On Interaction, Continuations and Defunctionalization[⋆]

Ulrich Schöpp

Ludwig-Maximilians-Universität München, Germany
Ulrich.Schoepp@ifi.lmu.de

Abstract. In game semantics and related approaches to programming language semantics, programs are modelled by interaction dialogues. Such models have recently been used in the design of new compilation methods, e.g. in Ghica's approach to hardware synthesis, or in joint work with Dal Lago on programming with sublinear space. This paper relates such semantically motivated non-standard compilation methods to more standard techniques in the compilation of functional programming languages, such as continuation passing and defunctionalization. We first show for the linear λ-calculus that interpretation in a model of computation by interaction can be described as a call-by-name CPS transformation followed by a defunctionalization procedure that takes into account control-flow information. We then use the interactive model to guide the extension of the compositional translation to source languages with full contraction and recursion.

1 Introduction

A successful approach in the semantics of programming languages is to model programs by interaction dialogues [11, 1]. Although dialogues are usually considered as abstract mathematical objects, it has also been argued that they are useful for *implementing* actual computation. Dialogues have been found useful especially for resource bounded computation, where they have given rise to nonstandard compilation methods for functional programming languages. For example Ghica et al. have developed methods for hardware synthesis based on game semantics [7]. A related semantic approach based on the *Int construction* [12] has been the used to design a functional programming language for sublinear space computation [4].

The aim of this paper is to relate such compilation methods based on interactive semantics to standard techniques in the compilation of functional programming languages. We consider the compilation of higher-order languages, such as PCF. A compiler would transform such a language to machine code by way of a number of intermediate languages. Typically, the higher-order source code would first be translated to first-order intermediate code, e.g. [3], from which the machine code is then generated. This paper is concerned with the first step, the translation from higher-order to first-order code. We consider two particular instances of well-known transformations that find application in compilers, CPS translation [17] and defunctionalization [18], and show that their composition is closely related to an interpretation of the source language in an interactive model of computation.

⋆ I thank the anonymous referees for their helpful feedback and suggestions.

This represents one step towards a general goal of capturing program transformations that find use in compilers by means of universal mathematical constructions. The interactive model that corresponds to CPS translation and defunctionalization is constructed using the Int construction [12], which captures a canonical way of constructing a model of higher-order computation from a first-order one. As this model validates call-by-name, it is natural to consider a call-by-name CPS translation; we use a variant of the one of Hofmann and Streicher [10].

To give an outline of how CPS translation, defunctionalization and the interpretation in an interactive model are related, we consider the very simple example of a function that increments a natural number: $\lambda x : \mathbb{N}.\, 1 + x$. We next outline how this function is translated by the two approaches and how the results compare.

1.1 CPS Translation and Defunctionalization

A compiler for PCF might first transform $\lambda x : \mathbb{N}.\, 1 + x$ into continuation passing style, perhaps apply some optimisations, and then use defunctionalization to obtain a first-order intermediate program, ready for compilation to machine language.

Hofmann and Streicher's call-by-name CPS transform [10] translates the source term $\lambda x : \mathbb{N}.\, 1 + x$ to the term $\lambda \langle x, k\rangle.\, (\lambda k.k\ 1)\ (\lambda u.x\ (\lambda n.k\ (u+n))) \colon \neg(\neg\neg\mathbb{N} \times \neg\mathbb{N})$, where we write $\neg A$ for $A \to \bot$, as usual. The argument in this term is a pair $\langle x, k\rangle$ of a continuation $k \colon \neg\mathbb{N}$ that accepts the result and a variable $x \colon \neg\neg\mathbb{N}$ that supplies the function argument. To obtain the actual function argument, one applies x to a continuation (here $\lambda n.k\ (u+n)$) to ask for the actual argument to be thrown into the supplied continuation.

Defunctionalization [18] translates this higher-order term into a first-order one. The basic idea is to give each function a name and to pass around not the function itself, but only its name and the values of its free variables. To this end, each lambda abstraction is named with a label: $\lambda^{l_1} \langle x, k\rangle.\, (\lambda^{l_2} k.k\ 1)\ (\lambda^{l_3} u.x\ (\lambda^{l_4} n.k\ (u+n)))$. The whole term can be represented simply by the label l_1. The function l_3 has free variables x and k and is represented by the label together with the values of x and k, which we write as $l_3(x,k)$.

Each application $s\ t$ is replaced by a procedure call $\mathit{apply}(s,t)$, as s is now only a function name and not the function itself. The procedure *apply* is defined by case distinction on the function name and behaves like the body of the respective λ-abstraction in the original term. In the example we have the following definition of *apply*:

$$\begin{aligned} \mathit{apply}(l_1, \langle x,k\rangle) &= \mathit{apply}(l_2, l_3(x,k)) & \mathit{apply}(l_2,k) &= \mathit{apply}(k,1) \\ \mathit{apply}(l_3(x,k),u) &= \mathit{apply}(x, l_4(k,u)) & \mathit{apply}(l_4(k,u),n) &= \mathit{apply}(k,u+n) \end{aligned}$$

To understand concretely how these equations represent the original term, it is perhaps useful see what happens when a concrete argument and a continuation are supplied: $(\lambda^{l_1} \langle x, k\rangle.\, (\lambda^{l_2} k.k\ 1)\ (\lambda^{l_3} u.x\ (\lambda^{l_4} n.k\ (u+n))))\ \langle \lambda^{l_5} k.\, k\ 42,\ \lambda^{l_6} n.\, \mathtt{print\_int}(n)\rangle$. Then we get the following cases for l_5 and l_6 in addition to the cases above

$$\mathit{apply}(l_5,k) = \mathit{apply}(k,42), \qquad \mathit{apply}(l_6,n) = \mathtt{print\_int}(n),$$

and the fully applied term defunctionalizes to $\mathit{apply}(l_1, \langle l_5, l_6\rangle)$. Executing it results in 43 being printed, as expected.

This outlines a naive defunctionalization method for translating a higher-order language into a first-order language with (tail) recursion. This method can be improved in various ways. The above *apply*-procedure performs a case distinction on the function name each time it is invoked. In this example, the label of the invoked function can be determined statically, however, so that the case distinction is not in fact necessary. Instead, we may define one function $apply_l$ for each label l and choose the appropriate label statically. The label l then does not need to be passed as an argument anymore. A defunctionalization procedure that takes into account control flow information in this way was introduced by Banerjee et al. [2]. If we apply it to this example and moreover simplify the result by removing unneeded function arguments, then we get:

$$\begin{array}{rr} apply_{l_1}() = apply_{l_2}() & apply_{l_2}() = apply_{l_3}(1) \\ apply_{l_3}(u) = apply_{l_5}(u) & apply_{l_4}(u,n) = apply_{l_6}(u+n) \end{array} \tag{1}$$

The term itself simplifies to $apply_{l_1}()$. The interface where these equations interact with the environment consists of the labels l_1, l_4, l_5 and l_6. Applying the term to concrete arguments as above amounts to extending the environment with the following equations:

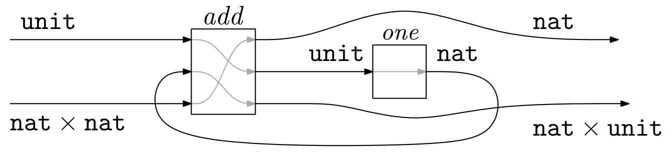
$$apply_{l_5}(u) = apply_{l_4}(u,42) \qquad apply_{l_6}(n) = \texttt{print\_int}(n)$$

The point of this paper is that the program (1) is the same as what we get from interpreting the source term in a model of computation by interaction.

1.2 Interpretation in an Interactive Computation Model

In computation by interaction the general idea is to study models of computation that interpret programs by interaction dialogues in the style of game semantics and to consider actual implementations of such dialogue interaction. For example, a function of type $\mathbb{N} \to \mathbb{N}$ may be implemented in interactive style by a program that, for a suitable type S, takes as input a value of type $\texttt{unit} + S \times \texttt{nat}$ and gives as output a value of type $\texttt{nat} + S \times \texttt{unit}$. The input $\mathsf{inl}(\langle\rangle)$ to this program is interpreted as a request for the return value of the function. An output of the form $\mathsf{inl}(n)$ means that n is the requested value. If the output is of the form $\mathsf{inr}(s,\langle\rangle)$, however, then this means that the program would like to know the argument of the function. It also requests that the value s is returned along with the answer, as programs here do not have state and s can thus not be stored until the request is answered. To answer the program's request, we can pass a value of the form $\mathsf{inr}(s,m)$, where m is our answer.

