ON THE RELATION OF INTERACTION SEMANTICS TO CONTINUATIONS AND DEFUNCTIONALIZATION

ULRICH SCHÖPP

Ludwig-Maximilians-Universität München, Germany
e-mail address: 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. for hardware synthesis or for 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, namely 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-translation followed by a defunctionalization procedure that takes into account control-flow information. We then establish a relation between these two compilation methods for the simply-typed λ-calculus and end by considering recursion.

1. INTRODUCTION

A successful approach in the semantics of programming languages is to model programs using interaction dialogues. It is fundamental to Game Semantics [23, 2], the Geometry of Interaction [17] and related lines of research. The idea goes back to the study of dialogical models of constructive logic [27], which explain the meaning of a logical sentence by how one can attack and defend it in a debate [6]. A proof of a sentence is a strategy for defending it against any possible attack. In programming language semantics, types take the place of sentences and attacks can be seen as requests for information. The meaning of a program is a strategy that explains how to answer any possible request. Programs are interpreted compositionally, so that the answer to a request depends only on how the parts of the programs answer to suitable requests. Computation is thus modelled as an interaction dialogue.

While interaction dialogues are typically considered as abstract mathematical objects, it has also been argued that they are useful for *implementing* actual computation. To compute the result of a program it is enough to have an implementation of the strategy that interprets it, i.e. a implementation that takes requests as input and that computes the strategies' answer as output. The compositional definition of the interactive interpretation guides the

construction of such an implementation. For example, one may implement the strategy for each program part by a separate module. The compositional translation of programs explain how to assemble such modules to obtain the implementation of a whole program. The modules interact with each other by a suitable form of message passing and implement the computation by playing out actual interaction dialogues. Implementations of this kind have been proposed for example in [12, 13, 8].

One main motivation for studying the implementation of interaction models is to guide the design of compilation methods for programming languages. Interaction models are typically quite concrete and suitable for implementation in simple low-level languages, but, at the same time, they have rich structure and provide accurate models for sophisticated programming languages, see e.g. [2, 23, 32].

The approach of using interactive semantics as an implementation technique for programming languages has been proposed in a variety of contexts. Mackie [28] uses ideas from the Geometry of Interaction for the implementation of functional languages. In later work it was noticed that such ideas are useful especially for the implementation of functional languages with strong resource constraints. Ghica et al. have developed methods for hardware synthesis based on Game Semantics [14, 16]. A related semantic approach has been used to design a functional programming language for sublinear space computation [8]. Other work has been motivated by the idea that strategies are implemented by communicating modules. This has inspired work on fully abstract translations from PCF to the π-calculus, such as [22, 5]. It has also been used to apply ideas from Game Semantics and the Geometry of Interaction for distributed computing [12]. In another direction, the Geometry of Interaction is being used as a basis for structuring quantum computation [20, 42]. This list of examples is certainly not exhaustive; it illustrates the wide range of applications of the implementation of interactive dialogues.

The aim of this paper is to relate compilation methods based on interaction semantics to standard techniques in the efficient compilation of functional programming languages. It has been observed before, for example by Melliès [29] and Levy [26], that interaction models are related to continuation passing, an important standard technique in the compilation of functional programming languages [3]. In this paper we make a further connection to defunctionalization [36].

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, 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 show that the composition of two well-known transformations, namely CPS-translation [35] and defunctionalization [36], is closely related to an interpretation of the source language in a model that implements interaction dialogues.

The interactive model we study in this paper is an instance of the *Int construction* [25]. This model is very basic and captures only what is needed for its intended application as an implementation technique. We believe that it is a good choice, as the Int construction has been identified as the core of a number of interactive semantics, so that our results apply to a number of interactive models. Indeed, in [1] it was shown that the (particle style) Geometry of Interaction can be seen as an instance of the Int construction with further structure. Abramsky-Jagadeesan-Malacaria (AJM) games [2] are also closely related to the Int construction. AJM games refine the Int construction by removing unwanted interaction

dialogues and by integrating a quotient to capture a good notion of program equality, see the construction in [2]. If one is interested only in implementing strategies, then one may restrict ones attention to the core given only by the Int construction.

In order to define an interpretation of a higher-order source language in an interactive model given by the Int construction, we build on work reported in [8]. As the resulting interpretation implements call-by-name, we relate it to a call-by-name CPS-translation – a variant of the one by Hofmann and Streicher [21].

Let us outline concretely how CPS-translation, defunctionalization and the interpretation in an interactive model are related by looking at 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-translation [21] translates the source term $\lambda x{:}\mathbb{N}.\, 1 + x$ to $\lambda \langle x, k \rangle.\, (\lambda k.k\ 1)\ (\lambda u.\, x\ (\lambda n.\, k\ (u+n)))\colon \neg(\neg\neg\mathbb{N} \times \neg\mathbb{N})$, where we write $\neg A$ for $A \to \bot$. This term defines a function, which takes as argument 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 [36] translates this higher-order term into a first-order program. 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 λ-abstraction is named with a unique 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 defines the function named with label l_1. It can be represented simply by the label l_1. The function with label 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 $apply(s, t)$, as s is now only the name of a function and not a 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 $apply$:

$$
\begin{aligned}
apply(f, a) = \ &\mathsf{case}\ f\ \mathsf{of}\ l_1 \Rightarrow \mathsf{let}\ \langle x, k \rangle = a\ \mathsf{in}\ apply(l_2, l_3(x, k)) \\
&\mid l_2 \Rightarrow apply(a, 1) \\
&\mid l_3(x, k) \Rightarrow apply(x, l_4(k, a)) \\
&\mid l_4(k, u) \Rightarrow apply(k, u + a)
\end{aligned}
$$

This definition should be understood as the recursive definition of a function $apply$ with two arguments. The definition is untyped, as in Reynold's original definition of defunctionalization [36].

To understand concretely how this definition represents the original term, it is perhaps useful to see what happens when a concrete argument and a continuation are supplied: $(\lambda^{l_1}\langle x, k \rangle.\, (\lambda^{l_2}k.k\ 1)\ (\lambda^{l_3}u.\, x\ (\lambda^{l_4}n.\, k\ (u+n))))\ \langle \lambda^{l_5}k.\, k\ 42,\ \lambda^{l_6}n.\, \mathtt{print\_int}(n) \rangle$. The definition

of *apply* then has two cases for l_5 and l_6 in addition to the cases above:

$$apply(l, a) = \text{case } l \text{ of } \dots$$
$$\mid l_5 \Rightarrow apply(a, 42)$$
$$\mid l_6 \Rightarrow \texttt{print\_int}(n)$$

The fully applied term defunctionalizes to $apply(l_1, \langle l_5, l_6 \rangle)$. Executing it results in 43 being printed. When we evaluate $apply(l_1, \langle l_5, l_6 \rangle)$, the first case in the definition of *apply* applies and results in the call $apply(l_2, l_3(l_5, l_6))$. For this call, the second case applies, so that the call $apply(l_3(l_5, l_6), 1)$ is made. The computation continues in this way with calls to $apply(l_5, l_4(l_6, 1))$, $apply(l_4(l_6, 1), 42)$, $apply(l_6, 43)$, and finally $\texttt{print\_int}(43)$.

   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*-function performs a case distinction on the function name each time it is invoked. However, in the example it is possible to determine the label in the first argument of each appearance of *apply* statically, so that the case distinction is not necessary. Instead, we may define one function $apply_l$ for each label l and replace $apply(l(x), a)$ by $apply_l(x, a)$. The label l thus 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. [4]. If we apply it to this example and moreover simplify the result by removing unneeded function arguments, then we get four mutually recursive functions:

$$apply_{l_1}() = apply_{l_2}() \qquad\qquad apply_{l_2}() = apply_{l_3}(1)$$
$$apply_{l_3}(u) = apply_{l_5}(u) \qquad\qquad apply_{l_4}(u, n) = apply_{l_6}(u + n) \tag{1.1}$$

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

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

   The point of this paper is that the program (1.1) is just 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 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 be returned along with the answer. Programs here do not have state and have no persistent memory to store any data until a request is answered. The program can however encode any data that it needs later in the value s and ask for this value to be returned unchanged with the answer to its request. For the outside, the value s is opaque. We do not

know anything about what is encoded in the value s, only that we have to give it back with the answer to the request. To answer the program's request, we pass a value of the form $\mathsf{inr}(s, m)$, where m is our answer.

The particular function $\lambda x{:}\mathbb{N}.\,1 + x$ is implemented by the program specified in the following diagram, where S is $\mathtt{nat}$. 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.
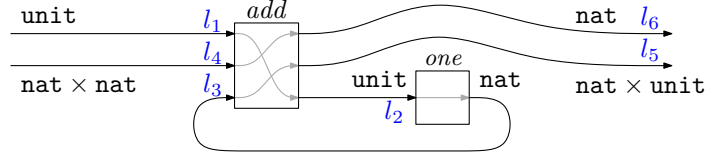

In this diagram, add has three input ports of type $\mathtt{unit}$, $\mathtt{nat} \times \mathtt{nat}$ and $\mathtt{nat}$ respectively from top to bottom. It may output a message on one of its three output ports, which have type $\mathtt{nat}$, $\mathtt{nat} \times \mathtt{unit}$ and $\mathtt{unit}$ from top to bottom. Its behaviour is given as follows: if it receives message $\langle\rangle$ on the topmost input port (a request for the sum), then it outputs $\langle\rangle$ on the bottom output port (a request to provide the first summand); if it receives n on the bottom input port (the first summand), then it outputs $\langle n, \langle\rangle\rangle$ on the middle output port (a request to provide the second summand and to hold on to the first summand until the request is answered); and if it receives $\langle n, m \rangle$ on the middle input port (both summands), it outputs $n + m$ on the topmost output port. 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 Section 6.

Compare the above interaction diagram to the definitions in (1.1) obtained by defunctionalization. The labels l_1, l_4 and l_3 there correspond to the three input ports 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 (1.1) 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.

1.3. **Overview.** The subject of this paper is the relation of the two translations that we have just outlined. The paper studies the following situation of two translations from a source language that is a variant of PCF into a simple first-order target language with tail recursion.
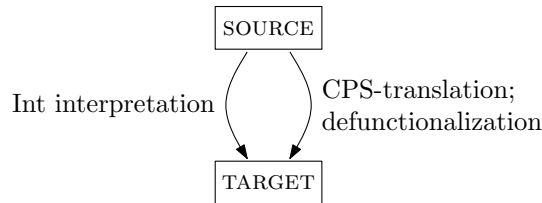
After giving definitions of the source and target language in the following two sections, we find it useful to present and analyse the above situation step by step. We define the two translations and study their relationship for a number of fragments of the source language of increasing strength.

$$\text{CORE} \subseteq \text{LIN} \subseteq \text{STL} \subseteq \text{SOURCE}$$

In Section 6 we start by studying the translation for the source fragment CORE. This fragment contains just a bare minimum of linear λ-abstraction and application. It is nevertheless instructive to consider this fragment as a setting in which to develop the infrastructure for the translation of higher-order functions.

In Section 7 we consider the fragment LIN, which extends CORE with a base type of natural numbers. Rather than studying the above situation directly with LIN in place of SOURCE, we argue that it is useful to take a detour over a calculus LIN_{EXP}, which is a version of LIN with additional type annotations. These type annotations are useful for understanding the Int-interpretation.

In Section 8 we then add contraction and come to the simply typed fragment STL of the source language. We first continue to use additional type annotations and extend LIN_{EXP} to STL_{EXP}. We then come back to the unannotated source fragment STL by showing how STL can be translated into STL_{EXP} (Prop. 8.7).

In Section 9 we finally then extend the translation to the full source language by adding recursion.

2. Target Language

Programs in the target language consist of mutually tail-recursive definitions of first-order functions, such as the *apply*-equations above. One should think of the target language as a simple variant of SSA-form compiler intermediate languages, e.g. [7], in which function definitions are often presented as labelled blocks that end with a jump to a label.

The target language does not model function calls or a calling convention; it models only what one would use for the compilation of a single unit. Certain function labels are designated as entry or exit points. In the following example target program the labels *const* and *pow* are intended as entry points.

$$const(x) = const\_ret(23)$$
$$pow(\langle x, y \rangle) = pow\_loop(\langle x, y \rangle)$$
$$pow\_loop(\langle x, y \rangle) = \textsf{case iszero}(x) \textsf{ of inl}(z) \Rightarrow pow\_ret(y)$$
$$; \textsf{inr}(z) \Rightarrow pow\_loop(\langle x - 1, y * y \rangle)$$

The function labels *const_ret* and *pow_ret* are exit points that are assumed to be defined externally and that are used to return the results of computations.

A target program will be a set of equations together with lists of entry and exit labels that specify the interface of the program. Target programs are defined in detail in the rest of this section. Upon first reading, the reader may wish to skim this section only.

Target programs are typed. The set of *target types* is defined by the grammar below. Recursive types will be needed at the end of Section 8 only. *Target expressions* are standard

terms for these types, see e.g. [34]:

$$\text{Types:} \qquad A, B ::= \alpha \mid \text{unit} \mid \text{nat} \mid A \times B \mid A + B \mid \mu\alpha.\,A$$

$$\text{Expressions:} \quad e, e_1, e_2 ::= x \mid \langle\rangle \mid n \mid e_1 + e_2 \mid \text{iszero}(e)$$
$$\mid \langle e_1, e_2 \rangle \mid \text{let } \langle x, y \rangle = e_1 \text{ in } e_2$$
$$\mid \text{inl}(e) \mid \text{inr}(e) \mid \text{case } e \text{ of inl}(x) \Rightarrow e_1; \text{inr}(y) \Rightarrow e_2$$
$$\mid \text{fold}_A(e) \mid \text{unfold}_A(e)$$

In the syntax, α ranges over type variables, x over expression variables, and n over natural numbers as constants. We identify terms up to renaming of bound variables. The term let $\langle x, y \rangle = e_1$ in e_2 binds the variables x and y in e_2 and case e of inl$(x) \Rightarrow e_1$; inr$(y) \Rightarrow e_2$ binds variable x in e_1 and variable y in e_2. The term $\text{iszero}(e)$ is intended to have type $\text{unit} + \text{unit}$, with $\text{inl}(\langle\rangle)$ representing true.

We remark that the type nat is used solely to encode values of the source type of natural numbers $\mathbb{N}$. For applications to compilation, one may be interested in restricting the natural numbers to, say, 64-bit integers. Such a restriction can be made without affecting the results in this paper.

Target expressions are typed with a standard type system, see Figure 1. A judgement $\Gamma \vdash e : A$ therein expresses that e has type A in context Γ, where Γ is a finite mapping from variables to target types.

For convenience, we allow ourselves ML-like data type notation for working with recursive types. For example, for a type of lists we may write

$$\beta \text{ list} = \text{datatype nil of unit} \mid \text{cons of } \beta \times (\beta \text{ list})$$

instead of $\mu\alpha.\,\text{unit} + \beta \times \alpha$, as $\text{cons}(x, l)$ is more readable than $\text{fold}_{\mu\alpha.\,\text{unit}+\beta\times\alpha}(\text{inr}(\langle x, l \rangle))$.

For the operational semantics of target expressions we define a standard call-by-value small-step reduction relation. We use the concepts of *target values* and *evaluation contexts*:

$$\text{Values:} \quad v, w ::= \langle\rangle \mid n \mid \langle v, w \rangle \mid \text{inl}(v) \mid \text{inr}(v) \mid \text{fold}_A(v)$$

$$\text{Evaluation Contexts:} \quad C ::= [] \mid C + e \mid v + C \mid \text{iszero}(C)$$
$$\mid \langle C, e \rangle \mid \langle v, C \rangle \mid \text{let } \langle x, y \rangle = C \text{ in } e$$
$$\mid \text{inl}(C) \mid \text{inr}(C) \mid \text{case } C \text{ of inl}(x) \Rightarrow e_1; \text{inr}(y) \Rightarrow e_2$$

The small-step reduction relation is then defined to be the smallest relation $\longrightarrow$ satisfying the following clauses:

$$n_1 + n_2 \longrightarrow n_3 \text{ if } n_3 \text{ is the sum of } n_1 \text{ and } n_2$$
$$\text{iszero}(0) \longrightarrow \text{inl}(\langle\rangle)$$
$$\text{iszero}(n) \longrightarrow \text{inr}(\langle\rangle) \text{ if } n \text{ is non-zero}$$
$$\text{let } \langle x, y \rangle = \langle v_1, v_2 \rangle \text{ in } e \longrightarrow e[v_1/x, v_2/y]$$
$$\text{case inl}(v) \text{ of inl}(x) \Rightarrow e_1; \text{inr}(y) \Rightarrow e_2 \longrightarrow e_1[v/x]$$
$$\text{case inr}(v) \text{ of inl}(x) \Rightarrow e_1; \text{inr}(y) \Rightarrow e_2 \longrightarrow e_2[v/x]$$
$$\text{unfold}(\text{fold}(v)) \longrightarrow v$$
$$C[e_1] \longrightarrow C[e_2] \text{ if } e_1 \longrightarrow e_2$$

Proposition 2.1. *For each $\vdash e : A$ there exists a unique value v satisfying $e \longrightarrow^* v$.*

$$\frac{x \colon A \text{ in } \Gamma}{\Gamma \vdash x \colon A} \qquad \overline{\Gamma \vdash \langle\rangle \colon \mathtt{unit}}$$

$$\overline{\Gamma \vdash n \colon \mathtt{nat}} \qquad \frac{\Gamma \vdash e_1 \colon \mathtt{nat} \qquad \Gamma \vdash e_2 \colon \mathtt{nat}}{\Gamma \vdash e_1 + e_2 \colon \mathtt{nat}} \qquad \frac{\Gamma \vdash e \colon \mathtt{nat}}{\Gamma \vdash \mathsf{iszero}(e) \colon \mathtt{unit} + \mathtt{unit}}$$

$$\frac{\Gamma \vdash e_1 \colon A \qquad \Gamma \vdash e_2 \colon B}{\Gamma \vdash \langle e_1, e_2 \rangle \colon A \times B} \qquad \frac{\Gamma \vdash e_1 \colon A \times B \qquad \Gamma, x \colon A, y \colon B \vdash e_2 \colon C}{\Gamma \vdash \mathsf{let} \ \langle x, y \rangle = e_1 \ \mathsf{in} \ e_2 \colon C}$$

$$\frac{\Gamma \vdash e \colon A}{\Gamma \vdash \mathsf{inl}(e) \colon A + B} \qquad \frac{\Gamma \vdash e \colon B}{\Gamma \vdash \mathsf{inr}(e) \colon A + B}$$

$$\frac{\Gamma \vdash e \colon A + B \qquad \Gamma, x \colon A \vdash e_1 \colon C \qquad \Gamma, y \colon B \vdash e_2 \colon C}{\Gamma \vdash \mathsf{case} \ e \ \mathsf{of} \ \mathsf{inl}(x) \Rightarrow e_1; \ \mathsf{inr}(y) \Rightarrow e_2 \colon C}$$

$$\frac{\Gamma \vdash e \colon A[\mu\alpha. \, A/\alpha]}{\Gamma \vdash \mathsf{fold}_{\mu\alpha.A}(e) \colon \mu\alpha.A} \qquad \frac{\Gamma \vdash e \colon \mu\alpha.A}{\Gamma \vdash \mathsf{unfold}_{\mu\alpha.A}(e) \colon A[\mu\alpha. \, A/\alpha]}$$

Figure 1: Typing of Target Expressions

Having defined target expressions, we are now ready to define target *programs*. These consist of a set of first-order function definitions. Fix an infinite set $\mathcal{L}$ of function labels.

