Gamma \vdash t\colon B^\perp}{\Gamma \vdash p^* t\colon A^\perp}$$

$$\langle\rangle\text{I}\ \frac{\Gamma \vdash s\colon A^\perp \qquad \Delta \vdash t\colon B^\perp}{\Gamma, \Delta \vdash \langle s, t\rangle\colon (A+B)^\perp} \qquad \langle\rangle\text{E}\ \frac{\Gamma \vdash s\colon (A+B)^\perp \qquad \Delta, x\colon A^\perp, y\colon B^\perp \vdash t\colon X}{\Gamma, \Delta \vdash \mathsf{let}\ \langle x, y\rangle = s\ \mathsf{in}\ t\colon X}$$

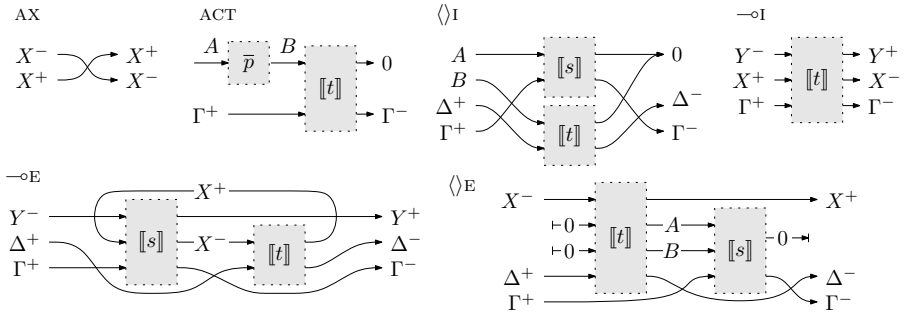
$$\multimap\text{I}\ \frac{\Gamma,\, x\colon X \vdash t\colon Y}{\Gamma \vdash \lambda x{:}X.\, t\colon X \multimap Y} \qquad \multimap\text{E}\ \frac{\Gamma \vdash s\colon X \multimap Y \qquad \Delta \vdash t\colon X}{\Gamma, \Delta \vdash s\ t\colon Y}$$

$$\forall\text{I}\ \frac{\Gamma \vdash t\colon X}{\Gamma \vdash \Lambda\alpha.\, t\colon \forall\alpha.\, X}\ \alpha \text{ not in } \Gamma \qquad \forall\text{E}\ \frac{\Gamma \vdash t\colon \forall\alpha.\, X}{\Gamma \vdash t\ A\colon X[A/\alpha]}$$

Fig. 1. Linear Intermediate Language



The translation is defined by induction on the derivation and is given in graphical notation below. Strictly speaking, we define an interpretation of derivations rather than sequents. We write only $[\![\Gamma \vdash t\colon X]\!]$, as different derivations of the same sequent will be equal in the sense defined in the next section. We also write just $[\![t]\!]$ when context and type are clear from the context. For readability, we omit the operation $\overline{(-)}$ on types – it is assumed to be applied to all types in the figure below.



The basic rules AX, WEAK, EXCH, —∘I and —∘E amount to a standard interpretation of the linear λ-calculus in the monodial closed structure that one obtains from applying the categorical Int construction to the low-level language, see [7]. It is the same structure that one finds in the Geometry of Interaction [1] and in Game Semantics [2]. The rules for functions formalise the view that the type $X \multimap Y$ is implemented by programs that, when linked with a program implementing interface X, implement interface Y. Rule —∘E implements the linking of a function program to its argument program. Rule —∘I amounts

to a reinterpretation of the interface of $[\![t]\!]$, so that an application will link the argument program to variable x.

Polymorphism is implemented by using nat in place of any type variable α. Since the values of any type can be encoded into nat, we can recover any type instance by using the encoding and decoding programs defined above. Rule $\forall$I is the identity, just like $\multimap$I, as the type variable α has already been substituted by nat. Rule $\forall$E realises the encoding and decoding of values of type A into ones of type nat. Let $F[\cdot] = \overline{X^-[\cdot/\alpha]}$ and $G[\cdot] = \overline{X^+[\cdot/\alpha]}$. We then have programs $F[\mathsf{encode}_{\overline{A}}] \colon F[\overline{A}] \to F[\mathsf{nat}]$ and $G[\mathsf{decode}_{\overline{A}}] \colon G[\mathsf{nat}] \to G[\overline{A}]$. Note $F[\mathsf{nat}] = \overline{X^-}$ and $\overline{X^+} = G[\mathsf{nat}]$. To obtain $[\![t\ A]\!]$, we pre- and post-compose $[\![t]\!]$ with these programs.

It remains to define the structural rules and 0. Weakening is defined such that any input on the weakened port results in a diverging computation, and exchange permutes the input and output wires. The conclusion in rule 0 is interpreted as $id_0 \colon 0 \to 0$.

3.1 Equational Theory

We intend LIN to be used as language for constructing low-level programs and for reasoning about them. For equational reasoning, the notion of equality for LIN should be defined to reflect a reasonable notion of equality of low-level programs. In this section we define a notion of equality for LIN that allows us to reason about program correctness: closed terms of base type are equal if and only if they translate to equal programs.

To define equality, one possible option would be to consider two terms equal when they translate to equal low-level programs. This definition would validate the β-equality $(\lambda x{:}X.\, t)\ s = t[s/x]$ if s is closed, but it would not validate other reasonable instances of it. For example, if t does not contain x, then s is dead code and the β-equality would correspond to reasonable dead code elimination. However, if s has free variables, then it may be possible to distinguish the low-level programs interpreting $(\lambda x{:}X.\, t)\ s$ and $t[s/x]$ by interacting with the dead code that would otherwise never be used.

This motivates the definition of a coarser equality relation for LIN. We define it so that we can use the parametricity of polymorphism for equality reasoning. We therefore use a relational definition of term equality for LIN, where terms are equal if they map related arguments to related results, just like in standard models of polymorphism [27].

A *value type relation* is given by a triple (A, A', R) of two closed value types A and A' and a binary relation R between the closed values of type A and those of type A'. We write $R \subseteq A \times A'$ for the triple (A, A', R).

A *value type environment* ρ is a mapping from type variables to value type relations. If σ and σ' are both mappings from type variables to closed value types, then we write $\rho \subseteq \sigma \times \sigma'$ if $\rho(\alpha)$ is a relation $R_\alpha \subseteq \sigma(\alpha) \times \sigma'(\alpha)$. For such σ we write $X\sigma$ for the type obtained from X by substituting any variable α with $\sigma(\alpha)$.

For each value type A and each $\rho \subseteq \sigma \times \sigma'$, define a relation $[\![A]\!]_\rho \subseteq A\sigma \times A\sigma'$ by:

$$
\begin{aligned}
[\![\alpha]\!]_\rho &= \rho(\alpha) \\
[\![A + B]\!]_\rho &= \{(\mathsf{inl}(v), \mathsf{inl}(v')) \mid v\ [\![A]\!]_\rho\ v'\} \cup \{(\mathsf{inr}(w), \mathsf{inr}(w')) \mid w\ [\![B]\!]_\rho\ w'\} \\
[\![A \times B]\!]_\rho &= \{(\langle v, w\rangle, \langle v', w'\rangle) \mid v\ [\![A]\!]_\rho\ v', w\ [\![B]\!]_\rho\ w'\} \\
[\![A]\!]_\rho &= \{(v, v) \mid v \text{ value of type } A\} \text{ if } A \text{ is a base type } (0, \mathsf{unit}, \mathsf{nat}).
\end{aligned}
$$

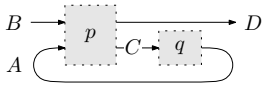
For any interface type X and any type environment $\rho \subseteq \sigma \times \sigma'$ we define a relation $[\![X]\!]_\rho$ between low-level programs of type $(X\sigma)^- \to (X\sigma)^+$ and $(X\sigma')^- \to (X\sigma')^+$:

$$p\ [\![A^\perp]\!]_\rho\ p' \text{ iff } \forall v, v'.\ v\ [\![A]\!]_\rho\ v' \Rightarrow \forall o.\ ((\exists b.\ entry_p(v) \xrightarrow{o}_p b) \Leftrightarrow (\exists b'.\ entry_{p'}(v') \xrightarrow{o}_{p'} b'))$$

$$p\ [\![X \multimap Y]\!]_\rho\ p' \text{ iff } \forall q, q'.\ q\ [\![X]\!]_\rho\ q' \Rightarrow \mathsf{app}(p,q)\ [\![Y]\!]_\rho\ \mathsf{app}(p',q')$$

$$p\ [\![\forall \alpha.X]\!]_\rho\ p' \text{ iff } \forall R \subseteq A \times A'.\ \mathsf{inst}_{(\forall\alpha.X)\sigma}(p,A)\ [\![X]\!]_{\rho[R/\alpha]}\ \mathsf{inst}_{(\forall\alpha.X)\sigma'}(p',A')$$

In these cases, we use the following notation. For programs $p\colon A + B \to C + D$ and $q\colon C \to A$, we write $\mathsf{app}(p,q)$ for the following program of type $B \to D$.

