Defunctionalisation as Modular Closure Conversion

Ulrich Schöpp
Ludwig-Maximilians-Universität München
Oettingenstr. 67
Munich, Germany
schoepp@tcs.ifi.lmu.de

ABSTRACT

We study the problem of translating from call-by-value PCF to a first-order low-level language. Such translations are typically defined by induction on the structure of the source term. Each sub-term is translated to a low-level program fragment and the translation of the whole term is a composition of these fragments. It is desirable to follow this compositional approach also in reasoning about such translations, e.g. to show correctness of the translation by verifying the low-level fragments individually. In this paper, we define a defunctionalisation method in which the low-level program fragments are considered as little modules with a well-defined interface. We show correctness of the translation by decomposing it into a number of steps that each allows compositional reasoning. The main step is a typed closure conversion that translates PCF into a calculus based on interaction semantics. It takes into account low-level information, e.g. on closure representation and stack shape, that is obtained by global program analysis. We capture such information using an annotated type system for PCF and show that suitable annotations can be computed by type inference.

CCS CONCEPTS

• **Software and its engineering → Compilers**; **Modules / packages**; • **Theory of computation → Linear logic**;

KEYWORDS

defunctionalisation, ML-style modules, call-by-value PCF

ACM Reference format:
Ulrich Schöpp. 2017. Defunctionalisation as Modular Closure Conversion. In *Proceedings of PPDP'17, Namur, Belgium, October 9–11, 2017,* 21 pages.
https://doi.org/10.1145/3131851.3131868

1 INTRODUCTION

In higher-order programming languages there is no strict distinction between program code and data. Functions may be computed and passed around as data, while they can also be considered program code to be executed. Compilers have the task of implementing higher-order functions in low-level languages that separate code and data. The aim is to produce efficient low-level machine code, of course. But being able to translate a whole source program to efficient machine code is just one aspect of compilation.

There are a number of situations where a more detailed understanding of the low-level machine code produced by compiling source code is needed. For practical applications, it is desirable to decompose a source program into modules, translate them to low-level code independently and link the resulting code on a low level. This is desirable to improve compilation times, for example, or to allow linking of code produced by different compilers or written in different languages. Implementing such low-level linking of code coming from different source requires the definition of a low-level calling convention. If the source language is a functional programming language and the compiled modules contain higher-order functions, then one must know how the compiler decomposes higher-order functions into low-level code and data, as such details are exposed in low-level code.

There are also theoretical reasons for trying to understand the low-level code produced by compilation of (parts of) high-level programs. One main motivation for the work in this paper has been the study of the relation of practical compilation methods to game semantics. Game semantics can be seen as a method for interpreting high-level programs by low-level interaction strategies. Connections between game semantics and defunctionalisation have been identified for call-by-name languages in [15], but for call-by-value languages the relations are not fully understood. To relate compilation methods to game semantics, one needs to understand how they translate source code to low-level code at a similar abstraction level as the low-level interaction strategies. Since practical compilation methods are typically defined as a composition of many small steps, this involves understanding the low-level code produced by a number of such compilation steps. Another theoretical motivation is the specification and verification of the low-level code produced by compilation. Work on compositional compilation with CompCert shows that this is a non-trivial task [2].

Different ways of compiling higher-order functional programming languages vary with regard to how difficult it is to understand the low-level code produced by them. A popular way of implementing functional programming languages is using closures with a function pointer. Function values are encoded by a record containing a pointer p to the code for the function body and a tuple $\vec{v}$ of the values of the free variables in the function value. The application of a function represented by $\langle p, \vec{v}\rangle$ to an argument w becomes simply a call $(*p)(\vec{v}, w)$ (in C-notation). The low-level code produced by this implementation of functions is relatively easy to work with. Typically, closures are stored on the heap, so that all functions are encoded uniformly by a pointer. It is then not hard, for example,

PPDP'17, October 9–11, 2017, Namur, Belgium
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5291-8/17/10…$15.00
https://doi.org/10.1145/3131851.3131868

to define a calling convention and to link compiled low-level code with programs written in C, for example.

Defunctionalisation is another well-known method for implementing higher-order functional languages. It is useful for example in order to compile to a target language that does not have pointers or indirect calls. An extreme example is hardware synthesis for FPGAs. Indeed, the Geometry of Synthesis [7] implements a form of Geometry of Interaction, which can be considered a variant of defunctionalisation [15]. Another reason for using defunctionalisation is that it can be used to translate higher-order programs into simple first-order programs that are relatively easy to optimise well, as has been demonstrated by the MLton compiler [3, 18].

Defunctionalisation can be seen as a variant of closure conversion. The difference is that the code for function application is not identified by a pointer p, but by a tag t that is sufficient to identify the code. The pairs $\langle t, \vec{v}\rangle$ that represent functions are typically encoded using an algebraic data type that also statically controls the length and type of the vector $\vec{v}$. In the simplest case, the tag would be a unique name of the function. For application one implements a static procedure $\mathtt{apply}(\langle t, \vec{v}\rangle, w) = \mathtt{case}\ t\ \mathtt{of} \dots$ that performs case distinction on the tag t. Application of $\langle t, \vec{v}\rangle$ to w thus becomes a call to $\mathtt{apply}(\langle t, \vec{v}\rangle, w)$.

Practical implementations of defunctionalisation are more complicated, however. First, one typically uses some control-flow information to reduce the number of possible cases in the case-distinction of `apply`. For example, if one can determine statically that only two function values are possible at a certain call site, then one may invoke a smaller procedure $\mathtt{apply}'$ that performs case distinction only on these two possible cases, rather than all cases. The data type for the tag at this point can be optimised to allow only for the two possible cases. With such an optimisation, there are many small `apply`-procedures and control flow information determines which one to use in application. Such an optimised defunctionalisation method is implemented in the MLton compiler [3].

Such optimisations make it more difficult to understand the low-level code produced by defunctionalisation. Consider, for example, the situation where one has two separately compiled modules, one containing a function $\mathtt{f} : ((\mathtt{int} \to \mathtt{int}) \to \mathtt{int}) \to \mathtt{int}$ and the other an argument $g : (\mathtt{int} \to \mathtt{int}) \to \mathtt{int}$ for it. Suppose we want to perform the application of `f` to `g` by linking the compiled low-level code, perhaps because the two modules have been compiled by different compilers or perhaps because one is implemented as a circuit on an FPGA. If they are to be compiled separately, the two low-level modules must somehow contain `apply`-procedures for `f` and `g` respectively. Note that if `f` wants to apply its argument to some value, then it needs to invoke an `apply`-procedure for `g`, which would be defined in the other module. Additionally, this code must also be able to invoke an `apply`-procedure for `g`'s argument, which would be found in the module for `f` and which may not be the same as the one for invoking `f` itself. So, in order to be able to compose the two compiled low-level code for the two modules, one must explain how they implement suitable `apply`-procedures and one must provide an interface such that both modules can find and invoke the right `apply`-procedure in the other module.

To realise such linking of low-level modules, one must keep track of a number of low-level details. For example, the modules need to pass (encoded) closures back and forth. The choice of data type to represent the tags in defunctionalisation in one module may affect the choice of data types in the other module. This choice of data types, moreover, depends on control flow information that is global to the program. It is not immediately clear how to keep track of such details and there are many details to manage. That it is possible to do so was shown by Fourtounis et al. [6], who have presented a defunctionalisation method with separate compilation. It is desirable to further clarify these issues and to account for many possible solutions in a systematic, clear and general way.

This is especially important for compositional specification and formal verification. In which sense does the low-level code for $\mathtt{f} : ((\mathtt{int} \to \mathtt{int}) \to \mathtt{int}) \to \mathtt{int}$ correctly implement the source function of this type? We would like to be able to specify and verify correctness without having the source code for the function argument. This is motivated by allowing linking with terms written in other languages and by the investigation of the relation to game semantics. Existing correctness proofs for defunctionalisation, e.g. [1, 12] do not appear to apply directly in such a situation.

In this paper we show how to address such issues of low-level modularity by considering defunctionalisation as a translation into *modules* with a well-defined signature. In essence, we use a module system to keep track of many `apply`-functions and their composition. All parts of a program will be translated into tiny modules and the translation of the whole program will be obtained by composing these modules. This use of modules in the formulation of defunctionalisation will not incur runtime overhead: our modules will just be incomplete fragments of low-level program code. Viewing code fragments as modules allows us to make sense of incomplete code fragments, as is required e.g. if one wants to consider `f` and `g` from above in isolation. To show correctness of the translation, we show that it can be described as a type-correct instance of a general typed closure conversion method. We stress that the main point of the paper is not just that defunctionalisation can be defined in a type-correct way or that it can be defined by induction on the source term. This has been done in previous work, e.g. [12].

2 CALL-BY-VALUE PCF

We take call-by-value PCF as a simple higher-order source language. Recall that this is a simply-typed functional programming language with types $X, Y ::= \mathbb{N} \mid X \to Y$. We use the following terms.

$$\begin{aligned} s, t ::= {} & x \mid \mathtt{zero} \mid \mathtt{succ}(t) \mid \mathtt{pred}(t) \mid \mathtt{ifz}\ s\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 \\ & \mid s\ t \mid \mathtt{fn}\, x \Rightarrow t \mid \mathtt{fix}\ f\ x \Rightarrow t \end{aligned}$$

The term $\mathtt{fix}\ f\ x \Rightarrow t$ denotes a recursive function definition; the bound variable f of type $X \to Y$ can be used for recursive calls.

3 LOW-LEVEL LANGUAGE

We aim to translate PCF to a simple low-level language that has a similar level of abstraction as compiler intermediate languages like LLVM. Since we focus on defunctionalisation in this paper, it suffices to use a very simple low-level language with nothing more than register variables and conditional jumps. We do not need a built-in call-stack, pointers or indirect jumps. This does not mean that such features cannot be used. We comment in Sec. 7 how a machine stack can be used, if desired. By using a very simple low-level language, we retain control over low-level implementation details, which is useful, for example, for implementing tail recursion or for identifying GC-roots. It also allows us to consider unusual targets, e.g. for hardware synthesis.

The low-level language has the following first-order types.

$$A, B ::= \alpha \mid 0 \mid \mathtt{unit} \mid \mathtt{int} \mid A \times B \mid A + B \mid A \cup B \mid \mu\alpha.\, A$$

Here, α ranges over an infinite set of type variables, 0 is an empty type and int represents $\mathbb{N}$-values (let us assume that these are 32-bit integers).

The main purpose of the type system is to capture information for data representation purposes, much like in LLVM. For example, the types are sufficient to represent tuples without overhead: a value of type $A \times B$ can be encoded simply as a value of type A and one of type B. The low-level type system is not intended to capture interesting correctness guarantees. Indeed, the untagged union type $A \cup B$ makes the type system weak. While it would be possible to work without union types, they allow a slightly more efficient representation of values. In Sec. 5.1, we shall see examples, where control-flow information allows us to omit defunctionalisation tags completely, even when more than one function value is possible.

The low-level values are defined by:

$$\begin{aligned} v, w ::= {} & x \mid () \mid n \mid \langle v, w\rangle \mid \mathtt{inl}_{A+B}(v) \mid \mathtt{inr}_{A+B}(v) \\ & \mid \mathtt{in}_{A,B}(v) \mid \mathtt{fold}_A(v) \end{aligned}$$

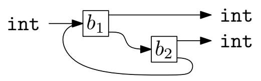
The value $\mathtt{in}_{A,B}(v)$ denotes an injection into a union type. The type system is defined so that $\mathtt{in}_{A,B}(v)$ is well-typed only when B is either $A \cup C$ or $C \cup A$. If $v\colon A$, then we have $\mathtt{in}_{A,A\cup C}(v)\colon A \cup C$ and $\mathtt{in}_{A,C\cup A}(v)\colon C \cup A$ and these terms denote the respective injections in the union.

Low-level programs are made up of blocks. A *block* has the form $f\,(x\colon A) = b$, where f is the block label, A is the argument type, x is a formal parameter of type A, and b is the body formed by the following grammar.

$$\begin{aligned} b ::= {} & f(v) \mid \mathtt{let}\ x = \mathtt{op}(v)\ \mathtt{in}\ b \mid \mathtt{let}\ \langle x, y\rangle = v\ \mathtt{in}\ b \\ & \mid \mathtt{let}\ \mathtt{in}_{A,B}(x) = v\ \mathtt{in}\ b \mid \mathtt{let}\ \mathtt{fold}_{\mu\alpha.\,A}(x) = v\ \mathtt{in}\ b \\ & \mid \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow g(w);\ \mathtt{inr}(y) \Rightarrow h(u) \end{aligned}$$

Here, f, g and h range over block labels. The expression $f(v)$ denotes a jump to the block with label f, the formal parameter of which is set to v (think of $f(v)$ as "$x := v$; goto f"). A block therefore consists of a number of let-bindings and ends with a (possibly conditional) jump with argument to another block. In the first let-term, op ranges over primitive operations, such as add, sub and mul, which take an argument of type int×int and produce a result of type int, as well as eq, which produces a result of type unit + unit, seen as a boolean. The other let-terms represent pattern matching operations. For example, the block $f\,(x\colon A{\times}B) = \mathtt{let}\ \langle x_1, x_2\rangle = x\ \mathtt{in}\ g(x_1)$ implements the first projection from $A \times B$ to A. The types guarantee that unpairing and unfolding always succeed. For values of union type, the deconstruction may be undefined. For example, ($\mathtt{let}\ \mathtt{in}_{A,A\cup B}(x) = \mathtt{in}_{B,A\cup B}(w)\ \mathtt{in}\ b$) is undefined if $A \neq B$.

We find it useful to depict sets of blocks as control flow graphs. Blocks are depicted as white boxes and we draw an arrow from one block to another if the first block ends with a jump to the latter. We sometimes annotate such edges with the type of the value that is passed as argument when the corresponding jump is taken. For example, two blocks $b_1 : f\,(x\colon \mathtt{int}) = \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}(y) \Rightarrow h_1(w);\ \mathtt{inr}(y) \Rightarrow g(r)$ and $b_2 : g\,(x\colon \mathtt{int}) = \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}(y) \Rightarrow f(w);\ \mathtt{inl}(y) \Rightarrow h_2(r)$ would be depicted as follows.

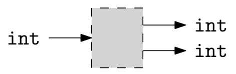


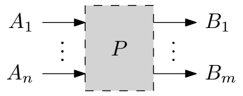
3.1 Program Fragments

We want to view low-level programs as being composed from modules. To this end we decompose control-flow graphs into fragments.

Program fragments are control-flow graphs with a choice of incoming and outgoing edges. Formally, a program fragment $P = (\mathit{entry}, S, \mathit{exit})$ is given by a set S of well-typed blocks, a list of entry labels $\mathit{entry} = (\mathit{entry}_i\colon A_i)_{i=1,\dots,n}$ and a list of exit labels $\mathit{exit} = (\mathit{exit}_i\colon B_i)_{i=1,\dots,m}$. We require that no two blocks in S have the same label. All blocks in S must be well-typed, given that the entry and exit labels have their specified type. Each entry label must be defined in S, while no exit label may be defined in it. We write $P\colon A_1, \dots, A_n \to B_1, \dots, B_m$ to indicate the types of the entry- and exit-labels of a program.

We depict a program fragment $P\colon A_1, \dots, A_n \to B_1, \dots, B_m$ by a grey box with dashed lines as shown on the right below. The example blocks above can be made into the following fragment $((f\colon \mathtt{int}), \{b_1, b_2\}, (h_1\colon \mathtt{int}, h_2\colon \mathtt{int}))$, which would be depicted as on the left below. One should think of such a grey box as representing the control-flow graph of the blocks in the fragment.





In the following, program fragments will be the building blocks from which larger programs are built. For example, if one has fragments $P\colon \vec{A} \to \vec{B}$ and $Q\colon \vec{B} \to \vec{C}$, then one can combine them to form a single fragment $(Q \circ P)\colon \vec{A} \to \vec{C}$. In essence, one just takes all the blocks from P and Q and renames the entry labels of Q to match the exit labels of P. That is, if P jumps to its i-th exit label, then the jump goes to the i-th entry label of Q. Some care must be take in case both P and Q define a block with the same label; in such cases one may freshly rename the label in one of the programs.

To allow renaming of labels, we identify program fragments up to (bijective) renaming of labels. Notice that the entry and exit labels are identified by their position in the respective lists, and can be renamed without harm.

Each program may be transformed into a program with a single entry and exit label. From a program $P\colon A_1, \dots, A_n \to B_1, \dots, B_m$, one obtains a program $P'\colon A_1 + \cdots + A_n \to B_1 + \cdots + B_m$ by adding case distinction to the entry and injection blocks into the exit. In the graphical notation, we make such conversions implicitly, i.e. we draw P' as we draw P above, when the types make such conversions clear.

We consider two programs $P, Q\colon A \to B$ without free value variables *equal* if they are equal extensionally: If one jumps with value v to the entry label of each of the programs, then they either both diverge, or they both jump to their exit label, with the same argument value. In general, two programs are equal if applying any closing substitution to them produces equal programs.

For types A and B we introduce a relation $A \lhd B$ that formalises that any value of type A can be encoded as a value of type B. We define $A \lhd B$ to hold if there exist two program fragments $\mathtt{in}\colon A \longrightarrow B$ and $\mathtt{out}\colon B \longrightarrow A$, such that their composition $b_2 \circ b_1 \colon A \longrightarrow A$ (obtained by taking the blocks from both programs and making the entry label of out the exit label of in) is equal to the identity. For the purposes of this paper, we shall use only instances of $\lhd$, where in and out are computationally inexpensive, e.g. $A \lhd (A \cup B)$.

3.2 Signatures

In the translation of PCF to the low-level language, we will view the low-level program fragments as tiny low-level *modules*. The translation will be strongly compositional in the sense that the translation of a program is a composition of the fragments obtained by translating its parts. Moreover, this composition of fragments will be type-correct: We next define SML-style signatures for program fragments that will specify their interfaces.

Signatures are defined by the following grammar.

$$S ::= A \to B \mid \mathtt{sig}\, X\colon S, \dots, X\colon S\ \mathtt{end} \mid \mathtt{functor}(X\colon S) \to S$$
$$\mid A \cdot S \mid \forall \alpha \lhd A.\, S \mid \exists \alpha \lhd A.\, S$$

Here, X ranges over variables. We allow signatures of the form $\mathtt{sig}\, X_1\colon S_1, \dots, X_n\colon S_n\ \mathtt{end}$ only if the variables X_i are pairwise distinct. The functor binds the variable in its argument as usual; we assume that the bound variable is named freshly.