The particular function $\lambda x\colon \mathbb{N}.\, 1+x$ is implemented by the program specified in the following diagram, where S is $\texttt{nat}$. This diagram is to be understood so that one may pass a message along any of its input wires. The message must be a value of the type labelling the wire. When a message arrives at an input of box, the box will react by sending a message on one of its outputs. Thus, at any time there is one message in the network. Computation ends when a message is passed along an output wire.



In this diagram, *add* takes as input messages of type $\texttt{unit} + (\texttt{nat} + (\texttt{nat} \times \texttt{nat}))$; the three input arrows in the figure represent the summands of this type. It outputs a message of type $\texttt{nat} + (\texttt{unit} + (\texttt{nat} \times \texttt{unit}))$. Its behaviour is given by the mappings $\mathsf{inl}(\langle\rangle) \mapsto \mathsf{inr}(\mathsf{inl}(\langle\rangle))$, $\mathsf{inr}(\mathsf{inl}(n)) \mapsto \mathsf{inr}(\mathsf{inr}(n, \langle\rangle))$ and $\mathsf{inr}(\mathsf{inr}(n,m)) \mapsto \mathsf{inl}(n+m)$. The box labelled *one* maps the request $\langle\rangle$ to the number 1.

This interactive implementation of $\lambda x : \mathbb{N}.\, 1 + x$ may be described as the interpretation of the term in a semantic model $\mathrm{Int}(\mathbb{T})$ built by applying the general categorical Int construction to a category $\mathbb{T}$ that is constructed from the target language, see Sec. 6.

Compare the above interaction diagram to the definitions in (1) obtained by defunctionalization. The labels l_1, l_3 and l_4 there correspond to the three inputs of the *add*-box (from top to bottom), l_2 is the input of box *one*, and l_5 and l_6 are the destination labels of the two outgoing wires. One may consider the *apply*-definitions in (1) a particular implementation of the diagram, where a call to $apply_l(m)$ means that message m is sent to point l in the diagram. A naive implementation would introduce a label for the end of each arrow in the diagram and implement the message passing accordingly.

This outlines the relation between the translations, which we now describe in detail.

2 Target Language

Programs in the target language consist of mutually (tail-)recursive definitions of first-order functions, such as the examples for *apply*-equations above. The target language is at the same abstraction level as SSA-form compiler intermediate languages, e.g. [3].

The set of *target types* is defined by the grammar below. Sum types and recursive types will be needed at the end of Sec. 8 only. *Target expressions* and *values* are standard terms for these types, see e.g. [16]:

$$\begin{array}{rl}
\text{Types:} & A, B ::= \alpha \mid \texttt{unit} \mid \texttt{nat} \mid A \times B \mid A + B \mid \mu\alpha.A \\
\text{Expressions:} & e, e_1, e_2 ::= x \mid \langle\rangle \mid n \mid e_1 + e_2 \mid \mathsf{iszero?}(e) \mid \langle e_1, e_2\rangle \mid \mathsf{let}\ \langle x, y\rangle = e_1\ \mathsf{in}\ e_2 \\
& \qquad \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow e_1;\ \mathsf{inr}(y) \Rightarrow e_2 \\
& \qquad \mid \mathsf{fold}(e) \mid \mathsf{unfold}(e) \\
\text{Values:} & v, v_1, v_2 ::= \langle\rangle \mid n \mid \langle v_1, v_2\rangle \mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \mid \mathsf{fold}(v)
\end{array}$$

Here, n ranges over natural numbers as constants and $\mathsf{iszero?}(e)$ is intended to have type $\texttt{unit} + \texttt{unit}$ with $\mathsf{inl}(\langle\rangle)$ representing true. We assume a standard (non-linear) typing and equality judgement, so that each well-typed closed expression e equals a unique value v of the same type, written as $e = v$.

Target programs consist of a set of first-order function definitions.

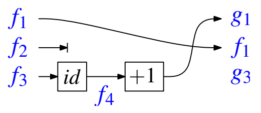
Definition 1. *Let $\mathscr{L}$ be an infinite set of program labels. A* definition of a label $f \in \mathscr{L}$ *is given by an equation of the form $f(x) = g(e)$ or the form $f(x) = \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(y) \Rightarrow g(e_1);\ \mathsf{inr}(z) \Rightarrow h(e_2)$, wherein $g, h \in \mathscr{L}$ and e, e_1 and e_2 range over target expressions.*

Definition 2. *A* target program $P = (\alpha, E, \beta)$ *consists of a set E of function definitions together with a list α of entry labels and a list β of exit labels. Both α and β must be lists of pairwise distinct labels. The set E of definitions must contain at most one definition for any label and must not contain any definition for the labels in β.*

The list α assigns an order to the function labels that may be used as entry points for the program and β identifies external labels as return points.

We allow ourselves to use syntactic sugar, such as writing $f(x,y) = g(e)$ for $f(z) = g(\mathsf{let}\ \langle x,y\rangle = z\ \mathsf{in}\ e)$ or writing just $f()$ for $f(\langle\rangle)$.

We use an informal graphical notation for target programs, depicting for example the program $(f_1 f_2 f_3, \{f_3(x) = f_4(x),\ f_4(x) = g_1(x+1)\}, g_1 f_1 g_3)$ as shown below.



Target programs can be typed in the evident way, so that in each function definition $f(x) = \dots$ the variable x is assigned a type and function calls must preserve types. If P is the program $(f_1 \dots f_n, E, g_1 \dots g_m)$, then we write $P\colon (A_1 \dots A_n) \to (B_1 \dots B_m)$ if the functions $f_1,\dots,f_n,g_1,\dots,g_m$ in it can be typed such that they accept values of type $A_1,\dots,A_n,B_1,\dots,B_m$ respectively.

We define a simple reduction semantics for programs. A *function call* is an expression of the form $f(v)$, where f is a function label and v is a value. A relation $\to_P$ formalises the function calls as they happen during the execution of a program P. It is the smallest relation satisfying the following conditions: if P contains a definition $f(x) = g(e)$ then $f(v) \to_P g(w)$ for all values v and w with $e[v/x] = w$; and if P contains a definition $f(x) = \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(y) \Rightarrow g(e_1);\ \mathsf{inr}(z) \Rightarrow h(e_2)$ then $f(v) \to_P g(w)$ for all values v and w with $\exists u.\, e[v/x] = \mathsf{inl}(u) \wedge e_1[u/y] = w$, and $f(v) \to_P h(w)$ for all values v and w with $\exists u.\, e[v/x] = \mathsf{inr}(u) \wedge e_2[u/z] = w$.

A *call-trace* of program P is a sequence $f_1(v_1) f_2(v_2) \dots f_n(v_n)$, such that $f_i(v_i) \to_P f_{i+1}(v_{i+1})$ holds for all $i \in \{1,\dots,n-1\}$.

Two programs $P,Q\colon (A_1 \dots A_n) \to (B_1 \dots B_m)$ are *equal* if, for any input, they give the same output, that is, if the entry labels of P and Q are $f_1,\dots,f_n$ and $g_1,\dots,g_n$ respectively and the exit labels are $h_1,\dots,h_m$ and $k_1,\dots,k_m$ respectively, then, for any v, w, i and j, P has a call-trace of the form $f_i(v) \dots g_j(w)$ if and only if Q has a call-trace of the form $h_i(v) \dots k_j(w)$.

The following notation is used in Sec. 7. For any list of target types $X = B_1 \dots B_n$ and any target type A, we write $A \cdot X$ for the list $(A \times B_1) \dots (A \times B_n)$. Given a program $P\colon X \to Y$, we write $A \cdot P\colon A \cdot X \to A \cdot Y$ for the program that passes on the value of type A unchanged and otherwise behaves like P. It may be defined by replacing each definition $f(x) = g(e)$ with $f(z,x) = g(z,e)$ for fresh z, and likewise for branching definitions.