$\mathsf{app}(p,q) =$ 

If p is a program of type $\overline{X^-} \to \overline{X^+}$ then we write $\mathsf{inst}_{\forall\alpha.X}(p,A)$ for the program obtained by pre- and post-composing with $F[\mathsf{encode}_A]$ and $G[\mathsf{decode}_A]$, where $F[\cdot] = \overline{X^-[\cdot/\alpha]}$ and $G[\cdot] = \overline{X^+[\cdot/\alpha]}$. We write $\mathsf{inst}_{\forall\vec{\alpha}.X}(p,\sigma)$ for iterated application of inst.

Definition 1 (Equality). *Suppose $\Gamma \vdash s\colon X$ and $\Gamma \vdash t\colon X$ are derivable. Suppose $\vec{\alpha}$ is a list of the free type variables in these sequents and suppose Γ is $x_1\colon X_1, \ldots, x_n\colon X_n$. Then we write $\Gamma \models s = t\colon X$ if: For any value type environment $\rho \subseteq \sigma \times \tau$ and all $\vec{p}$ and $\vec{q}$ with $p_i\ [\![X_i]\!]_\rho\ q_i$ for $i = 1, \ldots, n$, $\mathsf{app}(\mathsf{inst}_{(\forall\vec{\alpha}.\Gamma\multimap X)\sigma}([\![\Gamma \vdash s\colon X]\!], \sigma), \vec{p})$ and $\mathsf{app}(\mathsf{inst}_{(\forall\vec{\alpha}.\Gamma\multimap X)\tau}([\![\Gamma \vdash t\colon X]\!], \tau), \vec{q})$ are $[\![X]\!]_\rho$-related.*

This definition requires some justification, in particular that it does not depend on the choice of derivation of $\Gamma \vdash s\colon X$ or $\Gamma \vdash t\colon X$. It is possible to define equality for derivations and show coherence, i.e. that two derivations of the same derivation have equal interpretations, see [24] for more details. For the results in this paper, it would be just as well to work with derivations and without coherence, so we do not spell this out.

The parametricity lemma takes the form:

Lemma 1 (Parametricity). *If $\Gamma \vdash t\colon X$ then $\Gamma \models t = t\colon X$.*

Lemma 2 (Identity Extension). *For any σ, we have $\models s = t\colon X\sigma$ if and only if $[\![s]\!]\ [\![X]\!]_{\Delta_\sigma}\ [\![t]\!]$, where $\Delta_\sigma(\alpha) = \{(v,v) \mid v \text{ is closed value of type } \sigma(\alpha)\}$.*

Equality is symmetric and transitive and a congruence with respect to the term constructors. We state further properties of equality in the following lemmas. In them, as in rest of this paper, we write just $s = t$ to mean that $\Gamma \models s = t\colon X$ holds for any Γ and X that make both terms well-typed.

Lemma 3. *The following β-equalities are valid.*

$$(\lambda x{:}X.\ s)\ t = s[t/x] \quad (\Lambda\alpha.t)\ A = t[A/\alpha] \quad (\mathsf{let}\ \langle x,y\rangle{=}\langle s,t\rangle\ \mathsf{in}\ r) = r[s/x, t/y]$$

Lemma 4. *The equations $id^*t = t$ and $p^*(q^*t) = (q \circ p)^*t$ are valid. Moreover, if $p = q$ then $p^*t = q^*t$.*

We note that LIN allows duplication of values of type $A^\perp$. If $\Gamma, x_1 \colon A^\perp, x_2 \colon A^\perp \vdash t \colon Y$ then $\Gamma, x \colon A^\perp \vdash \mathsf{let}\ \langle x_1, x_2\rangle{=}\nabla_A^* x\ \mathsf{in}\ t \colon Y$, where $\nabla_A \colon A + A \to A$ is the canonical low-level program of its type, i.e. the one mapping both $\mathsf{inl}(v)$ and $\mathsf{inr}(v)$ to v. If one substitutes a closed term s for x, then one has $(\mathsf{let}\ \langle x_1, x_2\rangle{=}\nabla_A^* s\ \mathsf{in}\ t) = t[s/x_1, s/x_2]$, as the following lemma shows.

Lemma 5. *The equations* $(x \mapsto \mathsf{inl}(x))^*\langle s, t\rangle = s$ *and* $(x \mapsto \mathsf{inr}(x))^*\langle s, t\rangle = t$ *are valid. Moreover, if* t *is closed then we have* $\nabla_A^* t = \langle t, t\rangle$.

Relational parametricity is useful, as it justifies dinaturality properties, e.g. that $\vdash s \colon \forall\alpha.\, \alpha^\perp \multimap \alpha^\perp$ satisfies $(x \mapsto v)^*(s\ A\ t) = s\ B\ ((x \mapsto v)^* t)$ for any $\vdash t \colon A^\perp$ and any value v of appropriate type. It would be possible to write this paper without polymorphism in LIN, but at the expense of having to establish such equations explicitly as invariants in all constructions. In Lemma 9 we use the following instance of dinaturality. Its proof is much like the proof that parametricity implies dinaturality in [20].

Lemma 6. *Suppose* $\vdash s \colon \forall\alpha.\, (\forall\beta.\, X \multimap \forall\gamma.\, Y \multimap \alpha^\perp) \multimap \alpha^\perp$*, where* α *is not free in* X *or* Y*. Then* $(y \mapsto v)^*(s\ A\ t) = s\ B\ (\Lambda\beta.\, \lambda x{:}X.\, \Lambda\gamma.\, \lambda y{:}Y.\, (y \mapsto v)^*(t\ x))$ *holds for any value* v *and any closed* t *of correct type.*

4 Linear Call-by-Value

The rest of the paper is devoted to showing how a call-by-value λ-calculus can be translated to intermediate languages based on LIN. To handle the full call-by-value λ-calculus, we shall need to extend LIN with a form of duplication in Section 5. In this section we first show that a *linear* fragment of the call-by-value λ-calculus can be translated to LIN. We do so by refining Plotkin's call-by-value CPS-translation to target LIN instead of the λ-calculus. While the linear source language is very simple, it is instructive to consider the translation for it first, as this already requires us to develop the main infrastructure needed for the translation.

By composing the refined CPS-translation to LIN with the translation from LIN to the low-level language, we obtain a fully specified translation from higher-order source language to first-order low-level language. The point is that LIN is low-level enough to give us fine-grained control over low-level programs, for example to make the closure representation and memory management issues explicit, but at the same time it is high-level enough to carry out essentially the standard correctness proof of the CPS-translation.

As the source language we consider a λ-calculus with a base type of natural numbers and a diverging term Ω that allows one to observe the evaluation order.

Source Types	X, Y	$::= \mathbb{N} \mid X \to Y$
Source Values	V, W	$::= x \mid \lambda x{:}X.\, M \mid n$ (natural number constant)
Source Terms	M, N	$::= V \mid M\ N \mid \mathsf{add}(V, W) \mid \mathsf{if0}(V, M, N) \mid \Omega$

The source terms are typed as follows:

$$\frac{}{\Gamma, x\colon X \vdash x\colon X} \qquad \frac{\Gamma, x\colon X \vdash M\colon Y}{\Gamma \vdash \lambda x{:}X.\, M\colon X \to Y} \qquad \frac{\Gamma \vdash M\colon X \to Y \qquad \Gamma \vdash N\colon X}{\Gamma \vdash M\ N\colon Y}$$

$$\frac{}{\Gamma \vdash n\colon \mathbb{N}} \qquad \frac{\Gamma \vdash V_i\colon \mathbb{N}}{\Gamma \vdash \mathsf{add}(V_1, V_2)\colon \mathbb{N}} \qquad \frac{\Gamma \vdash V\colon \mathbb{N} \qquad \Gamma \vdash N_i\colon \mathbb{N}}{\Gamma \vdash \mathsf{if0}(V, N_1, N_2)\colon \mathbb{N}} \qquad \frac{}{\Gamma \vdash \Omega\colon \mathbb{N}}$$

In this section we moreover impose the linearity restriction that in a source term any variable may be used at most once. (Duplication of variables of type $\mathbb{N}$ could also be allowed in this section, but we do without for simplicity.)

The terms for addition and if-then-else are restricted to values for technical convenience; one can write an addition function as $\lambda x{:}\mathbb{N}.\, \lambda y{:}\mathbb{N}.\, \mathsf{add}(x, y)$, for example. The if-then-else is restricted to the base type $\mathbb{N}$; we discuss this in Section 5.

We use a standard call-by-value reduction semantics: $(\lambda x{:}X.\, M)\ V \longrightarrow M[V/x]$, $\mathsf{add}(m, n) \longrightarrow m + n$, $\mathsf{if0}(0, M, N) \longrightarrow M$, $\mathsf{if0}(n+1, M, N) \longrightarrow N$, $\Omega \longrightarrow \Omega$, $M_1\ N \longrightarrow M_2\ N$ if $M_1 \longrightarrow M_2$, and $V\ N_1 \longrightarrow V\ N_2$ if $N_1 \longrightarrow N_2$.