We next define what it means for a program fragment to have signature S. First we associate to each signature S two lists S^- and S^+ of low-level types. The program fragments of signature S will then be fragments $P\colon S^- \longrightarrow S^+$ with further properties depending on S.

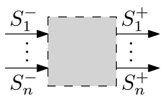
S	S^-	S^+
$A \longrightarrow B$	A	B
$\mathtt{sig}\, X_1\colon S_1, \dots, X_n\colon S_n\ \mathtt{end}$	$S_1^-, \dots, S_n^-$	$S_1^+, \dots, S_n^+$
$\mathtt{functor}(X\colon S_1) \longrightarrow S_2$	S_2^-, S_1^+	S_2^+, S_1^-
$A \cdot S_1$	$A \times S_1^-$	$A \times S_1^+$
$\forall \alpha \lhd A.\, S_1$	$S_1^-[A/\alpha]$	$S_1^+[A/\alpha]$
$\exists \alpha \lhd A.\, S_1$	$S_1^-[A/\alpha]$	$S_1^+[A/\alpha]$

Here we denote list concatenation using comma. We write $A \times S$ for the list obtained by replacing each B in S with $A \times B$.

In the rest of this section we define what it means for a program fragment to have signature S.

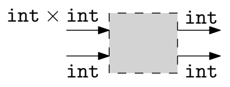
A *fragment of signature* $A \longrightarrow B$ is any fragment $P\colon A \longrightarrow B$ with a single entry label of type A, call it *entry*, and a single exit label of type B, call it *ret*. Such a fragment implements a function from A to B in the following sense: To apply the function to argument $v\colon A$, one executes the jump $\mathit{entry}(v)$. The program will then compute the result and return it by jumping to *ret*.

A *fragment of signature* $S = \mathtt{sig}\, X_1\colon S_1, \dots, X_n\colon S_n\ \mathtt{end}$ is a fragment $P\colon S^- \longrightarrow S^+$ that provides an implementation for each of the signatures $S_1, \dots, S_n$ in the following sense. If, for any i, one restricts the interface to just the entry and exit labels belonging to S_i, then the resulting fragment must be a fragment of signature S_i.



For example, a fragment of signature

```
sig add: int × int → int,
    square: int → int
end
```



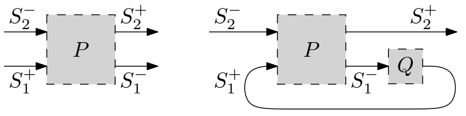
provides two entry labels *add.entry*, *square.entry* and two exit labels *add.ret*, *square.ret*. These labels are shown from top to bottom in the figure on the right.

A concrete fragment of this example signature would be:

$$\begin{aligned} \mathit{add.entry}(x) &= \mathtt{let}\ y = \mathtt{add}(x)\ \mathtt{in}\ \mathit{add.ret}(y) \\ \mathit{square.entry}(x) &= \mathtt{let}\ y = \mathtt{mul}(\langle x, x\rangle)\ \mathtt{in}\ \mathit{square.ret}(y) \end{aligned} \tag{1}$$

If we consider it as a fragment with just the labels *add.entry* and *add.ret*, then it must be a fragment of signature $\mathtt{int} \times \mathtt{int} \longrightarrow \mathtt{int}$.

A *fragment of signature* $S = \mathtt{functor}(X\colon S_1) \longrightarrow S_2$ is a fragment $P\colon S^- \longrightarrow S^+$ that represents a functor in the following sense: Whenever Q is a fragment of signature S_1, then the composition of P and Q shown on the right below is a fragment of signature S_2.



(2)

Recall from above that the composition may require renaming. We draw Q as having a single entry and exit label, as justified above.

For example, let S_1 be $\mathtt{sig}\ \mathit{add}\colon \mathtt{int}{\times}\mathtt{int} \longrightarrow \mathtt{int}, \mathit{square}\colon \mathtt{int} \longrightarrow \mathtt{int}\ \mathtt{end}$ as above and let S_2 be $\mathtt{sig}\ f\colon \mathtt{int} \longrightarrow \mathtt{int}\ \mathtt{end}$. Then, a fragment of signature $\mathtt{functor}(X\colon S_1) \longrightarrow S_2$ should have three entry labels, one coming from S_2^- and two from S_1^+. Suppose these labels are defined (in this order) by the following blocks.

$$\begin{aligned} f.\mathit{entry}(x\colon \mathtt{int}) &= X.\mathit{add.entry}(\langle x, x\rangle) \\ X.\mathit{add.ret}(y\colon \mathtt{int}) &= X.\mathit{square.entry}(y) \\ X.\mathit{square.ret}(z\colon \mathtt{int}) &= f.\mathit{ret}(z) \end{aligned} \tag{3}$$

Suppose the exit labels are $f.\mathit{ret}$, $X.\mathit{add.entry}$ and $X.\mathit{square.entry}$, in this order. To see how this defines functor, consider the fragment of signature S_1 defined in (1) above. To connect it to our functor as shown in (2), we just need to rename the entry and exit labels of the blocks in (1) appropriately and add these blocks to the blocks for the functor. In this case, renaming amounts to prefixing each label with 'X.', i.e. rename *add.entry* to $X.\mathit{add.entry}$, etc. The resulting blocks together with those from (3) are then a fragment of signature S_2, with entry label $f.\mathit{entry}$ and exit label $f.\mathit{ret}$. It implements the function $x \mapsto (x + x)^2$.

By itself, this simple realisation of functors is quite limited. Suppose, for example, that we want to implement f so that it implements the mapping $x \mapsto x^2 + x$. It would be natural to modify the above definitions to:

$$\begin{aligned} f.\mathit{entry}(x\colon \mathtt{int}) &= X.\mathit{square.entry}(x) \\ X.\mathit{add.ret}(z\colon \mathtt{int}) &= f.\mathit{ret}(z) \\ X.\mathit{square.ret}(y\colon \mathtt{int}) &= X.\mathit{add.entry}(y, ??) \end{aligned}$$

But we cannot complete the definition in the last line, as the initial value of x is not available there anymore. To address this problem, we use signatures of the form $A \cdot S$.

To explain the signatures of the form $A \cdot S_1$, we need the following notation. Given a fragment $F\colon A_1, \dots, A_n \longrightarrow B_1, \dots, B_m$, define a fragment $C \cdot F$ of type $(C{\times}A_1), \dots, (C{\times}A_n) \longrightarrow (C{\times}B_1), \dots, (C{\times}B_m)$

that behaves like F, the only difference being that it passes around an additional value of type C unchanged. The fragment $C \cdot F$ can be defined from F by giving each block a new first argument of type C, which is then passed on uninspected and unchanged. For example, $f(x \colon A) = g(v)$ becomes $f(\langle z, x\rangle \colon C \times A) = g(\langle z, v\rangle)$.

A *fragment of signature* $S = A \cdot S_1$ then is a program fragment $P \colon S^- \to S^+$ for which there exists a fragment Q of signature S_1, such that P equals $A \cdot Q$. In other words, P must behave like a fragment of signature S_1 that in addition passes around a value of type A unchanged and uninspected. The additional argument of type A may be seen as a *callee-save argument* (although, strictly speaking, we are talking about jumps here). The callee, P, saves this argument and guarantees to give it back on exit. We call this guarantee of all fragments of signature $A \cdot S_1$ the *callee-save invariant.*

Callee-save arguments are useful to address the problem that we had with defining the functor for $x \mapsto x^2 + x$ above. Let the signature S_1' be a modification of S_1 above with a callee-save argument: $\mathtt{sig}\ \mathit{add}\colon \mathtt{int}\times\mathtt{int} \to \mathtt{int}, \mathit{square}\colon \mathtt{int}\cdot(\mathtt{int}\to\mathtt{int})\ \mathsf{end}$. Then we can define a fragment of signature $\mathtt{functor}(X\colon S_1') \to S_2$:

$$\begin{aligned} f.\mathit{entry}(x\colon \mathtt{int}) &= X.\mathit{square}.\mathit{entry}(\langle x, x\rangle) \\ X.\mathit{add}.\mathit{ret}(z\colon \mathtt{int}) &= f.\mathit{ret}(z) \\ X.\mathit{square}.\mathit{ret}(z\colon \mathtt{int}\times\mathtt{int}) &= \mathtt{let}\ \langle x, y\rangle = z\ \mathtt{in}\ f.\mathit{add}.\mathit{entry}(y, x) \end{aligned}$$

The entry and exit labels for *square* now take a new callee-save argument of type `int`. The callee-save invariant for $\mathtt{int}\cdot(\mathtt{int}\to\mathtt{int})$ guarantees that the x in the first line and the x in the third line are the same values.

In this example, we have used a callee-save argument only in *square*. It would also be possible to use $(\mathtt{unit} \cup \mathtt{int}) \cdot S_1$ in place of S_1'. Then, both *add* and *square* would have a callee-save argument. The union type is useful to account for different uses of the callee-save argument: in jumps to *add* nothing (`unit`) needs to be stored, while in jumps to *square* an `int` is to be stored.

The reader may find it helpful to think of subexponentials as an abstract form of stack frames. In machine programs, one usually implements function calls using a call stack. Before a call, all local data that is written to the current stack frame. To make a call, one then writes the return address to the stack frame and jumps to the destination. The stack frame can be considered as a callee-save argument. Indeed, we will use the callee-save arguments provided by subexponentials much like stack frames. We will use them to store local data, as we have seen in the example above. We will also use them to store an abstract form of a return address, see the implementation of the contraction rule in Sec. 6.

Finally, signatures allow type quantification. Type quantification can be seen as an alternative to type declarations in signatures. For example, the SML signature $\mathtt{functor}(X : \mathtt{sig}\ \mathtt{type}\ t; \mathtt{val}\ g : \mathtt{int} \to t\ \mathsf{end}) \to \mathtt{sig}\ \mathtt{type}\ t; \mathtt{val}\ f : X.t \to t\ \mathsf{end}$ can be understood as $\forall\alpha.\, \mathtt{functor}(X : \mathtt{sig}\ g : \mathtt{int} \to \alpha\ \mathsf{end}) \to \exists\beta.\, \mathtt{sig}\ f : \alpha \to \beta\ \mathsf{end}$. Such a correspondence can be made precise in general [13], but we do not need the general case here. However, type quantification is bounded is bounded here, the intention being that α in $\forall\alpha \lhd A.\, S_1$ ranges over all low-level types B that satisfy $B \lhd A$.

The fragments of signature $\forall\alpha\lhd A.\, S_1$ and of signature $\exists\alpha\lhd A.\, S_1$ are the fragments of signature $S_1[A/\alpha]$. This definition reflects that polymorphism is not visible at the implementation level. For now, the reader should think of the quantification as ranging over all types B with $B \lhd A$. This means that in the low-level implementation, we can use A in place of the quantified variable, as any possible value there can be encoded into A. The introduction and elimination rules of the quantifiers then amount to encoding and decoding operations, see Sec. 8.

4 DEFUNCTIONALISING PCF INTO MODULES

In this section we outline how defunctionalisation can be defined as a typed closure conversion that targets a module calculus. Minamide et al. [11] have defined typed closure conversion as a type-correct translation to a type-safe programming language that replaces an abstraction $\mathtt{fn}\, x \Rightarrow s$ with free variables $\vec{x}$ by a pair $\langle p, \vec{x}\rangle$, in which p is a pointer that identifies the code of the closed function $\lambda\langle\vec{x}, x\rangle.\ \mathit{closure\text{-}convert}(s)$. Application is implemented by a single procedure $\mathtt{apply}(\langle p, \vec{v}\rangle, w) = (*p)(\vec{v}, x)$, which invokes the closed function identified by the pointer. For defunctionalisation, we want to replace the pointer p with a tag t that identifies the function.

In the Introduction, we have outlined that to capture the essence of realistic defunctionalisation methods requires more than just to replacing the pointer with a tag. In realistic defunctionalisation methods, the tag alone is not enough to identify the function; control-flow information must also be taken into account. There is not a single `apply`-procedure that performs case distinction on all possible tags, but many small `apply`-functions that branch only on the tags that are statically known to be possible at a particular call site. To translate an application, one needs the tag together with the information which of these small `apply`-functions to use.

It seems natural to use a module system to keep track of how the many small `apply`-procedures are defined and how they are linked together. With a suitable module system, defunctionalisation can be expressed as a form typed closure conversion into modules. To explain the general idea, let us first outline it using Standard ML as a module language. The reader should think of SML as being restricted to first-order types and of SML modules (structures) as representing fragments of low-level code, whose interface is specified by the module type (signatures).

The following signatures specify the type of modules that are suitable for representing the values and `apply`-functions that closure conversion should produce. Assume that the type `t` in these signatures is restricted to a low-level type.

```
signature Iℕ = sig        signature IX→Y = sig
  type t = int              type t (* abstract *)
end                         functor T (X: IX) : sig
                              structure T: IY
                              val apply: t * X.t -> T.t
                            end
                          end
```

These signatures can be used to represent values of call-by-value PCF using just first-order data as follows. A PCF value of type X is represented by a module `V` of type $\mathtt{I}_X$ and a value v of type `V.t`. For PCF values of type $\mathbb{N}$, this amounts to just a value of type `int`, so it is just a direct representation of the value. For PCF values of type $X \to Y$, the first-order type `V.t` represents the function value. A typical first-order representation of a function would be the tuple of its free variables. This type `V.t` is abstract, however, so we cannot inspect the value `v` that is supposed to represent the function. But the module `V` contains an `apply` function that we can use to apply the encoded function to any (encoded) argument. How can one apply such a function to a given argument? Suppose we have an encoding of a value of type X, i.e. a module `X`: $\mathtt{I}_X$ and a value w: `X.t`. Let `M` := `V.T(X)`. Then the result value of the function application is encoded by the module `M.T` of signature $\mathtt{I}_Y$ and the value of type `M.T.t` that one gets by evaluating `M.apply`(v, w).

With this representation of values, it is possible to represent the terms of PCF as follows. A typing sequent $x_1 : X_1, \dots, x_n : X_n \vdash t : Y$ maps to a module of type:

```
functor Mt (X1: IX1) ... (Xn: IXn) : sig
  include IY
  val eval: X1.t * ... * Xn.t -> t
end
```

The function `eval` takes the values of the free variables of the term and computes a representation of the term's value (possibly performing side-effects, if one allows constants that have any). The value of the term is then represented by the module `T` and the value of type `T.t` returned by `eval`.

There is more than one way to implement a translation from PCF to modules according to this specification. If we allow ourselves an operator to take the address of a function, then standard typed closure conversion [11] becomes an instance of this scheme. As outlined above, all application functions would be defined by: `fun apply`$((p, \vec{v}), w) = (*p)(\vec{v}, x)$.

To implement a defunctionalising translation, one chooses algebraic data types to represent functions and implements `apply` using case distinction. We outline this for a few examples.

Example 4.1. The term $\vdash \lambda y.\, y + 3 : (\mathbb{N} \to \mathbb{N})$ may be represented by the module:

```
functor M1 = struct
  type t = unit
  functor T(Y: IN) = struct
    structure T: IN = struct type t = int end
    fun apply ((f, y) : t * Y.t) : T.t = y + 3
  end
  fun eval() = ()
end
```

Since the term is a closed value, its value can be represented using the unit type. The function `eval` produces this value. The `apply` function implements function application. It takes the function value and the argument y and returns $y + 3$.

Example 4.2. The term $x : (\mathbb{N} \to \mathbb{N}) \vdash \lambda y.\, x\ (y + 1) : (\mathbb{N} \to \mathbb{N})$ may be translated to:[1]

```
functor M2 (X: IN→N) = struct
  type t = X.t
  functor T(Y: IN) = struct
    structure T: IN = struct type t = int end
    fun apply ((f, y) : t * Y.t) : T.t =
      X.T(Y).apply(f, y+1)
  end
  fun eval(x : X.t) : t = x
end
```

The function term $\lambda y.\, x\ (y + 1)$ has a free variable x, whose value fully identifies the function value, so we use it as the function code.

Example 4.3. The term $x : (\mathbb{N} \to \mathbb{N}) \vdash$ `ifz` $(x\ 2)$ `then` $\lambda y.\, y + 1$ `else` $\lambda y.\, x\ (y + 3) : (\mathbb{N} \to \mathbb{N})$ may be translated to:

[1] In the program we have written `X.T(Y).apply(f, y+1)` for readability. In Standard ML one would need to define `Z=X.T(Y)` and can then use `Z.apply`.

```
functor M3 (X: IN→N) = struct
  datatype t = Left of unit | Right of X.t
  functor T(Y: IN) = struct
    structure T: IN = struct type t = int end
    fun apply ((f, y) : t * Y.t) : T.t =
      case f of Left() => y + 1
              | Right(x) => X.T(Y).apply(x, y+3)
  end
  fun eval(x : X.t) : t =
    if X.T(struct type t=int end).apply(x, 2) = 0
    then Left() else Right(x)
end
```

The term could evaluate to two different functions, so we add a tag using an algebraic data type, so that we can distinguish both cases.

It is not hard to define a general translation that uses the tuple of free variables to represent function abstractions and that adds tags in case distinctions. In the next section, we shall do this, but with low-level program fragments in place of Standard ML structures.

5 ANNOTATED PCF

We now define a defunctionalising translation from PCF to low-level program fragments, considered as modules. It is compositional in the sense that the low-level translation of a term is obtained by composing the translations of its parts. Each translated part has a well-defined signature and correctness can be shown by verifying the parts independently.

We replace the SML signatures $\mathtt{I}_X$ from above with signatures for the low-level language. These signatures make visible more low-level information than the SML signatures, such as callee-save values. In order to record such additional information, we use a PCF type system with added annotations. One should think of the annotations as capturing the low-level implementation details that will be visible in the interfaces of compiled low-level code. They are not intended for the programmer and will be inferred automatically.

Annotated PCF types are defined by the grammar

$$\mathcal{X}, \mathcal{Y} ::= A \cdot \mathbb{N} \mid A \cdot (\mathcal{X} \xrightarrow{C}_B \mathcal{Y}) \enspace ,$$

in which A, B and C range over low-level types. We write just X for $\mathtt{unit} \cdot X$. The PCF typing judgements are also annotated and now have the form $x_1 : \mathcal{X}_1, \dots, x_n : \mathcal{X}_n \vdash_D t : \mathcal{Y}$, where t is a normal PCF term, where the type $\mathcal{Y}$ and the $\mathcal{X}_i$ are annotated PCF types, and where D is a low-level type.