Definition 2.2. A *function definition* for label $f \in \mathcal{L}$ is given by an equation of one of the two forms

$$f(x) = g(e), \qquad\qquad f(x) = \mathsf{case} \ e \ \mathsf{of} \ \mathsf{inl}(y) \Rightarrow g(e_1); \ \mathsf{inr}(z) \Rightarrow h(e_2),$$

wherein $g, h \in \mathcal{L}$ and e, e_1 and e_2 range over target expressions.

We allow ourselves to use syntactic sugar, writing $f()$ for $f(\langle\rangle)$ and $f(x, y) = t$ for $f(z) = t[\mathsf{let} \ \langle x, y \rangle = z \ \mathsf{in} \ x/x, \mathsf{let} \ \langle x, y \rangle = z \ \mathsf{in} \ y/y]$, for example.

Definition 2.3. A *target program* $P = (i, D, o)$ consists of a set D of function definitions together with a list $i \in \mathcal{L}^*$ of entry labels and a list $o \in \mathcal{L}^*$ of exit labels. Both i and o must be lists of pairwise distinct labels. The set D of definitions must contain at most one definition for any label and must not contain any definition for the labels in o.

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

We use an informal graphical notation for target programs, depicting for example the program $(const \ pow \ f, D, const\_ret \ pow\_ret)$ as shown below.
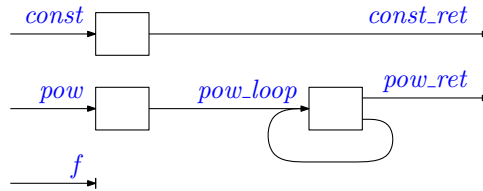
The boxes correspond to the function definitions. The arrows indicate for example that one may send a value v to label pow, which amounts to the function call $pow(v)$. As a result a value will be sent to pow_loop.

Target programs are well-typed if each function symbol f can be assigned an argument type $A(f)$ such that each definition is well-typed: A definition of the form $f(x) = g(e)$ is well-typed if $x\colon A(f) \vdash e\colon A(g)$ is derivable; and a definition of the form $f(x) = \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(y) \Rightarrow g(e_1); \mathsf{inr}(z) \Rightarrow h(e_2)$ is well-typed if $x\colon A(f) \vdash e\colon C_1 + C_2$ and $y\colon C_1 \vdash e_1\colon A(g)$ and $z\colon C_2 \vdash e_2\colon A(h)$ are derivable for some C_1 and C_2. If P is the program $(f_1 \ldots f_n, D, g_1 \ldots g_m)$, then we write $P\colon (A_1 \ldots A_n) \to (B_1 \ldots B_m)$ if the argument types of $f_1, \ldots, f_n, g_1, \ldots, g_m$ are $A_1, \ldots, A_n, B_1, \ldots, B_m$ respectively.

We define a simple evaluation semantics for target 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 $\longrightarrow_P$ 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 $f(v) \longrightarrow_P g(w)$ for all values v and w with $e[v/x] \longrightarrow^* w$; 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 $f(v) \longrightarrow_P g(w)$ for all values v and w with $\exists u.\ e[v/x] \longrightarrow^* \mathsf{inl}(u) \wedge e_1[u/y] \longrightarrow^* w$, and $f(v) \longrightarrow_P h(w)$ for all values v and w with $\exists u.\ e[v/x] \longrightarrow^* \mathsf{inr}(u) \wedge e_2[u/z] \longrightarrow^* w$.

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

Definition 2.4 (Program Equality). Two programs $P, Q\colon (A_1 \ldots A_n) \to (B_1 \ldots B_m)$ are *equal* if, for any input, they give the same output, that is, suppose the entry labels of P and Q are $f_1, \ldots, f_n$ and $g_1, \ldots, g_n$ respectively and the exit labels are $h_1, \ldots, h_m$ and $k_1, \ldots, k_m$ respectively, then, for any v, w, i and j, P has a call-trace of the form $f_i(v) \ldots h_j(w)$ if and only if Q has a call-trace of the form $g_i(v) \ldots k_j(w)$.

Programs are thus equal, if the same input value on the same input port leads to the same output value (if any) on the same output port in both programs.

The following notation is used in Section 7. For any list of target types $X = B_1 \ldots B_n$ and any target type A, we write $A \cdot X$ for the list $(A \times B_1) \ldots (A \times B_n)$. 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 of the form $f(x) = g(e)$ in P with $f(u, x) = g(u, e)$ for a fresh variable u, and each definition of the form $f(x) = \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(y) \Rightarrow g(e_1); \mathsf{inr}(z) \Rightarrow h(e_2)$ with $f(u, x) = \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(y) \Rightarrow g(u, e_1); \mathsf{inr}(z) \Rightarrow h(u, e_2)$, again for fresh u.

We observe that target programs can be organised into a category $\mathbb{T}$ that has enough structure so that we can apply the Int construction [25, 19] (with respect to coproducts) to it and obtain a category $\mathrm{Int}(\mathbb{T})$ that models interactive computation.

Target programs can be organised into a category $\mathbb{T}$. Its objects are finite lists of target types. A morphism from X to Y is given by a program $P\colon X \to Y$. Two programs $P\colon X \to Y$ and $Q\colon X \to Y$ represent the same morphism if and only if they are equal in the sense of Definition 2.4. Thus, the morphisms from X to Y of $\mathbb{T}$ are the equivalence classes of programs of type $X \to Y$ with respect to program equality.

Lemma 2.5. $\mathbb{T}$ *is a category.*

Proof outline. The identity on $X = A_1 \ldots A_n$ is the program $(f_1 \ldots f_n, \emptyset, f_1 \ldots f_n)$. For the composition of $P\colon X \to Y$ and $Q\colon Y \to Z$, we first note that we can rename the

labels P and Q such that we have $P = (i, D_P, m)$ and $Q = (m, D_Q, o)$. The composition $Q \circ P\colon X \to Z$ is then given simply by the program $(i, D_P \cup D_Q, o)$. $\qquad\square$

Lemma 2.6. *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* [19] *with respect to these coproducts.*

Proof outline. A simple proof can be given by observing that there is a faithful embedding from $\mathbb{T}$ to the category of sets and partial functions. The equations that are required to show then follow from the fact that the category of sets and partial functions has the desired structure. $\qquad\square$

While we would like to emphasise the mathematical structure of target programs given by the Int construction, in the rest of the paper we shall spell it out concretely rather than referring to categorical notions in order to make the paper easier to read.

3. Source Language

Our source language is a variant of PCF, a simply-typed λ-calculus with a basic type $\mathbb{N}$ of natural numbers and associated constants, as well as a fixed-point combinator for recursion. The intended evaluation strategy is call-by-name.

The source language has the following types and terms.

Types: $X, Y \ ::= \ 1 \ \mid \ X \to Y \ \mid \ \mathbb{N}$

Terms: $s, t \ ::= \ * \ \mid \ \lambda x{:}X.\, t \ \mid \ s\ t \ \mid \ n \ \mid \ s + t \ \mid \ \mathsf{if0}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \ \mid \ \mathsf{fix}_X$

We write $\neg X$ as an abbreviation for the type $X \to \bot$. Again, we identify terms up to renaming of bound variables.

The typing judgement has the form $\Gamma \vdash t\colon X$, where Γ is a finite list of variable declarations $x_1\colon X_1, \ldots, x_n\colon X_n$. We formulate the typing rules so that it is easy to consider fragments of the source language of varying expressiveness. The core rules are those of a linear λ-calculus and are given in Figure 2. The rules for natural numbers appear in Figure 3. We allow an addition operation $s + t$ instead of the standard successor operation $\mathsf{succ}(s)$, as this gives a simple example to explain the issues with the compilation of multinary operation. Rules for contraction and for the fixed point combinator are given in Figures 4 and 5.

4. CPS-Translation

We use a variant of Hofmann and Streicher's *call-by-name CPS-translation* [21], which translates the source language extended with the following rules for product types as well as a type $\bot$ without any rules.

$$\times\mathrm{I}\ \frac{\Gamma \vdash s\colon X \qquad \Delta \vdash t\colon Y}{\Gamma,\, \Delta \vdash \langle s, t\rangle\colon X \times Y} \qquad \times\mathrm{E}\ \frac{\Gamma \vdash s\colon X \times Y \qquad \Delta,\, x\colon X,\, y\colon Y \vdash t\colon Z}{\Gamma,\, \Delta \vdash \mathsf{let}\ \langle x, y\rangle = s\ \mathsf{in}\ t\colon Z}$$

For each source type X, the type $\underline{X}$ of its continuations is defined by:

$$\underline{1} = \neg 1 \qquad\qquad \underline{\mathbb{N}} = \neg\mathbb{N} \qquad\qquad \underline{X \to Y} = \neg\underline{X} \times \underline{Y}$$

A continuation for type $X \to Y$ is thus a pair of a continuation of type $\underline{Y}$, using which the result can be returned, and a function $\neg\underline{X}$ to access the argument. A function can request

$$\text{AX } \frac{}{x\colon X \vdash x\colon X} \qquad \text{1I } \frac{}{\vdash *\colon 1}$$

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

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

Figure 2: Source Language (CORE) – Linear Core

$$\text{NUM } \frac{}{\vdash n\colon \mathbb{N}} \qquad \text{ADD } \frac{\Gamma \vdash s\colon \mathbb{N} \qquad \Delta \vdash t\colon \mathbb{N}}{\Gamma,\, \Delta \vdash s+t\colon \mathbb{N}} \qquad \text{IF } \frac{\Gamma \vdash s\colon \mathbb{N} \qquad \Delta_1 \vdash t_1\colon \mathbb{N} \qquad \Delta_2 \vdash t_2\colon \mathbb{N}}{\Gamma,\, \Delta_1,\, \Delta_2 \vdash \mathsf{if0}\; s\; \mathsf{then}\; t_1\; \mathsf{else}\; t_2\colon \mathbb{N}}$$

Figure 3: Source Language (LIN) – Natural Numbers

$$\text{CONTR } \frac{\Gamma,\, y\colon X,\, z\colon X \vdash t\colon Y}{\Gamma,\, x\colon X \vdash t[x/y, x/z]\colon Y}$$

Figure 4: Source Language (STL) – Contraction

$$\text{FIX } \frac{}{\vdash \mathsf{fix}_X\colon (X \to X) \to X}$$

Figure 5: Source Language – Recursion

its argument by applying this function to a continuation of type $\underline{X}$. The argument will then be provided to this continuation.

For computation in continuation passing style, we often use the type $\neg\underline{X}$, which we denote by $\overline{X}$.

The CPS-translation translates the source language into itself, translating any typing derivation of $x_1\colon X_1, \ldots, x_n\colon X_n \vdash t\colon Y$ into a derivation of $x_1\colon \overline{X_1}, \ldots, x_n\colon \overline{X_n} \vdash \underline{t}\colon \overline{Y}$. It is defined by induction on the given typing derivation. Figure 6 shows how each typing rule on the left is translated to a derived rule on the right.

This CPS-translation differs from the standard call-by-name CPS-translation of [21] in the use of η-expansion in the rules for variables and contraction. These expansions will allow us to use compositional reasoning in Sections 6-8. The term $\eta(t, X)$ is defined by induction on the type X:

$$\eta(t, X) = t \text{ if } X \text{ is a base type } (1, \mathbb{N} \text{ or } \bot)$$
$$\eta(t, X \times Y) = \mathsf{let}\; \langle x, y \rangle = t \text{ in } \langle \eta(x, X), \eta(y, Y) \rangle$$
$$\eta(t, X \to Y) = \lambda x.\, \eta(t\; \eta(x, X), Y) \text{ where } x \text{ is fresh.}$$

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

$$\frac{}{\vdash *\colon 1} \qquad \Longrightarrow \qquad \frac{}{\vdash \lambda k.\, k\, *\colon \overline{1}}$$

$$\frac{\Gamma \vdash t\colon Y}{\Gamma,\, x\colon X \vdash t\colon Y} \qquad \Longrightarrow \qquad \frac{\overline{\Gamma} \vdash \underline{t}\colon \overline{Y}}{\overline{\Gamma},\, x\colon \overline{X} \vdash \underline{t}\colon \overline{Y}}$$

$$\frac{\Gamma,\, y\colon Y,\, x\colon X,\, \Delta \vdash t\colon Z}{\Gamma,\, x\colon X,\, y\colon Y,\, \Delta \vdash t\colon Z} \qquad \Longrightarrow \qquad \frac{\overline{\Gamma},\, y\colon \overline{Y},\, x\colon \overline{X},\, \overline{\Delta} \vdash \underline{t}\colon \overline{Z}}{\overline{\Gamma},\, x\colon \overline{X},\, y\colon \overline{Y},\, \overline{\Delta} \vdash \underline{t}\colon \overline{Z}}$$

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

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

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

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

$$\frac{\Gamma \vdash s\colon \mathbb{N} \quad \Delta_1 \vdash t_1\colon \mathbb{N} \quad \Delta_2 \vdash t_2\colon \mathbb{N}}{\Gamma,\, \Delta_1,\, \Delta_2 \vdash \mathsf{if0}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\colon \mathbb{N}} \qquad \Longrightarrow$$

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

$$\frac{\Gamma,\, y\colon X,\, z\colon X \vdash t\colon Y}{\Gamma,\, x\colon X \vdash t[x/y, x/z]\colon Y} \qquad \Longrightarrow \qquad \frac{\overline{\Gamma},\, y\colon \overline{X},\, z\colon \overline{X} \vdash \underline{t}\colon \overline{Y}}{\overline{\Gamma},\, x\colon \overline{X} \vdash \underline{t}[\eta(x,\overline{X})/y, \eta(x,\overline{X})/z]\colon \overline{Y}}$$

$$\frac{}{\vdash \mathsf{fix}_X\colon (X \to X) \to X} \quad \Longrightarrow \quad \frac{}{\vdash \lambda\langle f,k\rangle.\mathsf{fix}_{\overline{X}}\, (\lambda g.\lambda k_1.\, f\ \langle \lambda k_2.\, g\ k, \lambda x.\, k_1\, x\rangle)\, k}$$
$$\colon \overline{(X \to X) \to X}$$

Figure 6: CPS-translation

The last equation is more general than what we need in this paper. We use only the special case $\eta(t, X \to \bot) = \lambda x.\, t\ \eta(x, X)$, as we apply η-expansion only to terms with types of the form $\overline{Z}$. For example, we have $\eta(x, \overline{\mathbb{N}}) = \eta(x, \neg\neg\mathbb{N}) = \lambda x_1.\, x\ (\lambda x_2.\, x_1\ x_2)$.

In the examples in the Introduction, we have not applied this η-expansion for better readability.

5. Defunctionalization

In the translation from source to target, we first apply CPS-translation and then use defunctionalization. In this paper we use the flow-based defunctionalization procedure introduced by Banerjee, Heintze and Riecke [4]. This procedure uses control flow information, so the CPS-translated term is first annotated with control flow information and then defunctionalized using this information.

In this section we define a particularly simple special case of the flow-based defunctionalization procedure. It is too weak to handle the whole source language, but it allows

for a simple explanation of the relation to the Int construction. We will extend the de-functionalization procedure in Section 8 to cover the CPS-translation of the whole source language.

Control flow information is added to the terms in the form of labelling annotations. In the simple variant of the defunctionalization procedure that we describe here, each function abstraction and application is annotated with a single label from $\mathcal{L}$. Thus, the terms $\lambda x{:}X.\,t$ and $s\,t$ are replaced by $\lambda^l x{:}X.\,t$ and $s\,@_l\,t$ respectively, where l ranges over $\mathcal{L}$. The function type $X \to Y$ is replaced by $X \xrightarrow{l} Y$, again for any $l \in \mathcal{L}$. We write $\neg_l X$ for $X \xrightarrow{l} \bot$.

We require that each abstraction be uniquely identified by its label, that is, 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 is defined by an abstraction with label l. The typing rules for abstraction and application are modified as follows to enforce that terms are annotated with correct control flow information.

$$\frac{\Gamma,\,x\colon X \vdash t\colon Y}{\Gamma \vdash \lambda^l x{:}X.\,t\colon X \xrightarrow{l} Y} \qquad \frac{\Gamma \vdash s\colon X \xrightarrow{l} Y \qquad \Delta \vdash t\colon X}{\Gamma,\,\Delta \vdash s\,@_l\,t\colon Y}$$

Allowing function types and applications to be annotated with a single label only is a real restriction. For example, it is not possible to label and type terms such as $\lambda x.\,\langle x\,(\lambda y.\,0), x\,(\lambda z.\,1)\rangle$. The two abstractions $\lambda y.\,0$ and $\lambda z.\,1$ would each have to be given a unique label, say l_1 and l_2 respectively. But then in the two uses of the variables x, its types would have to be $X \xrightarrow{l_1} Y$ and $X \xrightarrow{l_2} Y$ respectively. With the above rules, this is not possible, as l_1 and l_2 are different labels. In general, one needs to allow types such as $X \xrightarrow{\{l_1,l_2\}} Y$ with more than a single label for more than one possible definition site, as in e.g. [4]. We come back to this in Section 8, but up until then the variant with a single label suffices and simplifies the exposition.