4.1 CPS-Translation

The aim is now to move Plotkin's call-by-value CPS-translation to LIN as the target language. We first recall Plotkin's CPS-translation [19] and its typing [12], that is, a variant that always puts the continuation first [8] and that covers our source language.

To any source type X, we assign a continuation type $\mathcal{K}(X)$ defined by:

$$\mathcal{K}(X) = \mathcal{A}(X) \to \bot \qquad \mathcal{A}(\mathbb{N}) = \mathbb{N} \qquad \mathcal{A}(X \to Y) = \mathcal{K}(Y) \to \mathcal{K}(X)$$

A source term of type $x_1\colon X_1, \ldots, x_n\colon X_n \vdash M\colon X$ is translated to a term of type $x_1\colon \mathcal{A}(X_1), \ldots, x_n\colon \mathcal{A}(X_n) \vdash \mathsf{cps}(M)\colon \mathcal{T}(X)$ where $\mathcal{T}(X) = \mathcal{K}(X) \to \bot$:

$$\begin{aligned}
\mathsf{cps}(x) &= \lambda k.\, k\ x & \mathsf{cps}(\lambda x.\, M) &= \lambda k.\, k\ (\lambda k_1.\, \lambda x.\, \mathsf{cps}(M)\ k_1)\\
\mathsf{cps}(n) &= \lambda k.\, k\ n & \mathsf{cps}(M\ N) &= \lambda k.\, \mathsf{cps}(M)\ (\lambda f.\, \mathsf{cps}(N)\ (\lambda x.\, f\ k\ x))\\
\mathsf{cps}(\Omega) &= \lambda k.\, \Omega_\bot & \mathsf{cps}(\mathsf{add}(V, W)) &= \lambda k.\, \mathsf{cps}(V)\ (\lambda x.\, \mathsf{cps}(W)\ (\lambda y.\, k\ (x + y)))\\
& & \mathsf{cps}(\mathsf{if0}(V, M, N)) &= \lambda k.\, \mathsf{cps}(V)\ (\lambda x.\, \mathsf{if0}(x, \mathsf{cps}(M)\ k, \mathsf{cps}(N)\ k))
\end{aligned}$$

4.2 Refining the CPS-Translation

While the CPS-translation above is completely linear, it is not obvious how to adapt it to target LIN. The problem is that in LIN we must make explicit which values to keep for later use. Consider for example the addition term $x\colon \mathbb{N}, y\colon \mathbb{N} \vdash x + y\colon \mathbb{N}$ used in the above translation. We cannot just represent $\mathbb{N}$ in LIN by an interface from which we can get a single number at a time: in order to compute the sum we must have both summands at the same time. We address this issue by making the environment of variables explicit.

We decompose $\mathcal{A}(X)$ into two parts: First there is a *code type* $\mathcal{C}_\varphi(X)$, which is a value type. This is the type of codes that represent the values of type X. The code for a natural number would be just the number itself, while the code for a function might be the tuple of the values of the free variables in the body of the function, or similar.

Second, there is an *access type* $\mathcal{A}_\varphi(X)$, which is a LIN-type. The terms of type $\mathcal{A}_\varphi(X)$ will represent low-level programs that can interpret codes of type $\mathcal{C}_\varphi(X)$. They allow us to use such codes without knowledge of the encoding details.

The parameter φ in $\mathcal{C}_\varphi(X)$ and $\mathcal{A}_\varphi(X)$ is a value type variable that allows for information hiding. The code for a natural number should always be the number itself, but for a function, any possible encoding should be acceptable. We do not need to inspect the codes for functions; it suffices that they are accepted by access types. The type variable φ represents a type of abstract codes that we do not know anything about other than that it is accepted by the terms of access type.

For the given source language, the type of continuations is thus refined to:

$$\begin{aligned} \mathcal{K}_\alpha(X) &= \forall\varphi.\, \mathcal{A}_\varphi(X) \multimap (\mathcal{C}_\varphi(X) \times \alpha)^\perp \\ \mathcal{C}_\varphi(\mathbb{N}) = \mathsf{nat} \quad & \mathcal{C}_\varphi(X \to Y) = \varphi \\ \mathcal{A}_\varphi(\mathbb{N}) = 0^\perp \quad & \mathcal{A}_\varphi(X \to Y) = \forall\beta.\, \mathcal{K}_\beta(Y) \multimap \mathcal{K}_{\varphi\times\beta}(X) \end{aligned}$$

The type $\mathcal{K}_\alpha(X)$ of continuations refines $\mathcal{K}(X)$. A continuation accepts inputs that may be encoded using any encoding type φ. For this to work, the continuation must know how to use such an encoded value, which is what the argument of type $\mathcal{A}_\varphi(X)$ provides. Then, any value of type $\mathcal{C}_\varphi(X) \times \alpha$, i.e. the code of an actual value (we ignore α for now and come back to it below), can be thrown into the continuation.

The access types are defined as explained above. For natural numbers we do not need an access type, as the code type does not use φ. Indeed, we have $\mathcal{K}_\alpha(\mathbb{N}) = \forall\varphi.\, 0^\perp \multimap (\mathsf{nat} \times \alpha)^\perp$ and this type is isomorphic to $(\mathsf{nat} \times \alpha)^\perp$. A continuation of type $\mathbb{N}$ therefore just expects to be passed a natural number (again, ignore α for now).

Finally, we define a type $\mathcal{T}_\gamma(X)$ by

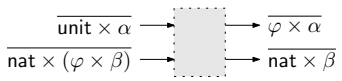
$$\mathcal{T}_\gamma(X) = \forall\alpha.\, \mathcal{K}_\alpha(X) \multimap (\gamma \times \alpha)^\perp \ .$$

The CPS-translation will be such that terms of source type X are translated to LIN-terms of type $\mathcal{T}_G(X)$, where $G = \mathsf{unit} \times \mathcal{C}_{\varphi_1}(X_1) \times \cdots \times \mathcal{C}_{\varphi_n}(X_n)$ is a type containing the codes for the values of the free variables of the term.

Example 1. Let us explain concretely how to understand the type

$$\begin{aligned} \mathcal{T}_{\mathsf{unit}}(\mathbb{N} \to \mathbb{N}) &\cong \forall\alpha.\, \Big(\forall\varphi.\, \mathcal{A}_\varphi(\mathbb{N} \to \mathbb{N}) \multimap (\mathcal{C}_\varphi(\mathbb{N} \to \mathbb{N}) \times \alpha)^\perp\Big) \multimap (\mathsf{unit} \times \alpha)^\perp \\ &\cong \forall\alpha.\, \Big(\forall\varphi.\, (\forall\beta.\, (\mathsf{nat} \times \beta)^\perp \multimap (\mathsf{nat} \times (\varphi \times \beta))^\perp) \multimap (\varphi \times \alpha)^\perp\Big) \multimap \alpha^\perp \ . \end{aligned}$$

Closed terms of this type are translated to programs with the following interface (up to removal of ports of type 0).



A function $f : \mathbb{N} \to \mathbb{N}$ is implemented by such a program in the following way. To start the evaluation of the function itself, we pass any value $\langle\langle\rangle, s\rangle : \overline{\mathsf{unit} \times \alpha}$ to the first input port. The low-level program then returns a pair $\langle c, s\rangle : \overline{\varphi \times \alpha}$, where the first component c

If Γ declares the variables $\vec{z}$ and these all appear free in M, then define $\mathsf{cps}(\Gamma \vdash M)$ by:

$$
\begin{aligned}
\mathsf{cps}(x\colon X \vdash x) &= \Lambda\alpha.\,\lambda k.\,(\langle\langle\langle\rangle, x\rangle, s\rangle \mapsto \langle x, s\rangle)^*(k\ \varphi_x\ x)\\
\mathsf{cps}(\vdash n) &= \Lambda\alpha.\,\lambda k.\,(\langle\langle\rangle, s\rangle \mapsto \langle n, s\rangle)^*(k\ \mathsf{unit}\ \star)\\
\mathsf{cps}(\Gamma \vdash \lambda x{:}X.\,M) &= \Lambda\alpha.\,\lambda k.\,k\ \mathcal{C}(\Gamma)\ (\Lambda\beta.\,\lambda k_1.\,\Lambda\varphi_x.\,\lambda x.\ (\langle a, \langle\vec{z}, t\rangle\rangle \mapsto \langle\langle\vec{z}, a\rangle, t\rangle)^*\\
&\qquad\qquad (\mathsf{cps}(\Gamma, x\colon X \vdash M)\ \beta\ k_1))\\
\mathsf{cps}(\Gamma \vdash M\ N) &= \Lambda\alpha.\,\lambda k.\,(\langle\vec{z}, s\rangle \mapsto \langle\vec{z}, \langle\vec{z}, s\rangle\rangle)^*\mathsf{cps}(\Gamma \vdash M)\ (\mathcal{C}(\Gamma) \times \alpha)\ t\\
&\qquad \text{where } t = (\Lambda\varphi.\,\lambda f.\ (\langle\varphi, \langle\vec{z}, s\rangle\rangle \mapsto \langle\vec{z}, \langle\varphi, s\rangle\rangle)^*\\
&\qquad\qquad \mathsf{cps}(\Gamma \vdash N)\ (\varphi \times \alpha)\ (\Lambda\beta.\,\lambda x.\,f\ \alpha\ k\ \beta\ x))\\
\mathsf{cps}(x\colon \mathbb{N}, y\colon \mathbb{N} \vdash \mathsf{add}(x, y)) &= \Lambda\alpha.\,\lambda k.\,(\langle\langle\langle\langle\rangle, m\rangle, n\rangle, s\rangle \mapsto \langle m + n, s\rangle)^*(k\ \mathsf{unit}\ \star)\\
\mathsf{cps}(x\colon \mathbb{N}, y\colon \mathbb{N} \vdash \mathsf{add}(y, x)) &= \Lambda\alpha.\,\lambda k.\,(\langle\langle\langle\langle\rangle, n\rangle, m\rangle, s\rangle \mapsto \langle m + n, s\rangle)^*(k\ \mathsf{unit}\ \star)\\
\mathsf{cps}(x\colon \mathbb{N} \vdash \mathsf{add}(x, n)) &= \Lambda\alpha.\,\lambda k.\,(\langle\langle\langle\rangle, m\rangle, s\rangle \mapsto \langle m + n, s\rangle)^*(k\ \mathsf{unit}\ \star)\\
\mathsf{cps}(x\colon \mathbb{N} \vdash \mathsf{add}(m, x)) &= \Lambda\alpha.\,\lambda k.\,(\langle\langle\langle\rangle, n\rangle, s\rangle \mapsto \langle m + n, s\rangle)^*(k\ \mathsf{unit}\ \star)\\
\mathsf{cps}(\vdash \mathsf{add}(m, n)) &= \Lambda\alpha.\,\lambda k.\,(\langle\langle\rangle, s\rangle \mapsto \langle m + n, s\rangle)^*(k\ \mathsf{unit}\ \star)\\
\mathsf{cps}(\Gamma \vdash \mathsf{if0}(x, M, N)) &= \Lambda\alpha.\,\lambda k.\,\mathsf{let}\ \langle k_1, k_2\rangle = \nabla^*_{\mathsf{nat}}(k\ \mathsf{unit}\ \star)\ \mathsf{in}\\
&\qquad\qquad (\langle\vec{z}, s\rangle \mapsto \mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{inl}(\langle\vec{z}, s\rangle)\ \mathsf{else}\ \mathsf{inr}(\langle\vec{z}, s\rangle))^*\\
&\qquad\qquad \langle\mathsf{cps}(\Gamma \vdash M)\ \alpha\ (\Lambda\varphi.\lambda x.k_1), \mathsf{cps}(\Gamma \vdash N)\ \alpha\ (\Lambda\varphi.\lambda x.k_2)\rangle\\
\mathsf{cps}(\Gamma \vdash \mathsf{if0}(0, M, N)) &= \mathsf{cps}(\Gamma \vdash M)\\
\mathsf{cps}(\Gamma \vdash \mathsf{if0}(n + 1, M, N)) &= \mathsf{cps}(\Gamma \vdash N)\\
\mathsf{cps}(\vdash \Omega_N) &= \Lambda\alpha.\,\lambda k.\,\Omega^*_{\mathsf{unit}}(k\ \mathsf{unit}\ \star)
\end{aligned}
$$

If Γ declares more than the free variables of M, let Δ be the subcontext declaring just the free variables of M, let $\vec{z}$ and $\vec{y}$ be the list of variables defined in Γ and Δ respectively and define $\mathsf{cps}(\Gamma \vdash M) = \Lambda\alpha.\,\lambda k.\,(\langle\vec{z}, s\rangle \mapsto \langle\vec{y}, s\rangle)^*(\mathsf{cps}(\Delta \vdash M)\ \alpha\ k)$.

Fig. 2. Call-by-Value CPS translation into LIN

is a code for the value of f and the second component is our initial value s (this follows from parametricity). We do not know how c encodes f, but we may use this code to apply the function to concrete arguments. We may pass $\langle n, \langle c, t\rangle\rangle\colon \overline{\mathsf{nat} \times (\varphi \times \beta)}$ to the second input port. What we get is the desired value $\langle f(n), t\rangle\colon \mathsf{nat} \times \beta$.

The values s and t are both returned unchanged. They are useful, as our simple low-level language cannot store values for later use. Instead of storing values, the values may be encoded in s and t and then decoded when these values are returned.

Having defined the types of the CPS-translation, we now come to the terms. In Fig. 2 we define the term $\mathsf{cps}(\Gamma \vdash M)$, where M is a source term and Γ is a finite list of source variable declarations $x_1\colon X_1, \ldots, x_n\colon X_n$ under which M is well-typed. In the figure we use the following notation. We choose a fresh type variable φ_x for each source variable x, write $\mathcal{C}(\Gamma)$ for the value type $\mathsf{unit} \times \mathcal{C}_{\varphi_{x_1}}(X_1) \times \cdots \times \mathcal{C}_{\varphi_{x_n}}(X_n)$, associated to the left, and $\mathcal{A}(\Gamma)$ for the LIN-context $x_1\colon \mathcal{A}_{\varphi_{x_1}}(X_1), \ldots, x_n\colon \mathcal{A}_{\varphi_{x_n}}(X_n)$. For a list of pairwise distinct variables $\vec{z}$, we also write $\vec{z}$ for tuples of variables, i.e. ε denotes $\langle\rangle$ and $\vec{z}, x$ denotes $\langle\vec{z}, x\rangle$. In Fig. 2 we have omitted type annotations in LIN-terms for better readability, as these will be uniquely determined by the types. The types of the

terms in that figure are specified by the following proposition, which one should keep in mind when reading the figure. This proposition is proved by induction on M.

Proposition 1. *If $\Gamma \vdash M : X$ in the source language with linearity restriction, then $\mathcal{A}(\Gamma) \vdash \mathsf{cps}(\Gamma \vdash M) : \mathcal{T}_{\mathcal{C}(\Gamma)}(X)$ in* LIN.

Let us now explain informally the CPS-translation of abstraction and application. Notice first that, by parametricity, values of type α and β cannot be inspected, but only be passed along. This is used in the translation to preserve certain values for later use.

For abstraction, consider a fully applied term of the form $\mathsf{cps}(\Gamma \vdash \lambda x{:}X.\, M)\ \alpha\ K$. It has type $(\mathcal{C}(\Gamma) \times \alpha)^\perp$ and it expects to be sent (the codes of) the values of its free variables and some arbitrary value s of type α. By definition, it sends this data unchanged to $(K\ \mathcal{C}(\Gamma)\ (\dots))$, which also has type $(\mathcal{C}(\Gamma) \times \alpha)^\perp$. But K expects as its first argument the type that encodes the function value and here we use $\mathcal{C}(\Gamma)$. This means that the tuple of the values of the free variables is now considered as the code of the function. The second argument to K is the access term that explains how this tuple can be used to apply the function to arguments. If we fully apply this second argument, then we obtain a term of type $(\mathcal{C}_{\varphi_x}(X) \times (\mathcal{C}(\Gamma) \times \beta))^\perp$. If we pass a value $\langle a, \langle \varphi, t\rangle\rangle$ to this term, then this value is transformed to $\langle\langle a, \varphi\rangle, t\rangle : \mathcal{C}(\Gamma,\, x{:}\, X) \times \beta$ and then passed to the CPS-translation of the function body M, as expected.

For application, consider $\mathsf{cps}(\Gamma \vdash M\ N)\ \alpha\ K$, which has type $(\mathcal{C}(\Gamma) \times \alpha)^\perp$. If we pass to this term the pair $\langle \vec{z}, s\rangle$, where $\vec{z}$ are the values of the free variables, then first $\langle \vec{z}, \langle \vec{z}, s\rangle\rangle$ is passed to the CPS-translation of M. When its evaluation is finished, it sends the value $\langle c, \langle \vec{z}, s\rangle\rangle$ to the continuation, where c is the code of the function value and the second component $\langle \vec{z}, s\rangle$ is returned unchanged. The CPS-translation of the application is defined such that then $\langle \vec{z}, \langle c, s\rangle\rangle$ is passed to the argument N. This causes the argument to be evaluated; it passes $\langle a, \langle c, s\rangle\rangle$ to the continuation, where a is the value of the function argument and the pair $\langle c, s\rangle$ is again returned unchanged. But the continuation is defined so that it just invokes the access term provided by the function M with the code c for the function and a for the argument. It thus invokes the program to perform the requested function application.

Example 2. We illustrate the CPS-translation and how it relates to low-level programs by translating the simple example term $\vdash (\lambda x{:}\mathbb{N}.\, \mathsf{add}(x, 5))\ 3 : \mathbb{N}$. In this example we let $\gamma := \mathsf{unit}$. In the following, α, β and φ are type variables and we have $\overline{\alpha} = \overline{\beta} = \overline{\varphi} = \mathsf{nat}$.