The SML-signature $\mathtt{I}_\mathcal{X}$ from above is now replaced by the signature $\mathcal{I}[\![\mathcal{X}]\!] := \exists \alpha \lhd \mathcal{B}[\![\mathcal{X}]\!].\, \mathcal{I}[\![\mathcal{X}]\!]_\alpha$, where $\mathcal{I}[\![\mathcal{X}]\!]_\alpha$ is defined by

$$\mathcal{I}[\![A \cdot \mathbb{N}]\!]_\alpha = A \cdot (\mathbf{sig\ end})$$

$$\begin{aligned}\mathcal{I}[\![A \cdot (\mathcal{X} \xrightarrow{C}_B \mathcal{Y})]\!]_\alpha = \forall \beta \lhd \mathcal{B}[\![\mathcal{X}]\!].\, A \cdot \big(&\mathtt{functor}(X : \mathcal{I}[\![\mathcal{X}]\!]_\beta) \to \\ &\exists \gamma \lhd \mathcal{B}[\![\mathcal{Y}]\!].\ \mathtt{sig} \\ &\quad T : \mathcal{I}[\![\mathcal{Y}]\!]_\gamma, \\ &\quad apply : B \cdot (\alpha \times \mathcal{X}.\mathtt{t}_\beta \to \mathcal{Y}.\mathtt{t}_\gamma) \\ &\mathtt{end}\big) \enspace ,\end{aligned}$$

where the bounds are $\mathcal{B}[\![B \cdot \mathbb{N}]\!] = \mathtt{unit}$ and $\mathcal{B}[\![A \cdot (\mathcal{X} \xrightarrow{C}_B \mathcal{Y})]\!] = C$, and where the type $\mathcal{X}.t_\alpha$ denotes `int` if $\mathcal{X}$ is $\mathbb{N}$ and α otherwise.

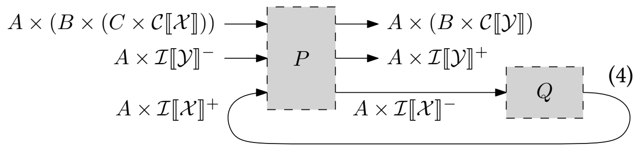
This definition of $\mathcal{I}[\![\mathcal{X}]\!]$ can be obtained from $\mathtt{I}_\mathcal{X}$ by first removing all type declarations. Transparent declarations `type t = int` can be substituted out. Abstract type declarations are replaced by type variables and are quantified appropriately. This translation can be seen as an example of F-ing modules [13]. In addition to these changes, $\mathcal{I}[\![\mathcal{X}]\!]$ contains low-level information in the form of callee-save arguments like $A \cdot -$, as well as bounds on quantification.

The definition of $\mathcal{I}[\![A \cdot (\mathcal{X} \xrightarrow{C}_B \mathcal{Y})]\!]$ shows that the annotations A and B represent subexponentials, i.e. they are the types of callee-save arguments. Their particular choice will be heavily dependent on implementation details, see Sec. 6. Note here that the scope of A comprises the whole type, which means that all parts of the interface contain a callee-save argument of this type. The subexponential $B \cdot -$ appears only in *apply* and allows more precise types (it could be removed at the expense of potentially making the type A larger).

The annotation C in $A \cdot (\mathcal{X} \xrightarrow{C}_B \mathcal{Y})$ is the low-level type that encodes functions of this type. To understand this recall first from Sec. 3.2 that in the definition of low-level interfaces quantified type variables are replaced by their upper bounds. This means that in the low-level interfaces, the argument of type $\alpha \times \mathcal{X}.\mathtt{t}_\beta$ to *apply* will become $C \times \mathcal{X}.\mathtt{t}_{\mathcal{B}[\![\mathcal{X}]\!]}$. The return type will become $\mathcal{Y}.\mathtt{t}_{\mathcal{B}[\![\mathcal{Y}]\!]}$.

We write short $\mathcal{C}[\![\mathcal{X}]\!]$ for $\mathcal{X}.\mathtt{t}_{\mathcal{B}[\![\mathcal{X}]\!]}$, that is $\mathcal{C}[\![B \cdot \mathbb{N}]\!] = \mathtt{int}$ and $\mathcal{C}[\![A \cdot (\mathcal{X} \xrightarrow{C}_B \mathcal{Y})]\!] = C$. One should think of $\mathcal{C}[\![\mathcal{X}]\!]$ as the low-level *code type* that is being used to encode the PCF values of type $\mathcal{X}$. In particular, the type over a function arrow is the type that is used for low-level closure encoding.

With this notation, we can spell out the low-level interface of the program fragments of signature $\mathcal{I}[\![\mathcal{X}]\!]$. A fragment of interface $\mathcal{I}[\![\mathbb{N}]\!]$ is a program with an empty interface. This corresponds to the fact that no code is necessary to implement $\mathtt{I}_\mathbb{N}$. A fragment P of signature $\mathcal{I}[\![A \cdot (\mathcal{X} \xrightarrow{C}_B \mathcal{Y})]\!]$ has the interface shown in the figure below. The three entry and exit points of P correspond to *apply*, the module T and the functor argument X (from top to bottom). Notice that A and B only have the role of callee-save arguments. In essence, *apply* takes input $C \times \mathcal{C}[\![\mathcal{X}]\!]$ (code of function value and argument) and returns $\mathcal{C}[\![\mathcal{Y}]\!]$.


$A \times (B \times (C \times \mathcal{C}[\![\mathcal{X}]\!]))$
$A \times \mathcal{I}[\![\mathcal{Y}]\!]^-$
$A \times \mathcal{I}[\![\mathcal{X}]\!]^+$
P
$A \times (B \times \mathcal{C}[\![\mathcal{Y}]\!])$
$A \times \mathcal{I}[\![\mathcal{Y}]\!]^+$
$A \times \mathcal{I}[\![\mathcal{X}]\!]^-$
Q
(4)

The signature $\mathtt{M}_t$ from the previous section adapts much like $\mathtt{I}_\mathcal{X}$. A judgement $x_1 \colon \mathcal{X}_1, \dots, x_n \colon \mathcal{X}_n \vdash_A t \colon \mathcal{Y}$ will be translated to a low-level module of signature:

$$\begin{aligned}
&\forall \alpha_1 \lhd \mathcal{B}[\![\mathcal{X}_1]\!]. \dots \forall \alpha_n \lhd \mathcal{B}[\![\mathcal{X}_n]\!].\\
&\mathtt{functor}(x_1 \colon \mathcal{I}[\![\mathcal{X}_1]\!]_{\alpha_1}) \dots (x_n \colon \mathcal{I}[\![\mathcal{X}_n]\!]_{\alpha_n}) \to \exists \gamma \lhd \mathcal{B}[\![\mathcal{Y}]\!].\, \mathtt{sig}\\
&\quad \mathtt{eval} : A \cdot (\mathcal{X}_1.\mathtt{t}_{\alpha_n} \times \dots \times \mathcal{X}_1.\mathtt{t}_{\alpha_n} \to \mathcal{Y}.\mathtt{t}_\gamma), \qquad (5)\\
&\quad T : \mathcal{I}[\![\mathcal{Y}]\!]_\gamma\\
&\mathtt{end}
\end{aligned}$$

We have now defined annotations to PCF types that allow us to translate PCF types to low-level signatures directly. To be able to implement these signatures, we need to instantiate the annotations appropriately. The annotated PCF type systems records the constraints on the annotations that will be needed in the implementation. The typing rules appear in Fig. 1. We use the following notation. This definition of $\mathcal{C}[\![\mathcal{X}]\!]$ extends to contexts: $\mathcal{C}[\![\Gamma]\!]$ is defined by $\mathcal{C}[\![\text{empty}]\!] = \mathtt{unit}$, $\mathcal{C}[\![x \colon \mathcal{X}]\!] = \mathcal{C}[\![\mathcal{X}]\!]$ and $\mathcal{C}[\![\Delta, x \colon \mathcal{X}]\!] = \mathcal{C}[\![\Delta]\!] \times \mathcal{C}[\![\mathcal{X}]\!]$. A low-level value of type $\mathcal{C}[\![\Gamma]\!]$ is a tuple of the low-level values that encode the values of the variables in Γ. We further write $A \cdot \Gamma$ for the context one obtains by replacing each declaration $x \colon B \cdot X$ in Γ with $x \colon (A \times B) \cdot X$.

The subtyping rule for $\mathbb{N}$ allows us to replace any $E \cdot \mathbb{N}$ by $\mathtt{unit} \cdot \mathbb{N}$. This is justified, as $\mathcal{I}[\![\mathbb{N}]\!]$ is an empty signature. We will therefore just write $\mathbb{N}$ for $A \cdot \mathbb{N}$.

Let us sketch the meaning of the annotations of rule (APP). To evaluate $s\ t$ in call-by-value, one first evaluates s, then t and then invokes the application code with the obtained function and argument values. The conclusion of (APP) states that the *eval*-code offers a callee-save value of type U (cf. (5)). When we jump to the *eval*-code for s, we must remember this value together with the values of the variables that we need to evaluate t. This is why we require a callee-save argument of type $U \times \mathcal{C}[\![\Delta]\!]$ in the left premise of (APP). When we then jump to the *eval*-code for the argument t, we need to remember the value of type U and the already computed function value of type C, which explains the annotation $U \times C$ for t. When we have both function and argument value, we invoke the code for function application. But while doing so, we still need to remember the value of type U that we need to return unchanged at the end. This explains the appearance of U in the type of s: it is a callee-save argument for the *apply*-code.

One part of the type system that deserves comment are the rules for joining interfaces. The joining judgement is used in rule (IF) to give both branches of the case distinction the same type. Since the type system has a subtyping statement, the reader may expect this to be just a common upper bound for the types of the two branches. Here, however, joining may have actual computational content. Take, for example the case where $\mathcal{X}_1$ and $\mathcal{X}_2$ are both $\mathbb{N} \xrightarrow{\mathtt{unit}}_B \mathbb{N}$. This means that both branches return a function that is encoded without defunctionalisation tag. In this case $\mathcal{X}$ must be $\mathbb{N} \xrightarrow{\mathtt{unit+unit}}_B \mathbb{N}$, i.e. the result has a tag of a single bit. This corresponds to what we have done in Ex. 4.3. So joining amounts to tagging in this case. It is explained in more detail in Sec. 6.

5.1 Typing Examples

Let t be a term of the form $\mathtt{if0}\ \dots\ \mathtt{then}\ f\ x_1\ \mathtt{else}\ f\ x_2$. One derives $f \colon \mathtt{unit} \cdot (\mathcal{X} \xrightarrow{C}_A \mathbb{N}),\ x \colon \mathcal{X} \vdash_A f\ x \colon \mathbb{N}$ using (APP) and (VAR). From two instances of this, one derives using (IF) and (C) the judgement

$$f \colon (\mathtt{unit} + \mathtt{unit}) \cdot (\mathcal{X} \xrightarrow{C}_A \mathbb{N}),\ x_1 \colon \mathcal{X},\ x_2 \colon \mathcal{X} \vdash_A t \colon \mathbb{N}\ .$$

The annotation $\mathtt{unit} + \mathtt{unit}$ says the that the $\mathtt{apply}$-code in the module for f has a callee-save argument of this type. This argument will be used as an abstract form of the return address. The term t contains two applications of f, so a jump to the $\mathtt{apply}$-code for f in the compiled code may have two different origins. With this jump we pass along a callee-save value that identifies which of the two applications we are currently evaluating. This is needed in general so that we know where to continue with the computation when the return label of $\mathtt{apply}$ is reached.

In the example, x_1 and x_2 have the same type, which is somewhat restrictive. If $\mathcal{X}$ is a function type $B \cdot (\mathbb{N} \xrightarrow{C}_E \mathbb{N})$, then this will mean that both variables denote functions represented with exactly the same closure representation, for example. This constraint can be

$$\textsc{var}\ \frac{}{x\colon \mathcal{X} \vdash_A x\colon \mathcal{X}} \qquad \textsc{subl}\ \frac{\Gamma,\ x\colon \mathcal{X} \vdash_A t\colon \mathcal{Z}}{\Gamma,\ x\colon \mathcal{Y} \vdash_A t\colon \mathcal{Z}}\ \mathcal{Y} \le \mathcal{X} \qquad \textsc{subr}\ \frac{\Gamma \vdash_B t\colon \mathcal{X}}{\Gamma \vdash_A t\colon \mathcal{Y}}\ \mathcal{X} \le \mathcal{Y}, A \lhd B \qquad \textsc{w}\ \frac{\Gamma \vdash_A t\colon \mathcal{Y}}{\Gamma,\ x\colon \mathcal{X} \vdash_A t\colon \mathcal{Y}}$$

$$\textsc{e}\ \frac{\Gamma,\ x\colon \mathcal{X},\ y\colon \mathcal{Y},\ \Delta \vdash_A t\colon \mathcal{Z}}{\Gamma,\ y\colon \mathcal{Y},\ x\colon \mathcal{X},\ \Delta \vdash_A t\colon \mathcal{Z}} \qquad \textsc{c}\ \frac{\Gamma,\ x\colon B\cdot X,\ y\colon C\cdot X \vdash_A t\colon \mathcal{Y}}{\Gamma,\ z\colon (B+C)\cdot X \vdash_A t[z/x, z/y]\colon \mathcal{Y}} \qquad \textsc{zero}\ \frac{}{\vdash_A \mathtt{zero}\colon B\cdot\mathbb{N}} \qquad \textsc{succ}\ \frac{\vdash_A t\colon B\cdot\mathbb{N}}{\vdash_A \mathtt{succ}(t)\colon B\cdot\mathbb{N}}$$

$$\textsc{pred}\ \frac{\vdash_A t\colon B\cdot\mathbb{N}}{\vdash_A \mathtt{pred}(t)\colon B\cdot\mathbb{N}} \qquad \textsc{if}\ \frac{\Gamma \vdash_{A\times C\llbracket\Delta_1\rrbracket\times C\llbracket\Delta_2\rrbracket} s\colon \mathbb{N} \qquad \Delta_i \vdash_A t_i\colon \mathcal{X}_i \qquad \mathcal{X}_1 \curlyvee \mathcal{X}_2 = \mathcal{X}}{\Gamma, \Delta_1, \Delta_2 \vdash_A \mathtt{ifz}\ s\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2\colon \mathcal{X}} \qquad \textsc{fn}\ \frac{\Gamma,\ x\colon \mathcal{X} \vdash_A t\colon \mathcal{Y}}{E\cdot\Gamma \vdash_F \mathtt{fn}\,x \Rightarrow t\colon E\cdot(\mathcal{X} \xrightarrow{C\llbracket\Gamma\rrbracket}_A \mathcal{Y})}$$

$$\textsc{app}\ \frac{\Gamma \vdash_{U\times C\llbracket\Delta\rrbracket} s\colon \mathcal{X} \xrightarrow{C}_U \mathcal{Y} \qquad \Delta \vdash_{U\times C} t\colon \mathcal{X}}{\Gamma,\ \Delta \vdash_U s\ t\colon \mathcal{Y}} \qquad \textsc{fix}\ \frac{\Gamma,\ f\colon G\cdot\big(\mathcal{X} \xrightarrow{C\llbracket\Gamma\rrbracket}_A \mathcal{Y}\big),\ x\colon \mathcal{X} \vdash_A t\colon \mathcal{Y}}{H\cdot\Gamma \vdash_F \mathtt{fix}\,f\ x \Rightarrow t\colon E\cdot(\mathcal{X} \xrightarrow{C\llbracket\Gamma\rrbracket}_A \mathcal{Y})}\ (E + (H\times G)) \lhd H$$

Subtyping

$$\frac{}{A\cdot\mathbb{N} \le B\cdot\mathbb{N}} \qquad \frac{A_2 \lhd A_1 \qquad \mathcal{X}_2 \le \mathcal{X}_1 \qquad \mathcal{Y}_1 \le \mathcal{Y}_2 \qquad C_2 \lhd C_1 \qquad D_1 \lhd D_2}{A_1\cdot(\mathcal{X}_1 \xrightarrow{D_1}_{C_1} \mathcal{Y}_1) \le A_2\cdot(\mathcal{X}_2 \xrightarrow{D_2}_{C_2} \mathcal{Y}_2)}$$

Joining Interfaces

$$\frac{}{A\cdot\mathbb{N} \curlyvee A\cdot\mathbb{N} = A\cdot\mathbb{N}} \qquad \frac{\mathcal{Y}_1 \curlyvee \mathcal{Y}_2 = \mathcal{Y} \qquad \mathcal{X}_i = B_i\cdot X \qquad \mathcal{X} = (B_1+B_2)\cdot X}{A\cdot(\mathcal{X}_1 \xrightarrow{D_1}_C \mathcal{Y}_1) \curlyvee A\cdot(\mathcal{X}_2 \xrightarrow{D_2}_C \mathcal{Y}_2) = A\cdot(\mathcal{X} \xrightarrow{D_1+D_2}_C \mathcal{Y})}$$

Figure 1: Annotated PCF Typing

weakened using subtyping. To derive the same judgement with $x_1\colon \mathcal{X}_1$ and $x_2\colon \mathcal{X}_2$, it suffices to take $\mathcal{X}_i$ to be $B_i\cdot(\mathbb{N} \xrightarrow{C_i}_{E_i} \mathbb{N})$, where E_i, C_i and B_i are such that $B \lhd B_i$ and $E \lhd E_i$ and $C_i \lhd C$ is true for all $i \in \{1,2\}$. Informally, this means that C is large enough to encode either the value denoted by x_1 or that denoted by x_2. We may choose $C_1 \cup C_2$, for example. The other bounds say that both x_1 and x_2 provide at least as much space for callee-save values as f requires of its argument.

The example illustrates that the type system captures some control flow information. When f is invoked, we have two pieces of data, one of type `unit + unit`, corresponding to which copy the call came from, and one of type $C_1 \cup C_2$ for the function value. The example also shows that the defunctionalisation method presented here use a little less space than ones from the literature [3]. Even though two functions can flow to the variable f, the closure can be stored without tag. The information that would normally be stored in the tag is already present in the return address (represented abstractly here as a value of type `unit + unit`).

Another source of tagging is case distinction of higher type. Consider a term s of the form `if0` ... `then` f `else fn` $x \Rightarrow$ `plus` $x\ y$, where `plus` is an externally defined addition function, that defines a function by case distinction.

$$f\colon \mathbb{N} \xrightarrow{C}_A \mathbb{N},\ y\colon \mathbb{N} \vdash_A s\colon \mathbb{N} \xrightarrow{C+\mathtt{int}}_A \mathbb{N}$$

The value of f is encoded using type C. The function `fn` $x \Rightarrow$ `plus` $x\ y$ is encoded by the tuple of its free variables, i.e. $C\llbracket y\colon \mathbb{N}\rrbracket =$ `int`. The two possible results of the case distinction are represented using the sum type $C +$ `int`.