In the rest of this section we explain how terms of the labelled source language (with product types) can be defunctionalized into programs in the target language. We defer the question of how to annotate the terms obtained by CPS-translation with labels to later sections (Lemmas 6.2 and 8.3).

The defunctionalization of a term t in the labelled source language consists of a target expression t^*, which denotes the defunctionalized term itself, and a set of target equations $D(t)$, which contains the *apply*-definitions for defunctionalized function application.

$$x^* = x$$
$$n^* = n$$
$$(\text{if0 } s \text{ then } t \text{ else } u)^* = \text{case iszero}(s^*) \text{ of inl}(\_) \Rightarrow t^*; \text{inr}(\_) \Rightarrow u^*$$
$$\langle s,t\rangle^* = \langle s^*,t^*\rangle$$
$$(\text{let } \langle x,y\rangle = t \text{ in } s)^* = \text{let } \langle x,y\rangle = t^* \text{ in } s^*$$
$$(s\,@_l\,t)^* = apply_l(s^*,t^*)$$
$$(\lambda^l x{:}A.\,t)^* = \langle x_1,\ldots,x_n\rangle \text{ where } \mathrm{FV}(\lambda x{:}A.\,t) = \{x_1,\ldots,x_n\}$$

In the last case for abstraction we assume some fixed global ordering on all variables, so that the order of the tuple is well-defined.

$$D(x) = \emptyset$$
$$D(n) = \emptyset$$
$$D(\text{if0 } s \text{ then } t \text{ else } u) = D(s) \cup D(t) \cup D(u)$$
$$D(\langle s, t \rangle) = D(s) \cup D(u)$$
$$D(\text{let } \langle x, y \rangle = t \text{ in } s) = D(t) \cup D(s)$$
$$D(s \,@_l\, t) = D(s) \cup D(t)$$
$$D(\lambda^l x{:}A.\, t) = D(t) \cup \{ apply_l(\langle x_1, \ldots, x_n \rangle, x) = t^* \}$$

In general, the set $D(t)$ need not consist of function definitions in the strict sense of Definition 2.2; it may contain nested case distinctions, for example. This technical issue could easily be solved in general at the expense of making the technical development a little more complicated. However, we shall use defunctionalization only for terms t for which $D(t)$ does in fact only consist of function definitions, so we stick with the above simple definitions.

Note that for closed terms of function type the target expression t^* is just $\langle \rangle$. Since all closed terms $\underline{t}$ obtained by CPS-translation are of function type, we therefore consider the definition set $D(\underline{t})$ as the main result of defunctionalization.

Example 5.1. With label annotations the example from the Introduction becomes the term $\underline{t}$ given by

$$\lambda^{l_1} z.\, \text{let } \langle x, k \rangle = z \text{ in } (\lambda^{l_2} k'.\, k' \,@_{l_3}\, 1) \,@_{l_2}\, (\lambda^{l_3} u.\, x \,@_{l_5}\, (\lambda^{l_4} n.\, k \,@_{l_6}\, (u + n))).$$

Its type is $\neg_{l_1}(\neg_{l_5}\neg_{l_4}\mathbb{N} \times \neg_{l_6}\mathbb{N})$. The set $D(\underline{t})$ consists of the definitions

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

Compared to the definitions given in (1.1) in 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.

An important point to note is that the defunctionalization procedure yields a set of definition equations, but that it does not specify an interface of entry and exit labels. When one applies defunctionalization to a whole closed source programs of ground type, as is usually done in compilation, choosing an interface is not important. One would typically just choose a single entry label $\texttt{main}$ and a single exit label $\texttt{exit}$. If one is interested in compositionality, however, then open terms and terms of higher types must be also considered. Then one needs to fix an interface that explains how the free variables are accessed and how higher types are to be used. In the above example term $\underline{t}$, 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 from 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 function arguments. An example of such an optimisation is *lightweight defunctionalization* of Banerjee et al. [4]. We shall argue that the Int construction captures one such optimisation of the defunctionalization procedure.

6. The Core Linear Fragment

To explain the basic idea of how CPS-translation and defunctionalization relate to a model of interactive computation (namely $\mathrm{Int}(\mathbb{T})$), we first consider the simplest non-trivial case. We consider the core fragment of the source language, whose syntax is

$$\text{Types:} \qquad X, Y \quad ::= \quad 1 \ \Big| \ X \to Y$$

$$\text{Terms:} \qquad s, t \quad ::= \quad * \ \Big| \ \lambda x{:}X.\,t \ \Big| \ s\ t$$

and whose rules are just those of Figure 2. We call this source fragment CORE.

6.1. Interactive Interpretation.
First we describe directly the interpretation of this fragment of CORE in $\mathrm{Int}(\mathbb{T})$. A type X is interpreted by an interface (X^-, X^+), which consists of two finite lists X^- and X^+ of target types. Closed terms of type X will be interpreted as programs of type $P\colon X^- \to X^+$. The interfaces are defined by induction on the type:

$$1^- = \texttt{unit} \qquad\qquad (X \to Y)^- = Y^- X^+$$

$$1^+ = \texttt{unit} \qquad\qquad (X \to Y)^+ = Y^+ X^-$$

Here, $X^- Y^-$ denotes the concatenation of the lists X^- and Y^- (and likewise for the other cases).

For a context $\Gamma = x_1\colon X_1, \ldots, x_n\colon X_n$, we write Γ^- and Γ^+ for the concatenations $X_n^- \ldots X_1^-$ and $X_n^+ \ldots X_1^+$.

The interpretation of CORE is defined by induction on typing derivations. A typing derivation of $\Gamma \vdash t\colon X$ is interpreted by a morphism

$$[\![\Gamma \vdash t\colon X]\!] \colon X^- \Gamma^+ \to X^+ \Gamma^-$$

in $\mathbb{T}$ (which amounts to a morphism from (Γ^-, Γ^+) to (X^-, X^+) in the category $\mathrm{Int}(\mathbb{T})$). This interpretation is given in Figure 7. The boxes in this figure represent the inductive interpretation of the direct sub-derivations of the individual rules.

It is a slight abuse of notation to write $[\![\Gamma \vdash t\colon X]\!]$, even though the interpretation is defined not just from the sequent, but from its derivation. We believe that it is possible to justify this notation by proving that any two derivations of the same sequent the same interpretation, but in this paper we concentrate on the relation of the interpretation to CPS-translation and defunctionalization and always work with derivations.
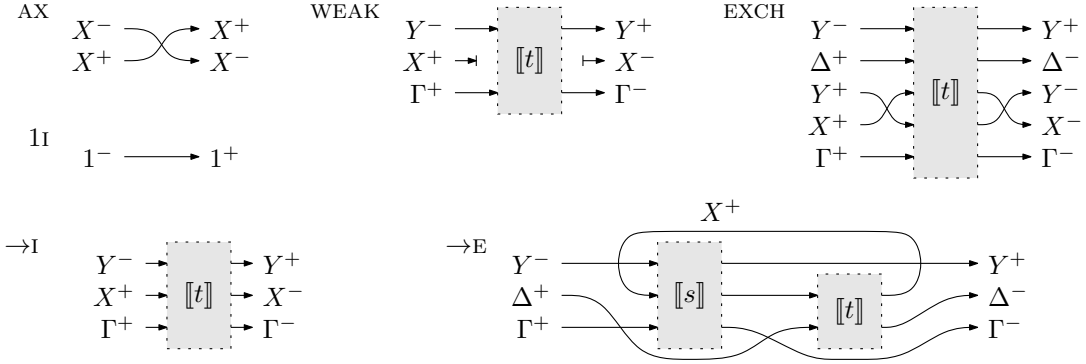
Figure 7: Int-interpretation of CORE

6.2. **CPS-translation and Defunctionalization.** The aim is now to demonstrate that this interpretation in $\mathrm{Int}(\mathbb{T})$ is closely related to CPS-translation followed by defunctionalization.

To apply flow-based defunctionalization, we must find suitable labellings of terms and types. We introduce special notation for labellings of types of the form $\overline{X}$.

Definition 6.1. For any type X and any $x^-, x^+ \in \mathcal{L}^*$ with $length(x^-) = length(X^-)$ and $length(x^+) = length(X^+)$, we define a type $\overline{X}[x^-, x^+]$ in the labelled variant of CORE inductively as follows:

(1) Define $\overline{1}[q, a]$ to be $\neg_q \neg_a 1$.
(2) If $\overline{X}[x^-, x^+]$ is defined and $\overline{Y}[y^-, y^+]$ is defined and of the form $\neg_q Y'$, then define $\overline{X \to Y}[y^- x^+, y^+ x^-]$ to be $\neg_q(\overline{X}[x^-, x^+] \times Y')$.

For example, $\overline{1 \to 1}[qa', aq']$ denotes $\neg_q(\neg_{q'} \neg_{a'} 1 \times \neg_a 1)$.

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

Readers familiar with game semantics may also want to compare the syntax trees of the types $\overline{X}[x^-, x^+]$ with game semantic arenas. The syntax tree induces a natural partial ordering on the labels appearing in it: $l_1 < l_2$ if there is a path from a node labelled $\xrightarrow{l_1}$ to one labelled $\xrightarrow{l_2}$ in the syntax tree. The Hasse diagrams of this ordering may be defined inductively as follows:
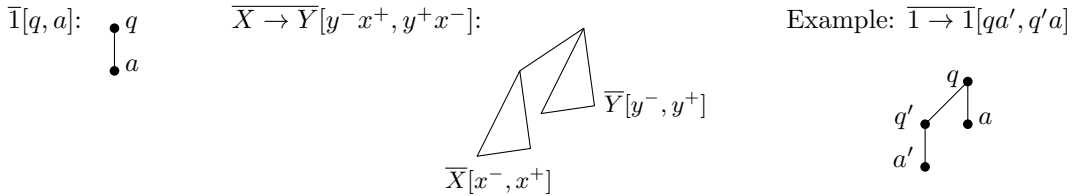
These diagrams correspond to the game semantic arenas for the corresponding types [23]. More information about the relation of game arenas and continuations can be found particularly in work of Levy [26] and Melliès [29].

If Γ is $x_1\colon X_1,\ldots,x_n\colon X_n$, then we write short $\overline{\Gamma}[x_n^-\ldots x_{\underline{1}}^-, x_n^+\ldots x_1^+]$ for the context $x_1\colon \overline{X_1}[x_1^-,x_1^+],\ldots,x_n\colon \overline{X_n}[x_n^-,x_n^+]$. 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.

Lemma 6.2. *If* $\Gamma \vdash t\colon X$ *is derivable in* CORE, *then the derivation of* $\overline{\Gamma} \vdash \underline{t}\colon \overline{X}$ *obtained by CPS-translation can be annotated with labels such that it derives the well-labelled sequent* $\overline{\Gamma}[\gamma^-,\gamma^+] \vdash \underline{t}\colon \overline{X}[x^-,x^+]$ *for some* $\gamma^-,\gamma^+,x^-,x^+ \in \mathcal{L}^*$.

The proof is a straightforward induction on derivations. We note that the η-expansion in the CPS-translation of variables is essential for this lemma to be true. For example, with the η-expansion a well-labelled $x\colon \overline{1}[q,a] \vdash \underline{x}\colon \overline{1}[q',a']$ is derivable; without it this would only be possible if $q = q'$ and $a = a'$. That η-expansions of variables can be labelled as needed follows from the more general property established in the proof of Lemma 8.2 below. The defunctionalization of $\underline{x}$ consists of definitions of $apply_{q'}$ and $apply_a$, which just forward their arguments to $apply_q$ and $apply_{a'}$ 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 possible second step.

We now define a function $\mathsf{CpsDefun}$ that combines CPS-translation and defunctionalization. Given any CORE-derivation of a judgement $\Gamma \vdash t\colon X$, 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. The function $\mathsf{CpsDefun}$ maps the source derivation of $\Gamma \vdash t\colon X$ to the target program $(x^-\gamma^+, D(\underline{t}), x^+\gamma^-)$, where $D(\underline{t})$ is the set of equations obtained by the defunctionalization of $\underline{t}$. It is not hard to see that the set $D(\underline{t})$ is indeed a target program whose definition does not depend on the choice of labels.

We define a single function $\mathsf{CpsDefun}$ rather than a composition of two general functions Cps and Defun, as in general there is no canonical choice of entry and exit labels for defunctionalization. Thus, the composition $\mathsf{Defun} \circ \mathsf{Cps}$ would only return a set of equations and not yet a 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 that erases all function arguments.

$$\mathsf{Erase}(i, E, o) := (i, \{f() = g() \mid f(x) = g(e) \in E\}, o)$$

In fact, Erase also removes all equations defined by case distinction, but these do not appear in $D(\underline{t})$ for this source language.

The composition $\mathsf{Erase} \circ \mathsf{CpsDefun}$ of these two functions takes (a typing derivation of) a source program, applies the CPS-translation, defunctionalizes and then 'optimises' the result by erasing all function arguments. The resulting program is in fact correct and it is what one obtains using the interpretation in $\mathrm{Int}(\mathbb{T})$:

Proposition 6.3. *Suppose* $\Gamma \vdash t\colon X$ *is derivable in* CORE. *Then the target program* $\mathsf{Erase}(\mathsf{CpsDefun}(\Gamma \vdash t\colon X))$ *has type* $X^-\Gamma^+ \to X^+\Gamma^-$ *and defines the same morphism in* $\mathbb{T}$ *as the Int-interpretation of* $\Gamma \vdash t\colon X$.

Since morphisms of type $X^-\Gamma^+ \to X^+\Gamma^-$ in $\mathbb{T}$ are defined to be equivalence classes of programs up to program equality (Definition 2.4), the Int-interpretation $[\![\Gamma \vdash t\colon X]\!]$ is an equivalence class of programs. The assertion of the proposition is therefore that the program $\mathsf{Erase}(\mathsf{CpsDefun}(\Gamma \vdash t\colon X))$ is an element of the equivalence class $[\![\Gamma \vdash t\colon X]\!]$.

Proof. The proof goes by induction on the derivation of $\Gamma \vdash t\colon X$. We continue by case distinction on the last rule in the derivation and show just the representative cases for variables and functions.

- Case AX.

$$\overline{x\colon X \vdash x\colon X}$$

In this case $\mathsf{Erase}(\mathsf{CpsDefun}(\Gamma, x\colon X \vdash x\colon X))$ has the form $(x_1^- x_2^+, D, x_1^+ x_2^-)$ for

$$D = \{ apply_{x_1^-(i)}() = apply_{x_2^-(i)}(), apply_{x_2^+(i)}() = apply_{x_1^+(i)}() \mid i = 1, \ldots, n \},$$

where we denote by $w(i)$ the i-th element in the sequence w and where n is the common length of x_1^-, x_1^+, x_2^- and x_2^+. This is clearly in the equivalence class of the Int-interpretation.

- Case $\to$I.

$$\vdots$$
$$\frac{\Gamma, x\colon X \vdash t\colon Y}{\Gamma \vdash \lambda x{:}X.\, t\colon X \to Y}$$

A CPS-translation of the derivation must have the following form in which $Y[y^-, y^+] = \neg_{q_t} Y'$ and $y^- = q_t z$ for $q_t \in \mathcal{L}$ and $z \in \mathcal{L}^*$.

$$\vdots$$
$$\frac{\overline{\Gamma}[\gamma^-, \gamma^+], x\colon \overline{X}[x^-, x^+] \vdash \underline{t}\colon \overline{Y}[y^-, y^+] \qquad \overline{k\colon Y' \vdash k\colon Y'}}{\dfrac{\overline{\Gamma}[\gamma^-, \gamma^+], x\colon \overline{X}[x^-, x^+], k\colon Y' \vdash \underline{t} @_{q_t} k\colon \bot}{\overline{\Gamma}[\gamma^-, \gamma^+] \vdash \lambda^q \langle x, k \rangle.\, \underline{t} @_{q_t} k\colon \overline{X \to Y}[qzx^+, y^+x^-]}}$$

By induction hypothesis, we know that the program $(y^- x^+ \gamma^+, \mathsf{Erase}(D(\underline{t})), y^+ x^- \gamma^-)$ is in the equivalence class of programs obtained by Int-interpretation of the given derivation of $\Gamma, x\colon X \vdash t\colon Y$.

We have to show that $(qzx^+ \gamma^+, \mathsf{Erase}(D(\lambda^q \langle x, k \rangle.\, \underline{t} @_{q_t} k)), y^+ x^- \gamma^-)$ is in the equivalence class of programs obtained by Int-interpretation of $\Gamma \vdash \lambda x{:}X.\, t\colon X \to Y$. But we have

$$\mathsf{Erase}(D(\lambda^q \langle x, k \rangle.\, \underline{t} @_{q_t} k)) = \{ apply_q() = apply_{q_t}() \} \cup \mathsf{Erase}(D(\underline{t}))$$

by definition. The definition of the Int-interpretation is such that the required assertion thus clearly holds.

- Case $\to$E.

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

A CPS-translation of this derivation has the form

$$\vdots$$
$$\frac{\overline{\Gamma}[\gamma^-, \gamma^+] \vdash \underline{s}\colon \overline{X \to Y}[y^- x^+, y^+ x^-] \qquad \dfrac{\overline{\Delta}[\delta^-, \delta^+] \vdash \underline{t}\colon \overline{X}[x^-, x^+] \qquad \overline{k\colon Y' \vdash k\colon Y'}}{\overline{\Delta}[\delta^-, \delta^+], k\colon Y' \vdash \langle \underline{t}, k \rangle\colon \overline{X}[x^-, x^+] \times Y'}}{\dfrac{\overline{\Gamma}[\gamma^-, \gamma^+], \overline{\Delta}[\delta^-, \delta^+], k\colon Y' \vdash \underline{s} @_{q_s} \langle \underline{t}, k \rangle\colon \bot}{\overline{\Gamma}[\gamma^-, \gamma^+], \overline{\Delta}[\delta^-, \delta^+] \vdash \lambda^q k.\, \underline{s} @_{q_s} \langle \underline{t}, k \rangle\colon \overline{Y}[qz, y^+]}}$$

where $y^- = q_s z$ and $\overline{Y}[qz, y^+] = Y'$ and $q_s \in \mathcal{L}$.

Applying the induction hypothesis to the left and right sub-derivations of the given derivation shows that $(y^- x^+ \gamma^+, \mathsf{Erase}(D(\underline{s})), y^+ x^- \gamma^-)$ implements the Int-interpretation of $\Gamma \vdash s \colon X \to Y$ and $(x^- \delta^+, \mathsf{Erase}(D(\underline{t})), x^+ \delta^-)$ implements the Int-interpretation of $\Delta \vdash t \colon X$.