First we have

$$\mathsf{cps}(\vdash 3) = \Lambda\alpha.\, \lambda k.\, (\langle\langle\rangle, s\rangle \mapsto \langle 3, s\rangle)^*(k\ \mathsf{unit}\ \star) : \mathcal{T}_\gamma(\mathbb{N})$$
$$\mathsf{cps}(x : \mathbb{N} \vdash \mathsf{add}(x, 5)) = \Lambda\alpha.\, \lambda k.\, (\langle\langle\langle\rangle, m\rangle, s\rangle \mapsto \langle m + 5, s\rangle)^*(k\ \mathsf{unit}\ \star) : \mathcal{T}_\gamma(\mathbb{N})\ .$$

These terms translate to the following programs.

$$[\![\mathsf{cps}(\vdash 3)]\!] = \quad \gamma \times \overline{\alpha} \longrightarrow \boxed{\langle\langle\rangle, s\rangle \mapsto \langle 3, s\rangle} \longrightarrow \mathsf{nat} \times \overline{\alpha}$$
$$[\![\mathsf{cps}(x : \mathbb{N} \vdash \mathsf{add}(x, 5))]\!] = \quad (\gamma \times \mathsf{nat}) \times \overline{\alpha} \longrightarrow \boxed{\langle\langle\langle\rangle, x\rangle, s\rangle \mapsto \langle x + 5, s\rangle} \longrightarrow \mathsf{nat} \times \overline{\alpha}$$

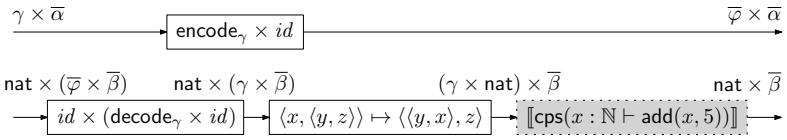
(The encoding and decoding programs arising from the type application with unit are the identity and have been removed.) Each of these programs takes a pair as an input.

The first component in this pair is the tuple of the code values for free variables of the term. The second component may be any value, which must be returned unchanged (by parametricity). A caller may use this second component to 'store' values across the call.

The translation of the abstraction $\mathsf{cps}(\vdash \lambda x{:}X.\,\mathsf{add}(x,5)) \colon \mathcal{T}_\gamma(\mathbb{N} \to \mathbb{N})$ is the term

$$\Lambda\alpha.\,\lambda k.\,k\;\gamma\;(\Lambda\beta.\,\lambda k_1.\,\Lambda\varphi_x.\,\lambda x.\;(\langle a,\langle\langle\rangle,t\rangle\rangle \mapsto \langle\langle\langle\rangle,a\rangle,t\rangle)^*\\ (\mathsf{cps}(x\colon \mathbb{N} \vdash \mathsf{add}(x,5))\;\beta\;k_1))$$

It translates to the following program, whose interface is as in Example 1.

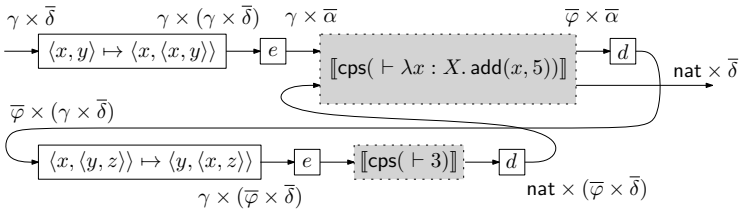


An input to the topmost input port corresponds to a request to compute the value of the function. It takes an argument of type $\gamma \times \overline{\alpha}$, whose first component is the tuple of the free values of the term (in this case none). The return value $\overline{\varphi} \times \overline{\alpha}$ must be the pair of the code for the function together and the value of type $\overline{\alpha}$ that was given as input. As the code for the function, the translation uses the (encoding of) tuple of the codes of the values of its free variables.

The second input port allows function application. It takes as input the function argument (of type nat) and the code for the function (of type $\overline{\varphi}$). Its third argument is again an arbitrary value that must be returned unchanged. To compute the function application, the program first constructs the tuple of (the codes of) the values of free variables of the body of the λ-term. It can do so, since the code for a function is just the tuple of (the codes of) its free variables, and the function argument value is also supplied. The program then just invokes the program for the body of the λ-term.

Notice the similarity to the implementation of functions using defunctionalization, such as [6]. Functions are represented by first order values and there is static program code for application of any function.

Finally, the CPS-translation of the application $\mathsf{cps}(\vdash (\lambda x{:}X.\,\mathsf{add}(x,5))\;3)$ is given by the term $\Lambda\delta.\,\lambda k.\,(\langle\langle\rangle,s\rangle \mapsto \langle\langle\rangle,\langle\langle\rangle,s\rangle\rangle)^*\mathsf{cps}(\vdash \lambda x{:}X.\,\mathsf{add}(x,5))\;(\gamma \times \delta)\;t$, where t abbreviates $(\Lambda\varphi.\,\lambda f.\,(\langle x,\langle\langle\rangle,s\rangle\rangle \mapsto \langle\langle\rangle,\langle x,s\rangle\rangle)^*\mathsf{cps}(\vdash 3)\;(\varphi \times \delta)\;(\Lambda\beta.\,\lambda x.\,f\;\delta\;k\;\beta\;x))$. This term translates to the following low-level program, in which each e abbreviates $(id \times \mathsf{encode}_A)$, i.e. $(\langle x,y\rangle \mapsto \langle x,\mathsf{encode}_A(y)\rangle)$ for an appropriate type A. Likewise, each d abbreviates $(id \times \mathsf{decode}_A)$ for an appropriate A.



In this program, the value of type γ is duplicated, as it is needed twice, once for the evaluation of the function and once for its argument. The copy is encoded in the callee-save value and the function value is computed. When this it is returned, the value of

type γ is restored, as it is needed in general to compute the function argument. While computing the function argument, the code of the function is encoded and put in the callee-save value. When the argument value is returned, the function code is decoded and both argument and function code are passed to the program for application.

This outlines how one can understand the translation on the level of low-level programs. On input $\langle\langle\rangle, s\rangle$, the above program outputs $\langle 8, s\rangle$, as expected.

One may also understand the program directly in LIN using the equations from Section 3.1. By using only the β-equations from Lemma 3, we get

$$\begin{aligned}&\mathsf{cps}(\vdash (\lambda x{:}X.\,\mathsf{add}(x,5))\ 3)\ \alpha\ k = (\langle\langle\rangle, s\rangle \mapsto \langle\langle\rangle, \langle\langle\rangle, s\rangle\rangle)^*\mathsf{cps}(\vdash \lambda x{:}X.\,\mathsf{add}(x,5))\ (\gamma\times\alpha)\ t\\ &= \cdots = (\langle\langle\rangle, s\rangle \mapsto \langle\langle\rangle, \langle\langle\rangle, s\rangle\rangle)^*(\langle\varphi, \langle\langle\rangle, s_1\rangle\rangle \mapsto \langle\langle\rangle, \langle\varphi, s_1\rangle\rangle)^*(\langle\langle\rangle, s_2\rangle \mapsto \langle 3, s_2\rangle)^*\\ &\qquad(\langle a, \langle\langle\rangle, t\rangle\rangle \mapsto \langle\langle\langle\rangle, a\rangle, t\rangle)^*(\langle\langle\langle\rangle, m\rangle, s_3\rangle \mapsto \langle m+5, s_3\rangle)^*(k\ \mathsf{unit}\ \star)\end{aligned}$$

Using Lemma 4, this term can be simplified to $(\langle\langle\rangle, s\rangle \mapsto \langle 8, s\rangle)^*(k\ \mathsf{unit}\ \star)$, which also shows that the term behaves as expected.

4.3 Soundness

In this section we show that correctness of the refined CPS-translation can be shown much in the same way as for the original CPS-translation. We show:

Theorem 1. *If $\vdash M : \mathbb{N}$ and $M \longrightarrow^* n$, where n is a value, then $\mathsf{cps}(\vdash M)\ \mathsf{unit}\ K = (\langle\vec{z}, s\rangle \mapsto \langle n, s\rangle)^*(K\ \mathsf{unit}\ \star)$ for any closed continuation $K : \mathcal{K}_{\mathsf{unit}}(\mathbb{N})$.*

Corollary 1. *If $\vdash M : \mathbb{N}$ then $[\![\mathsf{cps}(\vdash M)\ \mathsf{unit}]\!]$ is, up to isomorphism, a program of type $\mathsf{unit} \to \mathsf{nat}$ with the property that it maps $\langle\rangle$ to n if $M \longrightarrow^* n$ holds.*

Proof. We have $[\![\mathsf{cps}(\vdash M)\ \mathsf{unit}]\!] : (0+0) + \mathsf{unit}\times\mathsf{unit} \to (0 + \mathsf{nat}\times\mathsf{unit}) + 0$ by definition and the type is isomorphic to $\mathsf{unit} \to \mathsf{nat}$.