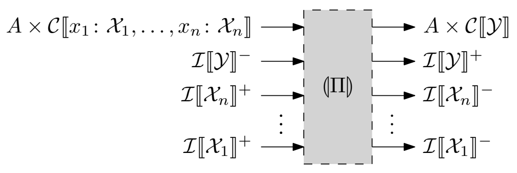
For a slightly larger example, consider the following term `step`:

```
fn f ⇒ fn x ⇒ let x₁ = pred(x) in let x₂ = pred(pred(x)) in
                    ifz x₁ then 1 else ifz x₂ then 1 else plus (f x₁) (f x₂)
```

It can be used to define a Fibonacci function by `fix` $f\ x \Rightarrow$ `step` $f\ x$. We have `step`: $A\cdot\big((B+B)\cdot(\mathbb{N} \xrightarrow{E}_S \mathbb{N}) \xrightarrow{\mathtt{unit}}_C B\cdot(\mathbb{N} \xrightarrow{E\times E}_D \mathbb{N})\big)$, where the annotation S is $(D\times \mathtt{int}\times E) \cup (D\times\mathtt{int})$. The function f is encoded using type E. The annotation S reflects that f is applied twice. In the first call, the callee-save argument contains x_2 and f, corresponding to $\mathtt{int}\times E$, while the second time it contains the `int`-value $f\ x_1$.

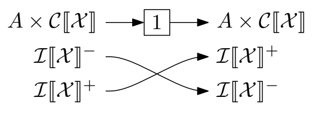
6 TRANSLATING ANNOTATED PCF

We next outline a direct translation from annotated PCF to low-level fragments. It goes by induction on typing derivations and maps a derivation Π of $x_1\colon \mathcal{X}_1, \dots, x_n\colon \mathcal{X}_n \vdash_A t\colon \mathcal{Y}$ to a low-level program $(\!|\Pi|\!)$ of the signature in (5). This means that $(\!|\Pi|\!)$ is a fragment of the following form.


$A \times \mathcal{C}\llbracket x_1\colon \mathcal{X}_1, \dots, x_n\colon \mathcal{X}_n\rrbracket$
$\mathcal{I}\llbracket\mathcal{Y}\rrbracket^-$
$\mathcal{I}\llbracket\mathcal{X}_n\rrbracket^+$
$\mathcal{I}\llbracket\mathcal{X}_1\rrbracket^+$
$(\!|\Pi|\!)$
$A \times \mathcal{C}\llbracket\mathcal{Y}\rrbracket$
$\mathcal{I}\llbracket\mathcal{Y}\rrbracket^+$
$\mathcal{I}\llbracket\mathcal{X}_n\rrbracket^-$
$\mathcal{I}\llbracket\mathcal{X}_1\rrbracket^-$

The topmost entry- and exit-point belong to the *eval*-function in the signature (5). The entry- and exit-points below come from the module T. The other ports are there to connect functor arguments.

The translation from annotated PCF to such low-level program fragments goes by induction on the typing derivation. We spell out representative cases, beginning with (VAR), where $(\!|\Pi|\!)$ is:


$A \times \mathcal{C}\llbracket\mathcal{X}\rrbracket$
1
$A \times \mathcal{C}\llbracket\mathcal{X}\rrbracket$
$\mathcal{I}\llbracket\mathcal{X}\rrbracket^-$
$\mathcal{I}\llbracket\mathcal{X}\rrbracket^+$
$\mathcal{I}\llbracket\mathcal{X}\rrbracket^+$
$\mathcal{I}\llbracket\mathcal{X}\rrbracket^-$

The block named 1 is the identity (note $\mathcal{C}\llbracket x\colon \mathcal{X}\rrbracket = \mathcal{C}\llbracket\mathcal{X}\rrbracket$).

The translation of rule (FN) is such that the closure is represented by the tuple of the values of the variables in Γ. It is given by:

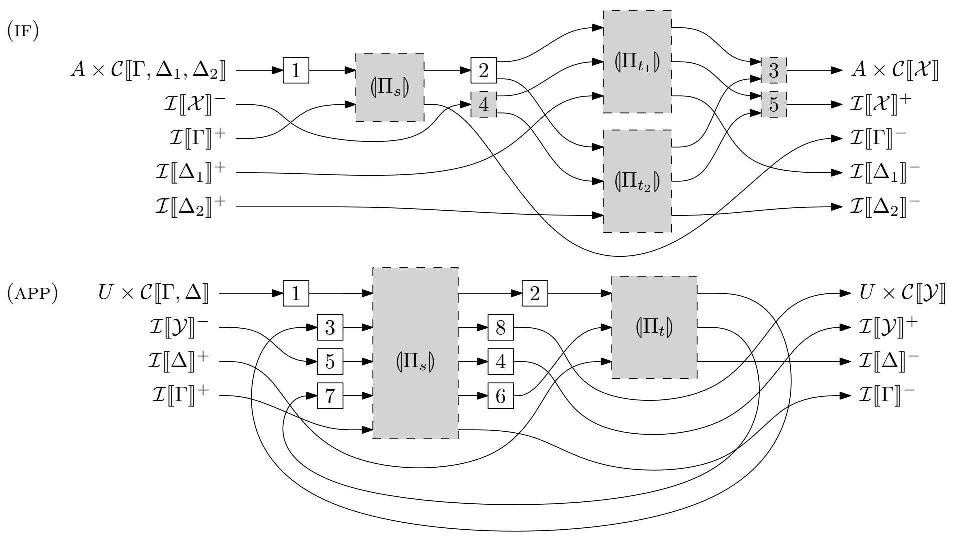
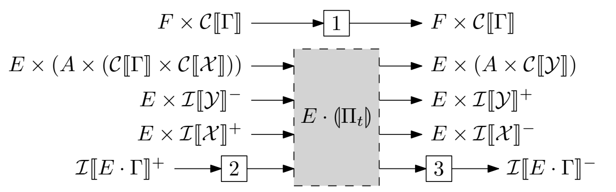

(IF)
$A \times \mathcal{C}[\![\Gamma, \Delta_1, \Delta_2]\!]$
$\mathcal{I}[\![\mathcal{X}]\!]^-$
$\mathcal{I}[\![\Gamma]\!]^+$
$\mathcal{I}[\![\Delta_1]\!]^+$
$\mathcal{I}[\![\Delta_2]\!]^+$
1
$(\!|\Pi_s|\!)$
2
4
$(\!|\Pi_{t_1}|\!)$
$(\!|\Pi_{t_2}|\!)$
3
5
$A \times \mathcal{C}[\![\mathcal{X}]\!]$
$\mathcal{I}[\![\mathcal{X}]\!]^+$
$\mathcal{I}[\![\Gamma]\!]^-$
$\mathcal{I}[\![\Delta_1]\!]^-$
$\mathcal{I}[\![\Delta_2]\!]^-$
(APP)
$U \times \mathcal{C}[\![\Gamma, \Delta]\!]$
$\mathcal{I}[\![\mathcal{Y}]\!]^-$
$\mathcal{I}[\![\Delta]\!]^+$
$\mathcal{I}[\![\Gamma]\!]^+$
1
3
5
7
$(\!|\Pi_s|\!)$
2
8
4
6
$(\!|\Pi_t|\!)$
$U \times \mathcal{C}[\![\mathcal{Y}]\!]$
$\mathcal{I}[\![\mathcal{Y}]\!]^+$
$\mathcal{I}[\![\Delta]\!]^-$
$\mathcal{I}[\![\Gamma]\!]^-$

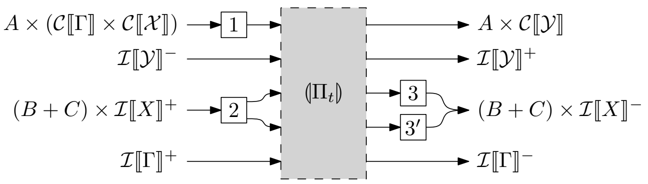
Figure 2: Translation of Representative Rules


$F \times \mathcal{C}[\![\Gamma]\!]$
1
$F \times \mathcal{C}[\![\Gamma]\!]$
$E \times (A \times (\mathcal{C}[\![\Gamma]\!] \times \mathcal{C}[\![\mathcal{X}]\!]))$
$E \times \mathcal{I}[\![\mathcal{Y}]\!]^-$
$E \times \mathcal{I}[\![\mathcal{X}]\!]^+$
$\mathcal{I}[\![E \cdot \Gamma]\!]^+$
2
$E \cdot (\!|\Pi_t|\!)$
$E \times (A \times \mathcal{C}[\![\mathcal{Y}]\!])$
$E \times \mathcal{I}[\![\mathcal{Y}]\!]^+$
$E \times \mathcal{I}[\![\mathcal{X}]\!]^-$
3
$\mathcal{I}[\![E \cdot \Gamma]\!]^-$

Thus, block 1 for *eval* is the identity. It returns the tuple of variables in Γ as the term's value. The next three entry and exit wires are the three components of $\mathcal{I}[\![E \cdot (\mathcal{X} \xrightarrow{C[\![\Gamma]\!]}_A \mathcal{Y})]\!]$, as spelled out in (4). The entry point with type $E \times (A \times (\mathcal{C}[\![\Gamma]\!] \times \mathcal{C}[\![\mathcal{X}]\!]))$ is the entry point for `apply`. In addition to the two callee-save arguments of type E and A, it gets the function and argument values as input. This entry point is defined to jump to the `eval`-code for t. The result will be returned at the exit of type $E \times (A \times \mathcal{C}[\![\mathcal{Y}]\!])$, which is where `apply` passes its return value. Thus, function application for $\texttt{fn}\, x \Rightarrow t$ is defined by jumping to the evaluation code for t. Finally, the blocks $2 : \langle\langle e, x\rangle, y\rangle \mapsto \langle e, \langle x, y\rangle\rangle$ and $3 : \langle e, \langle x, y\rangle\rangle \mapsto \langle\langle e, x\rangle, y\rangle$ are just there to reorganise the callee-save arguments in the context (Rather than defining these blocks explicitly, we state just how they map a value at the entry label to a value at the exit label. The labels are clear from the control-flow graph.)

The translation of rule (APP) in Fig. 2 implements the description of the typing rule given above. To evaluate the application, one first evaluates s, which is what block $1 : \langle u, c\rangle \mapsto \langle\langle u, \pi_\Delta(c)\rangle, \pi_\Gamma(c)\rangle$ does. In its definition we write π_Γ and π_Δ for the evident projections from $\mathcal{C}[\![\Gamma, \Delta]\!]$ to $\mathcal{C}[\![\Gamma]\!]$ and $\mathcal{C}[\![\Delta]\!]$ respectively. Block 1 thus puts the values in Δ in a callee-save argument for later use and jumps to the `eval`-code for s. Block $2 : \langle\langle u, d\rangle, f\rangle \mapsto \langle\langle u, f\rangle, d\rangle$ takes the result of the evaluation of s, i.e. the function value f. It retrieves the values in Δ from the callee-save argument (by invariant, d must be $\pi_\Delta(c)$), puts the function value f in a callee-save argument for later use and jumps to the `eval`-code for t. This code passes its result x to block $3 : \langle\langle u, f\rangle, x\rangle \mapsto \langle(), \langle u, \langle f, x\rangle\rangle\rangle$, which now constructs the pair $\langle f, x\rangle$ of function value and argument and jumps with it to the `apply`-code for s. The result is passed to block 8, which returns it. This and the remaining blocks are defined by $4, 6, 8 : \langle(), x\rangle \mapsto x$ and $5, 7 : y \mapsto \langle(), y\rangle$. They connect the code provided by t to s appropriately for application, as outlined above.

Next we show the contraction rule (C), which records control-flow information by tagging callee-save arguments. In the case of (C), we define $(\!|\Pi_t|\!)$ to be the following program fragment.


$A \times (\mathcal{C}[\![\Gamma]\!] \times \mathcal{C}[\![\mathcal{X}]\!])$
1
$\mathcal{I}[\![\mathcal{Y}]\!]^-$
$(B + C) \times \mathcal{I}[\![X]\!]^+$
2
$\mathcal{I}[\![\Gamma]\!]^+$
$(\!|\Pi_t|\!)$
3
3′
$A \times \mathcal{C}[\![\mathcal{Y}]\!]$
$\mathcal{I}[\![\mathcal{Y}]\!]^+$
$(B + C) \times \mathcal{I}[\![X]\!]^-$
$\mathcal{I}[\![\Gamma]\!]^-$

Block $1 : \langle a, \langle c, z\rangle\rangle \mapsto \langle a, \langle\langle c, z\rangle, z\rangle\rangle$ is the entry point for `eval`, which takes as input a callee-save value a and the values of the variables in the context. It duplicates the value of z and invokes the `eval`-code of t. The effect is that t is evaluated with the value z for both the variables x and y. Should the program for t jump to the interface entry point belonging to either the variable x or y (that is, the code for application if they are functions), then this jump is forwarded to z by blocks $3 : \langle b, v\rangle \mapsto \langle \texttt{inl}(b), v\rangle$ and $3' : \langle c, v\rangle \mapsto \langle \texttt{inr}(c), v\rangle$. In the callee-save argument, these blocks add a tag to record whether the jump came from x or y. When the code for z returns by a jump to its exit label, block 2: $\langle e, v\rangle \mapsto$ $\texttt{case}\ e\ \texttt{of}\ \texttt{inl}(b) \Rightarrow \mathit{exit.top}(\langle b, v\rangle);\ \texttt{inr}(c) \Rightarrow \mathit{exit.bottom}(\langle c, v\rangle)$ performs a case distinction over the callee-saved value and branches to the return point for either x or y.

The translation of contraction shows how return addresses are encoded as tags in callee-save arguments. This tagging relates directly to defunctionalisation [15]. In practice, one may use an n-ary contraction rule, so that one does not need to go through a binary decision tree and can branch directly. It is also possible to use actual addresses and direct jumps in a low-level language with pointers.

The implementation of (IF) in Fig. 2 is such that the *eval*-code first evaluates s and then, depending on the result, jumps either to the *eval*-code for t_1 or t_2. This is implemented by defining

1: $\langle a, c\rangle \mapsto \langle\langle a, \pi_{\Delta_1}(c), \pi_{\Delta_2}(c)\rangle, \pi_\Gamma(c)\rangle$ and 2 : $\langle\langle a, c_1, c_2\rangle, n\rangle \mapsto$ $\mathtt{let}\, b = \mathsf{eq}(n, 0)\, \mathtt{in}\, \mathtt{case}\, b\, \mathtt{of}\, \mathtt{inl}(y) \Rightarrow \mathit{exit.top}(\langle a, c_1\rangle);\ \mathtt{inr}(y) \Rightarrow$ $\mathit{exit.bot}(\langle a, c_2\rangle)$.

Next, the module T with entry labels $\mathcal{I}[\![\mathcal{X}]\!]^-$ and exit labels $\mathcal{I}[\![\mathcal{X}]\!]^+$ must be provided. Depending on whether t_1 or t_2 was evaluated, this fragment should behave like the T provided by t_1 or that by t_2. To implement this, fragment 3 tags the result value of the if-then-else with just enough information so that we can identify from it which T to use. Fragments 4 and 5 then make use of this information to branch to the right code. The rules for joining interfaces in Fig. 1 formalise what information is encoded in fragment 3. For example, if $t_1 : \mathbb{N} \xrightarrow{C_1}_A \mathbb{N}$ and $t_2 : \mathbb{N} \xrightarrow{C_2}_A \mathbb{N}$, then $\mathtt{ifz}\ \ldots\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2$ will have type $\mathbb{N} \xrightarrow{C_1+C_2}_A \mathbb{N}$. In this case, fragment 3 maps $\langle a, c_1\rangle$ on the topmost input to $\langle a, \mathtt{inl}(c_1)\rangle$ on its output and $\langle a, c_2\rangle$ on the other input to $\langle a, \mathtt{inl}(c_2)\rangle$. If $\mathcal{X}$ is $\mathbb{N} \xrightarrow{C_1+C_2}_A \mathbb{N}$, then fragment 4 only has one non-vacuous entry point of type $A \cdot ((C_1 + C_2) \times \mathtt{int})$ (for *apply*). It is defined by $\langle a, \langle c, n\rangle\rangle \mapsto \mathtt{case}\, c\, \mathtt{of}\, \mathtt{inl}(c_1) \Rightarrow \mathit{exit.top}(\langle a, \langle c_1, n\rangle\rangle);\ \mathtt{inr}(c_2) \Rightarrow$ $\mathit{exit.bottom}(\langle a, \langle c_2, n\rangle\rangle)$. The fragment 5 is the identity in this case; it just passes on the returned value of the function (an integer).

We state a simple correctness result, which follows from Theorem 8.1 in Sec. 8.

Corollary 6.1 (Correctness). *Suppose* Π *derives* $\vdash_A t : B \cdot \mathbb{N}$. *Then* t *reduces to a value* v *in a standard call-by-value operational semantics if and only if we have* $(\!|\Pi|\!) : \langle a, ()\rangle \mapsto \langle a, v\rangle$ *for any closed low-level value* $a : A$.

7 ANNOTATION INFERENCE

We have described a translation from annotated PCF to the low-level language. To use this translation for the compilation of PCF it remains to find annotations for any given PCF program.

A type inference algorithm can be developed using a standard approach. One can first eliminate the subtyping rules by closing all other rules under subtyping. The resulting rules are all ordinary PCF rules with type annotations. The task then reduces to decorating ordinary PCF typing annotations, obtained by standard type inference, with annotations. We choose a fresh type variable for each annotation and solve the side-conditions. All side-conditions have the form $A \lhd \alpha$, i.e. with a type variable as upper bound, and can be solved for one type variable at a time [4, 16]. To solve for β, one gathers all constraints with β as an upper bound, say $B_1 \lhd \beta$, $\ldots, B_n \lhd \beta$. If β is not free in the B_i, then $\beta := B_1 \cup \cdots \cup B_n$ is a solution. Otherwise $\beta := \mu\beta.\, B_1 \cup \cdots \cup B_n$ solves the constraints.

Theorem 7.1. *If* $\Gamma \vdash t : X$ *in* PCF, *then there exist a derivable sequent* $\Gamma_1 \vdash t_1 : \mathcal{X}_1$ *in annotated* PCF, *such that* $\Gamma = |\Gamma_1|$, $t = |t_1|$ *and* $X = |\mathcal{X}_1|$, *where* $|-|$ *denotes the removal of all type annotations.*

Note that the algorithm implements just one of many ways of solving constraints. Different solutions correspond to different choices of how to manage low-level details. They can be made without modifying the translation to low-level code, simply by controlling type inference. In many cases the low-level language will have access to a stack that can store callee-save arguments. A stack may be added to the low-level language as a type $\mathtt{Stack}$ with $(\mathtt{Stack} \times A) \lhd \mathtt{Stack}$ for all A (implemented by push and pop). Then, one may use $\mathtt{Stack}$ for callee-save annotations and use typing sequents have the form $\Gamma \vdash_{\mathtt{Stack}} t : \mathcal{X}$. What is stored in callee-save arguments is then just a stack pointer. With some further linearity analysis, it is possible to realise $\mathtt{Stack}$ using a machine stack.