The program obtained by CPS-translation and defunctionalization is

$$(qz\delta^+\gamma^+, \{apply_q() = apply_{q_s}()\} \cup D(\underline{s}) \cup D(\underline{t}), y^+\delta^-\gamma^-).$$

By definition, $\mathsf{Erase}(D(\lambda^q k.\, \underline{s} \,@_{q_s} \langle \underline{t}, k \rangle))$ has the form $\{apply_q() = apply_{q_s}()\} \cup \mathsf{Erase}(D(\underline{s})) \cup \mathsf{Erase}(D(\underline{t}))$. This corresponds to the Int-interpretation of the sequent $\Gamma, \Delta \vdash s\, t \colon Y$. $\quad\square$

While only for the very small source fragment CORE, we have now seen how one can associate interfaces with higher-order types and show that the Int-interpretation implements these interfaces in the same way as CPS-translation and defunctionalization. In Lemma 6.2 we have seen how η-expansion helps with compositional reasoning.

7. BASE TYPES

We now work towards extending the result to a more expressive source language, starting with a fragment that extends CORE with non-trivial base types. Define LIN to be the source fragment with the syntax shown below and the typing rules from Figures 2 and 3.

Types: $\quad X, Y \ ::= \ 1 \ \big| \ X \to Y \ \big| \ \mathbb{N}$

Terms: $\quad s, t \ ::= \ * \ \big| \ \lambda x{:}X.\, t \ \big| \ s\, t \ \big| \ n \ \big| \ s + t \ \big| \ \mathsf{if0}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$

That is, we add the type of natural numbers $\mathbb{N}$ with constant numbers, addition and case distinction, but still consider only a linear source language.

The example in the Introduction shows that for LIN it is not possible to remove all arguments from the *apply*-functions, as we have done for CORE. At least certain natural numbers must be passed as arguments.

7.1. **Interactive Interpretation.** Let us first consider the interpretation of LIN in $\mathrm{Int}(\mathbb{T})$. To this end we extend the definition of the interface (X^-, X^+) as follows:

$$1^- = \mathtt{unit} \qquad\qquad \mathbb{N}^- = \mathtt{unit} \qquad\qquad (X \to Y)^- = Y^- X^+$$

$$1^+ = \mathtt{unit} \qquad\qquad \mathbb{N}^+ = \mathtt{nat} \qquad\qquad (X \to Y)^+ = Y^+ X^-$$

The single value of type $\mathbb{N}^-$ encodes the request to compute a particular number. The values of type $\mathbb{N}^+$ are the possible answers.

It is not completely straightforward to extend the Int-interpretation described in the previous section. Consider for example the case of an addition $s + t$ of two closed terms $\vdash s \colon \mathbb{N}$ and $\vdash t \colon \mathbb{N}$. Suppose we already have programs (q_s, D_s, a_s) and (q_t, D_t, a_t) for s and t. It is not possible to construct a program for $s + t$ from these programs without modifying at least one of them. The problem is that after evaluating the first summand, we have no way of storing the result while we invoke the second program to compute the second summand. A natural way of constructing a program for $s + t$ would be to take the program (q, D, 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 D_s, and the equations from $\mathtt{nat} \cdot D_t$ (recall the notation $\mathtt{nat} \cdot -$ from Section 2). Here we use $\mathtt{nat} \cdot D_t$ instead of D_t in order to keep the value x of the first

summand available until the second summand is computed, so that we can compute the sum.

One solution to this issue was proposed by Dal Lago and the author in the form of IntML [8]. We consider here a simple special case of this system. The basic idea is to annotate the domain of each function type $X \to Y$ with a *subexponential* A, which is a target type, so that function types have the form $A \cdot X \to Y$.

We define $\textsc{lin}_{\textsc{exp}}$, a variant of $\textsc{lin}$ with subexponential annotations. It has the same terms as $\textsc{lin}$, but the grammar of types is modified as follows.

$$X, Y \ ::= \ 1 \ \bigm| \ A \cdot X \to Y \ \bigm| \ \mathbb{N}$$

In this grammar, A ranges over target types.

The subexponential annotations may be explained such that a term s of type $A \cdot X \to Y$ 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, the application $s\, t$ is such that 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 $\mathtt{unit} \cdot \mathbb{N} \to \mathtt{nat} \cdot \mathbb{N} \to \mathbb{N}$, as it needs to remember the already queried value of the first argument (having type $\mathtt{nat}$) when it queries the second argument.

It is interesting to note that Appel and Shao [39, §3.2] use a similar approach of preserving values by passing them as arguments for the optimisation of programs in CPS style.

Conceptually, subexponentials may be understood as a generalisation of the exponentials of Linear Logic. The special case $\omega \cdot X \to Y$, where the subexponential is the type $\omega = \mu\alpha.\,\mathtt{unit} + \alpha$ of unbounded natural numbers, may be understood as $!X \to Y$. This view corresponds to the construction of the exponential $!X$ in Game Semantics [2] or in Geometry of Interaction situations [1]. We make the generalisation to subexponentials because it allows us to make only the assumptions that are really needed, e.g. with respect to assuming recursive types in the target language. It also allows us to avoid unnecessary encoding operations. In the above outline of the translation of $s + t$, we could have used $\omega \cdot D_t$ instead of $\mathtt{nat} \cdot D_t$, but then in the definition of $apply_{a_s}$ we would need to encode x of type $\mathtt{nat}$ into a value of type ω and in the definition of $apply_{a_t}$ we would need to decode again.

The typing rules of $\textsc{lin}_{\textsc{exp}}$ are annotated version of the rules of $\textsc{lin}$, formulated to keep track of subexponential annotations. In $\textsc{lin}_{\textsc{exp}}$ contexts are finite lists of variable declarations of the form $x \colon A \cdot X$. The typing rules with subexponential annotations are are shown in Figure 8. In these rules, we write $A \cdot \Gamma$ for the context obtained by replacing each declaration $x \colon B \cdot X$ with $x \colon (A \times B) \cdot X$.

With subexponential annotations, it is straightforward to define the Int-interpretation. Extend the definition of $(-)^-$ and $(-)^+$ to $\textsc{lin}_{\textsc{exp}}$ by

$$(A \cdot X \to Y)^- = Y^-(A \times X^+) \qquad\qquad \Gamma^- = A_n \times X_n^- \ldots A_1 \times X_1^-$$
$$(A \cdot X \to Y)^+ = Y^+(A \times X^-) \qquad\qquad \Gamma^+ = A_n \times X_n^+ \ldots A_1 \times X_1^+,$$

where Γ is $x_1 \colon A_1 \cdot X_1, \ldots, x_n \colon A_n \cdot X_n$.

The interpretation of the rules is shown graphically in Figure 9. The interpretation of rule $\textsc{ax}$ remains essentially the same, but now uses the isomorphism $\mathtt{unit} \times A \simeq A$ to treat the subexponential. The cases for $\to\textsc{i}$ and $\to\textsc{e}$ must also be modified to take

$$\text{AX} \frac{}{x : \mathtt{unit} \cdot X \vdash x : X} \qquad \text{NUM} \frac{}{\vdash n : \mathbb{N}}$$

$$\text{WEAK} \frac{\Gamma \vdash t : Y}{\Gamma, x : A \cdot X \vdash t : Y} \qquad \text{EXCH} \frac{\Gamma, y : B \cdot Y, x : A \cdot X, \Delta \vdash t : Z}{\Gamma, x : A \cdot X, y : B \cdot Y, \Delta \vdash t : Z}$$

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

$$\text{ADD} \frac{\Gamma \vdash s : \mathbb{N} \qquad \Delta \vdash t : \mathbb{N}}{\Gamma, \mathtt{nat} \cdot \Delta \vdash s + t : \mathbb{N}} \qquad \text{IF} \frac{\Gamma \vdash s : \mathbb{N} \qquad \Delta_1 \vdash t_1 : \mathbb{N} \qquad \Delta_2 \vdash t_2 : \mathbb{N}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{if0} \ s \ \mathsf{then} \ t_1 \ \mathsf{else} \ t_2 : \mathbb{N}}$$

Figure 8: LIN_{EXP} – LIN with subexponential annotations
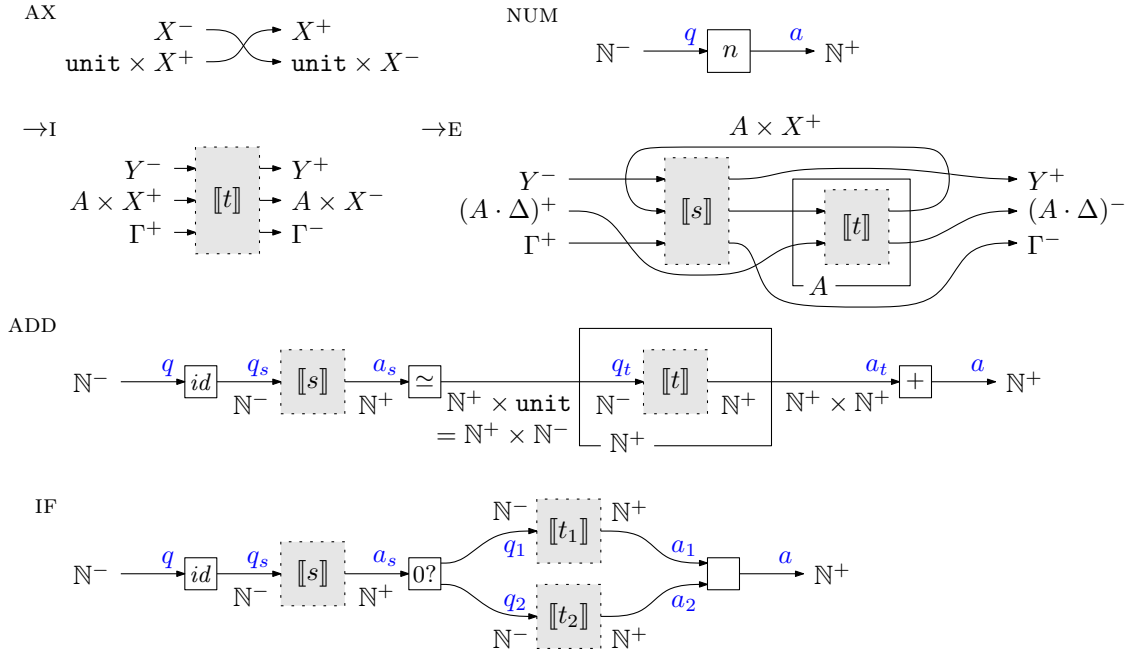
Figure 9: Int-interpretation of LIN_{EXP}

subexponentials into account. In the case for $\rightarrow\text{E}$ the box labelled with A represents the program obtained by applying the operation $A \cdot (-)$ to the content of the box. In this case we moreover make the isomorphisms $(A \cdot \Delta)^+ \simeq A \cdot \Delta^+$ and $(A \cdot \Delta)^- \simeq A \cdot \Delta^-$ implicit. In the cases for ADD and IF we omit the contexts Γ, Δ, Δ_1 and Δ_2 for better readability. They are handled as in the case for $\rightarrow\text{E}$. We omit the rules for pairs, which are also modified like the ones for functions [8]. In the case for IF, we write "0?" for the program given by $apply_{a_s}(x) = \mathsf{case} \ \mathsf{iszero}(x) \ \mathsf{of} \ \mathsf{inl}(y) \Rightarrow apply_{q_1}(y); \mathsf{inr}(z) \Rightarrow apply_{q_2}(z)$. A concrete definition of the Int-interpretation in terms of target equations can also be found in the proof of Proposition 7.2 below.

7.2. **CPS-translation and Defunctionalization.** Let us now outline how this interpretation using the Int-construction relates to the translation given by CPS-translation and defunctionalization, wherein the subexponential annotations are ignored.

A constant number n has the CPS-translation $\lambda^q k.\, k \,@_a\, n \colon \overline{\mathbb{N}}[q,a]$, where $\overline{\mathbb{N}}[q,a] = \neg_q \neg_a \mathbb{N}$. This defunctionalizes to $apply_q(\langle\rangle, k) = apply_a(k, n)$. The Int-interpretation yields the definition $apply_q() = apply_a(n)$, which differs only in that arguments have been removed.

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. For the sake of illustration we assume that s and t are closed.

$$D(\underline{s}) \cup D(\underline{t}) \cup \{ \, apply_q(\langle\rangle, k) = apply_{q_s}(\underline{s}^*, \langle k \rangle),$$
$$apply_{a_s}(\langle k \rangle, x) = apply_{q_t}(\underline{t}^*, \langle k, x \rangle),$$
$$apply_{a_t}(\langle k, x \rangle, y) = apply_a(k, x + y)\}.$$

The program obtained in this way has the same shape as the program obtained by Int-interpretation. The program in Figure 9 is annotated with labels to show the correspondence to the equations. The programs are not exactly equal. For example, $apply_q$ takes a pair as an argument, while the program obtained by Int-interpretation expects a single value of type $\mathbb{N}^-$. We study the relation of the two programs in the rest of this section.

In a similar manner, the term if0 s then t_1 else t_2 is CPS-translated to the labelled term $\lambda^q k.\, \underline{s} \,@_{q_s}\, (\lambda^{a_s} x.\, \text{if0 } x \text{ then } \underline{t_1} \,@_{q_1}\, (\lambda^{a_1} y.\, k \,@_a\, y) \text{ else } \underline{t_2} \,@_{q_2}\, (\lambda^{a_2} y.\, k \,@_a\, y))$. Defunctionalization gives us the equations

$$D(\underline{s}) \cup D(\underline{t_1}) \cup D(\underline{t_2}) \cup \{ \, apply_q(\langle\rangle, k) = apply_{q_s}(\underline{s}^*, \langle k \rangle),$$
$$apply_{a_s}(\langle k \rangle, x) = \text{case ifzero}(x) \text{ of } \text{inl}(\_) \Rightarrow apply_{q_1}(\underline{t_1}^*, \langle k \rangle)$$
$$; \text{inr}(\_) \Rightarrow apply_{q_2}(\underline{t_2}^*, \langle k \rangle)$$
$$apply_{a_1}(\langle k \rangle, y) = apply_a(k, y),$$
$$apply_{a_2}(\langle k \rangle, y) = apply_a(k, y)\},$$

and it can be observed that they correspond to the Int-interpretation given in Figure 9.

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.

Definition 7.1. We say that two target 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 $f(x) = \text{case } e \text{ of } \text{inl}(x) \Rightarrow g(e_1); \text{inr}(y) \Rightarrow h(e_2)$, then the other contains $f(x) = \text{case } e' \text{ of } \text{inl}(x) \Rightarrow g(e_1'); \text{inr}(y) \Rightarrow h(e_2')$ for some e', e_1' and e_2'.

We note that for LIN_{EXP} Lemma 6.2 continues to hold and that CpsDefun can be defined exactly as for CORE above.

Proposition 7.2. *For any derivation of $\Gamma \vdash t\colon X$ in LIN_{EXP} there exists a program $\mathsf{Int}(\Gamma \vdash t\colon X)$ that is a representative of the Int-interpretation $[\![\Gamma \vdash t\colon X]\!]$ (which is a morphism in $\mathbb{T}$ and as such an equivalence class of programs up to program equality) and that has the same skeleton as $\mathsf{CpsDefun}(\Gamma \vdash t\colon X)$.*

Proof. Recall that CpsDefun first translates the derivation of $\Gamma \vdash t\colon X$ to a labelled derivation of the CPS-translated term $\Gamma[\gamma^-, \gamma^+] \vdash \underline{t}\colon X[x^-, x^+]$, which is then mapped to the program $(x^- \gamma^+, D(\underline{t}), x^+ \gamma^-)$.

Here we show how to translate the derivation of $\Gamma[\gamma^-, \gamma^+] \vdash \underline{t} \colon X[x^-, x^+]$ to a set of equations $I(t)$, such that the assertion of the proposition is satisfied when we choose $\mathsf{Int}(\Gamma \vdash t \colon X) := (x^-\gamma^+, I(t), x^+\gamma^-)$. The definition of $I(t)$ is given by induction on the original derivation by the following clauses:

- Rule AX.

$$\overline{}$$
$$x \colon \overline{\mathtt{unit} \cdot X}[q_1 \ldots q_n, a_1 \ldots a_n] \vdash \eta(\underline{x}, \overline{X}) \colon \overline{X}[q'_1 \ldots q'_n, a'_1 \ldots a'_n]$$

We define $I(x) := \{ apply_{q'_i}(x) = apply_{q_i}(\langle\rangle, x) \mid i = 1, \ldots, n \} \cup \{ apply_{a_i}(\langle\rangle, x) = apply_{a'_i}(x) \mid i = 1, \ldots, n \}$

- Rule NUM.

$$\overline{}$$
$$\vdash \underline{n} \colon \overline{\mathbb{N}}[q, a]$$

Define $I(n) := \{ apply_q() = apply_a(n) \}$.

- Rule →I.

$$\frac{\overline{\Gamma}, \, x \colon \overline{A \cdot X}[x^-, x^+] \vdash \underline{t} \colon \overline{Y}[q_t z, y^+]}{\overline{\Gamma} \vdash \underline{\lambda x. t} \colon \overline{(A \cdot X \to Y)}[qzx^-, y^+x^+]}$$

Define $I(\lambda x. t) := I(t) \cup \{ apply_q() = apply_{q_t}() \}$.

- Rule →E.

$$\frac{\overline{\Gamma} \vdash \underline{s} \colon \overline{(A \cdot X \to Y)}[q_s zx^+, y^+x^-] \qquad \overline{\Delta}[\delta^-, \delta^+] \vdash \underline{t} \colon \overline{X}[x^-, x^+]}{\overline{\Gamma}, \, A \cdot \Delta \vdash \underline{s\ t} \colon \overline{Y}[qz, y^+]}$$

In the Int-interpretation we must account for the isomorphisms $(A \cdot \Delta)^- \simeq A \cdot \Delta^-$ and $(A \cdot \Delta)^+ \simeq A \cdot \Delta^+$. The set of equations $A \cdot I(t)$ gives rise to a program of type $(A \cdot X^-)(A \cdot \Delta^+) \to (A \cdot X^+)(A \cdot \Delta^-)$. It is easy to define from it a program of type $(A \cdot X^-)(A \cdot \Delta)^+ \to (A \cdot X^+)(A \cdot \Delta)^-$: Each definition $apply_q(u, x) = e$ for $q \in \delta^-$ is replaced by $apply_q(\langle u, v \rangle, y) = e[\langle v, y \rangle / x]$; and each call $apply_a(u, x)$ for $a \in \delta^+$ is replaced by $apply_a(\langle u, \mathsf{fst}(x) \rangle, \mathsf{snd}(x))$, where fst and snd are the evident projections. Write $I(A \cdot t)$ for the program obtained in this way.