Lemma 1. *Target programs can be organised into a category $\mathbb{T}$ whose objects are finite lists of target types and whose morphisms from X to Y are given by an equivalence classes of programs $P\colon X \to Y$ with respect to program equality.*

Lemma 2. *The category $\mathbb{T}$ has finite coproducts, such that the initial object 0 is given by the empty list and the object $X+Y$ is given by the concatenation of the lists X and Y. Moreover, $\mathbb{T}$ has a uniform trace [9] with respect to these coproducts.*

These lemmas show that $\mathbb{T}$ has enough structure so that we can apply the Int construction [12, 9] (with respect to coproducts) to it and obtain a category $\mathrm{Int}(\mathbb{T})$ that models interactive computation. We shall describe the interpretation of terms in $\mathrm{Int}(\mathbb{T})$ concretely and refer to loc. cit. for the categorical structure.

3 Source Language

Our source language is $\lambda^{\to,\mathbb{N}}$, the simply-typed λ-calculus with a basic type $\mathbb{N}$ of natural numbers and associated terms for numeral constants $n\colon \mathbb{N}$, addition $s+t$ and case distinction $\mathsf{if0}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$. The typing rules are straightforward and can be found in the CPS translation below. There they are formulated with an explicit contraction rule in order to make it easy to consider linear fragments of the source language.

We shall first consider a linear fragment of the source language in Sec. 6, then add the base type $\mathbb{N}$ in Sec. 7 and finally discuss the extension to the full language in Sec. 8.

If one adds a fixed point combinator, this language becomes as expressive as PCF.

4 CPS Translation

We use a variant of the *call-by-name CPS translation* [10], which translates the source language into $\lambda^{\to,\times,\mathbb{N},\bot}$, the calculus that extends $\lambda^{\to,\mathbb{N}}$ with with product types and an empty type $\bot$.

For each type X, the type $\underline{X}$ of its continuations is defined by $\underline{\mathbb{N}} = \neg\mathbb{N}$ and $\underline{X \to Y} = \neg\underline{X} \times \underline{Y}$. We write $\overline{X}$ for $\neg\underline{X}$. A continuation for type $X \to Y$ is thus a pair consisting of a continuation $\underline{Y}$, where the result can be returned, and a function $\neg\underline{X}$ to access the argument. A function can request its argument by applying this function to a continuation of type $\underline{X}$. The argument will then be provided to this continuation.

The CPS translation takes any sequent $x_1\colon X_1,\dots,x_n\colon X_n \vdash t\colon Y$ derivable in $\lambda^{\to,\mathbb{N}}$ to a sequent $x_1\colon \overline{X_1},\dots,x_n\colon \overline{X_n} \vdash \underline{t}\colon \overline{Y}$ derivable in $\lambda^{\to,\times,\mathbb{N},\bot}$. It is given by the following translation of typing rules of $\lambda^{\to,\mathbb{N}}$ on the left to derived rules of $\lambda^{\to,\times,\mathbb{N},\bot}$ on the right.

$$\frac{}{\Gamma, x\colon X \vdash x\colon X} \quad\Longrightarrow\quad \frac{}{\overline{\Gamma}, x\colon \overline{X} \vdash \eta(x,\overline{X})\colon \overline{X}}$$

$$\frac{\Gamma, x\colon X \vdash t\colon Y}{\Gamma \vdash \lambda x : X.t\colon X \to Y} \quad\Longrightarrow\quad \frac{\overline{\Gamma}, x\colon \overline{X} \vdash \underline{t}\colon \overline{Y}}{\overline{\Gamma} \vdash \lambda\langle x,k\rangle.\underline{t}\,k\colon \overline{X \to Y}}$$

$$\frac{\Gamma \vdash s\colon X \to Y \qquad \Delta \vdash t\colon X}{\Gamma, \Delta \vdash s\,t\colon Y} \quad\Longrightarrow\quad \frac{\overline{\Gamma} \vdash \underline{s}\colon \overline{X \to Y} \qquad \overline{\Delta} \vdash \underline{t}\colon \overline{X}}{\overline{\Gamma}, \overline{\Delta} \vdash \lambda k.\underline{s}\,\langle \underline{t},k\rangle\colon \overline{Y}}$$

$$\frac{}{\Gamma \vdash n\colon \mathbb{N}} \quad\Longrightarrow\quad \frac{}{\overline{\Gamma} \vdash \lambda k.k\,n\colon \overline{\mathbb{N}}}$$

$$\frac{\Gamma \vdash s\colon \mathbb{N} \qquad \Delta \vdash t\colon \mathbb{N}}{\Gamma, \Delta \vdash s+t\colon \mathbb{N}} \quad\Longrightarrow\quad \frac{\overline{\Gamma} \vdash \underline{s}\colon \overline{\mathbb{N}} \qquad \overline{\Delta} \vdash \underline{t}\colon \overline{\mathbb{N}}}{\overline{\Gamma}, \overline{\Delta} \vdash \lambda k.\underline{s}\,(\lambda x.\underline{t}\,(\lambda y.k\,(x+y)))\colon \overline{\mathbb{N}}}$$

$$\frac{\Gamma \vdash s\colon \mathbb{N} \quad \Delta_1 \vdash t_1\colon \mathbb{N} \quad \Delta_2 \vdash t_2\colon \mathbb{N}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{if0}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\colon \mathbb{N}} \quad\Longrightarrow\quad \frac{\overline{\Gamma} \vdash \underline{s}\colon \overline{\mathbb{N}} \qquad \overline{\Delta_1} \vdash \underline{t_1}\colon \overline{\mathbb{N}} \qquad \overline{\Delta_2} \vdash \underline{t_2}\colon \overline{\mathbb{N}}}{\begin{array}{r}\overline{\Gamma}, \overline{\Delta_1}, \overline{\Delta_2} \vdash \lambda k.\underline{s}\,(\lambda x.\mathsf{if}\ x\ \mathsf{then}\ \underline{t_1}\,(\lambda y.k\,y)\colon \overline{\mathbb{N}}\\ \mathsf{else}\ \underline{t_2}\,(\lambda y.k\,y))\end{array}}$$

$$\frac{\Gamma, x\colon X, y\colon X \vdash t\colon Y}{\Gamma, x\colon X \vdash t[x/y]\colon Y} \quad\Longrightarrow\quad \frac{\overline{\Gamma}, x\colon \overline{X}, y\colon \overline{X} \vdash \underline{t}\colon \overline{Y}}{\overline{\Gamma}, x\colon \overline{X} \vdash \underline{t}[x/y]\colon \overline{Y}}$$

This CPS translation differs from the standard call-by-name CPS translation [10] in the translation of variables. Instead of letting $\underline{x}$ be just x, we take it to be an η-expansion $\eta(t,X)$ of x. The term $\eta(t,X)$, is defined by induction on X as follows: $\eta(t,\mathbb{N}) := t$ and

$\eta(t, X \to Y) := \lambda x.\, \eta(t\ \eta(x, X), Y)$, where x is a fresh variable. For example, we have $\eta(f, (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}) = \lambda x_1.\, f\ (\lambda x_2. x_1\ x_2)$ and $\eta(f, \mathbb{N} \to (\mathbb{N} \to \mathbb{N})) = \lambda x_1.\, \lambda x_2.\, f\ x_1\ x_2$.

The use of η-expansion allows us to use compositional reasoning in Sec. 7. In the example in the Introduction, we have not applied this η-expansion for better readability.

5 Defunctionalization

After the CPS transform, the term is annotated with control flow information and then translated into the target language by a defunctionalization procedure that takes the control flow information into account. In this section we define a particularly simple variant of such a procedure, which suffices to show the relation to the Int construction. It is a special case of the flow-based defunctionalization described by Banerjee et al. [2].