Of theoretical interest is the possibility to solve the constraints by taking all annotations to be $\omega := \mu\alpha.\, \mathtt{unit} + \alpha$. This corresponds to exponentials in the Geometry of Interaction, see e.g. [9]. It is possible because we have $A \lhd \omega$ for all A. However, it does not seem suitable for implementation in practice, as the encoding and decoding functions realising $A \lhd \omega$ can be expensive. It would be more efficient to use a type of trees in place of ω, which results in data storage much like in abstract machines [5, 10].

Type annotation inference can also be understood as simple space usage analysis. For example, one may be interested in constant space programs, e.g. for embedded applications or for hardware synthesis [7]. A sufficient condition for restricting to constant space programs is that the $\lhd$-constraints can be solved without recursive types. This rules out (FIX), but tail recursion on first-order types is still possible with the typing rule below.

$$\textsc{tailfix}\ \frac{\Gamma,\ f : \mathtt{unit} \cdot (\mathbb{N} \xrightarrow{C[\![\Gamma]\!]}_A \mathbb{N}),\ x : \mathbb{N} \vdash_A t : \mathbb{N}}{(E \times A) \cdot \Gamma \vdash_F \mathtt{tailfix}\, f\ x \Rightarrow t : E \cdot (\mathbb{N} \xrightarrow{C[\![\Gamma]\!]}_A \mathbb{N})}$$

It is also possible to modify rule (FIX) to capture recursion with bounded depth n by taking H to be $E + (E \times G) + \cdots + (E \times G^n)$.

8 CORRECTNESS

We have defined the translation from annotated PCF to the low-level language directly, in order to be able to make all low-level implementation choices visible. Once one has understood the signatures of the translated low-level program fragments, it is not hard to work directly on this level, e.g. for implementing the translation. However, in the presence of recursion, proving correctness at this level of detail would be very unwieldy.

In this section we outline how the translation can be factored in two steps: a typed closure conversion to a polymorphic λ-calculus, which is then implemented in the low-level language. To verify the whole translation, it then suffices to show these two steps correct.

The first step can be understood as a variant of the typed closure conversion method from [17], which translates call-by-value PCF into a polymorphic λ-calculus. Here we replace the target λ-calculus with a calculus INT$'$ that has a direct low-level implementation. The resulting translation from PCF to the low-level language amounts to the translation described in Sec. 6. Already in [17], the idea of using the calculus INT [16] is mentioned. However, just using INT as is would not quite allow us to capture the low-level invariants from Sec. 6. More importantly, the equational theory from [16] does not suffice for the correctness argument from [17].

8.1 The Calculus INT$'$

We introduce the calculus INT$'$, a generalisation of INT [16]. The types of INT$'$ are defined by the following grammar, in which A ranges over low-level types.

$$\begin{aligned} X, Y \ :=\ & [A] \mid A \to X \mid I \mid X \otimes Y \mid X \multimap Y \mid A \cdot X \\ & \mid \forall \alpha \lhd A.\, X \mid \exists \alpha \lhd A.\, X \end{aligned}$$

Compared to INT, the subexponential $A \cdot X$ is now a first class type (not restricted to appear only as in $A \cdot X \multimap Y$), there are existential types and quantification over low-level types is bounded.

With the given space constraints, we cannot define the full type system of INT′ in detail. The terms and typing rules for INT are adapted from [16].

The reader may think of INT′ as a sub-system of System F. It has all the standard terms from System F, in particular abstraction $\lambda x\colon X.\, t$, application $s\ t$, type abstraction $\Lambda\alpha.\, t$ and type application $t\ A$. There are also standard terms for pairs and existential types. These terms are given types with a linear type system that makes low-level implementation details explicit. This is what the types I, $X \otimes Y$, $X \multimap Y$, $A \cdot X$ and $\forall \alpha \lhd A.\, X$ and $\exists \alpha \lhd A.\, X$ are for. The type $A \cdot X$ may be understood as a bounded version of the exponential $!X$ from Linear Logic, in the spirit of Bounded Linear Logic [8]. Indeed, contraction of a variable $x\colon (A+B)\cdot X$ gives $x_1\colon A\cdot X$ and $x_2\colon B \cdot X$.

In addition, INT′ also has a base type $[A]$ and a call-by-value function type $A \rightarrow X$. The type $[A]$ is intended to represent a computation that returns a value of type A. There are monadic terms (`return` and `let`) for this type. The type $A \rightarrow X$ represents a call-by-value function space that takes as argument low-level values of type A. There are special abstraction and application terms $\mathtt{fn}\ x\colon A \Rightarrow t$ and $t(v)$ for this type. For example, an incrementation operation may be given type $\mathtt{succ}\colon \mathtt{int} \rightarrow [\mathtt{int}]$ and the term $(\mathtt{fn}\ x\colon \mathtt{int} \Rightarrow \mathtt{let}\ y = \mathtt{succ}(x)\ \mathtt{in}\ \mathtt{return}(x, y))$ has type $\mathtt{int} \rightarrow [\mathtt{int} \times \mathtt{int}]$.

The terms and types of INT′ have a direct low-level interpretation. The types represent the interfaces of low-level programs, exactly as the signatures in Sec. 3.2 do. The signatures there may in fact be considered as special cases of INT′ types: $A \longrightarrow B$ corresponds to $A \rightarrow [B]$; and $\mathtt{functor}(X : S_1) \longrightarrow S_2$ corresponds to $S_1 \multimap S_2$; and I is the empty signature; and $\mathtt{sig}\ X_1 : S_1, \dots, X_n : S_n\ \mathtt{end}$ corresponds to $S_1 \otimes \cdots \otimes S_n$.

The terms of INT′ represent program fragments. A term $t\colon X$ represents a fragment $(\!|t|\!)\colon X^- \longrightarrow X^+$ (the definitions of X^- and X^+ are as for signatures). For example, a term $t\colon \forall \alpha \lhd A.\, X$ represents a fragment shown in the figure below. If, for some B with $B \lhd A$, one connects the fragments `in` and `out` obtained from $B \lhd A$ as shown, then the resulting fragment corresponds to the term $t\ B$.

$$\begin{tikzcd}
{} \arrow[r, "{X^-[B/\alpha]}"] & \boxed{\mathtt{in}} \arrow[r, "{X^-[A/\alpha]}"] & \boxed{(\!|t|\!)} \arrow[r, "{X^+[A/\alpha]}"] & \boxed{\mathtt{out}} \arrow[r, "{X^+[B/\alpha]}"] & {}
\end{tikzcd}$$

For existential types, if s is a term of type $X[B/\alpha]$, then $\mathtt{pack}(B, t)$ has type $\exists \alpha \lhd A.\, X$ whenever $B \lhd A$. This term represents the following fragment:

$$\begin{tikzcd}
{} \arrow[r, "{X^-[A/\alpha]}"] & \boxed{\mathtt{out}} \arrow[r, "{X^-[B/\alpha]}"] & \boxed{(\!|s|\!)} \arrow[r, "{X^+[B/\alpha]}"] & \boxed{\mathtt{in}} \arrow[r, "{X^+[A/\alpha]}"] & {}
\end{tikzcd}$$

The constructions in Sec. 3.2 can be read as describing application terms for $A \longrightarrow [B]$ and $X \multimap Y$.

8.2 Correctness

To show correctness of the translation from Sec 6, we now first translate PCF into INT′ in such a way that the low-level interpretation of the resulting INT′ terms agrees (up to simplification) with the programs from Sec. 6.

With the above view of signatures as special cases of INT′ types, the signatures from Sec. 6 can be seen as INT′ types. We define a translation that takes a derivation Π of the annotated PCF judgement $x_1\colon \mathcal{X}_1, \dots, x_k\colon \mathcal{X}_k \vdash_S t\colon Y$ in to an INT′ term $TCC'(\Pi)$ of the type in (5). We define it such that the translation from Sec. 6 appears as a (slightly simplified) direct description of $(\!|TCC'(\Pi)|\!)$.

The definition of $TCC'(\Pi)$ follows the lines of the typed closure conversion from [17]. However, it needs to be modified to produce an efficient low-level implementation. Consider, for example, the case for application, i.e. where Π ends with rule (APP). If we write Π_s and Π_t for the sub-derivations of the two premises, then $TCC'(\Pi)$ has a definition of the following form, which is much like in [17].

$$\begin{aligned}
TCC'(\Pi) = \Lambda\vec{\alpha}.\, \lambda\vec{x}\vec{y}.\ & \mathtt{let}\ \mathtt{pack}(\alpha, \langle \mathsf{T}_f, \mathtt{eval}_f\rangle) = TCC'(\Pi_s)\ \vec{\alpha}\ \vec{x}\ \mathtt{in}\\
& \mathtt{let}\ \mathtt{pack}(\beta, \langle \mathsf{T}_x, \mathtt{eval}_x\rangle) = TCC'(\Pi_t)\ \vec{\alpha}\ \vec{y}\ \mathtt{in}\\
& \mathtt{let}\ \mathtt{pack}(\rho, \langle \mathsf{T}_y, \mathtt{apply}\rangle) = \mathsf{T}_f\ \beta\ \mathsf{T}_x\ \mathtt{in}\\
& \mathtt{pack}(\rho, \langle \mathsf{T}_y, \mathtt{eval}\rangle)
\end{aligned}$$

In [17], the term `eval`, is defined to be

$$\begin{aligned}
\mathtt{fn}\ \langle \vec{x}, \vec{y}\rangle \Rightarrow\ & \mathtt{let}\ v_f = \mathtt{eval}_f(\vec{x})\ \mathtt{in}\\
& \mathtt{let}\ v_x = \mathtt{eval}_x(\vec{y})\ \mathtt{in}\ \mathtt{apply}(\langle v_f, v_x\rangle).
\end{aligned}$$

This means that `eval` first computes function value, then argument value, and then invokes the application code for the function.

In this paper we use a slightly different implementation of `eval` that defines the same function (extensionally), but that has a better low-level implementation. The low-level implementation of `eval` above is not optimal with respect to its space usage. It stores the variables in $\vec{x}$ until $apply(\langle v_f, v_x\rangle)$ returns, even though they could be disposed of earlier. Hence we use a more space efficient implementation of `eval` that discards these values earlier. This optimisation is already reflected in the type annotations (one may choose to use `eval` as above, but then needs to weaken the type annotations).

Besides adding low-level annotations, one must also extend the interpretation of (IF), which was restricted to base types in [17] for simplicity. To do so, one defines an INT′ term $\mathtt{join}\colon \mathcal{I}[\![\mathcal{X}_1]\!]_{A_1} \otimes \mathcal{I}[\![\mathcal{X}_2]\!]_{A_2} \multimap \mathcal{I}[\![\mathcal{X}]\!]_{A_1+A_2}$ whenever $\mathcal{X}_1 \curlyvee \mathcal{X}_2 = \mathcal{X}$. The term is defined by induction on the derivation of the latter. It represents the low-level programs 4 and 5 in the translation of (IF) in Sec. 6 that join the interfaces of $(\!|t_1|\!)$ and $(\!|t_2|\!)$ into a single one.

Having defined $TCC'(\Pi)$, the task is then to show:

THEOREM 8.1. *Suppose* Π *derives* $\vdash_A t\colon B \cdot \mathbb{N}$. *Then* t *reduces to a value* v *in a standard call-by-value operational semantics if and only if we have* $(\!|TCC'(\Pi)|\!)\colon \langle a, ()\rangle \mapsto \langle a, v\rangle$ *for any closed low-level value* $a\colon A$.

To show this, we would like to use the proof from [17], but the equational theory of INT from [16] is insufficient to do so, e.g. because its extension to recursion is unclear.

To support reasoning like in [17], we define a notion of equality that lets us ignore low-level annotations and consider terms as if they were System F terms. The idea is that two terms whose types differ only up to low-level annotations (e.g. because they use callee-save arguments differently) may still be considered as implementing the same program. With bounded quantification and unrestricted subexponentials $A \cdot X$, such a generalisation appears to be necessary to allow a useful reasoning. We formalise it by defining a logical relation $(\!|t|\!) \sim_{X,\rho} [\![t]\!]$ that relates the low-level program $(\!|t|\!)$ obtained from an INT′-term t to a standard domain-theoretic interpretation $[\![t]\!]$ that ignores low-level annotations and interprets t as if it were a System F-term. At base type, the logical relation allows us to conclude that $(\!|t|\!)$ returns a number n if and only if $[\![t]\!] = n$. This means that we may use the domain-theoretic interpretation $[\![t]\!]$, i.e. to ignore low-level annotations, and still obtain useful results about low-level programs.

With these definitions, we can use the argument from [17] with very few modifications to show correctness. The term $TCC'(\Pi)$ is not the same $TCC(\Pi)$ from [17] intensionally, but has the same (extensional) domain-theoretic denotation. The proof from [17] can then be used to show correctness of the denotational interpretation. One uses a logical relation to relate PCF to the denotational interpretation of $TCC(\Pi)$, as in [17]. To relate $TCC(\Pi)$ to low-level programs, we use the logical relation $\sim_{X,\rho}$ outlined above.

While Thm. 8.1 applies only to closed terms of base type, the two logical relations can also be used to give a specification of the low-level program fragments obtained by translating (possibly open) terms of higher-order type. A possible application would be to link the translated low-level programs to low-level code written by hand or produced by other compilers. Note that the choice of closure representation is abstract and the encoding of closures may be different for each abstraction.

9 CONCLUSION

Defunctionalisation is often presented as a whole-program transformation, which makes modular reasoning and implementation difficult. The translation of a part of a program may depend on global choices and these choices are also visible in the interfaces of produced code fragments. In this paper we have shown how to define and reason about defunctionalisation in a simple modular way. By relating it to a typed closure conversion and managing the low-level details in INT′, we were able to obtain a strongly compositional correctness proof for a realistic defunctionalisation method that is almost as simple as one for typed closure conversion alone.

An annotated type system for PCF captures all the global information that are needed for defunctionalisation; the correctness argument works for any possible choice of annotations. In a translation from PCF to the low-level language, one must make many encoding choices (closure encoding, implementation of callee-save arguments, etc), some of which can be made in many ways. The annotated type system allows flexible control over such choices by recording only constraints, as opposed to making ad hoc choices. Its types contain enough information to specify interfaces and a call-by-value calling-convention for compiled low-level code.

The annotated type system may be useful for separate compilation. For example, one to defer the choice of (some) closure representation to linking time. This can be done on the level of annotates PCF types. One may use type variables for all annotations in the types of public functions and record the $\lhd$-constraints. The module can then be compiled so that the `in`- and `out`-terms corresponding to the pending constraints are left undefined. The constraints are then solved at linking time and concrete code for the `in`- and `out`-terms is inserted then. Another option for separate compilation would be to use a standard closure representation using pointers for public functions of a module and use defunctionalisation internally. Our method makes it easy to use different closure representation in different places.

A simple implementation of type inference, translation to the low-level language and from there to machine code via LLVM can be found at http://www.github.com/uelis/modular. It demonstrates that type inference and the translation can be implemented efficiently, and also that effectful operations can be added without problem. For a meaningful performance evaluation, the implementation is currently too simple (it uses a naive garbage collector, for example). One may nevertheless look at a few example PCF-programs to check that the generated machine code is reasonable. The raytracing example from folder `Tests` is tail-recursive and translates to a low-level program that does not use the heap. The code runs in 1.6s without inlining and 0.6s with inlining on an x64 Intel i7-4770. Version of the same example take 1.7s and 2.6s when compiled with `ocamlopt` 4.02.3 and MLton 20100608 respectively. The reason may well be the quality of LLVM. For a higher-order example `euler` that computes digits of e (using streams represented by functions), the implementation produces a program that executes in 0.35s, versus 0.24s for OCaml and 0.13s for MLton. This result is most likely due to unnecessary heap allocations because the optimisation from Sec. 7 is currently not implemented, so that each recursive call incurs heap allocation and copying.

REFERENCES

[1] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In *Theoretical Aspects of Computer Software, TACS 2001*, pages 420–447, 2001.
[2] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. Verified compilation for shared-memory C. In *European Symposium on Programming, ESOP 2014*, volume 8410 of *Lecture Notes in Computer Science*, pages 107–127. Springer, 2014.
[3] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *European Symposium on Programming, ESOP 2000*, pages 56–71, 2000.
[4] Ugo Dal Lago and Ulrich Schöpp. Computation by interaction for space bounded functional programming. Information and Computation, 2016.
[5] Vincent Danos, Hugo Herbelin, and Laurent Regnier. Game semantics and abstract machines. In *Logic in Computer Science, LICS 1996*, pages 394–405. 1996.
[6] Georgios Fourtounis, Nikolaos S. Papaspyrou, and Panagiotis Theofilopoulos. Modular polymorphic defunctionalization. *Comput. Sci. Inf. Syst.*, 11(4):1417–1434, 2014.
[7] Dan R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In *Principles of Programming Languages, POPL 2007*, pages 363–375. ACM, 2007.
[8] 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.
[9] Naohiko Hoshino. A modified goi interpretation for a linear functional programming language and its adequacy. In *Foundations of Software Science and Computational Structures, FOSSACS 2011*, pages 320–334, 2011.
[10] Ian Mackie. The geometry of interaction machine. In *Principles of Programming Languages, POPL 1995*, pages 198–208. ACM, 1995.
[11] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In *Principles of Programming Languages, POPL 1996*, pages 271–283. ACM, 1996.
[12] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19(1):125–162, 2006.
[13] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. *J. Funct. Program.*, 24(5):529–607, 2014.
[14] Ulrich Schöpp. *Computation-by-Interaction for Structuring Low-Level Computation*. Habilitation thesis, Ludwig-Maximilians-Universität München, 2015.
[15] Ulrich Schöpp. On the relation of interaction semantics to continuations and defunctionalization. *Logical Methods in Computer Science*, 10(4), 2014.
[16] Ulrich Schöpp. Organising low-level programs using higher types. In *Principles and Practice of Declarative Programming, PPDP 2014*, New York, NY, 2014. ACM.
[17] Ulrich Schöpp. From call-by-value to interaction by typed closure conversion. In *Asian Symposium on Programming Languages and Systems, APLAS 2015*, volume 9458 of *Lecture Notes in Computer Science*, pages 251–270. Springer, 2015.
[18] Lukasz Ziarek, Stephen Weeks, and Suresh Jagannathan. Flattening tuples in an SSA intermediate representation. *Higher-Order and Symbolic Computation*, 21(3):333–358, 2008.