With this notation, we can conclude this case by defining

$$I(s\ t) := I(s) \cup I(A \cdot t) \cup \{ apply_q() = apply_{q_s}() \}.$$

- Rule ADD.

$$\frac{\overline{\Gamma} \vdash \underline{s} \colon \overline{\mathbb{N}}[q_s, a_s] \qquad \overline{\Delta} \vdash \underline{t} \colon \overline{\mathbb{N}}[q_t, a_t]}{\overline{\Gamma}, \, \mathtt{nat} \cdot \Delta \vdash \underline{s + t} \colon \overline{\mathbb{N}}[q, a]}$$

We use the notation $I(\mathtt{nat} \cdot t)$, which is as in the case for →E above, and define:

$$\begin{aligned} I(s + t) := I(s) \cup I(\mathtt{nat} \cdot t) \cup \{ \, & apply_q() = apply_{q_s}(), \\ & apply_{a_s}(x) = apply_{q_t}(x, \langle\rangle), \\ & apply_{a_t}(x, y) = apply_a(x + y) \} \end{aligned}$$

- Rule IF.

$$\frac{\overline{\Gamma} \vdash \underline{s} \colon \overline{\mathbb{N}}[q_s, a_s] \qquad \overline{\Delta_1} \vdash \underline{t_1} \colon \overline{\mathbb{N}}[q_1, a_1] \qquad \overline{\Delta_2} \vdash \underline{t_2} \colon \overline{\mathbb{N}}[q_2, a_2]}{\overline{\Gamma}, \, \Delta_1, \, \Delta_2 \vdash \underline{\mathsf{if0}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2} \colon \overline{\mathbb{N}}[q, a]}$$

Let $I(\mathsf{if0}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)$ be

$$I_{\underline{s}} \cup I_{\underline{t_1}} \cup I_{\underline{t_2}} \cup \{\ apply_q() = apply_{q_s}(),$$
$$apply_{a_s}(x) = \mathsf{case\ iszero}(x)\ \mathsf{of}\ \mathsf{inl}(y) \Rightarrow apply_{q_1}(y)$$
$$;\ \mathsf{inr}(z) \Rightarrow apply_{q_2}(z),$$
$$apply_{a_1}(x) = apply_a(x),$$
$$apply_{a_2}(x) = apply_a(x)\}. \qquad \qquad \square$$

The proposition establishes a simple connection between the general shape of the programs.

Let us now compare the values that are being passed around during program execution. Consider closed terms of type $\mathbb{N}$. It follows by soundness of each of the two translations that the program obtained by defunctionalization and that for the Int-interpretation will return the same number as their end result. If we consider the programs with the same skeleton constructed above, then we can say more, however. We can show that during the computation the two programs jump to the same labels in the same order. The argument values of these jumps are not exactly the same, however. One may consider the values appearing in the program obtained by Int-interpretation as simplifications of the values appearing at the same time in the traces of the program obtained by defunctionalization. The following example illustrates the correspondence informally.

Example 7.3. Consider the source term $((\lambda x.\,1 + x)\ 42)$ of type $\mathbb{N}$. The result of CPS-translation and labelling is the term

$$\lambda^{l_0} k.\, \underline{t} \,@_{l_1} \langle \lambda^{l_5} k''.\, k'' \,@_{l_4} 42, k \rangle : \neg_{l_0} \neg_{l_6} \mathbb{N}\ ,$$

where $\underline{t}$ is spelled out in Example 5.1. Defunctionalization gives us the following definitions.

$$apply_{l_0}(\langle\rangle, k) = apply_{l_1}(\langle\rangle, \langle\langle\rangle, k\rangle), \qquad apply_{l_1}(\langle\rangle, \langle x, k\rangle) = apply_{l_2}(\langle\rangle, \langle x, k\rangle),$$
$$apply_{l_2}(\langle\rangle, k') = apply_{l_3}(k', 1), \qquad apply_{l_3}(\langle x, k\rangle, u) = apply_{l_5}(x, \langle k, u\rangle),$$
$$apply_{l_4}(\langle k, u\rangle, n) = apply_{l_6}(k, u + n), \qquad apply_{l_5}(\langle\rangle, k'') = apply_{l_4}(k'', 42).$$

The program of the same skeleton obtained by Int-interpretation is:

$$apply_{l_0}() = apply_{l_1}(), \qquad\qquad apply_{l_1}() = apply_{l_2}(),$$
$$apply_{l_2}() = apply_{l_3}(1), \qquad\qquad apply_{l_3}(m) = apply_{l_5}(m),$$
$$apply_{l_4}(m, n) = apply_{l_6}(m + n), \qquad\qquad apply_{l_5}(m) = apply_{l_4}(m, 42).$$

Both programs have entry label l_0 and exit label l_6.

Let us now compare how these programs compute their result. A call trace of the first program, in which a closed continuation represented by $\langle\rangle$ is given as argument, is:

$$apply_{l_0}(\langle\rangle, \langle\rangle)\ apply_{l_1}(\langle\rangle, \langle\langle\rangle, \langle\rangle\rangle)\ apply_{l_2}(\langle\rangle, \langle\langle\rangle, \langle\rangle\rangle)\ apply_{l_3}(\langle\langle\rangle, \langle\rangle\rangle, 1)\ apply_{l_5}(\langle\rangle, \langle\langle\rangle, 1\rangle)$$
$$apply_{l_4}(\langle\langle\rangle, 1\rangle, 42)\ apply_{l_6}(\langle\rangle, 43)$$

The call trace of the second program is:

$$apply_{l_0}()\ apply_{l_1}()\ apply_{l_2}()\ apply_{l_3}(1)\ apply_{l_5}(1)\ apply_{l_4}(1, 42)\ apply_{l_6}(43)$$

The point is that the traces are the same, up to simplification of values by removing unneeded $\langle\rangle$-values.

In the rest of this section we study the relation of the traces of the programs obtained by the two translations. The example illustrates that the traces of both programs jump to the same labels in the same order. The main issue is to compare the argument values of each such jump. We compare not the argument values themselves (keeping track of the technical details appears to be non-trivial), but only what needs to be stored in order to encode these values, i.e. what a compiler needs to store in machine code.

For any target value v, we define a multiset $\mathcal{V}(v)$ of the numbers it contains as follows: if $v = n$ then $\mathcal{V}(v) = \{n\}$, if $v = \langle v_1, v_2 \rangle$ then $\mathcal{V}(v) = \mathcal{V}(v_1) \cup \mathcal{V}(v_2)$, and $\mathcal{V}(v) = \emptyset$ otherwise (values of recursive types or sum types cannot appear). The definition of $\mathcal{V}(v)$ is motivated by considering how the value v would eventually be encoded on a machine. A good compiler back-end would need to store in memory only the values in $\mathcal{V}(v)$, as the rest of the information in v is given statically by the type. We say that a value v *simplifies* a value w if $\mathcal{V}(v) \subseteq \mathcal{V}(w)$. 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.

We shall analyse the behaviour of the program $\mathsf{Int}(\Gamma \vdash t\colon X)$. We use the notation $\mathsf{Int}(A \cdot (\Gamma \vdash t\colon X))$ for the program obtained from $A \cdot \mathsf{Int}(\Gamma \vdash t\colon X)$ by inserting the isomorphisms $(A \cdot \Gamma)^+ \to A \cdot \Gamma^+$ and $A \cdot \Gamma^- \to (A \cdot \Gamma)^-$, as described in the proof of Proposition 7.2 above (case $\to$E).

Theorem 7.4. *Let $\vdash t\colon \mathbb{N}$, let $(q, D_t, a) := \mathsf{CpsDefun}(\vdash t\colon \mathbb{N})$ and let $\mathsf{Int}(\vdash t\colon \mathbb{N})$ be the program from Proposition 7.2. Then, any call-trace of $\mathsf{Int}(\vdash t\colon \mathbb{N})$ beginning with $apply_q()$ simplifies the call-trace of $\mathsf{CpsDefun}(\vdash t\colon X)$ of the same length that begins with $apply_q(\langle\rangle, \langle\rangle)$.*

This theorem 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 [4]. 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.

To prove the theorem we use a few lemmas. The first two are substitution lemmas.

Lemma 7.5. *If $\Gamma, x\colon X \vdash s\colon Y$ and $\vdash t\colon X$ are derivable in* LIN, *then so is $\Gamma \vdash s[t/x]\colon Y$. Moreover, there exist a set of labels $E \subseteq \mathcal{L}$ and a bijective renaming $\rho\colon \mathcal{L} \to \mathcal{L}$, such that: If $apply_{l_1}(v_1) \dots apply_{l_n}(v_n)$ is a trace of $\mathsf{CpsDefun}(\Gamma, x\colon X \vdash s\colon Y) \cup \mathsf{CpsDefun}(\vdash t\colon X)$, then $c_1 \dots c_n$ is a trace of $\mathsf{CpsDefun}(\Gamma \vdash s[t/x]\colon Y)$, where*

$$c_i = \begin{cases} apply_{\rho(l_i)}(v_i) & \text{if } l_i \notin E, \\ \varepsilon & \text{otherwise.} \end{cases}$$

Furthermore, all traces of $\mathsf{CpsDefun}(\Gamma \vdash s[t/x]\colon Y)$ arise in this way.

Proof outline. This lemma is proved by induction on the derivation of $\Gamma, x\colon X \vdash s\colon Y$. The only interesting case is that where the last rule is AX and s is x. In this case the definitions in $\mathsf{CpsDefun}(\Gamma \vdash s[t/x]\colon Y)$ and $\mathsf{CpsDefun}(\Gamma, x\colon X \vdash s\colon Y) \cup \mathsf{CpsDefun}(\vdash t\colon X)$ differ only in that the latter contains equations of the form $apply_l(x) = apply_{l'}(x)$ that come from the

η-expansion of x. The traces of the two programs thus differ only up to removal of these indirections. For the set E we choose the labels of the calls that must be removed. A renaming ρ may be necessary to deal with different choices of names in CpsDefun. $\square$

Lemma 7.6. *If $\Gamma, x\colon A \cdot X \vdash s\colon Y$ and $\vdash t\colon X$ are derivable in $\mathrm{LIN_{EXP}}$, then so is $\Gamma \vdash s[t/x]\colon Y$. Moreover, the set of labels $E \subseteq \mathcal{L}$ and the bijective renaming $\rho\colon \mathcal{L} \to \mathcal{L}$ from Lemma 7.5 have the following property: If $apply_{l_1}(v_1)\dots apply_{l_n}(v_n)$ is a trace of $\mathsf{Int}(\Gamma, x\colon X \vdash s\colon Y) \cup \mathsf{Int}(A \cdot (\vdash t\colon X))$, then $\mathsf{Int}(\Gamma \vdash s[t/x]\colon Y)$ has a trace of the form $c_1 \dots c_n$, where*

$$c_i = \begin{cases} apply_{\rho(l_i)}(w_i) & \text{if } l_i \notin E, \\ \varepsilon & \text{otherwise.} \end{cases}$$

and $\mathcal{V}(v_i) = \mathcal{V}(w_i)$. Furthermore, all traces of $\mathsf{Int}(\Gamma \vdash s[t/x]\colon Y)$ arise in this way.

This lemma is again proved by induction on the derivation of the first sequent. The statement is slightly weaker, as the traces of the two sets of equations may differ also up to applications of the isomorphism $(\mathtt{unit} \times A) \simeq A$, as can be seen by considering the case where the last rule deriving $\Gamma, x\colon X \vdash s\colon Y$ is AX and s is x. Thus, we only get $\mathcal{V}(v_i) = \mathcal{V}(w_i)$.

The next lemma says that any closed program of type $\mathbb{N}$ will indeed eventually give an answer, as would already follow from soundness, and moreover, the continuation that accepts the final answer is just passed along in the course of the computation; the computation itself does not depend on the continuation.

Lemma 7.7. *Let $\vdash t\colon \mathbb{N}$ and let $\underline{t}$ be labelled such that $\vdash \underline{t}\colon \overline{\mathbb{N}}[q, a]$ is derivable. Then the following are true.*

(1) *Any call trace of $D(\underline{t})$ beginning with a call of the form $apply_q(\langle\rangle, k)$, for some k, can be extended to end with a call $apply_a(k, v)$ for some value v.*

(2) *If $apply_q(\langle\rangle, k_1)\dots apply_l(v_1, v_2)$ and $apply_q(\langle\rangle, k_2)\dots apply_{l'}(v_1', v_2')$ are two call traces of $D(\underline{t})$ of the same length, then $l = l'$ and there exist expressions e_1 and e_2 and variables x_1 and x_2, such that $v_1 = e_1[k_1/x_1]$ and $v_2 = e_2[k_2/x_2]$ holds.*

Note that the second point implies that if $f(w)$ is a call in a call trace beginning with $apply_q(\langle\rangle, k)$, then k must simplify w.

Proof. In the proof we do not need subexponential annotations, so we formulate it for LIN.

For each type X we define a set $R(X)$ of closed terms as follows: $R(\mathbb{N})$ consists of all closed terms t that satisfy the assertion of the lemma; $R(X \to Y)$ consists of all closed terms s of type $X \to Y$ such that $t \in R(X)$ implies $s\,t \in R(Y)$. For any LIN-context Γ, we define $R(\Gamma)$ to be the set of all substitutions σ that map each variable declared in Γ to a closed term, such that $x\colon X \in \Gamma$ implies $\sigma(x) \in R(X)$.

The proof of the lemma then goes by showing by induction on the derivation that each derivable $\Gamma \vdash t\colon X$ has the property $\forall \sigma \in R(\Gamma).\, t\sigma \in R(X)$. The case for λ-abstraction follows using Lemma 7.5. $\square$

Proof of Theorem 7.4. The proof goes by induction on the size of the term t. We continue by case distinction and consider representative cases. To simplify the notation, we just write $\mathsf{Int}(t)$ instead of $\mathsf{Int}(\Gamma \vdash t\colon X)$.

- t is $s_1 + s_2$, i.e. the derivation of $\vdash t\colon \mathbb{N}$ ends with rule ADD.

  We observe that a labelling of the term $\underline{(s_1 + s_2)}$ must have the following form

$$\vdash \lambda^q k.\, \underline{s_1} @_{q_1} (\lambda^{a_1} x.\, \underline{s_2} @_{q_2} (\lambda^{a_2} y.\, k @_a (x + y)))\colon \overline{\mathbb{N}}[q, a],$$

where q and a are fresh and where $\underline{s_1}$ and $\underline{s_2}$ are labelled such that $\vdash \underline{s_1}\colon \overline{\mathbb{N}}[q_1, a_1]$ and $\vdash \underline{s_2}\colon \overline{\mathbb{N}}[q_2, a_2]$ are derivable.

The program $D(\underline{s_1 + s_2})$ consists of the set of equations

$$D(\underline{s_1}) \cup D(\underline{s_2}) \cup \{ \; apply_q(\langle\rangle, k) = apply_{q_1}(\langle\rangle, \langle k\rangle),$$
$$apply_{a_1}(\langle k\rangle, x) = apply_{q_2}(\langle\rangle, \langle k, x\rangle),$$
$$apply_{a_2}(\langle k, x\rangle, y) = apply_a(k, x + y)\}.$$

On the other hand, the program $\mathsf{Int}(s_1 + s_2)$ consists of the equations

$$\mathsf{Int}(s_1) \cup \mathsf{Int}(\mathtt{nat} \cdot s_2) \cup \{ \; apply_q() = apply_{q_1}(),$$
$$apply_{a_1}(x, \langle\rangle) = apply_{q_2}(x),$$
$$apply_{a_2}(x, y) = apply_a(x + y)\}.$$

By the above Lemma 7.7, we know that the call-trace of $D(\underline{t})$ beginning with $apply_q(\langle\rangle, \langle\rangle)$ must have the form

$$apply_q(\langle\rangle, \langle\rangle) \; \tau_1 \; \tau_2 \; apply_a(\langle\rangle, x + y),$$

where τ_1 and τ_2 must have the following forms:

$$\tau_1 = apply_{q_1}(\langle\rangle, \langle\langle\rangle\rangle) \ldots apply_{a_1}(\langle\rangle, x)$$
$$\tau_2 = apply_{q_2}(\langle\rangle, \langle\langle\rangle, x\rangle) \ldots apply_{a_2}(\langle\langle\rangle, x\rangle, y)$$

Applying the induction hypothesis for s_1 shows that the trace of $\mathsf{Int}(s_1)$ starting with $apply_{q_1}()$ simplifies the trace of $D(\underline{t})$ starting with $apply_{q_1}(\langle\rangle, \langle\rangle)$. Using Lemma 7.7, we get the desired property for τ_1. Similarly, the induction hypothesis for s_2 shows that the trace of $\mathsf{Int}(s_2)$ starting with $apply_{q_2}()$ simplifies the trace of $D(\underline{t})$ starting with $apply_{q_2}(\langle\rangle, \langle\rangle)$. By Lemma 7.7, the trace from $apply_{q_2}(\langle\rangle, \langle\langle\rangle, x\rangle)$ differs only in that it replaces $\langle\rangle$ with $\langle\langle\rangle, x\rangle$ in each call and at least one position. But this shows that the trace of $\mathsf{Int}(\mathtt{nat} \cdot s_2)$ starting with $apply_{q_2}(x)$ simplifies τ_2. Together this shows the desired property of the whole trace.

- t cannot be a λ-abstraction, as its type is $\mathbb{N}$.
- t is an application. In this case, t must have the form $(\lambda x.s) \; t_1 \; \ldots \; t_n$, as it is a closed term. Notice that the term $s[t_1/x] \; t_2 \; \ldots \; t_n$ is shorter and $\vdash s[t_1/x] \; t_2 \; \ldots \; t_n\colon \mathbb{N}$ is still derivable. Hence, we can apply the induction hypothesis to it.