If we choose K to be $\Lambda\varphi.\,\lambda a.\,(x{:}\mathsf{nat} \mapsto \mathsf{let}\ \_{=}\mathsf{print}(x)\ \mathsf{in}\ \Omega)$, then Theorem 1 tells us that $[\![\mathsf{cps}(\vdash M)\ \mathsf{unit}\ K]\!]$ is a program of type $\mathsf{unit}\times\mathsf{unit} \to 0$ that on input $\langle\langle\rangle, \langle\rangle\rangle$ prints n if and only if M reduces to n. By definition of the translation from LIN to low-level programs, this means that $[\![\mathsf{cps}(\vdash M)\ \mathsf{unit}]\!]$ must map $\mathsf{inr}(\langle\langle\rangle, \langle\rangle\rangle)$ to $\mathsf{inl}(\mathsf{inr}(\langle n, \langle\rangle\rangle))$, from which the result follows. □

We now come to the proof of Theorem 1, which follows that of Plotkin [19]. This is an important point of this paper; it shows that with little adaptation, existing proof methods can be applied to a translation going directly into first-order code.

For any source value V with $\Gamma \vdash V : X$, we define a value type $\mathcal{P}(\Gamma \vdash V)$, a low-level value $\eta(V)$ of type $\mathcal{C}_{\mathcal{P}(\Gamma\vdash V)}(X)$ and a LIN-term $\Psi(\Gamma \vdash V)$ by

$$\begin{aligned}\eta(x) &= x & \mathcal{P}(\Gamma\vdash x) &= \varphi_x & \Psi(\Gamma\vdash x) &= x\\ \eta(n) &= n & \mathcal{P}(\Gamma\vdash n) &= \mathtt{unit} & \Psi(\Gamma\vdash n) &= \star\\ \eta(\lambda x{:}X.\,t) &= \vec{y} & \mathcal{P}(\Gamma\vdash\lambda x{:}X.\,M) &= \mathcal{C}(\Delta)\end{aligned}$$
$$\Psi(\Gamma\vdash\lambda x{:}X.\,t) = \Lambda\beta.\,\lambda k_1.\,\Lambda\varphi_x.\,\lambda x.\,\mathsf{cps}(\Delta,\ x{:}\ X\vdash t)\ \beta\ k_1,$$

where in the last two lines Δ is the context obtained from Γ by deleting all declarations of variables that are not free in $\lambda x{:}X.\,M$ and $\vec{y}$ is the tuple of variables declared in Δ. The notation Ψ comes from [19].

We shall often write just $\mathcal{P}(M)$ and $\Psi(M)$, when a typing context Γ for M is clear from the context. The point of these definitions is the following lemma.

Lemma 7. $\mathsf{cps}(\Gamma \vdash V) = \Lambda\alpha.\, \lambda k.\, (\langle \vec{z}, s\rangle \mapsto \langle \eta(V), s\rangle)^*(k\ \mathcal{P}(V)\ \Psi(\Gamma \vdash V))$ *for any value* V.

Lemma 8. *Let M be a source term that is well-typed in context Γ. Let Δ be a reordering of the context Γ. Let $\vec{z}$ and $\vec{y}$ be the lists of variables declared in Γ and Δ respectively. Then* $\mathsf{cps}(\Gamma \vdash M) = \Lambda\alpha.\, \lambda k.\, (\langle \vec{z}, s\rangle \mapsto \langle \vec{y}, s\rangle)^*(\mathsf{cps}(\Delta \vdash M)\ \alpha\ k)$.

Lemma 9. *For any value V, any α and any closed K of the appropriate type, we have*

$$\begin{aligned}&\mathsf{cps}(\Gamma \vdash M[V/x])\ \alpha\ K = \\ &\quad (\langle \vec{z}, s\rangle \mapsto \langle\langle \vec{z}, \eta(V)\rangle, s\rangle)^*(\mathsf{cps}(\Gamma,\, x\colon X \vdash M)[\Psi(V)/x, \mathcal{P}(V)/\varphi_x]\ \alpha\ K)\ .\end{aligned}$$

Proof. The proof goes by induction on M. The most difficult case is that when M is a λ-abstraction $\lambda y{:}Y.\, N$. We show the sub-case where $x \neq y$ and where y is free in N.

$$\begin{aligned}
&(\langle \vec{z}, s\rangle \mapsto \langle\langle \vec{z}, \eta(V)\rangle, s\rangle)^*(\mathsf{cps}(\Gamma,\, x\colon X \vdash \lambda y.\, N)[\Psi(V)/x, \mathcal{P}(V)/\varphi_x]\ \alpha\ K) \\
&= (\langle \vec{z}, s\rangle \mapsto \langle\langle \vec{z}, \eta(V)\rangle, s\rangle)^* \\
&\quad (K\ \varphi_{\vec{z},x}\ (\Lambda\beta.\, \lambda k_1.\, \Lambda\varphi_y.\, \lambda y.\, \mathsf{cps}(\Gamma,\, x\colon X,\, y\colon Y \vdash N)[\Psi(V)/x, \mathcal{P}(V)/\varphi_x]\ \beta\ k_1)) \\
&= (\langle \vec{z}, s\rangle \mapsto \langle\langle \vec{z}, \eta(V)\rangle, s\rangle)^* && \text{(Lemma 9)} \\
&\quad (K\ \varphi_{\vec{z},x}\ (\Lambda\beta.\, \lambda k_1.\, \Lambda\varphi_y.\, \lambda y.\ (\langle\langle\langle \vec{z}, x\rangle, y\rangle, s\rangle \mapsto \langle\langle\langle \vec{z}, y\rangle, x\rangle, s\rangle)^* \\
&\qquad\qquad (\mathsf{cps}(\Gamma,\, y\colon Y,\, x\colon X \vdash N)[\Psi(V)/x, \mathcal{P}(V)/\varphi_x]\ \beta\ k_1))) \\
&= K\ \varphi_{\vec{z}}\ (\Lambda\beta.\, \lambda k_1.\, \Lambda\varphi_y.\, \lambda y.\ (\langle\langle \vec{z}, a\rangle, s\rangle \mapsto \langle\langle \vec{z}, \eta(V)\rangle, a\rangle, s\rangle)^* && \text{(Lemma 6)} \\
&\qquad\qquad (\langle\langle\langle \vec{z}, x\rangle, y\rangle, s\rangle \mapsto \langle\langle\langle \vec{z}, y\rangle, x\rangle, s\rangle)^* \\
&\qquad\qquad (\mathsf{cps}(\Gamma,\, y\colon Y,\, x\colon X \vdash N)[\Psi(V)/x, \mathcal{P}(V)/\varphi_x]\ \beta\ k_1)) \\
&= K\ \varphi_{\vec{z}}\ (\Lambda\beta.\, \lambda k_1.\, \Lambda\varphi_y.\, \lambda y.\ (\langle\langle \vec{z}, a\rangle, s\rangle \mapsto \langle\langle \vec{z}, a\rangle, \eta(V)\rangle, s\rangle)^* \\
&\qquad\qquad (\mathsf{cps}(\Gamma,\, y\colon Y,\, x\colon X \vdash N)[\Psi(V)/x, \mathcal{P}(V)/\varphi_x]\ \beta\ k_1)) \\
&= K\ \varphi_{\vec{z}}\ (\Lambda\beta.\, \lambda k_1.\, \Lambda\varphi_y.\, \lambda y.\, \mathsf{cps}(\Gamma,\, y\colon Y \vdash |N[V/x]|)\ k_1) && \text{(IH)} \\
&= \mathsf{cps}(\Gamma \vdash \lambda y{:}Y.\, N[V/x])\ \alpha\ K && \square
\end{aligned}$$

To prove correctness of the CPS-translation, Plotkin defines a term $M{:}K$, which is the term that one gets from $\mathsf{cps}(M)\ K$ by reduction of administrative redexes. We adapt this definition to the current situation and define $M :^{\Gamma}_{A} K$ as follows.

$$\begin{aligned}
N :^{\Gamma}_{\alpha} K &= (\langle \vec{z}, s\rangle \mapsto \langle \eta(N), s\rangle)^*(K\ \mathcal{P}(N)\ \Psi(N)) && (N \text{ is closed value}) \\
M\ N :^{\Gamma}_{\alpha} K &= (\langle \vec{z}, s\rangle \mapsto \langle \vec{z}, \langle \vec{z}, s\rangle\rangle)^* && (M \text{ is not a value}) \\
&\quad (M :^{\Gamma}_{\alpha} (\Lambda\varphi.\, \lambda f.\ (\langle \varphi, \langle \vec{z}, s\rangle\rangle \mapsto \langle \vec{z}, \langle \varphi, s\rangle\rangle)^* \\
&\qquad\qquad \mathsf{cps}(\Gamma \vdash N)\ (\varphi \times \alpha)\ (\Lambda\beta.\, \lambda x.\, (f\ \alpha\ K\ \beta\ x))) \\
M\ N :^{\Gamma}_{\alpha} K &= (\langle \vec{z}, s\rangle \mapsto \langle \vec{z}, \langle \eta(M), s\rangle\rangle)^* && (M \text{ is a value, } N \text{ is not a value}) \\
&\quad (N :^{\Gamma}_{\mathcal{P}(M)\times\alpha} (\Lambda\beta.\, \lambda x.\, (\Psi(M)\ \alpha\ K\ \beta\ x))) \\
M\ N :^{\Gamma}_{\alpha} K &= (\langle \vec{z}, s\rangle \mapsto \langle \eta(N), \langle \eta(M), s\rangle\rangle)^* && (M \text{ and } N \text{ are values}) \\
&\quad (\Psi(M)\ \alpha\ K\ \mathcal{P}(N)\ \Psi(N)) \\
\mathsf{if0}(0, M, N) :^{\Gamma}_{\alpha} K &= M :^{\Gamma}_{\alpha} K \qquad \mathsf{if0}(n+1, M, N) :^{\Gamma}_{\alpha} K = N :^{\Gamma}_{\alpha} K \\
\mathsf{add}(m, n) :^{\Gamma}_{\alpha} K &= (m + n) :^{\Gamma}_{\alpha} K
\end{aligned}$$

We show the following lemma by case distinction on M.