A PCF

For reference, we include the typing rules for our variant of PCF in Fig. 3.

B LOW-LEVEL LANGUAGE

For low-level programs, the typing rules for values are shown in Fig. 4. In these rules, Γ is a value context, a finite mapping from variables to low-level types. To define the well-typed low-level programs, we define a judgement $\Gamma \mid \Phi \vdash b$ that identifies well-typed bodies of blocks. Therein, Φ is a *label context*, which is a list of declarations of the form $f : \neg A$, expressing that the block with label f takes arguments of type A. For each label, Φ must contain at most one declaration. The typing rules for this judgement are defined in Fig. 5.

A program fragment P is *well-typed in context* Γ if there exists a label context Φ such that, for each block definition $f\,(x : A) = b$ in P, both $\Gamma, x : A \mid \Phi \vdash b$ is derivable and $f : \neg A$ is in Φ.

The operational semantics for low-level program fragments is defined for closed low-level programs. For a closed program fragment P it is given by a relation $b_1 \to_P b_2$, which expresses that body term b_1 reduces to body term b_2. It is defined to be the smallest relation transitive relation satisfying $f(v) \to_P b[v/x]$ if p contains a block definition $f\,(x : A) = b$, and such that the following basic transitions hold.

$$\begin{aligned}
\texttt{let}\ \langle x, y\rangle = \langle v, w\rangle\ \texttt{in}\ b \quad &\to_P \quad b[v/x, w/y]\\
\texttt{case}\ \texttt{inl}(v)\ \texttt{of}\ \texttt{inl}(x) \Rightarrow b_1;\ \texttt{inr}(y) \Rightarrow b_2 \quad &\to_P \quad b_1[v/x]\\
\texttt{case}\ \texttt{inr}(w)\ \texttt{of}\ \texttt{inl}(y) \Rightarrow b_1;\ \texttt{inr}(y) \Rightarrow b_2 \quad &\to_P \quad b_2[v/y]\\
\texttt{let}\ \texttt{fold}_{\mu\alpha.\,A}(x) = \texttt{fold}_{\mu\alpha.\,A}(v)\ \texttt{in}\ b \quad &\to_P \quad b[v/x]\\
\texttt{let}\ \texttt{in}_{A,B}(x) = \texttt{in}_{A,B}(v)\ \texttt{in}\ b \quad &\to_P \quad b[v/x]
\end{aligned}$$

We omit the straightforward transition for the primitive operations `add`, `mul`, etc.

We consider two closed programs $P, Q : A \to B$ *equal* if they are equal extensionally. Write $P : v \mapsto w$ if jumping to the (wlog single) entry label $\mathit{entry}(v) \to_P^* \mathit{exit}(w)$. Two programs are equal if $P : v \mapsto w$ holds if and only if $Q : v \mapsto w$ does.

C ANNOTATED PCF

C.1 Translation

We give the missing cases for the translation from Sec. 6.

For defining the translation, it is convenient to use the following notation. Recall that $A \lhd B$ was defined in terms of encoding and decoding programs `in` and `out`. For working with them, it is convenient to use the notation $\texttt{let}\ y = \texttt{in}_{A \lhd B}(v)\ \texttt{in}\ b_1$ and $\texttt{let}\ \texttt{in}_{A \lhd B}(x) = w\ \texttt{in}\ b_2$ for definable fragments with the following behaviour. In the fragment $\texttt{let}\ y = \texttt{in}_{A \lhd B}(v)\ \texttt{in}\ b_1$, the value v must have type A. When executing this fragment, it encodes v as an element of B, as the `in` program would do, binds the result to y and then executes b_1. The other fragment does the analogous decoding.

Case (ZERO): $(\!|\Pi|\!)$ must have type $\texttt{void} + A \times \texttt{unit} \to \texttt{void} + A \times \texttt{int}$. As `void` is the empty type, it suffices to give a program of type $A \times \texttt{unit} \to A \times \texttt{int}$, which we define by $\langle a, ()\rangle \mapsto \langle a, 0\rangle$.

Case (SUCC): Write Π_t for the derivation of the premise of this rule. After simplification as in the case for (ZERO), $(\!|\Pi_t|\!)$ amounts to a program $A \times \texttt{unit} \to A \times \texttt{int}$ that computes the value of t. We obtain $(\!|\Pi|\!)$ by appending to $(\!|\Pi_t|\!)$ a block that increments the return value.

The case for (PRED) is treated analogously.

Case (IF): The translation of this rule is already given in Sec. 6, but it remains to define the fragments 3, 4 and 5. To define them, we use the side condition $\mathcal{X}_1 \curlyvee \mathcal{X}_2 = \mathcal{X}$ from the premise of (IF). For any derivation Π of $\mathcal{X}_1 \curlyvee \mathcal{X}_2 = \mathcal{X}$, we define three programs $b_\Pi : C[\![\mathcal{X}_1]\!] + C[\![\mathcal{X}_2]\!] \to C[\![\mathcal{X}]\!]$ and $\texttt{in}_\Pi : \mathcal{I}[\![\mathcal{X}_1]\!]^- + \mathcal{I}[\![\mathcal{X}_2]\!]^- \to \mathcal{I}[\![\mathcal{X}]\!]^-$ and $\texttt{out}_\Pi : \mathcal{I}[\![\mathcal{X}]\!]^+ \to \mathcal{I}[\![\mathcal{X}_1]\!]^+ + \mathcal{I}[\![\mathcal{X}_2]\!]^+$. Then we take $A \cdot b_\Pi$, $\texttt{in}_\Pi$ and $\texttt{out}_\Pi$ for 3, 4 and 5, respectively.

The programs b_Π, $\texttt{in}_\Pi$ and $\texttt{out}_\Pi$ are defined by induction on Π. In the base case $A \cdot \mathbb{N} \curlyvee A \cdot \mathbb{N} = A \cdot \mathbb{N}$, the programs $\texttt{out}_\Pi$ and $\texttt{in}_\Pi$ are vacuous, as their interfaces are void. For b_Π we choose the program $(x \mapsto \texttt{case}\ x\ \texttt{of}\ \texttt{inl}(y) \Rightarrow y;\ \texttt{inl}(z) \Rightarrow z)$. In the induction case, Π ends with the joining rule for functions. In this case, we let b_Π be the identity and define the other two programs in Fig. 6. Blocks 1, 5 and 6 perform case distinction over $D_1 + D_2$ or $B_1 + B_2$. Programs 2, 3 and 4 are b, `out` and `in` from the induction hypothesis for $\mathcal{Y}_1 \curlyvee \mathcal{Y}_2 = \mathcal{Y}$.

Cases (SUBL) and (SUBR): To translate these rules, we define, for any derivation Π of $\mathcal{X} \le \mathcal{Y}$, low-level programs $\texttt{in}_\Pi : \mathcal{I}[\![\mathcal{Y}]\!]^- \to \mathcal{I}[\![\mathcal{X}]\!]^-$ and $\texttt{out}_\Pi : \mathcal{I}[\![\mathcal{X}]\!]^+ \to \mathcal{I}[\![\mathcal{Y}]\!]^+$. The definition goes by induction on the derivation Π. The base case for $\mathbb{N}$ is trivial, as the interface $\mathcal{I}[\![\mathbb{N}]\!]$ consists of empty types. If Π ends with the subtyping rule for functions, then we write $\Pi_\mathcal{X}$ and $\Pi_\mathcal{Y}$ for the subderivation deriving its two premises and the desired programs as in Fig. 7.

Case (FIX): The implementation of recursion is shown in Fig. 8. For clarity the interface of the program obtained by translating the premise is also shown there. The translation of the conclusion implements recursion in a standard way. By the side condition on (FIX), the type H can encode values that represent a call stack for the recursion. Recursive calls are managed by using a callee-save value of type H to represent the call stack. For example, if the program jumps to the application code for the recursive function f, then it jumps to block $2'$ in the figure. This block "pushes" the current stack frame g on the stack h and jumps with the resulting stack h' to the application code for t. If t is finished with evaluation, it jumps to block 5. If the call stack h is empty, then this block returns the result, otherwise it pops off the topmost stack frame g and acts like a return from a recursive call.

C.2 Examples

For Examples 4.1-4.3, the translation to the low-level language is spelled out in detail in Fig. 9.

C.3 Algorithmic Formulation of Typing Rules

The algorithmic typing rules used in the proof of Theorem 7.1 are shown in Fig. 10. There, we write $\Gamma \lhd \Delta$ if Γ has the form $x_1 : A_1 \cdot X_1, \dots, x_n : A_n \cdot X_n$ and Δ has the form $x_1 : B_1 \cdot X_1, \dots, x_n : B_n \cdot X_n$ and we have $A_i \lhd B_i$ for $i = 1, \dots, n$. This restricted form of subtyping is enough for type inference.

D CORRECTNESS

D.1 Organising Low-Level Programs

We define the type system for INT$'$ and define the logical relation that relates INT$'$ terms to low-level programs.

$$\textsc{var}\ \frac{}{x\colon X \vdash x\colon X} \qquad \textsc{w}\ \frac{\Gamma \vdash t\colon Y}{\Gamma,\ x\colon X \vdash t\colon Y} \qquad \textsc{e}\ \frac{\Gamma,\ x\colon X,\ y\colon Y,\ \Delta \vdash t\colon Z}{\Gamma,\ y\colon Y,\ x\colon X,\ \Delta \vdash t\colon Z} \qquad \textsc{c}\ \frac{\Gamma,\ x\colon X,\ y\colon X \vdash t\colon Y}{\Gamma,\ z\colon X \vdash t[z/x,\ z/y]\colon Y} \qquad \textsc{zero}\ \frac{}{\vdash \mathsf{zero}\colon \mathbb{N}}$$

$$\textsc{succ}\ \frac{\vdash t\colon \mathbb{N}}{\vdash \mathsf{succ}(t)\colon \mathbb{N}} \qquad \textsc{pred}\ \frac{\vdash t\colon \mathbb{N}}{\vdash \mathsf{pred}(t)\colon \mathbb{N}} \qquad \textsc{if}\ \frac{\Gamma \vdash s\colon \mathbb{N} \qquad \Delta_1 \vdash t_1\colon X \qquad \Delta_2 \vdash t_2\colon X}{\Gamma,\ \Delta_1,\ \Delta_2 \vdash \mathsf{ifz}\ s\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\colon X} \qquad \textsc{fn}\ \frac{\Gamma,\ x\colon X \vdash t\colon Y}{\Gamma \vdash \mathsf{fn}\, x \Rightarrow t\colon X \to Y}$$

$$\textsc{app}\ \frac{\Gamma \vdash s\colon X \to Y \qquad \Delta \vdash t\colon X}{\Gamma,\ \Delta \vdash s\ t\colon Y} \qquad \textsc{fix}\ \frac{\Gamma,\ f\colon X \to Y,\ x\colon X \vdash t\colon Y}{\Gamma \vdash \mathsf{fix}\, f\ x \Rightarrow t\colon X \to Y}$$

Figure 3: Call-by-Value PCF

$$\frac{}{\Gamma,\ x\colon A \vdash x\colon A} \qquad \frac{}{\Gamma \vdash ()\colon \mathsf{unit}} \qquad \frac{}{\Gamma \vdash n\colon \mathsf{int}} \qquad \frac{\Gamma \vdash v\colon A \qquad \Gamma \vdash w\colon B}{\Gamma \vdash \langle v,\ w\rangle\colon A \times B}$$

$$\frac{\Gamma \vdash v\colon A}{\Gamma \vdash \mathsf{inl}_{A+B}(v)\colon A + B} \qquad \frac{\Gamma \vdash v\colon B}{\Gamma \vdash \mathsf{inr}_{A+B}(v)\colon A + B} \qquad \frac{\Gamma \vdash v\colon A_i}{\Gamma \vdash \mathsf{in}_{A_i,\, A_1 \cup A_2}(v)\colon A_1 \cup A_2}\ i \in \{1, 2\} \qquad \frac{\Gamma \vdash v\colon A[\mu\alpha.\, A/\alpha]}{\Gamma \vdash \mathsf{fold}_{\mu\alpha.\, A}(v)\colon \mu\alpha.\, A}$$

Figure 4: Typing of Low-Level Values

$$\frac{\Gamma \vdash v\colon A \qquad \mathsf{op} \in \mathit{Prim}(A, B) \qquad \Gamma, x\colon B \mid \Phi \vdash b}{\Gamma \mid \Phi \vdash \mathsf{let}\ x = \mathsf{op}(v)\ \mathsf{in}\ b} \qquad \frac{\Gamma \vdash v\colon A \times B \qquad \Gamma, x\colon A, y\colon B \mid \Phi \vdash b}{\Gamma \mid \Phi \vdash \mathsf{let}\ \langle x, y\rangle = v\ \mathsf{in}\ b}$$

$$\frac{\Gamma \vdash v\colon A + B \qquad \Gamma, x\colon A \mid \Phi \vdash b_1 \qquad \Gamma, y\colon B \mid \Phi \vdash b_2}{\Gamma \mid \Phi \vdash \mathsf{case}\ v\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow b_1;\ \mathsf{inr}(y) \Rightarrow b_2} \qquad \frac{\Gamma \vdash v\colon \mu\alpha.\, A \qquad \Gamma, x\colon A[\mu\alpha.\, A/\alpha] \mid \Phi \vdash b}{\Gamma \mid \Phi \vdash \mathsf{let}\ \mathsf{fold}_{\mu\alpha.\, A}(x) = v\ \mathsf{in}\ b}$$

$$\frac{\Gamma \vdash v\colon A \cup B \qquad \Gamma, x\colon A \mid \Phi \vdash b}{\Gamma \mid \Phi \vdash \mathsf{let}\ \mathsf{in}_{A,B}(x) = v\ \mathsf{in}\ b} \qquad \frac{\Gamma \vdash v\colon A}{\Gamma \mid \Phi, f\colon \neg A, \Psi \vdash f(v)}$$

Figure 5: Typing of Low-Level Blocks

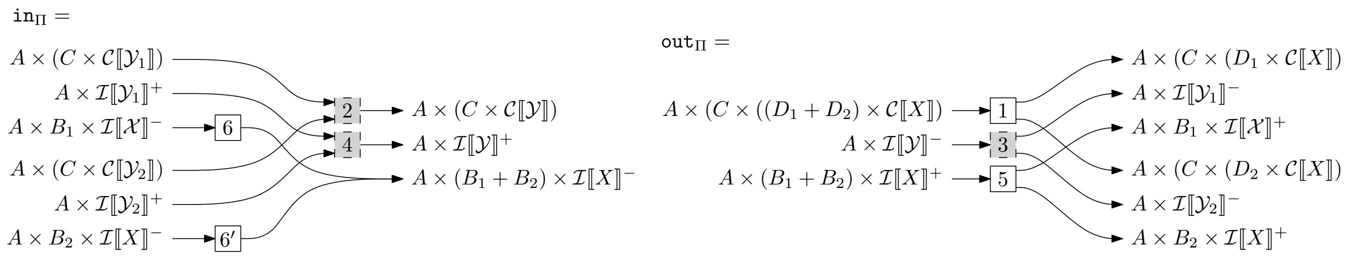
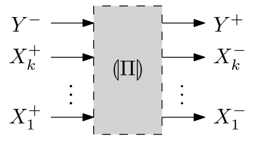
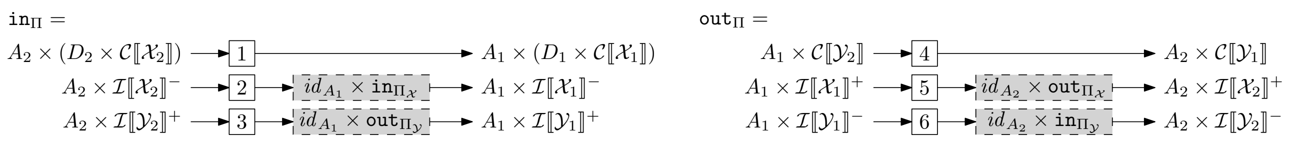

$\mathtt{in}_\Pi =$
$A \times (C \times \mathcal{C}[\![\mathcal{Y}_1]\!])$
$A \times \mathcal{I}[\![\mathcal{Y}_1]\!]^+$
$A \times B_1 \times \mathcal{I}[\![\mathcal{X}]\!]^-$
6
$A \times (C \times \mathcal{C}[\![\mathcal{Y}_2]\!])$
$A \times \mathcal{I}[\![\mathcal{Y}_2]\!]^+$
$A \times B_2 \times \mathcal{I}[\![X]\!]^-$
6′
2
4
$A \times (C \times \mathcal{C}[\![\mathcal{Y}]\!])$
$A \times \mathcal{I}[\![\mathcal{Y}]\!]^+$
$A \times (B_1 + B_2) \times \mathcal{I}[\![X]\!]^-$
$\mathtt{out}_\Pi =$
$A \times (C \times ((D_1 + D_2) \times \mathcal{C}[\![X]\!])$
$A \times \mathcal{I}[\![\mathcal{Y}]\!]^-$
$A \times (B_1 + B_2) \times \mathcal{I}[\![X]\!]^+$
1
3
5
$A \times (C \times (D_1 \times \mathcal{C}[\![X]\!])$
$A \times \mathcal{I}[\![\mathcal{Y}_1]\!]^-$
$A \times B_1 \times \mathcal{I}[\![\mathcal{X}]\!]^+$
$A \times (C \times (D_2 \times \mathcal{C}[\![X]\!])$
$A \times \mathcal{I}[\![\mathcal{Y}_2]\!]^-$
$A \times B_2 \times \mathcal{I}[\![X]\!]^+$

Figure 6: Joining of Interfaces in the Translation of (IF)

The INT′ typing judgement has the form $\Omega \mid \Gamma \mid \Phi \vdash t\colon X$, where $\Omega = \alpha_1 \lhd A_1, \ldots, \alpha_n \lhd A_n$ is a context declaring type variables and their bounds, where $\Gamma = y_1\colon B_1,\ \ldots,\ y_m\colon B_m$ is a context declaring value variables, and where $\Phi = x_1\colon X_1,\ \ldots,\ x_k\colon X_k$ is a context declaring module interface variables. In these contexts, the A_i and B_j range over low-level types and the X_l range over INT′ types. A derivation Π of the typing judgement $\Omega \mid \Gamma \mid \Phi \vdash t\colon X$ represents a low-level program, called $(\!|\Pi|\!)$ in a slight abuse of notation, of the following type.