  It follows from Lemmas 7.5 and 7.6 and the definition of the translations of λ-abstraction and application that the desired result for $(\lambda x.s) \; t_1 \; \ldots \; t_n$ follows from the result for $s[t_1/x] \; t_2 \; \ldots \; t_n$ obtained by induction hypothesis. $\qquad\square$

8. Simple Types

In this section, we strengthen the source language by adding contraction, explain how the Int-interpretation can be extended and how it relates CPS-translation and defunctionalization. With increasing expressiveness of the source language, the syntactic details of defunctionalization become harder to manage. For defunctionalization we now need a more expressive control flow analysis, and the translation uses the recursive types in the target language. We shall argue that a type system with subexponential annotations, adapted from IntML, offers a simple and conceptually clear way of managing such details. We concentrate in this section only on the relationship between program interfaces and skeletons.

We consider the source fragment STL of the simply-typed λ-calculus with the following syntax and the typing rules from Figures 2–4.

Types: $\qquad X, Y \;\; ::= \;\; 1 \;\mid\; X \to Y \;\mid\; \mathbb{N}$

Terms: $\qquad s, t \;\; ::= \;\; * \;\mid\; \lambda x{:}X.\, t \;\mid\; s\, t \;\mid\; n \;\mid\; s + t \;\mid\; \mathsf{if0}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$

8.1. CPS-translation and Defunctionalization.

The CPS-translation defined in Section 4 restricts to STL. The defunctionalization procedure described in Section 5, however, is too simple to handle contraction. The control-flow annotations therein are not sufficient; they need to be extended so that applications can be annotated with more than one label.

Banerjee et al. [4] use a calculus with control flow annotations, in which applications are annotated with sets of labels instead of just a single label. Thus, $s\,@_{\{l_1,\dots,l_n\}}\, t$ means that s is a term whose evaluation may have any of the functions with label $l_1, \dots, l_n$ as a result. Such an application is defunctionalized into a case distinction on the function that actually appears for s during evaluation:

$$(s\,@_{\{l_1,\dots,l_n\}}\, t)^* = \mathsf{case}\ s^*\ \mathsf{of}\ l_1(\vec{x}) \Rightarrow apply_{l_1}(l_1(\vec{x}), t^*); \dots; l_n(\vec{y}) \Rightarrow apply_{l_n}(l_n(\vec{y}), t^*).$$

Note that such a case distinction is possible only if labels are actually passed as values. To encode labels, one typically uses algebraic data types whose constructors correspond to the function labels. To handle the full simply-types λ-calculus, one must allow for recursive algebraic data types. An example is given in Example 8.6 on page 35.

We define a variant of the labelled λ-calculus of Banerjee et al. [4], which is suitable for the target language considered here (the target language in [4] has union types, while we use disjoint sums here).

Instead of sets of labels, we annotate applications with *label terms* formed by the following grammar, in which l ranges over all the labels from $\mathcal{L}$.

$$L_1, L_2 \;\; ::= \;\; l \;\mid\; L_1 + L_2$$

Write $\mathcal{L}_T$ for the set of all label terms.

In the labelled version of STL with product types, each abstraction $\lambda^l x{:}X.\, t$ is still annotated with a unique label $l \in \mathcal{L}$. Applications $s\,@_L\, t$, however, are now annotated with a label term. Function types are also annotated with a label term instead of just a single label. Moreover, we extend the type system with explicit coercion terms $\mathsf{coercl}_L(t)$ and $\mathsf{coercr}_L(t)$. The syntax of the labelled STL with products is therefore given as follows.

Types: $\qquad X, Y \;\; ::= \;\; 1 \;\mid\; X \xrightarrow{L} Y \;\mid\; X \times Y \;\mid\; \mathbb{N} \;\mid\; \bot$

Terms: $\qquad s, t \;\; ::= \;\; * \;\mid\; \lambda^l x{:}X.\, t \;\mid\; s\,@_L\, t \;\mid\; \langle s, t \rangle \;\mid\; \mathsf{let}\ \langle x, y \rangle = s\ \mathsf{in}\ t$

$\qquad\qquad\qquad\quad\; \mid\; n \;\mid\; s + t \;\mid\; \mathsf{if0}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$

$\qquad\qquad\qquad\quad\; \mid\; \mathsf{coercl}_L(t) \;\mid\; \mathsf{coercr}_L(t)$

The typing rules for the new and modified terms are:

$$\frac{\Gamma,\, x\colon X \vdash t\colon Y}{\Gamma \vdash \lambda^l x{:}X.\, t\colon X \xrightarrow{l} Y} \qquad\qquad \frac{\Gamma \vdash s\colon X \xrightarrow{L} Y \qquad \Delta \vdash t\colon X}{\Gamma,\, \Delta \vdash s\,@_L\, t\colon Y}$$

$$\frac{\Gamma \vdash t\colon X \xrightarrow{L_1} Y}{\Gamma \vdash \mathsf{coercl}_{L_1+L_2}(t)\colon X \xrightarrow{L_1+L_2} Y} \qquad \frac{\Gamma \vdash t\colon X \xrightarrow{L_2} Y}{\Gamma \vdash \mathsf{coercr}_{L_1+L_2}(t)\colon X \xrightarrow{L_1+L_2} Y}$$

The type $X \xrightarrow{(l_1+l_2)+l_3} Y$ thus is the type of functions with label l_1, l_2 or l_3. Annotating this type with the term $(l_1 + l_2) + l_3$, as opposed to the set $\{l_1, l_2, l_3\}$, is convenient for technical reasons, as our target language has disjoint sum types and not union types, which means that we cannot assume associativity.

We shall often omit the subscript L in the terms $\mathsf{coercl}_L(t)$ and $\mathsf{coercr}_L(t)$, when it can be reconstructed from type information.

With these changes to the label annotations, we can extend the defunctionalization procedure to cover the whole source language. The new terms are defunctionalized as follows (the notation $l(x_1, \ldots, x_n)$ is explained below).

$$(s \,@_L t)^* = apply_L(s^*, t^*)$$
$$(\lambda^l x{:}X.\, t)^* = l(x_1, \ldots, x_n) \text{ where } \mathrm{FV}(\lambda^l x{:}X.\, t) = \{x_1, \ldots, x_n\}$$
$$(\mathsf{coercl}_L(t))^* = \mathsf{inl}(t^*)$$
$$(\mathsf{coercr}_L(t))^* = \mathsf{inr}(t^*)$$

Definitions:

$$D(s \,@_L t) = D(s) \cup D(t) \cup D(L)$$
$$D(\lambda^l x{:}X.\, t) = D(t) \cup \{apply_l(l(x_1, \ldots, x_n), x) = t^*\}$$
$$D(\mathsf{coercl}_L(t)) = D(t) \cup D(L)$$
$$D(\mathsf{coercr}_L(t)) = D(t) \cup D(L)$$

where

$$D(l) = \emptyset$$
$$D(L_1 + L_2) = D(L_1) \cup D(L_2) \cup \{apply_{L_1+L_2}(f, x) = \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(f_1) \Rightarrow apply_{L_1}(f_1, x)$$
$$; \mathsf{inr}(f_2) \Rightarrow apply_{L_2}(f_2, x)\}$$

The term $l(x_1, \ldots, x_n)$ plays the role of a constructor in a functional language. For each label $l \in \mathcal{L}$ we assume a data type τ_l with a single constructor called l with arguments of appropriate type to make the above definition type correct. In ML-notation one would write

$$\tau_l = \mathsf{datatype}\ l\ \mathsf{of}\ A$$

for a suitable type A. We extend the definition of τ_l to label terms by letting $\tau_{L_1+L_2} = \tau_{L_1} + \tau_{L_2}$. The definition of these types is such that in a definition of $apply_L(f, x)$, the variable f will have type τ_L.

If one writes out all the data type definitions for a given term, then one may obtain a set of mutually recursive data type definitions, as the type τ_l may appear in the argument type A of its constructor (see Example 8.6). In cases where the definitions are not actually recursive, it would be possible to remove the constructors and work just with tuples instead.

Example 8.1. Let us illustrate the modified defunctionalization by considering the CPS-translation of $\lambda x{:}\mathbb{N}.\, x + x$. For this example, η-expansion is not important, so we omit it for

simplicity. The CPS-translated and simplified term may be annotated with label terms as follows:

$$\lambda^{l_1}\langle x, k\rangle.\, x \,@_{l_4}\, \mathsf{coercl}(\lambda^{l_2}m.\, x \,@_{l_4}\, \mathsf{coercr}(\lambda^{l_3}n.\, k \,@_{l_5}\, (m + n)))$$

Its type is $(((\mathbb{N} \xrightarrow{l_2+l_3} \bot) \xrightarrow{l_4} \bot) \times (\mathbb{N} \xrightarrow{l_5} \bot)) \xrightarrow{l_1} \bot$. As a concrete argument one may think of $\langle \lambda^{l_4}k.\, k \,@_{l_2+l_3}\, 42,\, \lambda^{l_5}n.\, \mathtt{print\_int}(n)\rangle$. Note the use of the label term $l_2 + l_3$. The two applications of x could not be typed using the simple labelled λ-calculus from Section 5.

Defunctionalization turns the term into the following definitions.

$$\begin{aligned}
apply_{l_1}(l_1(), \langle x, k\rangle) &= apply_{l_4}(x, \mathsf{inl}(l_2(x, k))) \\
apply_{l_2}(l_2(x, k), m) &= apply_{l_4}(x, \mathsf{inr}(l_3(m, k))) \\
apply_{l_3}(l_3(m, k), n) &= apply_{l_5}(k, m + n) \\
apply_{l_4}(l_4(), k) &= apply_{l_2+l_3}(k, 42) \\
apply_{l_2+l_3}(k, n) &= \mathsf{case}\ k\ \mathsf{of}\ \mathsf{inl}(f_1) \Rightarrow apply_{l_2}(f_1, n) \\
&\qquad\qquad\qquad\ ; \mathsf{inr}(f_2) \Rightarrow apply_{l_3}(f_2, n) \\
apply_{l_5}(l_5(), n) &= \mathtt{print\_int}(n)
\end{aligned}$$

The types of the constructors are:

$$\begin{aligned}
\tau_{l_1} &= \mathsf{datatype}\ l_1\ \mathsf{of}\ \mathtt{unit} & \tau_{l_2} &= \mathsf{datatype}\ l_2\ \mathsf{of}\ \tau_{l_4} \times \tau_{l_5} \\
\tau_{l_3} &= \mathsf{datatype}\ l_2\ \mathsf{of}\ \mathtt{nat} \times \tau_{l_5} & \tau_{l_4} &= \mathsf{datatype}\ l_4\ \mathsf{of}\ \mathtt{unit} \\
\tau_{l_5} &= \mathsf{datatype}\ l_5\ \mathsf{of}\ \mathtt{unit}
\end{aligned}$$

In this example, these types are not actually recursive, so we could remove the constructors, replacing $l_2(x, k)$ just by the tuple $\langle x, k\rangle$, etc.

Next we show how any CPS-translated STL-term can be suitably annotated with labels, so that defunctionalization can be applied.

We carry over the notation $\overline{X}[x^-, x^+]$ from Section 6, but now allow x^- and x^+ range over $\mathcal{L}_T^*$ instead of $\mathcal{L}^*$.

In order to label the CPS-translated terms, we now have to deal with the new case for contraction. Recall the CPS-translation of contraction from Figure 6:

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

The use of η expansions allows us to label the CPS-translated terms in a compositional way, much like in Lemma 6.2. For the sake of illustration, consider the case where Γ is empty and where X is $\mathbb{N}$, so that $\overline{X}$ is $\neg\neg\mathbb{N}$. Suppose we have already labelled the premise of the CPS-translation, say as in

$$y\colon \neg_{q_1}\neg_{a_1}\mathbb{N},\ z\colon \neg_{q_2}\neg_{a_2}\mathbb{N} \vdash \underline{t}\colon \overline{Y}[y^-, y^+].$$

Note that the types of the variables y and z will in general be annotated with different labels. This means that these variables have different types and we cannot use contraction to make them into a single variable x. However, we can annotate the η-expansion of x, i.e. the term $\eta(x, \neg\neg\mathbb{N})$, in the following two ways, in which a_1' and a_2' are fresh labels and q is any label term.

$$x\colon \neg_q\neg_{a_1'+a_2'}\mathbb{N} \vdash \lambda^{q_1}k.\, x \,@_q\, \mathsf{coercl}(\lambda^{a_1'}y.\, k \,@_{a_1}\, y)\colon \neg_{q_1}\neg_{a_1}\mathbb{N}$$

$$x\colon \neg_q\neg_{a_1'+a_2'}\mathbb{N} \vdash \lambda^{q_2}k.\, x \,@_q\, \mathsf{coercr}(\lambda^{a_2'}y.\, k \,@_{a_2}\, y)\colon \neg_{q_2}\neg_{a_2}\mathbb{N}$$

If we substitute the first term for y and the second term for z, then the resulting term is a labelled version of $x \colon \neg_q \neg_{a_1' + a_2'} \mathbb{N} \vdash \underline{t}[\eta(x, \overline{X})/y, \eta(x, \overline{X})/z] \colon \overline{Y}[y^-, y^+]$, i.e. the conclusion of the CPS-translation of contraction. This outlines how we can substitute η-expansions of x, whereas we could not just substitute x for both y and z,

Of course, it remains to be shown that it is possible to find a labelling of the whole term even when the type of x contains label terms, such as $a_1' + a_2'$ instead of just fresh labels. Note that in the type of x we could not have put a single label in place of $a_1' + a_2'$, as a_1' and a_2' are the unique labels of two different abstractions.

The next two lemmas show that it is indeed always possible to label CPS-translated terms appropriately. The following lemma first generalises the above labelling of η-expansion from $\mathbb{N}$ to an arbitrary type X. In the subsequent lemma, this is then used to deal with the case of contraction as outlined above.

For any term t of labelled STL, we write $|t|$ for the STL-term obtained by removing all label annotations and deleting all coercions. Likewise, we write $|X|$ for the type obtained by removing all label annotations. In the following lemma, we also extend the operation $+$ to sequences of label terms: If $r \in \mathcal{L}_T^*$ is $L_1 \ldots L_n$ and $r' \in \mathcal{L}_T^*$ is $L_1' \ldots L_n'$, then we write $r + r'$ for $(L_1 + L_1') \ldots (L_n + L_n')$.

Lemma 8.2. *For any* STL*-type X there exists labels $a_1', a_2', q_1, q_2 \in \mathcal{L}^*$ and labelled terms t_1 and t_2, such that $|t_1| = |t_2| = \eta(x, \overline{X})$ and such that the judgements*

$$x \colon \overline{X}[q, a_1' + a_2'] \vdash t_1 \colon \overline{X}[q_1, a_1] \qquad and \qquad x \colon \overline{X}[q, a_1' + a_2'] \vdash t_2 \colon \overline{X}[q_2, a_2]$$

are derivable for all label terms $q, a_1, a_2 \in \mathcal{L}_T^*$ for which the types $\overline{X}[q, a_1' + a_2']$, $\overline{X}[q_1, a_1]$ and $\overline{X}[q_2, a_2]$ are defined.

Proof. For any labelled type Y we define the list of label terms in positive/negative positions in it: $P(\mathbb{N}) = P(\bot) = N(\mathbb{N}) = N(\bot) = \varepsilon$ (the empty list), $N(Y \xrightarrow{L} Z) = L\, P(Y)\, N(Z)$ and $P(Y \xrightarrow{L} Z) = N(Y)\, P(Z)$.

Informally, the η-expansion of a variable is such that the label terms in positive position appear only as annotations of applications. There are no restrictions on the label terms in an application, so an η-expansion can be typed for arbitrary label terms in positive position. For the terms in negative position, there are constraints however. For each label term in a negative position the term contains a λ-abstraction. Since each abstraction must be annotated with a unique label, this leads to the constraint that the label terms in negative position can be obtained by coercion from the unique label of the abstraction in the term.

Formally, this can be expressed as follows: Let X be any STL-type and let X_1 and X_2 be labelled types with $|X_1| = |X_2| = X$. Then there exists a term t with $|t| = \eta(x, X)$ such that $x \colon X_1 \vdash t \colon X_2$ is derivable whenever the list $L_1 \ldots L_n := P(X_1) N(X_2)$ has the property that there are pairwise distinct labels $l_1, \ldots, l_n$ such that, for all $i \in \{1, \ldots, n\}$, the label l_i is a sub-term of the label term L_i.

The proof goes by induction on the type X. We spell out the case for function types.

- Case $Y \to Z$. We have $P(Y_1 \xrightarrow{L_1} Z_1)\, N(Y_2 \xrightarrow{L_2} Z_2) = N(Y_1)\, P(Z_1)\, L_2\, P(Y_2)\, N(Z_2)$, by definition. The assumption on this list implies that $P(Y_2)\, N(Y_1)$ has the property needed to apply the induction hypothesis to Y. Hence, there exists a labelled term t_y with $|t_y| = \eta(y, Y)$ and $y \colon Y_2 \vdash t_y \colon Y_1$.

  Likewise, the list $P(Z_1)\, N(Z_2)$ is such that we can apply the induction hypothesis to obtain a term t_z with $|t_z| = \eta(z, Z)$ and $z \colon Z_1 \vdash t_z \colon Z_2$.

If we define $t' = \lambda^l . t_z[f \mathbin{@}_{L_1} t_y / z]$, we therefore have $|t'| = \eta(f, Y \to Z)$ and $f \colon Y_1 \xrightarrow{L_1} Y_1 \vdash t' \colon Y_2 \xrightarrow{l} Y_2$.

By assumption, we know that we can choose l to be sub-term of L_2, without violating the constraint that each λ-abstraction must be uniquely identified by its label. The result therefore follows by applying coercions to t'.

The assertion now follows as a special case, where only coercions from a'_1 to $a'_1 + a'_2$ and from a'_2 to $a'_1 + a'_2$ are used. $\square$

With this lemma we can show that each derivation obtained by CPS-translation can be typed in the labelled variant of STL.