Lemma 10. *Let $\Gamma \vdash M : X$ in the source language. Then, for all α and all closed K such that $\mathsf{cps}(\Gamma \vdash M)\ \alpha\ K$ is well-typed, we have $\mathsf{cps}(\Gamma \vdash M)\ \alpha\ K = M :^{\Gamma}_{\alpha} K$.*

Lemma 11. *If $M \longrightarrow N$ then $M :^{\Gamma}_{\alpha} K = N :^{\Gamma}_{\alpha} K$ for any α and closed K of the appropriate type.*

Proof. We show the representative case where M is $(\lambda x.M_1)\ M_2$, M_2 is a value and x is free in M_1. Let Δ be the subcontext of Γ defining just the free variables of $(\lambda x.M_1)$.

$$
\begin{aligned}
&((\lambda x{:}X.\, M_1)\ M_2) :^{\Gamma}_{\alpha} K \\
&= (\langle \vec{z}, s\rangle \mapsto \langle\langle \eta(\lambda x.\, M_1), \eta(M_2)\rangle, s\rangle)^*(\Psi(\lambda x{:}X.\, M_1)\ \alpha\ K\ \mathcal{P}(M_2)\ \Psi(M_2)) \\
&= (\langle \vec{z}, s\rangle \mapsto \langle\langle \eta(\lambda x.\, M_1), \eta(M_2)\rangle, s\rangle)^* \\
&\quad ((\Lambda\beta.\, \lambda k'.\, \Lambda\varphi_x.\, \lambda x.\, \mathsf{cps}(\Delta, x\colon X \vdash M_1)\ \beta\ k')\ \alpha\ K\ \mathcal{P}(M_2)\ \Psi(M_2)) \\
&= (\langle \vec{z}, s\rangle \mapsto \langle\langle \eta(\lambda x.\, M_1), \eta(M_2)\rangle, s\rangle)^*(\mathsf{cps}(\Delta, x\colon X \vdash M_1)\ \alpha\ K)[\Psi(M_2)/x, \mathcal{P}(M_2)/\varphi_x] \\
&= (\langle \vec{z}, s\rangle \mapsto \langle\langle \vec{z}, \eta(M_2)\rangle, s\rangle)^*(\ (\langle\langle \vec{z}, x\rangle, s'\rangle \mapsto \langle\langle \eta(\lambda x.M_1), x\rangle, s'\rangle)^* \\
&\qquad\qquad (\mathsf{cps}(\Delta, x\colon X \vdash M_1)\ \alpha\ K))[\Psi(M_2)/x, \mathcal{P}(M_2)/\varphi_x] \\
&= \mathsf{cps}(\Gamma \vdash M_1[M_2/x])\ \alpha\ K = M_1[M_2/x] :^{\Gamma}_{\alpha} K
\end{aligned}
$$

□

Theorem 1 is now a direct consequence of Lemmas 10 and 11. Each reduction step in the source language can be followed by an equality in LIN and, for a value n, the definition of $n :_{\alpha} K$ is just as required.

The soundness argument shows that we can trace reduction steps in the source language with equalities in LIN. We are not aware of such results for direct translations from source to a first-order target language, e.g. for defunctionalization compilation.

4.4 On Resource Usage

Let us come back to the issue of memory space usage that has motivated this work. Consider a source program of the form $\mathsf{let}\ x_1{=}M_1\ \mathsf{in}\ \mathsf{let}\ x_2{=}M_2\ \mathsf{in}\ \ldots\ \mathsf{let}\ x_k{=}M_k\ \mathsf{in}\ N$, where $(\mathsf{let}\ x{=}M\ \mathsf{in}\ N)$ abbreviates $(\lambda x{:}X.\, N)\ M$. The refined call-by-value CPS-translation is defined such that the values of the terms $M_1, \ldots, M_k$ are computed in this order in the low-level language. Moreover, by construction, at the time when the computation of M_i starts, only values of variables are being stored that are free in N or some M_j with $j \geq i$. In particular, the value of each x_i is only kept as long as it is needed. This means that the proposed CPS-translation solves the issue discussed with an INTML-example in the Introduction.

Take the concrete example $M = \mathsf{let}\ x{=}5\ \mathsf{in}\ \mathsf{let}\ y{=}x+1\ \mathsf{in}\ \mathsf{let}\ z{=}y+4\ \mathsf{in}\ z+3$ from the introduction. The program $[\![M]\!]$ maps $\langle\langle\rangle, s\rangle$ to $\langle 13, s\rangle$. The following table traces the computation through this program. In the two steps where values are discarded, the case at the bottom of Figure 2 applies. For example, the definition of the CPS-translation $\mathsf{cps}(x\colon \mathbb{N},\ y\colon \mathbb{N} \vdash \mathsf{let}\ z{=}y+4\ \mathsf{in}\ z+3)$ is just

$$(\langle\langle\langle\langle\rangle, y\rangle, z\rangle, s\rangle \mapsto \langle\langle\langle\rangle, z\rangle, s\rangle)^*\mathsf{cps}(y\colon \mathbb{N} \vdash \mathsf{let}\ z{=}y+4\ \mathsf{in}\ z+3),$$

which explains that the value 5 is discarded from line three to four.

Value at entry	Subprogram
$\langle\langle\rangle, s\rangle$	$[\![\mathsf{cps}(\ \vdash \mathsf{let}\ x{=}5\ \mathsf{in}\ \mathsf{let}\ y{=}x+1\ \mathsf{in}\ \mathsf{let}\ z{=}y+4\ \mathsf{in}\ z+3)]\!]$
$\langle\langle\langle\rangle, 5\rangle, s\rangle$	$[\![\mathsf{cps}(x\colon \mathbb{N} \vdash \mathsf{let}\ y{=}x+1\ \mathsf{in}\ \mathsf{let}\ z{=}y+4\ \mathsf{in}\ z+3)]\!]$
$\langle\langle\langle\langle\rangle, 5\rangle, 6\rangle, s\rangle$	$[\![\mathsf{cps}(x\colon \mathbb{N},\ y\colon \mathbb{N} \vdash \mathsf{let}\ z{=}y+4\ \mathsf{in}\ z+3)]\!]$
$\langle\langle\langle\rangle, 6\rangle, s\rangle$	$[\![\mathsf{cps}(y\colon \mathbb{N} \vdash \mathsf{let}\ z{=}y+4\ \mathsf{in}\ z+3)]\!]$
$\langle\langle\langle\langle\rangle, 6\rangle, 10\rangle, s\rangle$	$[\![\mathsf{cps}(y\colon \mathbb{N}, z\colon \mathbb{N} \vdash z+3)]\!]$
$\langle\langle\langle\rangle, 10\rangle, s\rangle$	$[\![\mathsf{cps}(z\colon \mathbb{N} \vdash z+3)]\!]$

5 Call-by-Value

In this section we observe that the linearity restriction on the source language can be lifted if the intermediate language allows contraction. We describe a simple way of extending LIN with contraction by adding exponentials $!X$. We believe that for applications in compilation it would be better to use the more fine-grained *subexponentials* $A \cdot X$ of INTML [7], see [24, §6] for a discussion. For lack of space and for simplicity, we consider just exponentials here.

A simple way of expressing this is by defining an intermediate language EXP, which extends LIN with a type $!X$ and the following typing rules.

$$\frac{\Gamma, x\colon X \vdash t\colon Y}{\Gamma, x\colon !X \vdash t\colon Y} \qquad \frac{\Gamma, x\colon !X, y\colon !X \vdash t\colon Y}{\Gamma, x\colon !X \vdash t[x/y]\colon Y} \qquad \frac{\Gamma \vdash s\colon !X \multimap Y \qquad !\Delta \vdash t\colon X}{\Gamma, !\Delta \vdash s\ t\colon Y}$$

The translation of LIN to the low-level language can be extended to EXP by choosing $(!X)^- = \mathsf{nat} \times X^-$ and $(!X)^+ = \mathsf{nat} \times X^+$ and by interpreting the exponential rules in a standard way [1].