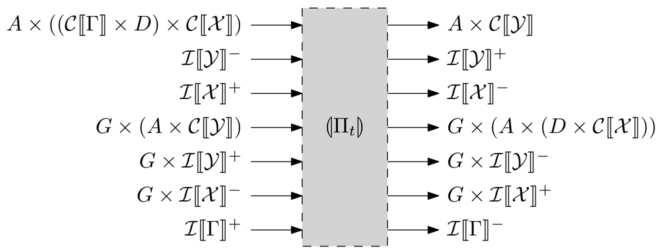
The program $(\!|\Pi|\!)$ may contain free occurrences of the type variables from Ω and the value variables from Γ. The intention is that a closed term of type X translates to a low-level fragment of type $X^- \to X^+$. For an open term like t above, the intention is that program fragments $X_1^- \to X_1^+, \ldots, X_k^- \to X_k^+$ are connected to it using iapp from Fig. 17. These fragments may come from closed terms, for example. Then one obtains a fragment of type $Y^- \to Y^+$


$\mathtt{in}_\Pi =$
$A_2 \times (D_2 \times \mathcal{C}\llbracket\mathcal{X}_2\rrbracket)$
$A_2 \times \mathcal{I}\llbracket\mathcal{X}_2\rrbracket^-$
$A_2 \times \mathcal{I}\llbracket\mathcal{Y}_2\rrbracket^+$
1
2
3
$id_{A_1} \times \mathtt{in}_{\Pi_\mathcal{X}}$
$id_{A_1} \times \mathtt{out}_{\Pi_\mathcal{Y}}$
$A_1 \times (D_1 \times \mathcal{C}\llbracket\mathcal{X}_1\rrbracket)$
$A_1 \times \mathcal{I}\llbracket\mathcal{X}_1\rrbracket^-$
$A_1 \times \mathcal{I}\llbracket\mathcal{Y}_1\rrbracket^+$
$\mathtt{out}_\Pi =$
$A_1 \times \mathcal{C}\llbracket\mathcal{Y}_2\rrbracket$
$A_1 \times \mathcal{I}\llbracket\mathcal{X}_1\rrbracket^+$
$A_1 \times \mathcal{I}\llbracket\mathcal{Y}_1\rrbracket^-$
4
5
6
$id_{A_2} \times \mathtt{out}_{\Pi_\mathcal{X}}$
$id_{A_2} \times \mathtt{in}_{\Pi_\mathcal{Y}}$
$A_2 \times \mathcal{C}\llbracket\mathcal{Y}_1\rrbracket$
$A_2 \times \mathcal{I}\llbracket\mathcal{X}_2\rrbracket^+$
$A_2 \times \mathcal{I}\llbracket\mathcal{Y}_2\rrbracket^-$

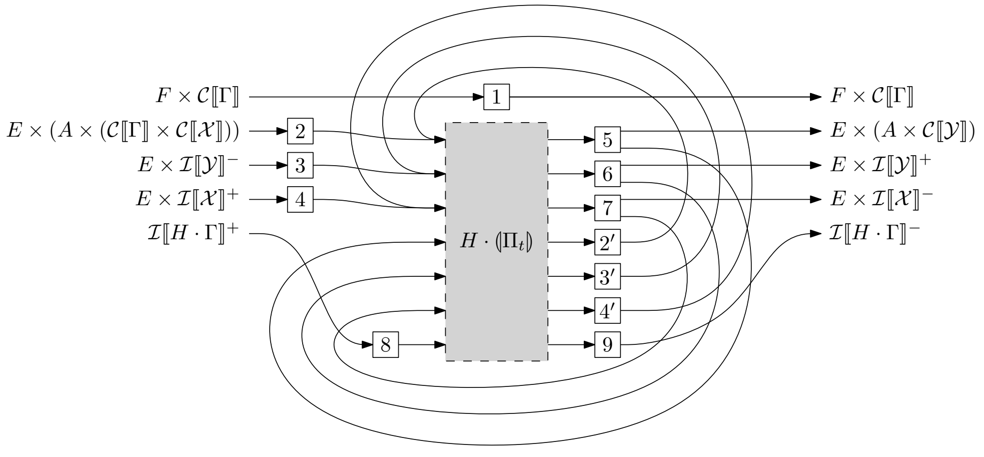
$$\begin{aligned}
1 &: \langle a, \langle d, x\rangle\rangle \mapsto \langle \mathtt{in}_{A_2 \lhd A_1}(a), \langle \mathtt{in}_{D_2 \lhd D_1}(d), \mathtt{in}_{\mathcal{C}\llbracket\mathcal{X}_2\rrbracket \lhd \mathcal{C}\llbracket\mathcal{X}_1\rrbracket}(x)\rangle\rangle \\
2,3 &: \langle a, x\rangle \mapsto \langle \mathtt{in}_{A_2 \lhd A_1}(a), x\rangle \\
4 &: \langle a_1, y_2\rangle \mapsto \mathtt{let}\ \mathtt{in}_{A_2 \lhd A_1}(a_2) = a_1\ \mathtt{in}\ \mathtt{let}\ \mathtt{in}_{\mathcal{C}\llbracket\mathcal{Y}_1\rrbracket \lhd \mathcal{C}\llbracket\mathcal{Y}_2\rrbracket}(y_1) = y_2\ \mathtt{in}\ \langle a_2, y_1\rangle \\
5,6 &: \langle a_1, x\rangle \mapsto \mathtt{let}\ \mathtt{in}_{A_2 \lhd A_1}(a_2) = a_1\ \mathtt{in}\ \langle a_2, x\rangle
\end{aligned}$$

Figure 7: Translation of Subtyping

Interface of translation of premise $\Gamma,\ f : G \cdot \big(\mathcal{X} \xrightarrow{\mathcal{C}\llbracket\Gamma\rrbracket}_A \mathcal{Y}\big),\ x : \mathcal{X} \vdash_A t : \mathcal{Y}$:


$A \times ((\mathcal{C}\llbracket\Gamma\rrbracket \times D) \times \mathcal{C}\llbracket\mathcal{X}\rrbracket)$
$\mathcal{I}\llbracket\mathcal{Y}\rrbracket^-$
$\mathcal{I}\llbracket\mathcal{X}\rrbracket^+$
$G \times (A \times \mathcal{C}\llbracket\mathcal{Y}\rrbracket)$
$G \times \mathcal{I}\llbracket\mathcal{Y}\rrbracket^+$
$G \times \mathcal{I}\llbracket\mathcal{X}\rrbracket^-$
$\mathcal{I}\llbracket\Gamma\rrbracket^+$
$(\!|\Pi_t|\!)$
$A \times \mathcal{C}\llbracket\mathcal{Y}\rrbracket$
$\mathcal{I}\llbracket\mathcal{Y}\rrbracket^+$
$\mathcal{I}\llbracket\mathcal{X}\rrbracket^-$
$G \times (A \times (D \times \mathcal{C}\llbracket\mathcal{X}\rrbracket))$
$G \times \mathcal{I}\llbracket\mathcal{Y}\rrbracket^-$
$G \times \mathcal{I}\llbracket\mathcal{X}\rrbracket^+$
$\mathcal{I}\llbracket\Gamma\rrbracket^-$

Translation of conclusion $H \cdot \Gamma \vdash_F \mathtt{fix}\ f\ x \Rightarrow t : E \cdot (\mathcal{X} \xrightarrow{\mathcal{C}\llbracket\Gamma\rrbracket}_A \mathcal{Y})$:


$F \times \mathcal{C}\llbracket\Gamma\rrbracket$
$E \times (A \times (\mathcal{C}\llbracket\Gamma\rrbracket \times \mathcal{C}\llbracket\mathcal{X}\rrbracket))$
$E \times \mathcal{I}\llbracket\mathcal{Y}\rrbracket^-$
$E \times \mathcal{I}\llbracket\mathcal{X}\rrbracket^+$
$\mathcal{I}\llbracket H \cdot \Gamma\rrbracket^+$
1
2
3
4
8
$H \cdot (\!|\Pi_t|\!)$
5
6
7
2′
3′
4′
9
$F \times \mathcal{C}\llbracket\Gamma\rrbracket$
$E \times (A \times \mathcal{C}\llbracket\mathcal{Y}\rrbracket)$
$E \times \mathcal{I}\llbracket\mathcal{Y}\rrbracket^+$
$E \times \mathcal{I}\llbracket\mathcal{X}\rrbracket^-$
$\mathcal{I}\llbracket H \cdot \Gamma\rrbracket^-$

$$\begin{aligned}
1 &: \langle f, v\rangle \mapsto \langle f, v\rangle \\
2 &: \langle e, \langle a, \langle c, x\rangle\rangle\rangle \mapsto \mathtt{let}\ h = \mathtt{in}_{E+H\times G \lhd H}(\mathtt{inl}(e))\ \mathtt{in}\ \langle h, \langle a, \langle\langle c, c\rangle, x\rangle\rangle\rangle \\
2' &: \langle h, \langle g, \langle a, \langle d, x\rangle\rangle\rangle\rangle \mapsto \mathtt{let}\ h' = \mathtt{in}_{E+H\times G \lhd H}(\mathtt{inr}(\langle h, g\rangle))\ \mathtt{in}\ \langle h', \langle a, \langle\langle d, d\rangle, x\rangle\rangle\rangle \\
3,4 &: \langle e, y\rangle \mapsto \mathtt{let}\ h = \mathtt{in}_{E+H\times G \lhd H}(\mathtt{inl}(e))\ \mathtt{in}\ \langle h, y\rangle \\
3',4' &: \langle h, \langle g, y\rangle\rangle \mapsto \mathtt{let}\ h' = \mathtt{in}_{E+H\times G \lhd H}(\mathtt{inr}(\langle h, g\rangle))\ \mathtt{in}\ \langle h', y\rangle \\
5,6,7 &: \langle h, y\rangle \mapsto \mathtt{let}\ \mathtt{in}_{E+H\times G \lhd H}(z) = h\ \mathtt{in}\ \mathtt{case}\ z\ \mathtt{of}\ \mathtt{inl}(e) \Rightarrow \mathit{exit\_top}(\langle e, y\rangle) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mathtt{inr}(\langle h, g\rangle) \Rightarrow \mathit{exit\_bottom}(\langle h, \langle g, y\rangle\rangle) \\
8 &: \langle\langle h, x\rangle, y\rangle \mapsto \langle h, \langle x, y\rangle\rangle \\
9 &: \langle h, \langle x, y\rangle\rangle \mapsto \langle\langle h, x\rangle, y\rangle
\end{aligned}$$

Figure 8: Translation of Recursion

- The term from Example 4.1 may be annotated as follows: $\vdash_A \lambda y.\, y + 3 \colon B \cdot (\mathbb{N} \xrightarrow{\mathtt{unit}}_C \mathbb{N})$.
  Its (slightly simplified) low-level translation is:

```
$eval.entry(a\colon A,\ u\colon \mathtt{unit}) = eval.ret(a, ())$
$apply.entry(b\colon B,\ c\colon C,\ \langle f, y\rangle \colon \mathtt{unit} \times \mathtt{int}) =$
    $\mathtt{let}\ r = \mathtt{add}(y, 1)\ \mathtt{in}\ apply.ret(b, c, r)$
```

- The term from Example 4.1 may be annotated as follows: $x\colon B \cdot (\mathbb{N} \xrightarrow{C}_D \mathbb{N}) \vdash_A \lambda y.\, x\ (y + 1) \colon B \cdot (\mathbb{N} \xrightarrow{C}_D \mathbb{N})$.
  Its (slightly simplified) low-level translation is:

```
$eval.entry(a\colon A,\ x\colon C) = eval.ret(a, x)$
$apply.entry(b\colon B,\ d\colon D,\ \langle f, y\rangle \colon C \times \mathtt{int}) =$
    $\mathtt{let}\ z = \mathtt{add}(y, 1)\ \mathtt{in}$
    $x.apply.entry(b, d, \langle f, z\rangle)$
$x.apply.ret(b\colon B,\ d\colon D,\ r\colon \mathtt{int}) = apply.ret(b, d, r)$
```

  Here, the labels $x.apply.entry$ and $x.apply.ret$ are the entry and exit labels of $\mathcal{I}[\![B \cdot (\mathbb{N} \xrightarrow{C}_D \mathbb{N})]\!]$ belonging to the variable x. In all definitions, all but the last argument are callee-save arguments.

- Next we give the translation of the term from Example 4.3. In this term, the variable x is used twice. To understand the translation of the term, it may be useful to first look at the translation without the contraction on x, i.e. where two different variables x_1 and x_2 are used for the two occurrences of x:

  $$x_1 \colon \mathtt{unit} \cdot (\mathbb{N} \xrightarrow{C_1}_{A \times C_2} \mathbb{N}),\ x_2 \colon B \cdot (\mathbb{N} \xrightarrow{C_2}_D \mathbb{N}) \vdash_A \mathtt{ifz}\ (x_1\ 3)\ \mathtt{then}\ \lambda y.\, y + 1\ \mathtt{else}\ \lambda.x.\, x_2\ (y + 1).\colon B \cdot (\mathbb{N} \xrightarrow{\mathtt{unit}+C_2}_D \mathbb{N}).$$

  The translation of this judgement is as follows. Here again, in all definition only the last argument is interesting; the rest are all callee-save arguments.

```
$eval.entry(a\colon A,\ \langle f_1, f_2\rangle \colon C_1 \times C_2) = x_1.apply.entry((), \langle a, f_2\rangle, \langle f_1, 3\rangle)$    // Evaluation starts by applying $x_1$ to 3.
$x_1.apply.ret(u\colon \mathtt{unit}, \langle a, f_2\rangle \colon A \times C_2, r\colon \mathtt{int}) =$
    $\mathtt{let}\ z = \mathtt{eq}(r, 0)\ \mathtt{in}$
    $\mathtt{case}\ z\ \mathtt{of}\ \mathtt{inl}() \Rightarrow eval.ret(a, \mathtt{inl}())$
                $\mathtt{inr}() \Rightarrow eval.ret(a, \mathtt{inr}(f_2))$
$apply.entry(b\colon B,\ d\colon D,\ \langle f, y\rangle \colon (\mathtt{unit} + C_2) \times \mathtt{int}) =$
    $\mathtt{let}\ y' = \mathtt{add}(y, 1)\ \mathtt{in}$
    $\mathtt{case}\ f\ \mathtt{of}\ \mathtt{inl}() \Rightarrow apply.ret(b, d, y')$
                $\mathtt{inr}(f_2) \Rightarrow x_2.apply.entry(b, d, \langle f_2, y'\rangle)$
$x_2.apply.ret(b\colon B,\ d\colon D,\ r\colon \mathtt{int}) = apply.ret(b, d, r)$
```

// Function x_1 returns the result r of its application. The other arguments are callee-save arguments that are unchanged since the jump to $x_1.apply.entry$. With the value r, we can continue evaluation.

// A caller asks to apply function value f to y. Depending on the tag of the function, we can either return the result immediately (left case), or we need jump to the application code for function x_2.

  Let us now come to the term from Example 4.3. It may be typed as

  $$x\colon (\mathtt{unit} + B) \cdot (\mathbb{N} \xrightarrow{C}_{(A \times C) \cup D} \mathbb{N}) \vdash \mathtt{ifz}\ (x\ 3)\ \mathtt{then}\ \lambda y.\, y + 1\ \mathtt{else}\ \lambda.x.\, x\ (y + 1) \colon B \cdot (\mathbb{N} \xrightarrow{\mathtt{unit}+C}_D \mathbb{N}),$$

  which translates to:

```
$eval.entry(a\colon A,\ f\colon C) = x.apply.entry(\mathtt{inl}(), \mathtt{in}_{A \times C, (A \times C) \cup D}(\langle a, f\rangle), \langle f, 3\rangle)$    // Evaluation starts by applying $x$ to 3.
$x.apply.ret(s\colon (\mathtt{unit} + B),\ u\colon (A \times C) \cup D,\ r\colon \mathtt{int}) =$
    $\mathtt{case}\ s\ \mathtt{of}\ \mathtt{inl}() \Rightarrow \mathtt{let}\ \mathtt{in}_{A \times C, (A \times C) \cup D}(a, f) = u\ \mathtt{in}$
                        $\mathtt{let}\ z = \mathtt{eq}(r, 0)\ \mathtt{in}$
                        $\mathtt{case}\ z\ \mathtt{of}\ \mathtt{inl}() \Rightarrow eval.ret(a, \mathtt{inl}())$
                                    $\mathtt{inr}() \Rightarrow eval.ret(a, \mathtt{inr}(f)))$
                $\mathtt{inr}(b) \Rightarrow \mathtt{let}\ \mathtt{in}_{D, (A \times C) \cup D}(d) = u\ \mathtt{in}$
                        $apply.ret(b, d, r)$
$apply.entry(b\colon B, d\colon D, \langle f, y\rangle \colon (\mathtt{unit} + C) \times \mathtt{int}) =$
    $\mathtt{let}\ y' = \mathtt{add}(y, 1)\ \mathtt{in}$
    $\mathtt{case}\ f\ \mathtt{of}\ \mathtt{inl}() \Rightarrow apply.ret(b, d, y')$
                $\mathtt{inr}(f_2) \Rightarrow x.apply.entry(\mathtt{inr}(b), \mathtt{in}_{D, (A \times C) \cup D}(d), \langle f, y'\rangle)$
```

// The application of x returns r; the callee-save arguments s contains the information, which of the two applications in the terms it was (note the actual arguments of $x.apply.entry$ in the other blocks). A case distinction jumps to the right destination.
(In this block we allow ourselves to continue with let-definitions in the branches of the case. Strictly speaking, only jumps are allowed there and one would need to define new blocks for the two branches.)