Lemma 8.3. *If $\Gamma \vdash t \colon X$ is derivable in* STL, *then there exist label terms $x^-, \gamma^+ \in \mathcal{L}_T^*$, such that, for all label terms $x^+, \gamma^- \in \mathcal{L}_T^*$ for which $\overline{\Gamma}[\gamma^-, \gamma^+]$ and $\overline{X}[x^-, x^+]$ are defined, the sequent $\overline{\Gamma}[\gamma^-, \gamma^+] \vdash t' \colon \overline{X}[x^-, x^+]$ is derivable for some labelled term t' with $|t'| = \underline{t}$.*

Proof. The proof goes by induction on the derivation of $\Gamma \vdash t \colon X$. We consider representative cases. To simplify the notation we write just $\overline{\Gamma}[\gamma^-, \gamma^+] \vdash \underline{t} \colon \overline{X}[x^-, x^+]$ to express that there exists a labelled term t' with $|t'| = \underline{t}$ for which $\overline{\Gamma}[\gamma^-, \gamma^+] \vdash t' \colon \overline{X}[x^-, x^+]$ is derivable.

- Case AX. This case follows directly from Lemma 8.2.
- Case $\to$E. By induction hypothesis, there exist label terms $y^- x_1^+, \gamma^+ \in^* L$ such that $\overline{\Gamma}[\gamma^-, \gamma^+] \vdash \underline{s} \colon \overline{X \to Y}[y^- x_1^+, y^+ x_1^-]$ is derivable for all label terms $y^+ x^-, \gamma^- \in \mathcal{L}_T^*$ for which all types in the sequent are defined. Also by induction hypothesis, there exists label terms $x_2^-, \delta^+ \in \mathcal{L}_T^*$, such that for all label terms x_2^+, δ^- the sequent $\overline{\Delta}[\delta^-, \delta^+] \vdash \underline{t} \colon \overline{X}[x_2^-, x_2^+]$ is derivable. In particular, we can choose x_2^+ to be x_1^+ and x_1^- to be x_2^+ and obtain $\overline{\Gamma}[\gamma^-, \gamma^+], \overline{\Delta}[\delta^-, \delta^+] \vdash \underline{s\,t} \colon \overline{Y}[y^-, y^+]$. Thus, we have shown that there exist label terms $y^-, \delta^+ \gamma^+$, such that the sequent is derivable for all label terms $y^+, \delta^- \gamma^-$, as was required to show.
- Case CONTR. By induction hypothesis, there exist label terms $y^-, \gamma^+, x_1^+, x_2^+$ such that $\overline{\Gamma}[\gamma^-, \gamma^+], x_1 \colon \overline{X}[x_1^-, x_1^+], x_2 \colon \overline{X}[x_2^-, x_2^+] \vdash \underline{t} \colon \overline{Y}[y^-, y^+]$ is derivable for all label terms $y^+, \gamma^-, x_1^-, x_2^-$. By Lemma 8.2, there exist labels $a'_1, a'_2 \in \mathcal{L}^*$ and we can annotate $\eta(x, \overline{X})$ to become t_1 and t_2 so that that $x \colon \overline{X}[x^-, a'_1 + a'_2] \vdash t_1 \colon \overline{X}[x_1^-, x_1^+]$ and $x \colon \overline{X}[x^-, a'_1 + a'_2] \vdash t_2 \colon \overline{X}[x_2^-, x_2^+]$ are derivable.

  We annotate the two copies of $\eta(x, \overline{X})$ in the CPS-translation of contraction as t_1 and t_2.

  Overall we obtain that there exist label terms $y^-, \gamma^+, a'_1 + a'_2$, such that

$$\overline{\Gamma}[\gamma^-, \gamma^+], x \colon \overline{X}[x^-, a'_1 + a'_2] \vdash \underline{t[x/x_1, x/x_2]} \colon \overline{Y}[y^-, y^+]$$

  is derivable for all y^+, γ^-, x^-, which shows the assertion. $\square$

This lemma justifies the definition of CpsDefun also for STL: Given a derivation of $\Gamma \vdash t \colon X$, annotate its CPS-translation using the lemma, so that $\Gamma[\gamma^-, \gamma^+] \vdash \underline{t} \colon X[x^-, x^+]$ becomes derivable, and take $\mathsf{CpsDefun}(\Gamma \vdash t \colon X) := (x^- \gamma^+, D(\underline{t}), x^+ \gamma^-)$.

8.2. Interactive Interpretation.

We now show how the Int-translation can be extended to STL and how it relates to defunctionalization. To this end, we again consider a variant of the type system with subexponential annotations. We extend LIN$_\text{EXP}$ to STL$_\text{EXP}$ by adding subexponential annotations to the contraction rule and by adding a new rule STRUCT for

$$\text{CONTR } \frac{\Gamma, y\colon A \cdot X, z\colon B \cdot X \vdash t : Y}{\Gamma, x\colon (A + B) \cdot X \vdash t[x/y, x/z] : Y}$$

$$\text{STRUCT } \frac{\Gamma, x\colon A \cdot X \vdash t : Y}{\Gamma, x\colon B \cdot X \vdash t : Y} \, A \lhd B$$

Figure 10: Additional Rules of STL_{EXP} over LIN_{EXP}
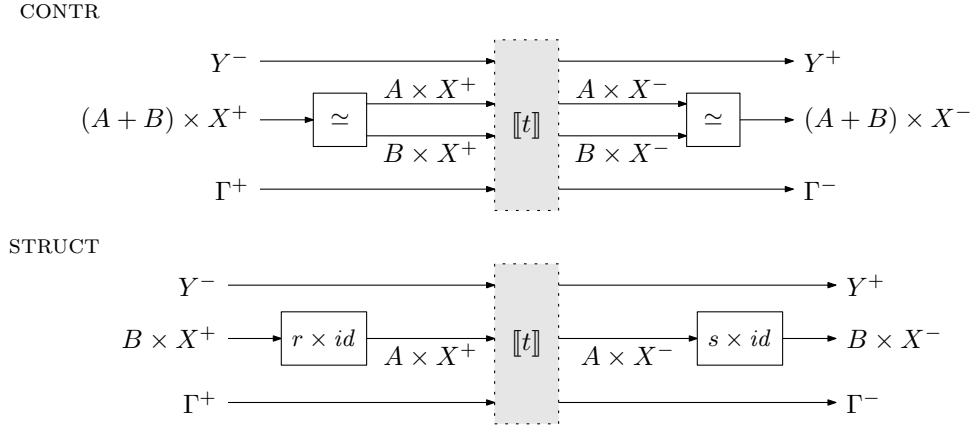
Figure 11: Int-Interpretation of new rules in STL_{EXP}

weakening of subexponential annotations. Rule STRUCT makes the type system more well-behaved and also increases the expressive power of the system. It is needed at the end of this Section in the proof of Proposition 8.7.

The new rules of STL_{EXP} are shown in Figure 10. To understand the annotations on CONTR, recall the explanation of subexponentials as making explicit the environment in which a variable is being used. The judgement in the premise of CONTR tells us that the variables y and z are used in environments with additional values of type A and B respectively. The subexponential $A + B$ in the conclusion tells 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. Rule STRUCT has a side condition $A \lhd B$, which expresses that A is a retract of B, i.e. that any value of type A can be encoded into one of type B. Formally, $A \lhd B$ holds if and only if there exist target expressions $x\colon A \vdash s : B$ and $y\colon B \vdash r : A$, such that $r[s[v/x]/y] \longrightarrow^* v$ holds for any target value v of type A. Notice in particular, that for isomorphic types $A \simeq B$, we have both $A \lhd B$ and $B \lhd A$.

The Int-translation of the new rules of STL_{EXP} is shown in Figure 11. Rule CONTR is interpreted by use of the isomorphism $(A + B) \times C \simeq A \times C + B \times C$, which is implemented using case distinction. A message of type $(A + B) \times X^+$ has the form $\langle \mathsf{inl}(a), x \rangle$ or $\langle \mathsf{inr}(b), x \rangle$. Depending on the case, the message is forwarded to the occurrence of either y or z. In the interpretation of rule STRUCT, one chooses s and r to witness $A \lhd B$ as defined above. The interpretation will be sound for any such choice of s and r, see.

8.3. Relating the Translations. We have now defined two translations from $\mathrm{STL}_{\mathrm{EXP}}$ to the target language. To relate them, we begin by spelling out a simple example to illustrate that both translations treat contraction in the same way.

Example 8.4. Consider again the source term $\lambda x{:}\mathbb{N}.\, x + x$. Its Int-interpretation may be depicted as follows.
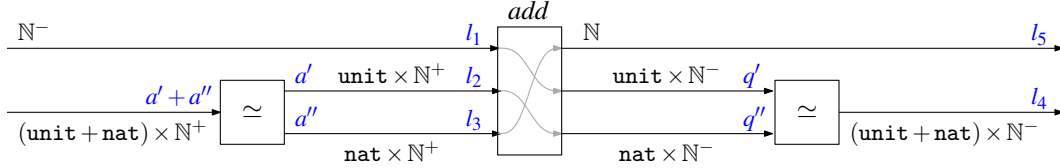
The box labelled *add* is defined as in the Introduction, up to uses of the isomorphism $\mathtt{unit} \times \mathbb{N}^+ \simeq \mathbb{N}^+$. The interpretation of rule CONTR inserts the two boxes labelled $\simeq$, which denote the canonical isomorphism of their type.

This program implements the term $\lambda x{:}\mathbb{N}.\, x + x$ as follows: To compute the result of the function when applied 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.

Consider now the CPS-translation of the term $\lambda x{:}\mathbb{N}.\, x + x$. If we omit the η-expansions in the translation of variables for simplicity, then we obtain the term

$$\lambda^{l_1}\langle x, k\rangle.\, t_1 \,@_{q'} (\lambda^{l_2} m.\, t_2 \,@_{q''} (\lambda^{l_3} n.\, k \,@_{l_5} (m + n))) \tag{8.1}$$

of type $(((\mathbb{N} \xrightarrow{a'+a''} \bot) \xrightarrow{q'} \bot) \times (\mathbb{N} \xrightarrow{l_5} \bot)) \xrightarrow{l_1} \bot$, wherein t_1 and t_2 are the η-expansions $(\lambda^{q'} k.\, x \,@_{l_4} \mathsf{coercl}(\lambda^{a'} n.\, k \,@_{l_2} n))$ and $(\lambda^{q''} k.\, x \,@_{l_4} \mathsf{coercr}(\lambda^{a''} n.\, k \,@_{l_3} n))$ respectively. These two η-expansions come from the CPS-translation of contraction. Defunctionalization of the term in (8.1) leads to the equations

$$apply_{l_1}(\langle\rangle, \langle x, k\rangle) = apply_{q'}(q'(x), l_2(x, k))$$
$$apply_{l_2}(l_2(x, k), m) = apply_{q''}(q''(x), l_3(m, k))$$
$$apply_{l_3}(l_3(m, k), n) = apply_{l_5}(k, m + n),$$

and the subterms t_1 and t_2 add the following equations:

$$apply_{q'}(q'(x), k) = apply_{l_4}(x, \mathsf{inl}(a'(k))) \qquad apply_{a'}(a'(k), n) = apply_{l_2}(k, n)$$
$$apply_{q''}(q''(x), k) = apply_{l_4}(x, \mathsf{inr}(a''(k))) \qquad apply_{a''}(a''(k), n) = apply_{l_3}(k, n)$$
$$apply_{a'+a''}(k, n) = \mathsf{case}\ k\ \mathsf{of}\ \mathsf{inl}(f_1) \Rightarrow apply_{a'}(f_1, n)$$
$$; \mathsf{inr}(f_2) \Rightarrow apply_{a''}(f_2, n)$$

$$\tag{8.2}$$

In order to understand how this program works, it is perhaps again useful to apply the above term to the argument $\langle \lambda^{l_4} k.\, k \,@_{a'+a''} 42, \lambda^{l_5} n.\, \mathtt{print\_int}(n)\rangle$. Defunctionalization then yields two additional equations.

$$apply_{l_4}(\langle\rangle, k) = apply_{a'+a''}(k, 42)$$
$$apply_{l_5}(\langle\rangle, n) = \mathtt{print\_int}(n)$$

Note how this program computes the result in the same way as the one obtained by Int-interpretation above. Both programs have the same skeleton. The points corresponding to the *apply*-equations are labelled in the Int-interpretation above. Notice in particular how

the equations (8.2) that come from the η-expansion in the CPS-translation of contraction correspond to the isomorphisms added by the interpretation of rule CONTR in the Int-interpretation.

Proposition 8.5. *For any $\Gamma \vdash t\colon X$ derivable in $\mathrm{STL}_{\mathrm{EXP}}$, there exists a target program $\mathsf{Int}(\Gamma \vdash t\colon X)$ that is a representative of the Int-interpretation of the derivation of the sequent and that has the same skeleton as $\mathsf{CpsDefun}(\Gamma \vdash t\colon X)$.*

Proof. The proof goes by induction on the derivation, just as for Proposition 7.2. The only new case is that for contraction. To handle this case, consider the defunctionalization of the two η-expansions $x\colon \overline{X}[q, a_1' + a_2'] \vdash t_1\colon \overline{X}[q_1, a_1]$ and $x\colon \overline{X}[q, a_1' + a_2'] \vdash t_2\colon \overline{X}[q_2, a_2]$ from Lemma 8.2. Let us write $q_1(i)$ for the i-th label term in the list q_1, and likewise for the other lists. Observe that the defunctionalization of the terms t_1 and t_2 yield equations of the following shape for all possible indices i:

$$apply_{q_1(i)}(-,-) = apply_{q(i)}(-, \mathsf{inl}(-))$$
$$apply_{q_2(i)}(-,-) = apply_{q(i)}(-, \mathsf{inr}(-))$$
$$apply_{(a_1' + a_2')(i)}(x, -) = \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}(y) \Rightarrow apply_{a_1(i)}(y, -)$$
$$; \mathsf{inr}(z) \Rightarrow apply_{a_2(i)}(z, -)$$
$$apply_{a_1'(i)}(-,-) = apply_{a_1(i)}(-,-)$$

These equations have the same skeleton as an appropriate choice of equations for the Int-interpretation of rule CONTR. One can thus choose a representative of the Int-interpretation having the same skeleton as the program obtained from CPS-translation and defunctionalization. □

The proposition establishes a relation of the Int-interpretation and CPS-translation followed by defunctionalization for terms typeable in $\mathrm{STL}_{\mathrm{EXP}}$. An obvious question is how much of STL is covered by this result. In the rest of this section we show that with rule STRUCT and recursive types in the target language, in fact any STL-term can be typed in $\mathrm{STL}_{\mathrm{EXP}}$.

We first give an example to show how recursive target types appear in the two translations. When we discussed defunctionalization for STL, we have already remarked that recursive types are needed in the target language to treat the full simply-typed λ-calculus. The following example (i) illustrates the use of STRUCT and recursive types in $\mathrm{STL}_{\mathrm{EXP}}$; and (ii) shows that recursive types may appear even in the defunctionalization of the simply-typed λ-calculus.

Example 8.6. An example that illustrates why without recursive types in the target language not every STL-term would be typeable in $\mathrm{STL}_{\mathrm{EXP}}$ is the application $t\ s$, where $t = \lambda g.\, g\ (\lambda x.\, g\ (\lambda y.x))$ and $s = \lambda f.\, f\ (f\ (\lambda x.x))$. The terms t and s can be given types $(\mathtt{unit} + \alpha) \cdot (\alpha \cdot (\alpha \cdot X \to X) \to X) \to X$ and $(\mathtt{unit} + \beta) \cdot (\beta \cdot X \to X) \to X$ respectively, for a certain type X. In these types the subexponential annotations have been simplified using only the isomorphism $(-) \times \mathtt{unit} \simeq (-)$, which can be used by means of rule STRUCT.

Without recursive types in the target language, the application $t\ s$ could not be typed, as this would require us to unify $(\mathtt{unit} + \beta) \cdot (\beta \cdot X \to X) \to X$ with $\alpha \cdot (\alpha \cdot X \to X) \to X$, which would require unifying β and $\mathtt{unit} + \beta$. With recursive types, however, we can simply let $B := \mu\beta.\, \mathtt{unit} + \beta$ and instantiate the type variable β to be B. Since we have $\mathtt{unit} + B \lhd B$, we can use rule STRUCT to give s the type $B \cdot (B \cdot X \to X) \to X$ and with this give $t\ s$ the type X .

It is interesting to note that CpsDefun maps $t\ s$ to a program that also uses recursive types. An annotation of the CPS-translation of $t\ s$ in the labelled version of STL is:

$$\underline{t} = \lambda^{l_1} \langle g, k \rangle.\, g @_{l_4} \langle \mathsf{coercl}(\lambda^{l_2}\langle x, k_1 \rangle.\, g @_{l_4} \langle \mathsf{coercr}(\lambda^{l_3}\langle y, k_3 \rangle.\, x @_{l_5 + l_6} k_3), k_1 \rangle), k \rangle$$

$$\underline{s} = \lambda^{l_4} \langle f, k \rangle.\, f @_{l_2 + l_3} \langle \mathsf{coercl}(\lambda^{l_5} k_2.\, f @_{l_2 + l_3} \langle \mathsf{coercr}(\lambda^{l_6}\langle x, k_1 \rangle.\, x @_{l_8} k_1), k_2 \rangle), k \rangle$$

$$\underline{t\ s} = \lambda^{l_7} k.\, \underline{t} @_{l_1} \langle \underline{s}, k \rangle$$

The types $\tau_{(-)}$ that appear in the defunctionalization are:

$$\tau_{l_1} = \mathsf{datatype}\ l_1\ \mathsf{of}\ \mathtt{unit} \qquad\qquad \tau_{l_2} = \mathsf{datatype}\ l_2\ \mathsf{of}\ \tau_{l_4}$$

$$\tau_{l_3} = \mathsf{datatype}\ l_3\ \mathsf{of}\ (\tau_{l_5} + \tau_{l_6}) \qquad\qquad \tau_{l_4} = \mathsf{datatype}\ l_4\ \mathsf{of}\ \mathtt{unit}$$

$$\tau_{l_5} = \mathsf{datatype}\ l_5\ \mathsf{of}\ (\tau_{l_2} + \tau_{l_3}) \qquad\qquad \tau_{l_6} = \mathsf{datatype}\ l_6\ \mathsf{of}\ \mathtt{unit}$$

The types τ_{l_3} and τ_{l_5} are mutually recursive.

The reader familiar with Game Semantics may recognise the term t as one of the Kierstead terms of order three that is often used to illustrate the need for justification pointers in Hyland-Ong-games. The other Kierstead term of order three $t' = \lambda g.\, g\ (\lambda x.\, g\ (\lambda y.y))$ can be given type $(\mathtt{unit} + \alpha) \cdot (\alpha \cdot (\mathtt{unit} \cdot X \to X) \to X) \to X$. With this term it *is* possible to give a type to $t'\ s$ without recursive types by setting $\alpha := \mathtt{unit} + \mathtt{unit}$ and $\beta := \mathtt{unit}$.