The input of the defunctionalization procedure is a term of the labelled λ-calculus $\lambda_\ell^{\to,\times,\mathbb{N},\perp}$. Its syntax differs from that of $\lambda^{\to,\times,\mathbb{N},\perp}$ only in that abstractions, applications and function types are each annotated with a label from $\mathscr{L}$. Thus, the terms $\lambda x : X.t$ and $s\ t$ are replaced by $\lambda^l x{:}X.t$ and $s @_l t$. The type $X \to Y$ becomes $X \xrightarrow{l} Y$.

We require that each abstraction is uniquely identified by its label, i.e. we allow only terms in which no two abstractions have the same label. In the application $s @_l t$ the label l expresses that the function s applied here will be defined by an abstraction with label l. Note that each application can be annotated with a single label l only, which means that for each application the label of the function that is being applied is statically known. In general, one needs to allow a set of labels for more than one possible definition site, as in e.g. [2]. We discuss this in Sec. 8, but until then the variant with a single label suffices and much simplifies the exposition.

The typing rules of $\lambda_\ell^{\to,\times,\mathbb{N},\perp}$ enforce that terms are annotated with correct control flow information. An abstraction $\lambda^l x{:}X.t$ has type $X \xrightarrow{l} Y$. If s has type $X \xrightarrow{l} Y$ and t has type X then $s @_l t$ has type Y.

In the rest of this section we explain how the terms of $\lambda_\ell^{\to,\times,\mathbb{N},\perp}$ are defunctionalized into target terms. We defer the question of how to annotate terms with labels to Sec. 6.

The defunctionalization of a term t in $\lambda_\ell^{\to,\times,\mathbb{N},\perp}$ is defined by the following judgement, which has the form $t \Downarrow t^* ; D_t$, where t^* is the defunctionalized term in the target language and D_t is a set of equations. In general, the set D_t need not be function definitions in the sense of Def. 1. We shall however use defunctionalizion only for terms t for which D_t consists only of function definitions.

$$\frac{}{x \Downarrow x ; \emptyset} \qquad \frac{}{n \Downarrow n ; \emptyset} \qquad \frac{s \Downarrow s^* ; D_s \qquad t \Downarrow t^* ; D_t}{s + t \Downarrow s^* + t^* ; D_s \cup D_t}$$

$$\frac{s \Downarrow s^* ; D_s \qquad t \Downarrow t^* ; D_t \qquad u \Downarrow u^* ; D_u}{\mathsf{if0}\ s\ \mathsf{then}\ t\ \mathsf{else}\ u \Downarrow \mathsf{case\ iszero?}(s^*)\ \mathsf{of\ inl}(\langle\rangle) \Rightarrow t^* ; \mathsf{inr}(\langle\rangle) \Rightarrow u^* ; D_s \cup D_t \cup D_u}$$

$$\frac{s \Downarrow s^* ; D_s \qquad t \Downarrow t^* ; D_t}{\langle s, t\rangle \Downarrow \langle s^*, t^*\rangle ; D_s \cup D_t} \qquad \frac{s \Downarrow s^* ; D_s \qquad t \Downarrow t^* ; D_t}{\mathsf{let}\ \langle x, y\rangle = t\ \mathsf{in}\ s \Downarrow \mathsf{let}\ \langle x, y\rangle = t^*\ \mathsf{in}\ s^* ; D_s \cup D_t}$$

$$\frac{s \Downarrow s^* ; D_s \qquad t \Downarrow t^* ; D_t}{s @_l t \Downarrow apply_l(s^*, t^*) ; D_s \cup D_t} \qquad \frac{t \Downarrow t^* ; D_t \qquad fv(\lambda^l x{:}A.t) = \{x_1, \ldots, x_n\}}{\lambda^l x{:}A.t \Downarrow \langle x_1, \ldots, x_n\rangle ; D_t \cup \{apply_l(\langle x_1, \ldots, x_n\rangle, x) = t^*\}}$$

In the rule for abstraction we assume a global ordering on all variables, so that the order of the tuple is well-defined.

Note that for closed terms of function type, such as the closed terms the form $\underline{t}$ obtained by CPS translation, t^* is just $\langle\rangle$. We therefore concentrate on the set D_t.

With annotations the example from the Introduction becomes the term $\underline{t}$ given by $\lambda^{l_1} z.\, \mathsf{let}\ \langle x,k\rangle = z\ \mathsf{in}\ (\lambda^{l_2} k'.\, k' @_{l_3} 1) @_{l_2} (\lambda^{l_3} u.\, x @_{l_5} (\lambda^{l_4} n.\, k @_{l_6} (u+n)))$, whose type is $\vdash \underline{t} \colon \neg_{l_1}(\neg_{l_5}\neg_{l_4}\mathbb{N} \times \neg_{l_6}\mathbb{N})$. The set $D_{\underline{t}}$ of definitions consists of

$$apply_{l_1}(\langle\rangle, \langle x,k\rangle) = apply_{l_2}(\langle\rangle, \langle x,k\rangle), \qquad apply_{l_2}(\langle\rangle, k') = apply_{l_3}(k', 1),$$
$$apply_{l_3}(\langle x,k\rangle, u) = apply_{l_5}(x, \langle k\rangle), \qquad apply_{l_4}(\langle k\rangle, n) = apply_{l_6}(k, u+n).$$

Compared with the Introduction, it appears that more data is being passed around in these *apply*-equations. However, consider once again the application of $\underline{t}$ to the concrete arguments from the Introduction. Then one gets the additional equations

$$apply_{l_5}(\langle\rangle, k) = apply_{l_4}(k, 42), \qquad apply_{l_6}(\langle\rangle, n) = \texttt{print\_int}(n),$$

and the fully applied term defunctionalizes to $apply_{l_1}(\langle\rangle, \langle\langle\rangle, \langle\rangle\rangle)$. Thus, all the variables in the *apply*-equations only ever store the value $\langle\rangle$ or tuples thereof, and these arguments may just as well be omitted.

Note that the defunctionalization procedure yields a set of equations, but it does not specify an interface of entry and exit labels. When one applies defunctionalization to closed source programs of base type, as is usually done in compilation, choosing an interface is not important. For the entry labels one would typically just choose a single entry point `main`, for example. For open terms or terms of higher type, however, one needs to fix the interface that matches the type. In the above example of $\underline{t}$, a suitable choice of entry and exit labels would be $l_1 l_4$ and $l_5 l_4$ respectively. We shall explain how to define an interface for terms in the image of the CPS translation in the next section.

Of course, the defunctionalization procedure described above is quite simple. In actual applications one would certainly want to apply optimisations, not least to remove unnecessary functions arguments. An example of such an optimisation is *lightweight defunctionalization* of Banerjee et al. [2]. We shall see that the Int construction captures one such optimisation of the defunctionalization procedure.

6 The Linear Fragment

To explain the basic idea of how the interpretation in a model of interactive computation (namely $\mathrm{Int}(\mathbb{T})$) relates to CPS translation and defunctionalization, we first consider the simplest non-trivial case. Consider the linear fragment of the source language and instead of the natural number type $\mathbb{N}$ just a type o without any term constructors:

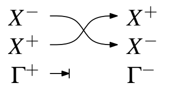
$$X, Y ::= o \mid X \multimap Y \qquad\qquad s, t ::= x \mid s\ t \mid \lambda x{:}X.\, t$$

The standard typing rules AX, $\multimap$I and $\multimap$E for this calculus are shown below.

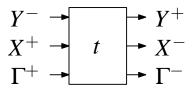
First we describe directly what the interpretation of this source language in $\mathrm{Int}(\mathbb{T})$ amounts to. A type X is modelled by an interface (X^-, X^+), which consists of two finite lists X^- and X^+ of target types. Closed terms of type X will be modelled by programs of type $P\colon X^- \to X^+$. The interfaces are defined by induction on the type: both o^- and o^+ are the singleton list `unit`, $(X \multimap Y)^-$ is Y^-X^+, i.e. the concatenation of Y^- and X^+, and $(X \multimap Y)^+$ is Y^+X^-. For a context $\Gamma = x_1 : X_1, \dots, x_n : X_n$, we write Γ^- and Γ^+ for the concatenations $X_n^- \dots X_1^-$ and $X_n^+ \dots X_1^+$.