The definition of equality is extended from LIN to EXP by letting $[\![!X]\!]_\rho$ be the smallest relation such that $q\ [\![X]\!]_\rho\ q'$ implies $(id_{\mathsf{nat}} \times q)\ [\![!X]\!]_\rho\ (id_{\mathsf{nat}} \times q')$. The results from Section 3.1 remain true.

This extension of LIN to EXP suffices to translate the full source language. Define $\mathcal{C}_\varphi(X)$, $\mathcal{A}_\varphi(X)$, $\mathcal{K}_\alpha(X)$ and $\mathcal{T}_\varphi(X)$ just as in Section 4, but with $!(-) \multimap (-)$ instead of $(-) \multimap (-)$. The CPS-translation of terms to EXP is defined exactly as in Fig. 2.

Proposition 2. *If $\Gamma \vdash M\colon X$ in the source language without the linearity restriction, then $!\mathcal{A}(\Gamma) \vdash \mathsf{cps}(\Gamma \vdash M)\colon \mathcal{T}_{\mathcal{C}(\Gamma)}(X)$ in* EXP.

Theorem 2. *If $\vdash M\colon \mathbb{N}$ in the source language without the linearity restriction and $M \longrightarrow^* n$, where n is a value, then $\mathsf{cps}(\vdash M)\ \mathsf{unit}\ K = (\langle \vec{z}, s\rangle \mapsto \langle n, s\rangle)^*(K\ \mathsf{unit}\ \star)$ for any closed continuation $K\colon \mathcal{K}_{\mathsf{unit}}(\mathbb{N})$.*

We end this section by noting that with contraction in the source language we can define a term $\mathsf{if0}(V, M, N)$ for M and N of any source type X. The definition goes by induction on X. At function type $X_1 \to X_2$ one defines $\lambda x{:}X_1.\, \mathsf{if0}(V, M\ x, N\ x)$, where the body is obtained from the induction hypothesis.

This definition may look undesirable for compilation, as the case distinction is performed every time the function is invoked. But consider what happens to terms of

the form $\mathsf{if0}(V, (\lambda x{:}\mathbb{N}.\, M), (\lambda x{:}\mathbb{N}.\, N))$ in defunctionalizing compilers, such as [6]. The two abstractions would be represented by two constructors, $C_1(\dots)$ and $C_2(\dots)$, say. The function value of the whole term could be either $C_1(\dots)$ or $C_2(\dots)$. When this function is applied, a case distinction over the two possible constructors is performed in order to execute the correct function, so a similar case distinction is performed here. It seems that is a certain amount of case distinction is unavoidable in order to find the right code to execute. The precise properties of the above definition of case distinction on higher types remain to be investigated, however.

6 Conclusion

We have given a new interpretation of call-by-value in an intermediate language that represents the structure of an interactive computation model constructed from a first-order low-level language. The interpretation makes low-level details explicit that are interesting for compilation, such as which values are stored at any point in the execution of the resulting low-level programs. The interpretation is motivated by game semantic models [3,14] which however abstract away some important (for us) low-level details.

We have shown that well-known logical tools, such as parametric polymorphism, are useful managing the low level details of the interpretation. We have formulated these principles in terms of a simple intermediate language LIN. Its use has simplified the handling of encoding details, for example in the encoding of function values. The utility of studying intermediate languages like LIN is also demonstrated by the fact that the soundness proof of the refined CPS-translation presented here can be carried out by a refinement of the standard proof. While it would certainly be possible to define the translation to the low-level language in a single step, it is not immediate how to track reduction on source terms on the level of compiled low-level programs. We hope that other logical tools will find use in the context of low level languages.

We should remark that for call-by-name a thunkifying translation [13] into EXP is possible. In essence, this is the translation of INTML [7]. Indeed, INTML integrates a call-by-name CPS-translation, see [23], that can be factored through EXP.

For further work, we are optimistic that it will be possible to extend the approach to more expressive source languages, e.g. with recursion. We should also like to clarify the relation to defunctionalizing compilation methods for call-by-value languages, such as the very successful [6].

In another direction, it should also be interesting to apply the results to the resource usage analysis of call-by-value languages. For instance, one may consider the design of a call-by-value functional language for LOGSPACE computation, using an approach similar to that of INTML [7] for call-by-name. To remain within LOGSPACE one would need to restrict nat in the low-level language. We believe that this can be done if one replaces $\forall\alpha.\, X$ in LIN by bounded quantification $\forall\alpha \lhd A.\, X$ using the ideas from [21]. One may also consider such bounded quantification to reduce the amount of encoding operations into nat arising from the low-level implementation of universal quantification. The situation should be similar that of exponentials and subexponentials [24, §6].

Acknowledgments. I am grateful for support by the Fondation Sciences Mathématiques de Paris as part of the PROOFS program.

References

1. Abramsky, S., Haghverdi, E., Scott, P.: Geometry of interaction and linear combinatory algebras. Mathematical Structures in Computer Science **12**(5) (2002) 625–665
2. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. Comput. **163**(2) (2000) 409–470
3. Abramsky, S., McCusker, G.: Call-by-value games. In Nielsen, M., Thomas, W., eds.: CSL. Volume 1414 of LNCS., Springer (1997) 1–17
4. Appel, A.W.: Compiling with Continuations. Cambridge University Press (2006)
5. Banerjee, A., Heintze, N., Riecke, J.G.: Design and correctness of program transformations based on control-flow analysis. In Kobayashi, N., Pierce, B.C., eds.: TACS. Volume 2215 of LNCS., Springer (2001) 420–447
6. Cejtin, H., Jagannathan, S., Weeks, S.: Flow-directed closure conversion for typed languages. In Smolka, G., ed.: ESOP. Volume 1782 of LNCS., Springer (2000) 56–71
7. Dal Lago, U., Schöpp, U.: Functional programming in sublinear space. In Gordon, A.D., ed.: ESOP. Volume 6012 of LNCS., Springer (2010) 205–225
8. Fischer, M.J.: Lambda calculus schemata. SIGACT News (14) (January 1972) 104–109
9. Fredriksson, O., Ghica, D.R.: Abstract machines for game semantics, revisited. In: LICS, IEEE (2013) 560–569
10. Ghica, D.R.: Geometry of synthesis: a structured approach to VLSI design. In Hofmann, M., Felleisen, M., eds.: POPL, ACM (2007) 363–375
11. Ghica, D.R., Smith, A., Singh, S.: Geometry of synthesis IV: compiling affine recursion into static hardware. In Chakravarty, M.M.T., Hu, Z., Danvy, O., eds.: ICFP. (2011) 221–233
12. Harper, R., Lillibridge, M.: Polymorphic type assignment and cps conversion. Lisp and Symbolic Computation **6**(3-4) (1993) 361–380
13. Hatcliff, J., Danvy, O.: Thunks and the lambda-calculus. J. Funct. Program. **7**(3) (1997) 303–319
14. Honda, K., Yoshida, N.: Game-theoretic analysis of call-by-value computation. Theor. Comput. Sci. **221**(1-2) (1999) 393–456
15. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. Math. Proc. Cambridge Philos. Soc. **119**(3) (1996) 447–468
16. Laird, J.: Game semantics and linear cps interpretation. Theor. Comput. Sci. **333**(1-2) (2005) 199–224
17. Melliès, P.A.: Game semantics in string diagrams. In: LICS, IEEE (2012) 481–490
18. Melliès, P.A., Tabareau, N.: Resource modalities in game semantics. In: LICS. (2007) 389–398
19. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. **1**(2) (1975) 125–159
20. Plotkin, G.D., Abadi, M.: A logic for parametric polymorphism. In Bezem, M., Groote, J.F., eds.: TLCA. Volume 664 of LNCS., Springer (1993) 361–375
21. Schöpp, U.: Stratified bounded affine logic for logarithmic space. In: LICS. (2007) 411–420
22. Schöpp, U.: Computation-by-interaction with effects. In Yang, H., ed.: APLAS. Volume 7078 of LNCS., Springer (2011) 305–321
23. Schöpp, U.: On interaction, continuations and defunctionalization. In Hasegawa, M., ed.: TLCA. Volume 7941 of LNCS., Springer (2013) 205–220
24. Schöpp, U.: Organising low-level programs using higher types. In PPDP 2014, to appear (2014)
25. Selinger, P.: A survey of graphical languages for monoidal categories. In: New Structures for Physics. Volume 813 of Lecture Notes in Physics., Springer (2011) 289–355
26. Shao, Z., Appel, A.W.: Efficient and safe-for-space closure conversion. ACM Trans. Program. Lang. Syst. **22**(1) (2000) 129–161
27. Wadler, P.: Theorems for free! In: FPCA. (1989) 347–359