We end this section by showing that with rule STRUCT and recursive types in the target language, $\mathrm{STL_{EXP}}$ can indeed type any STL-term. Suppose $\Gamma \vdash t \colon X$ is a typing judgment of $\mathrm{STL_{EXP}}$. Write $|X|$ and $|\Gamma|$ for the type and context of STL obtained by removing all subexponential annotations, i.e. replacing any $A \cdot Y \to Z$ with $Y \to Z$ and removing subexponentials in the context. With this notation we have:

Proposition 8.7. *If* $\Gamma \vdash t \colon X$ *is derivable in* STL, *then there exist* Δ *and* Y *with* $\Gamma = |\Delta|$ *and* $X = |Y|$, *such that* $\Delta \vdash t \colon Y$ *is derivable in* $\mathrm{STL_{EXP}}$.

Proof. Using rule STRUCT, the following rules are derivable.

$$\mathrm{AX}\ \frac{}{x \colon \alpha_1 \cdot X \vdash x : X}\ \mathtt{unit} \lhd \alpha_1$$

$$\to\!\mathrm{E}\ \frac{\Gamma \vdash s \colon A \cdot X \to Y \qquad x_1 \colon A_1 \cdot X_1, \ldots, x_n \colon A_n \cdot X_n \vdash t \colon X}{\Gamma, x_1 \colon \alpha_1 \cdot X_1, \ldots, x_n \colon \alpha_n \cdot X_n \vdash s\ t \colon Y}\ \begin{array}{l} A \times A_1 \lhd \alpha_1, \ldots, \\ A \times A_n \lhd \alpha_n \end{array}$$

$$\mathrm{ADD}\ \frac{\Gamma \vdash s \colon \mathbb{N} \qquad x_1 \colon A_1 \cdot X_1, \ldots, x_n \colon A_n \cdot X_n \vdash t \colon \mathbb{N}}{\Gamma, x_1 \colon \alpha_1 \cdot X_1, \ldots, x_n \colon \alpha_n \cdot X_n \vdash s + t \colon \mathbb{N}}\ \begin{array}{l} \mathtt{nat} \times A_1 \lhd \alpha_1, \ldots, \\ \mathtt{nat} \times A_n \lhd \alpha_n \end{array}$$

$$\mathrm{CONTR}\ \frac{\Gamma, y \colon A \cdot X, z \colon B \cdot X \vdash t \colon Z}{\Gamma, x \colon \alpha_1 \cdot X \vdash t[x/y, x/z] \colon Z}\ (A + B) \lhd \alpha_1$$

We only need STRUCT to derive these rules.

If we use these derived rules with fresh target type variables for the α_i and disregard the $\lhd$-side-conditions for now, then together with the unchanged rules WEAK, EXCH $\to$I, IF, NUM, we can construct a skeleton of a typing derivation for t. This exists because t is typeable in STL.

To make this into a proper $\mathrm{STL_{EXP}}$ type derivation, it just remains to solve all the $\lhd$-constraints. The constraints all have the form $A \lhd \alpha$, i.e. the right-hand side of any constraint is a type variable. With recursive types, it is easy to solve such constraints: Let $A_1 \lhd \alpha, \ldots, A_n \lhd \alpha$ be all constraints with α on the right-hand side. A solution for it is

$\alpha := \mu\alpha.\, A_1 + \cdots + A_n$. In this way, we can solve the constraints for the type variables one after the other and so obtain a correct typing derivation. $\qquad\square$

We note that the proof provides a simple type inference procedure for STL_{EXP}. It is adapted from the simple type inference algorithm in [9]. Since [9] is concerned with LOGSPACE-computation, recursive types are not allowed there, and the constraints are solved by trying to unify α with $A_1 + \cdots + A_n$ instead of setting $\alpha := \mu\alpha.\, A_1 + \cdots + A_n$. This simple heuristic does not work for all STL terms and we need to allow recursive types to prove the above proposition in general.
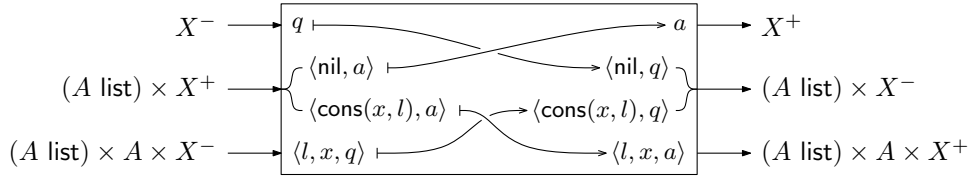
9. RECURSION

We conclude by explaining how the Int-interpretation and the subexponential annotations can be extended to handle the fixed point combinator of PCF.

Subexponential annotations for the fixed-point combinator can be given by

$$\text{FIX} \; \frac{}{\text{fix}_X \colon (A \text{ list}) \cdot (A \cdot X \to X) \to X}$$

where A list abbreviates $\mu\alpha.\, \texttt{unit} + A \times \alpha$.

The Int-interpretation of this term can be defined as follows:

Here, lists are used to implement a call stack. The function that we take the fixed point of has type $A \cdot X \to X$. This tells us that it needs to store a value of type A whenever it requests its argument. Thus, an activation record should be a stack of values of type A, which we encode as a list.

The appearance of the type $(A \text{ list})$ can also be explained from the subexponential type system. Clearly, the fixpoint combinator should have type $\text{fix} \colon B \cdot (A \cdot X \to X) \to X$ for some B. It should be defined to satisfy the equation $\text{fix } f = f \,(\text{fix } f)$. Consider typing judgements for the two terms in this equation. For the left-hand term we have $f \colon B \cdot (A \cdot X \to X) \vdash \text{fix } f : X$. As f appears twice in the right-hand term, we must use contraction to type it: $f \colon (1 + A \times B) \cdot (A \cdot X \to X) \vdash f \,(\text{fix } f) : X$. Notice now that we can give both terms the same type (using STRUCT) if we can solve the type equation $B \simeq 1 + A \times B$. We are thus naturally lead to choosing $B := A$ list. That case distinction appears in the above implementation of the fixed-point combinator is due to the duplicated use of the variable f.

The Int-interpretation implements recursion in a similar way as CPS-translation and defunctionalization. The CPS-translation of the fixed point combinator is

$$\lambda^{q_1}\langle f, k\rangle.\, \text{fix}_{\overline{X}} \,(\lambda^{q_4}g.\, \lambda k_1.\, f \,\langle \lambda^{q_3}k_2.\, g \; k_2,\, \lambda^{q_2}x.\, k_1 \; x\rangle) \; k \; .$$

A possible defunctionalization (without using control flow information) of this term is:

$$apply(q_1(), \langle f, k \rangle) = apply(q_4(f), k) \qquad apply(q_2(k_1), x) = apply(k_1, x)$$

$$apply(q_3(g), k_2) = apply(g, k_2) \qquad apply(q_4(f), k_1) = apply(f, \langle q_3(q_4(f)), q_2(k_1) \rangle)$$

Informally, the first three definitions to correspond to the inputs of the Int-interpretation above. A call to $apply(q_1(), \langle f, k \rangle)$ starts the recursion, a call to $apply(q_2(k_1), x)$ corresponds to the step function (that the fixed point is taken of) returning a result, and a call to $apply(q_3(g), k_2)$ corresponds to the step function requesting its argument. The final equation does not contribute to the external interface of the program and is used to implement the fixed point. The call stack, which above is encoded using lists, appears more implicitly in the continuations here.

10. 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. It may be seen as a simple nameless formulation of a combined CPS-translation and defunctionalization and it provides an alternative way of encoding continuations.

We have seen in this paper that working out the technical details of defunctionalization with explicit labels can become quite technical. The interactive interpretation admits a high-level description that abstracts from implementation details. Interactive model constructions may perhaps be useful in simplifying uses of defunctionalization. In the other direction, being aware that interactive models are related to standard compilation methods may help to improve non-standard compilation schemes based on interactive methods, such as [14, 8]. We may hope that some of the many existing techniques for compiler implementation can be adapted usefully to the non-standard schemes.

Types with subexponential annotations, in this form originally introduced in IntML [8], provide a logical account for the issues of managing value environments that are inherent to defunctionalization. With subexponential annotations, the type of a higher-order term fully specifies the interface of the target program obtained from it. The type system contains enough information in order to give a fully compositional definition of the translation to the target language.

The subexponential type system makes explicit issues that appear with defunctionalization and separate compilation. For example, in order to suppose we want to compile a function $f \colon A \cdot X \to Y$ separately from the rest of the program. Then one may compile the main program $f \colon B \cdot (A \cdot X \to Y) \vdash t \colon Z$ and $\vdash f \colon A \cdot X \to Y$ separately. A linker can combine the resulting two programs knowing only their types. Of course, the problems associated with defunctionalization and separate compilation do not just disappear. Suppose, for example, we only know the term f, but not the program t in which it will be used. Suppose further that X has the form $C \cdot \mathbb{N} \to \mathbb{N}$. Then the choice of the subexponential annotation C will limit which arguments f can be applied to; f can only be applied to arguments of type $D \cdot \mathbb{N} \to \mathbb{N}$ with $D \lhd C$. Choosing C without knowledge of t is possible, for example, if the target language has a type Heap with $D \lhd \mathsf{Heap}$ for any type D (anything can be stored on the heap). Then one may simply choose C to be Heap. This is not the only possible choice; for performance reasons one may consider performing a more precise

analysis in a linker or similar. The point is that the subexponential type system allows us to express such issues at a high level and to apply different possible solutions. Another example for this point is the explanation of the appearance of recursive target types in Section 8, which was given in terms of subexponential type annotations.

Subexponentials refine the exponentials in AJM games [2], where $!X$ is implemented using $\omega \cdot X$, where ω is a type of unbounded natural numbers. If we had used full exponentials in the Int-interpretation above, then we would have obtained a compilation method that encodes function values as values in ω, which is akin to storing closures on the heap. Subexponentials give us more control to avoid such encodings where unnecessary. The subexponential type system in this paper has its origin in Bounded Linear Logic [18, 38]. It is also similar to the type system for Syntactic Control of Concurrency (SCC) [15]. 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 [26] and in Melliès work on tensorial logic [29]. Connections of game models to compilation have also been made, e.g. [30]. Furthermore, it is well-known that continuation passing is related to message passing, see e.g. [41]. 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 Game Semantics. This suggests that techniques from Game Semantics help to analyse the possible traces of compiled machine code. For Java-like languages, there is recent work that connects fully abstract trace semantics [24] and Game Semantic models [33]. We hope that similar connections can be identified for the traces of machine code generated by compilers.

Work on concurrency and process calculus emphasises the interactive nature of computation. Milner's translation of the λ-calculus in the π-calculus [31] can be understood as a CPS-translation [37]. This connection was made more than once: see [37, §10] for historical notes. The interactive model that we have studied in this paper can be seen as a very static form of communicating processes, without process mobility or channel reconfiguration. In Milner's translation, channel names identify continuations and these are passed around explicitly. We have seen in this paper that by taking into account control flow information, it is possible to avoid passing continuations names, as they can be determined statically. It is an interesting direction for further work to find out if Milner's translation relates to CPS-translation and defunctionalization without control flow information. In this direction, it may perhaps be possible to connect to the interesting results of Berger et al. [5].

In further work, we should like to understand possible connections to Danvy's work on defunctionalized interpreters [11], or more generally to work on continuations and abstract machines, e.g. [10, 40]. 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-translation and defunctionalization may also help in identifying mathematical structure of efficient compilation methods. In particular, capturing call-by-value defunctionalizing compilation, perhaps similar to [7], should be interesting. Other interesting issues are efficient separate compilation and polymorphism:

the interpretation in $\text{Int}(\mathbb{T})$ is compositional and polymorphism can also be accounted for [38].

Finally, this paper clarifies the definition of IntML [8], which was introduced to capture LOGSPACE. In [8] we observed that IntML supports control operators, such as callcc, but their status remained somewhat unclear. It now turns out that the callcc combinator of [8] may be understood as the defunctionalization of a standard CPS implementation of callcc.

Acknowledgements. I would like to thank the anonymous referees for their constructive feedback and suggestions, which helped to improve the presentation of the results.

References

[1] Samson Abramsky, Esfandiar Haghverdi, and Philip J. Scott. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science*, 12(5):625–665, 2002.

[2] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409–470, 2000.

[3] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, TACS 2001*, volume 2215 of *Lecture Notes in Computer Science*, pages 420–447. Springer, 2001.

[5] Martin Berger, Kohei Honda, and Nobuko Yoshida. Sequentiality and the π-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, TLCA 2001*, volume 2044 of *Lecture Notes in Computer Science*, pages 29–45. Springer, 2001.

[6] Andreas Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56(1-3):183–220, 1992.

[7] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In Gert Smolka, editor, *European Symposium on Programming, ESOP 2000*, volume 1782 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 2000.

[8] Ugo Dal Lago and Ulrich Schöpp. Functional programming in sublinear space. In Andrew D. Gordon, editor, *European Symposium on Programming, ESOP 2010*, volume 6012 of *Lecture Notes in Computer Science*, pages 205–225. Springer, 2010.

[9] Ugo Dal Lago and Ulrich Schöpp. Type inference for sublinear space functional programming. In Kazunori Ueda, editor, *Asian Symposium on Programming Languages and Systems, APLAS 2010*, volume 6461 of *Lecture Notes in Computer Science*, pages 376–391. Springer, 2010.

[10] Vincent Danos, Hugo Herbelin, and Laurent Regnier. Game semantics and abstract machines. In *Logic in Computer Science, LICS 1996*, pages 394–405. IEEE Computer Society, 1996.

[11] Olivier Danvy. Defunctionalized interpreters for programming languages. In James Hook and Peter Thiemann, editors, *International Conference on Functional Programming, ICFP 2008*, pages 131–142. ACM, 2008.

[12] Olle Fredriksson and Dan R. Ghica. Seamless distributed computing from the geometry of interaction. In Catuscia Palamidessi and Mark Dermot Ryan, editors, *Trustworthy Global Computing, TGC 2012*, volume 8191 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2012.

[13] Olle Fredriksson and Dan R. Ghica. Abstract machines for game semantics, revisited. In *Logic In Computer Science, LICS 2013*, pages 560–569. IEEE Computer Society, 2013.

[14] Dan R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In Martin Hofmann and Matthias Felleisen, editors, *Principles of Programming Languages, POPL 2007*, pages 363–375. ACM, 2007.

[15] Dan R. Ghica, Andrzej S. Murawski, and C.-H. Luke Ong. Syntactic control of concurrency. *Theoretical Computer Science*, 350(2-3):234–251, 2006.

[16] Dan R. Ghica, Alex I. Smith, and Satnam Singh. Geometry of synthesis IV: compiling affine recursion into static hardware. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *International Conference on Functional Programming, ICFP 2011*, pages 221–233. ACM, 2011.

[17] Jean-Yves Girard. Towards a geometry of interaction. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, pages 69–108. American Mathematical Society, 1989.

[18] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science*, 97:1–66, 1992.

[19] Masahito Hasegawa. On traced monoidal closed categories. *Mathematical Structures in Computer Science*, 19(2):217–244, 2009.

[20] Ichiro Hasuo and Naohiko Hoshino. Semantics of higher-order quantum computation via geometry of interaction. In *Logic in Computer Science, LICS 2011*, pages 237–246. IEEE Computer Society, 2011.

[21] Martin Hofmann and Thomas Streicher. Continuation models are universal for lambda-mu-calculus. In *Logic in Computer Science, LICS 1997*, pages 387–395. IEEE Computer Society, 1997.

[22] J. M. E. Hyland and C.-H. Luke Ong. Pi-calculus, dialogue games and PCF. In *Functional Programming Languages and Computer Architecture, FPCA 1995*, pages 96–107, 1995.

[23] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163:285–408, December 2000.

[24] Alan Jeffrey and Julian Rathke. Java jr: Fully abstract trace semantics for a core Java language. In Shmuel Sagiv, editor, *European Symposium on Programming, ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer, 2005.

[25] André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Societ*, 119(3):447–468, 1996.

[26] Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, Berlin, Heidelberg, 2004.

[27] Paul Lorenzen. Ein dialogisches Konstruktivitatskriterium. *Infinitistic Methods*, 1961.

[28] Ian Mackie. The geometry of interaction machine. In Ron K. Cytron and Peter Lee, editors, *Principles of Programming Languages, POPL 1995*, pages 198–208. ACM, 1995.

[29] Paul-André Melliès. Game semantics in string diagrams. In *Logic in Computer Science, LICS 2012*, pages 481–490. IEEE, 2012.

[30] Paul-André Melliès and Nicolas Tabareau. An algebraic account of references in game semantics. *Electronic Notes in Theoretical Computer Science*, 249:377–405, 2009.

[31] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[32] Andrzej S. Murawski and Nikos Tzevelekos. Full abstraction for Reduced ML. *Annals of Pure and Applied Logic*, 164(11):1118–1143, 2013.

[33] Andrzej S. Murawski and Nikos Tzevelekos. Game semantics for interface middleweight java. In Suresh Jagannathan and Peter Sewell, editors, *Principles of Programming Languages, POPL 2014*, pages 517–528. ACM, 2014.

[34] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[35] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[36] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference - Volume 2*, ACM '72, pages 717–740. ACM, 1972.

[37] Davide Sangiorgi. From lambda to pi; or, rediscovering continuations. *Mathematical Structures in Computer Science*, 9(4):367–401, 1999.

[38] Ulrich Schöpp. Stratified bounded affine logic for logarithmic space. In *Logic in Computer Science, LICS 2007*, pages 411–420. IEEE, 2007.

[39] Zhong Shao and Andrew W. Appel. Efficient and safe-for-space closure conversion. *ACM Transactions on Programming Languages and Systems*, 22(1):129–161, 2000.

[40] Thomas Streicher and Bernhard Reus. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, 1998.

[41] Hayo Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, The University of Edinburgh, 1997.

[42] Akira Yoshimizu, Ichiro Hasuo, Claudia Faggian, and Ugo Dal Lago. Measurements in proof nets as higher-order quantum circuits. In Zhong Shao, editor, *European Symposium on Programming, ESOP 2014*, volume 8410 of *Lecture Notes in Computer Science*, pages 371–391. Springer, 2014.