The interpretation of a term $\Gamma \vdash t\colon X$ in $\mathrm{Int}(\mathbb{T})$ is a morphism $[\![\Gamma \vdash t\colon X]\!]\colon X^-\Gamma^+ \to X^+\Gamma^-$ in $\mathbb{T}$, i.e. an equivalence class of programs. This interpretation is defined by induction on the derivation; as depicted below.

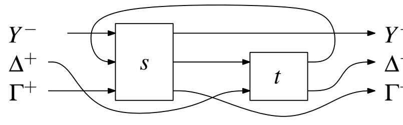
$$\textsc{ax}\ \frac{}{\Gamma, x\colon X \vdash x\colon X} \quad \Longrightarrow$$



$$\multimap\mathrm{I}\ \frac{\Gamma, x\colon X \vdash t\colon Y}{\Gamma \vdash \lambda x{:}X.\,t\colon X \multimap Y} \quad \Longrightarrow$$



$$\multimap\mathrm{E}\ \frac{\Gamma \vdash s\colon X \multimap Y \qquad \Delta \vdash t\colon X}{\Gamma, \Delta \vdash s\,t\colon Y} \quad \Longrightarrow$$



The aim is now to show that this interpretation in $\mathrm{Int}(\mathbb{T})$ is closely related to CPS translation followed by defunctionalization.

Our flow-based defunctionalization depends on suitable labellings of terms and types. We introduce notation for labellings of types of the form $\overline{X}$. For any type X and any $x^-, x^+ \in \mathscr{L}^*$ with $\mathit{length}(x^-) = \mathit{length}(X^-)$ and $\mathit{length}(x^+) = \mathit{length}(X^+)$, we define a type $\overline{X}[x^-, x^+]$ in the labelled λ-calculus as follows: $\overline{o}[q,a]$ is $\neg_q \neg_a \mathtt{unit}$ and $\overline{(X \multimap Y)}[y^-x^+, y^+x^-]$ is $\neg_q(\neg_r X' \times Y')$ if $\overline{X}[x^-, x^+]$ is $\neg_r X'$ and $\overline{Y}[y^-, y^+]$ is $\neg_q Y'$.

For example, $\overline{(o \multimap o)}[qa', aq']$ denotes $\neg_q(\neg_{q'}\neg_{a'}\mathtt{unit} \times \neg_a \mathtt{unit})$.

If Γ is $x_1 : X_1, \dots, x_n : X_n$, then we write short $\overline{\Gamma}[x_n^- \dots x_1^-, x_n^+ \dots x_1^+]$ for the context $x_1 : \overline{X_1}[x_1^-, x_1^+], \dots, x_n : \overline{X_n}[x_n^-, x_n^+]$. We say that a sequent $\overline{\Gamma}[\gamma^-, \gamma^+] \vdash t\colon \overline{X}[x^-, x^+]$ is *well-labelled* if the labels in $\gamma^-, \gamma^+, x^-, x^+$ are pairwise distinct.

Although defined as an abbreviation for a labelled type, one may alternatively think of $\overline{X}[x^-, x^+]$ as the type X together with a labelling of the ports of the interface (X^-, X^+).

Lemma 3. *If $\Gamma \vdash t\colon X$ is derivable in the linear type system, then the sequent $\overline{\Gamma} \vdash \underline{t}\colon \overline{X}$ obtained by CPS transform can be annotated with labels such that the sequent $\overline{\Gamma}[\gamma^-, \gamma^+] \vdash \underline{t}\colon \overline{X}[x^-, x^+]$ is well-labelled and derivable, for some $\gamma^-, \gamma^+, x^-, x^+ \in \mathscr{L}^*$.*

The proof is a straightforward induction on derivations. We note that the case for variables depends on the η-expanded form of the CPS translation. With the expansion a well-labelled $x\colon \overline{\mathbb{N}}[q,a] \vdash \underline{x}\colon \overline{\mathbb{N}}[q',a']$ is derivable; without it this would only be possible if $q = q'$ and $a = a'$. The defunctionalization of $\underline{x}$ consists of definitions of $\mathit{apply}_{q'}$ and apply_a, which just forward their arguments to apply_q and $\mathit{apply}_{a'}$ respectively. We believe that it is simpler to consider the case with these indirections first and study their removal (which is non-compositional, due to renaming) in a second step.

We now define a function CpsDefun that combines CPS transformation and defunctionalization. Given any judgement $\Gamma \vdash t\colon X$ derivable in the linear fragment of the source language, let $\overline{\Gamma}[\gamma^-,\gamma^+] \vdash \underline{t}\colon \overline{X}[x^-,x^+]$ be the judgement from the above lemma for a suitable choice of labels. Let $D_{\underline{t}}$ be the set of equations determined by the defunctionalization judgement $\underline{t} \Downarrow \underline{t}^* \,;D_{\underline{t}}$. The function $\mathsf{CpsDefun}$ maps the source judgement $\Gamma \vdash t\colon X$ to the target program $(x^-\gamma^+, D_{\underline{t}}, x^+\gamma^-)$. It is not hard to see the set $D_{\underline{t}}$ is such that this indeed a target program.

We use a single function $\mathsf{CpsDefun}$ rather than a composition of two functions general Cps and Defun, as we do not have a canonical choice of entry and exit labels for defunctionalization in general. Thus, the composition $\mathsf{Defun} \circ \mathsf{Cps}$ would only return a set of equations and not yet target program. With a combined function, it suffices to choose entry and exit labels for terms that are in the image of the CPS translation.

Define a further function Erase on target programs, which erases all function arguments (and removes all equations defined by case distinction, which cannot appear in $D_{\underline{t}}$ for this source language): $\mathsf{Erase}(\alpha,E,\beta) := (\alpha, \{f() = g() \mid f(x) = g(e) \in E\}, \beta)$.

The composed function $\mathsf{Erase} \circ \mathsf{CpsDefun}$ takes a source program, first applies the CPS translation and defunctionalization and then 'optimises' the result by erasing all function arguments. The resulting program is in fact correct and it is what one obtains from the interpretation in $\mathrm{Int}(\mathbb{T})$:

Proposition 1. *Suppose $\Gamma \vdash t\colon X$ is derivable in the linear type system. Then the target program $\mathsf{Erase}(\mathsf{CpsDefun}(\Gamma \vdash t\colon X))$ has type $X^-\Gamma^+ \to X^+\Gamma^-$ and is an element of the equivalence class of programs obtained by Int interpretation of $\Gamma \vdash t\colon X$.*

7 Base Types

We now work towards extending the result to a more expressive source language, starting with non-trivial base types. We replace the type o by the type of natural numbers $\mathbb{N}$ and add terms for constant numbers, addition and case distinction.

$$X,Y ::= \mathbb{N} \mid X \multimap Y \qquad s,t ::= n \mid s+t \mid \mathsf{if0}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \mid x \mid s\ t \mid \lambda x{:}X.\,t$$

The example from the Introduction illustrates that for this fragment of the source language it is not possible to remove all arguments from the *apply*-functions, as we have done above. At least certain natural numbers must be passed as arguments.

Again, we first consider the interpretation of the fragment in $\mathrm{Int}(\mathbb{T})$. To this end we define $\mathbb{N}^- = \mathtt{unit}$ (a request to compute the number) and $\mathbb{N}^+ = \mathtt{nat}$ (the actual number as an answer). It is however not completely straightforward to extend the Int interpretation described in the previous section. Consider for example the case of an addition $s+t$ for two closed terms $\vdash s\colon \mathbb{N}$ and $\vdash t\colon \mathbb{N}$. Suppose we already have programs (q_s,E_s,a_s) and (q_t,E_t,a_t) for s and t. For $s+t$ it would be natural to use the program (q,E,a) with equations $apply_q() = apply_{q_s}()$, $apply_{a_s}(x) = apply_{q_t}(x,\langle\rangle)$, $apply_{a_t}(x,y) = apply_a(x+y)$, the equations from E_s, and the equations from $\mathtt{nat}\cdot E_t$ (recall the notation $\mathtt{nat}\cdot -$ from Sec. 2). We use $\mathtt{nat}\cdot E_t$ instead of E_t in order to keep the value x available until the end when we want to compute the sum. The difficulty is to decide which values must, like x, must be preserved in which equations.

One solution to this issue was proposed by Dal Lago and the author in the form of IntML [4]. We consider here a simple special case of this system. The basic idea is to annotate the each function type $X \multimap Y$ with a *subexponential* A, which is a target type:

$$X, Y ::= \mathbb{N} \mid A \cdot X \multimap Y$$

The subexponential annotation may be explained such that a term s of type $A \cdot X \multimap Y$ is a function that uses its argument within an environment that contains an additional value of type A. The function s may be applied to any argument t of type X. In the interactive interpretation of the application $s\,t$, whenever s sends a query to t it needs to preserve a value of type A. It does so by sending the value along with the query, expecting it to be returned unmodified along with a reply. For example, addition naturally gets the type $\mathtt{unit} \cdot \mathbb{N} \multimap \mathtt{nat} \cdot \mathbb{N} \multimap \mathbb{N}$, as it needs to remember the already queried value of the first argument (having type `nat`) when it queries the second one.

In the type system, subexponential annotations are integrated by letting contexts consist of variable declarations of the form $x\colon A \cdot X$. The typing rules are shown below.

$$\textsc{ax}\ \frac{}{\Gamma, x\colon \mathtt{unit} \cdot X \vdash x : X} \qquad \textsc{const}\ \frac{}{\Gamma \vdash n : \mathbb{N}}$$

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

$$\textsc{add}\ \frac{\Gamma \vdash s : \mathbb{N} \qquad \Delta \vdash t : \mathbb{N}}{\Gamma, \mathtt{nat} \cdot \Delta \vdash s + t : \mathbb{N}} \qquad \textsc{if}\ \frac{\Gamma \vdash s : \mathbb{N} \qquad \Delta_1 \vdash t_1 : X \qquad \Delta_2 \vdash t_2 : X}{\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{if0}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : X}$$

In these rules, we write $A \cdot \Gamma$ for the context obtained by replacing each $x\colon B \cdot X$ with $x\colon (A \times B) \cdot X$. We note that rule IF enforces more linearity than usual. We have chosen to treat this linear version here, as for the defunctionalization of the non-linear version with $\Delta_1 = \Delta_2$, we would need to extend the labelled type system, and this case is already covered when we allow contraction in the next section.

With subexponential annotations, it is straightforward to define the Int interpretation. Define $(A \cdot X \multimap Y)^- = Y^-(A \cdot X^+)$ and $(A \cdot X \multimap Y)^+ = Y^+(A \cdot X^-)$. We write Γ^- and Γ^+ for $A_n \cdot X_n^- \dots A_1 \cdot X_1^-$ and $A_n \cdot X_n^+ \dots A_1 \cdot X_1^+$ if $\Gamma = x_1\colon A_1 \cdot X_1, \dots, x_n\colon A_n \cdot X_n$.

The interpretation of rule AX changes from the linear case only by insertion of isomorphisms of the form $A \simeq \mathtt{unit} \times A$. The interpretation of rule CONST is given by the program $(\mathit{apply}_q \gamma^+, \{\mathit{apply}_q() = \mathit{apply}_a(n)\}, \mathit{apply}_a \gamma^-)$. The interpretation of the other rules is shown graphically in Fig. 7. In the cases for $\multimap$E and ADD, the boxes labelled with A and $\mathbb{N}^+$ respectively denote the program obtained by applying the operations $A \cdot (-)$ and $\mathbb{N}^+ \cdot (-)$ respectively to the contents of the box. For ADD, CONST and IF we do not show the contexts for brevity. In the case for IF, we write 0? for the program given by $\mathit{apply}_{a_s}(x) = \mathsf{case\ iszero?}(x)\ \mathsf{of\ inl}(y) \Rightarrow \mathit{apply}_{q_1}(y);\ \mathsf{inr}(z) \Rightarrow \mathit{apply}_{q_2}(z)$.

Let us now consider how the above interpretation relates to the one obtained by CPS translation and defunctionalization, where the subexponential annotations are ignored.

A CPS translation of n is given by $\lambda^q k. k @_a n\colon \overline{\mathbb{N}}[q,a]$, where $\overline{\mathbb{N}}[q,a] = \neg_q \neg_a \mathtt{nat}$, which defunctionalizes to $\mathit{apply}_q(\langle\rangle, k) = \mathit{apply}_a(k, n)$. The Int interpretation yields the definition $\mathit{apply}_q() = \mathit{apply}_a(n)$, which differs only in the removal of arguments.

For addition $s + t$ a CPS translation is $\lambda^q k. \underline{s} @_{q_s} (\lambda^{a_s} x. \underline{t} @_{q_t} (\lambda^{a_t} y. k @_a (x + y)))$. Defunctionalization leads to the following set of equations: $D_{\underline{s}} \cup D_{\underline{t}} \cup \{\mathit{apply}_q(f, k) = \mathit{apply}_{q_s}(\underline{s}^*, \langle k \rangle), \mathit{apply}_{a_s}(\langle k \rangle, x) = \mathit{apply}_{q_t}(\underline{t}^*, \langle k, x \rangle), \mathit{apply}_{a_t}(\langle k, x \rangle, y) = \mathit{apply}_a(k, x + y)\}$.

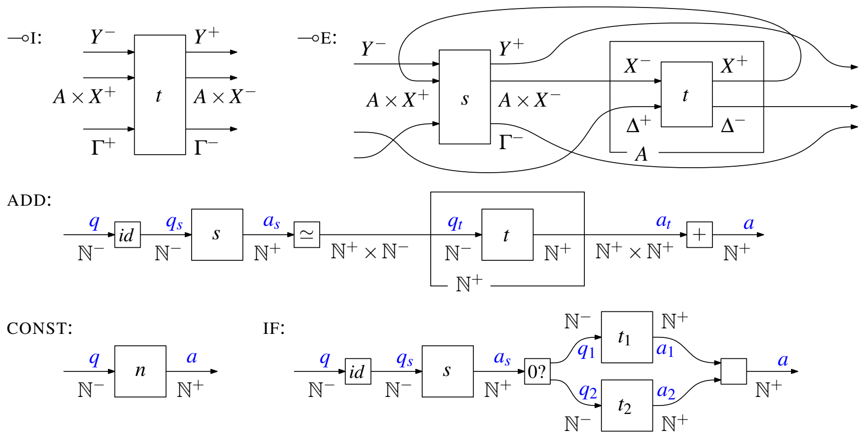


Fig. 1. Int Interpretation of Rules with Subexponentials

Comparing this to the Int interpretation, we can see that this set of equations has the same shape, albeit with different arguments.

Similarly, the source term $\mathsf{if0}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$ is CPS translated to the labelled term $\lambda^q k.\, \underline{s}@_{q_s}(\lambda^{a_s} x.\, \mathsf{if0}\ x\ \mathsf{then}\ \underline{t_1}@_{q_1}(\lambda^{a_1} y.\, k@_a y)\ \mathsf{else}\ \underline{t_2}@_{q_2}(\lambda^{a_2} y.\, k@_a y))$. The equations obtained by defunctionalization have the same shape as those of the Int interpretation.

The observation that the programs obtained by Int interpretation and CPS translation followed by defunctionalization have the same shape can be made precise as follows. We say that two programs have the same *skeleton* whenever they have the same interface and the following holds: if one of the programs contains the definition $f(x) = g(e)$, then the other contains $f(x) = g(e')$ for some e'; and if one of the programs contains $f(x) = \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow g(e_1); \mathsf{inr}(y) \Rightarrow h(e_2)$, then the other contains $f(x) = \mathsf{case}\ e'\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow g(e'_1); \mathsf{inr}(y) \Rightarrow h(e'_2)$ for some e', e'_1 and e'_2.

We note that Lemma 3 continues to hold and that $\mathsf{CpsDefun}$ can be defined as above.

Proposition 2. *For any $\Gamma \vdash t\colon X$ there exists a program $I_{\underline{t}}$ that is a representative of the Int interpretation of the derivation of the sequent and that has the same skeleton as* $\mathsf{CpsDefun}(\Gamma \vdash t\colon X)$.

The proposition establishes a simple connection between the general shape of the programs. Let us now compare the values that are being passed around in them. We show that the values appearing in call traces of the program obtained by Int interpretation can be seen as simplifications of the values appearing at the same time in the traces of the program obtained by defunctionalization.

For any value v, we define a multiset of $\mathscr{V}(v)$ of the numbers it contains as follows: $\mathscr{V}(v) = \{n\}$ if $v = n$, $\mathscr{V}(v) = \mathscr{V}(v_1) \cup \mathscr{V}(v_2)$ if $v = \langle v_1, v_2 \rangle$ and $\mathscr{V}(v) = \emptyset$ otherwise (values of recursive types or sum types cannot appear). We say that a value v *simplifies* a value w if $\mathscr{V}(v) \subseteq \mathscr{V}(w)$. For example, the value $\langle 2, \langle 3,3 \rangle \rangle$ simplifies $\langle 1, \langle \langle 2, \langle \rangle \rangle, \langle 3, \langle 2,3 \rangle \rangle \rangle \rangle$, but not $\langle 2,3 \rangle$. We say that a call trace $f_1(v_1) \dots f_n(v_n)$ *simplifies* the call trace $g(w_1) \dots g_n(w_n)$ if, for any $i \in \{1,\dots,n\}$, $f_i = g_i$ and v_i simplifies w_i.

With this terminology, we can express that the Int interpretation of any term simplifies its CPS translation and defunctionalization in the sense that it differs only in that unused function arguments are removed and function arguments are rearranged.

Proposition 3. *Let* $\vdash t\colon \mathbb{N}$*, let* $(q, D_{\underline{t}}, a) := \mathsf{CpsDefun}(\vdash t\colon \mathbb{N})$ *and let* $I_{\underline{t}}$ *be the program from Prop. 2. Then, any call-trace of* $I_{\underline{t}}$ *beginning with* $apply_q()$ *simplifies the call-trace of* $\mathsf{CpsDefun}(\vdash t\colon \mathbb{N})$ *of the same length that begins with* $apply_q(\langle\rangle, \langle\rangle)$*.*

This proposition allows us to consider the Int interpretation as a simplification of the program obtained by defunctionalization. This simplification seems quite similar to other optimisations of defunctionalization, in particular lightweight defunctionalization [2]. However, we do not know any variant of defunctionalization in the literature that gives exactly the same result. One may consider the Int interpretation as a new approach to optimising the defunctionalization of programs in continuation passing style.

8 Simple Types and Recursion

In this section, we strengthen the source language, explain how the Int interpretation can be extended to translate this language to the target language and relate this translation to CPS transform and defunctionalization. Since with increasing expressiveness of the source language it becomes harder to keep track of the syntactic details of defunctionalization, we investigate the relation less formally than in the previous section. We argue that a type system with subexponential annotations, adapted from IntML, offers a simple and conceptually clear way of managing the details.

First, we consider contraction in the source language. The CPS translation will remain unchanged, of course. The defunctionalization procedure described in Sec. 5, however, is too simple to handle this case. The control-flow annotations used therein do not suffice for the simply-typed case; they would need to be extended so that functions and applications are annotated with sets of labels. For instance, $s@_{\{l_1,\dots,l_n\}}t$ would mean that the label of s may not be uniquely determined, but that it is known to be among $l_1, \dots, l_n$. Such an application is transformed by the defunctionalization into a case distinction on the function that actually appears for s during evaluation: $(s@_{\{l_1,\dots,l_n\}}t)^* = \mathsf{case}\ s^*\ \mathsf{of}\ l_1(\vec{x}) \Rightarrow apply_{l_1}(l_1(\vec{x}), t^*); \dots; l_n(\vec{y}) \Rightarrow apply_{l_n}(l_n(\vec{y}), t^*)$. Details appear in [2]. Note that such a case distinction is only possible if labels are actually passed as values; they cannot be omitted as in Sec. 5. To encode labels, one typically uses algebraic data types whose constructors correspond to the function labels. To handle the full λ-calculus, one must allow for recursive algebraic data types.

Let us now consider how the Int translation can be extended to the simply-typed λ-calculus and how it relates to defunctionalization. To this end, we can extend the type system from the previous section with a rule COPY for explicit copying. We also add a rule STRUCT for weakening of subexponential annotations, which is needed for Prop. 4.

$$\textsc{copy}\ \frac{\Gamma \vdash s : X \qquad \Delta, x\colon A\cdot X, y\colon B\cdot X \vdash t : Z}{\Delta, (A+B)\cdot\Gamma \vdash \mathsf{copy}\ s\ \mathsf{as}\ x,y\ \mathsf{in}\ t : Z} \qquad \textsc{struct}\ \frac{\Gamma, x\colon A\cdot X \vdash t : Y}{\Gamma, x\colon B\cdot X \vdash t : Y}\ A \lhd B$$

The side condition $A \lhd B$ means that any value of type A can be encoded into one of type B. Formally, $A \lhd B$ if and only if there are target expressions $x\colon A \vdash s\colon B$ and $y\colon B \vdash r\colon A$, such that $r[s[v/x]/y] = v$ holds for any value v of type A.

Recall the explanation of subexponentials as making explicit the value environment in which a variable is being used. The sequent in the premise of COPY tells us that x_1 and x_2 are used in environments with an additional value of type A and B respectively. The subexponential $A+B$ in the conclusion tell us that x may be used in two ways: first in an environment that contains an additional variable of type A and second in one with an additional variable of type B. The coproduct identifies the two copies of x.

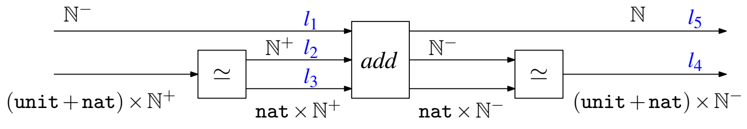
In the Int translation, copy is implemented by case distinction. Depending on whether the subexponential value is $\mathsf{inl}(a)$ or $\mathsf{inr}(b)$, the message is sent to x_1 or x_2 respectively. This case distinction mirrors the case distinction used in defunctionalization.

Let us outline the relation concretely by modifying the example from the Introduction. Consider the term $\lambda x:\mathbb{N}.\,x+x$. Its CPS transform is (with slight simplification) the term $\lambda^{l_1}\langle x,k\rangle.\,x@_{l_4}(\lambda^{l_2}m.\,x@_{l_4}(\lambda^{l_3}n.\,k@_{l_5}(m+n)))$. Applying this term to the argument $\langle\lambda^{l_4}k.\,k@_{\{l_2,l_3\}}42, \lambda^{l_5}n.\,\mathtt{print\_int}(n)\rangle$ gives us an example of an application that needs to be annotated with a set of labels.

If we apply the defunctionalization procedure of Banerjee et al. [2] and then manually remove unneeded arguments, then we get the following equations, in which we consider labels as constructors of an algebraic data type.

$$\begin{aligned} \mathit{apply}_{l_1}() &= \mathit{apply}_{l_4}(l_2()) & \mathit{apply}_{l_4}(k) &= \mathsf{case}\ k\ \mathsf{of}\ l_2() \Rightarrow \mathit{apply}_{l_2}(42)\\ \mathit{apply}_{l_2}(m) &= \mathit{apply}_{l_4}(l_3(m)) & &\qquad\qquad\quad \mid\ l_3(m) \Rightarrow \mathit{apply}_{l_3}(m,42)\\ \mathit{apply}_{l_3}(m,n) &= \mathit{apply}_{l_5}(m+n) & \mathit{apply}_{l_5}(n) &= \mathtt{print\_int}(n) \end{aligned}$$

We compare these equations to what we obtain by applying the Int interpretation to the term $\lambda x.\,\mathsf{copy}\ x\ \mathsf{as}\ x_1,x_2\ \mathsf{in}\ x_1+x_2$ of type $(\mathtt{unit}+\mathtt{nat})\cdot\mathbb{N}\multimap\mathbb{N}$:



$\mathbb{N}^-$
l_1
$\mathbb{N}^+$
l_2
l_3
$\simeq$
$(\mathtt{unit}+\mathtt{nat})\times\mathbb{N}^+$
$\mathtt{nat}\times\mathbb{N}^+$
add
$\mathbb{N}^-$
$\mathtt{nat}\times\mathbb{N}^-$
$\simeq$
$\mathbb{N}$
l_5
l_4
$(\mathtt{unit}+\mathtt{nat})\times\mathbb{N}^-$

The interpretation of COPY inserts the boxes labelled $\simeq$, which denote the canonical isomorphism of their type.

To apply the program given by the diagram to the actual argument 42, one connects the output of type $(\mathtt{unit}+\mathtt{nat})\times\mathbb{N}^-$ to the input of type $(\mathtt{unit}+\mathtt{nat})\times\mathbb{N}^+$ such that when the value $\langle k',\langle\rangle\rangle$ arrives at the output port, then the value $\langle k',42\rangle$ is fed back to the input port. This is what the equation for apply_{l_4} in the defunctionalized program does. The two possible cases of k, namely $\mathsf{inl}(\langle\rangle)$ or $\mathsf{inr}(m)$, correspond to $l_2()$ and $l_3(m)$ in the equation. Thus, in the interactive implementation of $\lambda x:\mathbb{N}.\,x+x$, duplication is treated just as in defunctionalization above. The points corresponding to the apply_{l_i}-equations are indicated in the diagram.

We note that with rules COPY and STRUCT the type system is as expressive as the simply-typed λ-calculus, if the target language has recursive types. This mirrors the use of recursive types in defunctionalization. Write $|t|$ for the term obtained by replacing in it any subterm of the form $\mathsf{copy}\ s_1\ \mathsf{as}\ x,y\ \mathsf{in}\ s_2$ with $s_2[s_1/x, s_1/y]$. Write $|X|$ and $|\Gamma|$ for the type and context of the simply-typed λ-calculus obtained by replacing any $A\cdot Y\multimap Z$ with $Y\to Z$ and removing subexponentials in the context.

Proposition 4. *If $\Gamma \vdash t\colon X$ is derivable in $\lambda^{\to,\mathbb{N}}$, then there exist Δ, s and Y with $\Gamma = |\Delta|$, $t = |s|$ and $X = |Y|$, such that $\Delta \vdash s\colon Y$ is derivable.*

The proof goes by using the simple type inference procedure from [5] and noting that the constraints generated therein can solved easily in the presence of recursive types. Thus, there exists a simple type inference algorithm that finds the derivation of $\Delta \vdash s\colon Y$.

Finally, we mention that recursion can be accounted for by a fixed point combinator of type $\mathsf{fix}_{A,X}\colon (A\ \mathsf{list}) \cdot (A \cdot X \multimap X) \multimap X$, where A list abbreviates $\mu\alpha.\mathtt{unit} + A \times \alpha$.

9 Conclusion

We have observed that the non-standard compilation methods based on computation by interaction are closely related to CPS translation and defunctionalization. The interpretation in an interactive model may be regarded as a simple direct description of the combination of CPS translation, defunctionalization and a final optimisation of arguments. Subexponential types, in this form originally introduced in IntML, provide a logical account for the issues of managing value environments inherent to defunctionalization. The use of recursive algebraic data types in defunctionalization is explained by type theoretic means in rule COPY, Prop. 4 and the type of the fixed point combinator.

Subexponentials refine the exponentials in AJM games [1], where $!X$ is implemented using $\mathbb{N} \cdot X$. If we had used full exponentials in the Int interpretation above, then we would have obtained a compilation that encodes function values as numbers in $\mathbb{N}$, which is akin to storing closures on the heap. Subexponentials give us more control to avoid such encodings where unnecessary.

The subexponential type system bears resemblance to the linear logic with subexponentials of [15], which is where the terminology of subexponentials was first used. The type system is also quite similar to the type system for Syntactic Control of Concurrency (SCC) in [8]. A main difference appears to be that while SCC controls the number of program threads, subexponentials account for both the threads and their local data.

The observation that there is a connection between game models and continuations is not new. It appears for example in Levy's work on a jump-with-argument calculus [13]. Connections of game models to compilation have been made in [14], for example. It is also well-known that continuation passing is related to message passing, see e.g. [21]. However, we are not aware of work that makes explicit a connection to defunctionalization. We believe that the connection between game models and machine languages deserves to be better known and studied further. The call traces in this paper, for example, should have the same status as plays in a game semantic model.

In further work, we should like to understand if there are connections to Danvy's work on defunctionalized interpreters [6], or more generally to work on abstract machines, e.g. [20, **?**]. A relation is not obvious: Danvy considers the defunctionalization of particular implementations of interpreters, while here we show that the whole compilation itself may be described extensionally by the Int construction.

In another direction, an interactive view of CPS transform and defunctionalization may also give insight into issues that are often seen as problematic in the context of defunctionalization, such as compositional compilation and polymorphism: the interpretation in Int($\mathbb{T}$) is compositional and polymorphism can also be accounted for [19].

References

1. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. Comput. 163(2), 409–470 (2000)
2. 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. Lecture Notes in Computer Science, vol. 2215, pp. 420–447. Springer (2001)
3. Cejtin, H., Jagannathan, S., Weeks, S.: Flow-directed closure conversion for typed languages. In: Smolka, G. (ed.) ESOP. Lecture Notes in Computer Science, vol. 1782, pp. 56–71. Springer (2000)
4. Dal Lago, U., Schöpp, U.: Functional programming in sublinear space. In: Gordon, A.D. (ed.) ESOP. Lecture Notes in Computer Science, vol. 6012, pp. 205–225. Springer (2010)
5. Dal Lago, U., Schöpp, U.: Type inference for sublinear space functional programming. In: Ueda, K. (ed.) APLAS. Lecture Notes in Computer Science, vol. 6461, pp. 376–391. Springer (2010)
6. Danvy, O.: Defunctionalized interpreters for programming languages. In: Hook, J., Thiemann, P. (eds.) ICFP. pp. 131–142. ACM (2008)
7. Ghica, D.R.: Geometry of synthesis: a structured approach to VLSI design. In: Hofmann, M., Felleisen, M. (eds.) POPL. pp. 363–375. ACM (2007)
8. Ghica, D.R., Murawski, A.S., Ong, C.H.L.: Syntactic control of concurrency. Theor. Comput. Sci. 350(2-3), 234–251 (2006)
9. Hasegawa, M.: On traced monoidal closed categories. Mathematical Structures in Computer Science 19(2), 217–244 (2009)
10. Hofmann, M., Streicher, T.: Continuation models are universal for lambda-mu-calculus. In: LICS. pp. 387–395. IEEE Computer Society (1997)
11. Hyland, J.M.E., Ong, C.H.L.: On full abstraction for PCF: I, II, and III. Inf. Comput. 163, 285–408 (December 2000)
12. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. Math. Proc. Cambridge Philos. Soc. 119(3), 447–468 (1996)
13. Levy, P.B.: Call-By-Push-Value: A Functional/Imperative Synthesis, Semantics Structures in Computation, vol. 2. Springer-Verlag, Berlin, Heidelberg (2004)
14. Melliès, P.A., Tabareau, N.: An algebraic account of references in game semantics. Electr. Notes Theor. Comput. Sci. 249, 377–405 (2009)
15. Nigam, V., Miller, D.: Algorithmic specifications in linear logic with subexponentials. In: Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'09). pp. 129–140. ACM, New York, NY (2009)
16. Pierce, B.C.: Types and programming languages. MIT Press (2002)
17. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. 1(2), 125–159 (1975)
18. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the ACM annual conference - Volume 2. pp. 717–740. ACM '72, ACM (1972)
19. Schöpp, U.: Stratified bounded affine logic for logarithmic space. In: Proceedings of the 22nd IEEE Symposium on Logic in Computer Science (LICS'07). pp. 411–420. IEEE (2007)
20. Streicher, T., Reus, B.: Classical logic, continuation semantics and abstract machines. J. Funct. Program. 8(6), 543–572 (1998)
21. Thielecke, H.: Categorical Structure of Continuation Passing Style. Ph.D. thesis, University of Edinburgh (1997)