Figure 9: Translation of Examples 4.1-4.3

$$\textsc{var}\ \frac{}{x\colon \mathcal{X} \vdash_A x\colon \mathcal{X}} \qquad \textsc{w}\ \frac{\Gamma \vdash_A t\colon \mathcal{X}}{\Gamma, \Delta \vdash_A t\colon \mathcal{X}} \qquad \textsc{e}\ \frac{\Gamma,\ x\colon \mathcal{X},\ y\colon \mathcal{Y},\ \Delta \vdash_A t\colon \mathcal{Z}}{\Gamma,\ y\colon \mathcal{Y},\ x\colon \mathcal{X},\ \Delta \vdash_A t\colon \mathcal{Z}} \qquad \textsc{c}\ \frac{\Gamma,\ x\colon B\cdot X,\ y\colon C\cdot X \vdash_A t\colon \mathcal{Y}}{\Gamma,\ z\colon D\cdot X \vdash_A t[z/x,\, z/y]\colon \mathcal{Y}}\ (B+C) \lhd D$$

$$\textsc{zero}\ \frac{}{\vdash_A \mathtt{zero}\colon B\cdot\mathbb{N}} \qquad \textsc{succ}\ \frac{\vdash_A t\colon B\cdot\mathbb{N}}{\vdash_A \mathtt{succ}(t)\colon B\cdot\mathbb{N}} \qquad \textsc{pred}\ \frac{\vdash_A t\colon B\cdot\mathbb{N}}{\vdash_A \mathtt{pred}(t)\colon B\cdot\mathbb{N}} \qquad \textsc{abs}\ \frac{\Gamma,\ x\colon \mathcal{X} \vdash_A t\colon \mathcal{Y}}{\Gamma' \vdash_F \mathtt{fn}\, x \Rightarrow t\colon E\cdot(\mathcal{X} \xrightarrow{D}_A \mathcal{Y})}\ C[\![\Gamma]\!] \lhd D,\ E\cdot\Gamma \lhd \Gamma'$$

$$\textsc{app}\ \frac{\Gamma \vdash_A s\colon D\cdot(\mathcal{X} \xrightarrow{C}_U \mathcal{Y}) \qquad \Delta \vdash_B t\colon \mathcal{X}}{\Gamma,\ \Delta \vdash_U s\ t\colon \mathcal{Y}}\ U\times C[\![\Delta]\!] \lhd A,\ U\times C \lhd B,\ \mathtt{unit} \lhd D$$

$$\textsc{if}\ \frac{\Gamma \vdash_B s\colon \mathbb{N} \quad \Delta_1 \vdash_A t_1\colon \mathcal{X}_1 \quad \Delta_2 \vdash_A t_2\colon \mathcal{X}_2 \quad \mathcal{X}_1 \curlyvee \mathcal{X}_2 = \mathcal{X}}{\Gamma, \Delta_1, \Delta_2 \vdash_A \mathtt{ifz}\ s\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2\colon \mathcal{X}}\ A\times C[\![\Delta_1]\!]\times C[\![\Delta_2]\!] \lhd B$$

$$\textsc{rec}\ \frac{\Gamma,\ f\colon G\cdot(\mathcal{X} \xrightarrow{D}_A \mathcal{Y}),\ x\colon \mathcal{X} \vdash_A t\colon \mathcal{Y}}{\Gamma' \vdash_F \mathtt{fix}\, f\ x \Rightarrow t\colon E\cdot(\mathcal{X} \xrightarrow{D}_A \mathcal{Y})}\ C[\![\Gamma]\!] \lhd D,\ (E+(H\times G)) \lhd H,\ H\cdot\Gamma \lhd \Gamma'$$

Figure 10: Algorithmic Annotated PCF typing

that corresponds to t with the free variables $x_1, \ldots, x_k$ bound to the fragments.

The typing rules for INT′ are shown in Figs. 11–16. This variant of INT′ has first-class types $A \cdot X$ with rules adapted from linear logic. We omit rules for union types and recursive types, as we need these types only to solve $\lhd$-constraints. The rules make reference to a judgement $\Omega \vdash A \lhd B$, which we define to mean that $A' \lhd B'$ holds, where A' and B' are the closed types obtained from A and B by replacing each type variable from Ω with its upper bound.

The translation from INT′ to the low-level language is defined directly by induction on the typing derivation. We refer to [14, 16] for details on the translation for all but existential types. In most cases the translation is essentially forced by the interfaces. Looking at the interfaces of the programs for the premises of any rule often suggests how to construct the program in the conclusion. Let us outline a few examples.

The type $X \multimap Y$ can be understood as a type that explains low-level program linking. We outline the case for application of a closed term f of type $X \multimap Y$ to a closed argument. The term f represents a low-level program fragment $p : X^+ + Y^- \to X^- + Y^+$. It can be understood as a fragment that is intended to be linked to a fragment with interface X; the result of linking is a fragment with interface Y. Suppose we have closed a term g of type X, which represents a low-level program fragment $q\colon X^- \to X^+$. Then function application $f\ g$ then represents the low-level program $\mathtt{iapp}(p, q)$ defined in Fig. 17.

The type $A \to X$ captures value passing. A term f of type $A \to X$ represents a low-level program fragment $r\colon A \times X^- \to X^+$. It expects to be given a value of type A together with any input. The fragment can be seen as a function from A to X as follows. Suppose we have a value $v\colon A$. Then we can construct the program fragment $\mathsf{vapp}(r, v)$ from Fig. 17.

The introduction for existential types is interpreted using pack from Fig. 17.

The interpretation of the abstraction $\lambda x{:}X.\, t$ is essentially the identity and determined by the types. The abstractions $\mathtt{fn}\, x \Rightarrow t$ and $\Lambda\alpha.\, t$ are interpreted using the combinators vabs and tabs from Fig. 17. The box around p should be understood as an operation that binds the variable x that may appear free in p. It is defined by modifying each block, so that the value of the variable x is passed around unchanged as the first argument of each block, rather than appearing freely. For example $f(y) = g(v)$ becomes $f(z) = \mathtt{let}\ \langle x, y\rangle = z\ \mathtt{in}\, g(\langle x, v\rangle)$ and other blocks are changed analogously. The program differs from $A \cdot p$ in that the first component may be read in the places where x is free in p.

For existential types, the introduction rule is interpreted using pack. The elimination rule is interpreted by substituting A for α in $(\!|t|\!)$ (the interpretation of the derivation of the right-hand premise) and then connecting the program $(\!|s|\!)$.

In addition to the terms in the typing rules, we assume a constant for tail composition

$$\mathtt{comp}_{A,B,C}\colon \mathtt{unit}\cdot(A \to [B]) \multimap \mathtt{unit}\cdot(B \to [C]) \multimap (A \to [C])$$

(composition is definable in INT′, but it would keep A in a callee-save argument when evaluating the second function, i.e. the subexponential for the second argument would not be $\mathtt{unit}$), a strength

$$\mathtt{str}_{A,B,C}\colon B\cdot(A \to [C]) \multimap (A \times B \to [C \times B])$$

and a fixed-point combinator

$$\mathtt{fix}_{X,A,B}\colon B\cdot(A\cdot X \multimap X) \multimap X$$

where $\mathtt{unit} + (B \times A) \lhd B$.

In this paper, we do not add low-level annotations to the terms of INT′. The terms express only what a program computes. The same term may have different typing derivations, which correspond to different low-level implementations of the same behaviour.

In the rest of this section we make this precise in which sense the low-level program obtained from a derivation implements the term. We first define a denotational semantics for the terms, which defines the behaviour of terms. Then we define how low-level programs implement this behaviour. Informally, the rest of this section should be clear. It just says that the low-level implementation of INT′ correctly implements its terms when one ignores the type annotations.

D.2 Denotational Semantics

The meaning of terms is defined by a standard domain theoretic semantics. We interpret INT′ types by ω-cpos and terms by continuous functions.

$$\frac{}{\Omega \mid \Gamma \mid x\colon X \vdash x\colon X} \qquad \frac{\Omega \mid \Gamma \mid \Phi \vdash t\colon Y}{\Omega \mid \Gamma \mid \Phi,\ x\colon X \vdash t\colon Y} \qquad \frac{\Omega \mid \Gamma \mid \Phi,\ \Psi,\ \Psi',\ \Phi' \vdash t\colon Z}{\Omega \mid \Gamma \mid \Phi,\ \Psi',\ \Psi,\ \Phi' \vdash t\colon Z}$$

Figure 11: Typing Rules for INT′: Axiom and Structural Rules

$$\frac{\Gamma \vdash v\colon A}{\Omega \mid \Gamma \mid - \vdash \mathtt{return}(v)\colon [A]} \qquad \frac{\Omega \mid \Gamma \mid \Phi \vdash s\colon [A] \qquad \Omega \mid \Gamma,\ x\colon A \mid \Psi \vdash t\colon [B]}{\Omega \mid \Gamma \mid \Phi,\ A\cdot\Psi \vdash \mathtt{let}\ x = s\ \mathtt{in}\ t\colon [B]}$$

$$\frac{\Gamma \vdash v\colon A+B \qquad \Omega \mid \Gamma,\ x\colon A \mid \Phi \vdash s\colon X \qquad \Omega \mid \Gamma,\ y\colon B \mid \Phi \vdash t\colon X}{\Omega \mid \Gamma \mid \Phi \vdash \mathtt{case}\ v\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow s;\ \mathtt{inr}(y) \Rightarrow t\colon X}$$

Figure 12: Typing Rules for INT′: Basic Computations

$$\frac{\Omega \mid \Gamma,\ x\colon A \mid \Phi \vdash t\colon X}{\Omega \mid \Gamma \mid A\cdot\Phi \vdash \mathtt{fn}\ x \Rightarrow t\colon A \to X} \qquad \frac{\Omega \mid \Gamma \mid \Phi \vdash t\colon A \to X \qquad \Omega \mid \Gamma \vdash v\colon A}{\Omega \mid \Gamma \mid \Phi \vdash t(v)\colon X}$$

Figure 13: Typing Rules for INT′: Value Passing

$$\frac{\Omega \mid \Gamma \mid \Phi,\ x\colon X \vdash t\colon Y}{\Omega \mid \Gamma \mid \Phi,\ x\colon \mathtt{unit}\cdot X \vdash t\colon Y} \qquad \frac{\Omega \mid \Gamma \mid \Phi,\ x\colon A\cdot B\cdot X \vdash t\colon Y}{\Omega \mid \Gamma \mid \Phi,\ x\colon (A\times B)\cdot X \vdash t\colon Y} \qquad \frac{\Omega \mid \Gamma \mid \Phi \vdash t\colon X}{\Omega \mid \Gamma \mid A\cdot\Phi \vdash t\colon A\cdot X}$$

$$\frac{\Omega \mid \Gamma \mid \Phi,\ x\colon A\cdot X \vdash t\colon Y \qquad \Omega \vdash A \lhd B}{\Omega \mid \Gamma \mid \Phi,\ x\colon B\cdot X \vdash t\colon Y} \qquad \frac{\Omega \mid \Gamma \mid \Phi,\ x\colon A\cdot X,\ y\colon B\cdot X \vdash t\colon Y}{\Omega \mid \Gamma \mid \Phi,\ z\colon (A+B)\cdot X \vdash t[z/x, z/y]\colon Y}$$

Figure 14: Typing Rules for INT′: Subexponentials

A *type environment* ρ is a mapping from type variables to closed low-level types. Each low-level type is interpreted as the ordinary set $[\![A]\!]_\rho$ consisting of the set of closed values of type $A[\rho]$, where $(-)[\rho]$ is the type substitution that replaces α with $\rho(\alpha)$.

INT′ types are interpreted as ω-cpos. We define an ω-cpo $[\![X]\!]_\rho$ for each type X inductively as follows:

$$\begin{aligned}
[\![[A]]\!]_\rho &= ([\![A]\!]_\rho)_\perp\\
[\![A\cdot X]\!]_\rho &= [\![X]\!]_\rho\\
[\![X \multimap Y]\!]_\rho &= [\![X]\!]_\rho \Rightarrow [\![Y]\!]_\rho\\
[\![\forall\alpha \lhd A.\, X]\!]_\rho &= \textstyle\prod_{B\ \mathrm{type}} [\![X]\!]_{\rho[\alpha\mapsto B]}\\
[\![A \to X]\!]_\rho &= [\![X]\!]_\rho^{[\![A]\!]_\rho}\\
[\![\exists\alpha \lhd A.\, X]\!]_\rho &= \textstyle\sum_{B\ \mathrm{type}} [\![X]\!]_{\rho[\alpha\mapsto B]}
\end{aligned}$$

Here, $\Rightarrow$ denotes the ω-cpo of continuous functions, $\prod$ and X^A denote products and Σ denotes coproducts. Notice that the denotational semantics ignores low-level annotations, so it is essentially a semantics of Idealised Algol.

Terms are interpreted relative to a type environment ρ, a value environment σ and a module environment ϕ. A value environment assigns meaning to value variables. A variable y of type A is mapped to an element of $[\![A]\!]_\rho$. A module environment maps INT′ variables to their denotation; a variable of type x is mapped to an element of $[\![X]\!]_\rho$. The interpretation of a term t of type X is then defined as a function $[\![t]\!]$ that maps ρ, σ and ϕ to $[\![t]\!]_{\rho,\sigma,\phi} \in [\![X]\!]_\rho$. It is defined by induction on t in Fig. 18. We use π_1 and π_2 to denote the strict projection functions.

D.3 Relating Implementation and Denotation

Define a family of relations $\sim_{X,\rho}$ between low-level programs and elements of $[\![X]\!]_\rho$ by induction on the type X:

- $p \sim_{[A],\rho} d$ iff: $p\colon \mathtt{unit} \to A[\rho]$ and $p\colon () \mapsto v$ iff $d = \lfloor v \rfloor$.
- $p \sim_{A\cdot X,\rho} d$ iff: there exists q with $p = A\cdot q$ and $q \sim_{X,\rho} d$.
- $p \sim_{X\otimes Y,\rho} d$ iff: there exist p_1, p_2 and d_1, d_2 with $p = p_1 \otimes p_2$ and $d = \langle d_1, d_2\rangle$ and $p_1 \sim_{X,\rho} d_1$ and $p_2 \sim_{Y,\rho} d_2$.
- $p \sim_{X\multimap Y,\rho} f$ iff: whenever $q \sim_{X,\rho} d$, then $\mathtt{iapp}(p,q) \sim_{Y,\rho} f(d)$.
- $p \sim_{A\to X,\rho} f$ iff: for all $v \in [\![A]\!]_\rho$, we have $\mathtt{vapp}(p,v) \sim_{X,\rho} f(v)$.
- $p \sim_{\forall\alpha\lhd A.\,X,\rho} f$ iff: we have $\mathtt{tapp}_X(p,B) \sim_{X,\rho[\alpha\mapsto B]} f(B)$ for all $B \lhd A$.
- $p \sim_{\exists\alpha\lhd A.\,X,\rho} f$ iff: there exist B, q, d, such that $f = (B,d)$ and $p = \mathtt{pack}(q,B)$ and $q \sim_{X,\rho[\alpha\mapsto B]} d$.

This definition is extended to terms in context in a logical way. Let $[\![\Omega]\!]$ be the set of all type environments ρ, such that $(\alpha \lhd B) \in \Omega$ implies $\rho(\alpha) \lhd B$. If Γ is the value context $y_1\colon B_1, \dots, y_n\colon B_n$, then $[\![\Gamma]\!]_\rho$ is a the set of environments σ that map y_i to $[\![B_i]\!]_\rho$, for $i = 1,\dots,n$. If Φ is $x_1\colon X_1, \dots, x_n\colon X_n$, and ϕ is an environment mapping each x_i to $[\![X_i]\!]_\rho$, and if we have $p_i \sim_{X_i,\rho} \phi(x_i)$ for all $i = 1,\dots,n$, then we write $\vec{p} \sim_{\Phi,\rho} \phi$. With this notation, we extend $\sim$ to the denotations of terms as follows. Write $q \sim_{\Omega|\Gamma|\Phi,X} f$ if, whenever $\rho \in [\![\Omega]\!]$, $\sigma \in [\![\Gamma]\!]_\rho$ and $\vec{p} \sim_{\Phi,\rho} \phi$, then we have $\mathtt{iapp}(q[\rho][\sigma], \vec{p}) \sim_{X,\rho} f_{\rho,\sigma,\phi}$.

Finally, in order to account for recursion, we close the relation under limits. We write $p \approx_{\Omega|\Gamma|\Phi,X} f$ if there exist ω-chains $(p_i)_{i\geq 0}$, $(f_i)_{i\geq 0}$ with $p = \bigsqcup_i p_i$ and $f = \bigsqcup_i f_i$ and $p_i \sim_{\Omega|\Gamma|\Phi,X} f_i$ for all i. Here, we consider low-level programs as partial functions from low-level values to low-level values with the usual ordering.

With these definitions, we have the following fundamental lemma, which says that the low-level implementation of a typing derivation for a term t implements the denotation of t, according to $\approx$. For

$$\frac{\Omega \mid \Gamma \mid \Phi,\ x : X \vdash t : Y}{\Omega \mid \Gamma \mid \Phi \vdash \lambda x : X.\, t : X \multimap Y} \qquad \frac{\Omega \mid \Gamma \mid \Phi \vdash s : X \multimap Y \qquad \Omega \mid \Gamma \mid \Psi \vdash t : X}{\Omega \mid \Gamma \mid \Phi,\ \Psi \vdash s\ t : Y}$$

$$\frac{\Omega \mid \Gamma \mid \Phi \vdash s : X \qquad \Omega \mid \Gamma \mid \Psi \vdash t : Y}{\Omega \mid \Gamma \mid \Phi,\ \Psi \vdash s \otimes t : X \otimes Y} \qquad \frac{\Omega \mid \Gamma \mid \Phi \vdash s : X \otimes Y \qquad \Omega \mid \Gamma \mid \Psi,\ x : X,\ y : Y \vdash t : Z}{\Omega \mid \Gamma \mid \Phi,\ \Psi \vdash \mathtt{let}\ \langle x, y\rangle = s\ \mathtt{in}\ t : Z}$$

Figure 15: Typing Rules for INT′: Functions and Pairs

$$\frac{\Omega, \alpha \lhd A \mid \Gamma \mid \Phi \vdash t : X}{\Omega \mid \Gamma \mid \Phi,\ \Psi \vdash \Lambda\alpha.\, t : \forall \alpha \lhd A.\, X} \qquad \frac{\Omega \mid \Gamma \mid \Phi \vdash t : \forall \alpha \lhd A.\, X \qquad \Omega \vdash B \lhd A}{\Omega \mid \Gamma \mid \Phi,\ \Psi \vdash t\ B : X[B/\alpha]}$$

$$\frac{\Omega \mid \Gamma \mid \Phi \vdash t : X[B/\alpha] \qquad \Omega \vdash B \lhd A}{\Omega \mid \Gamma \mid \Phi,\ \Psi \vdash \mathsf{pack}(B, t) : \exists \alpha \lhd A.\, X} \qquad \frac{\Omega \mid \Gamma \mid \Phi \vdash s : \exists \alpha \lhd A.\, X \qquad \Omega,\ \alpha \lhd A \mid \Gamma \mid \Psi,\ x : X \vdash t : Y}{\Omega \mid \Gamma \mid \Phi,\ \Psi \vdash \mathtt{let}\ \mathsf{pack}(\alpha, x) = s\ \mathtt{in}\ t : Y}$$

Figure 16: Typing Rules for INT′: Quantification

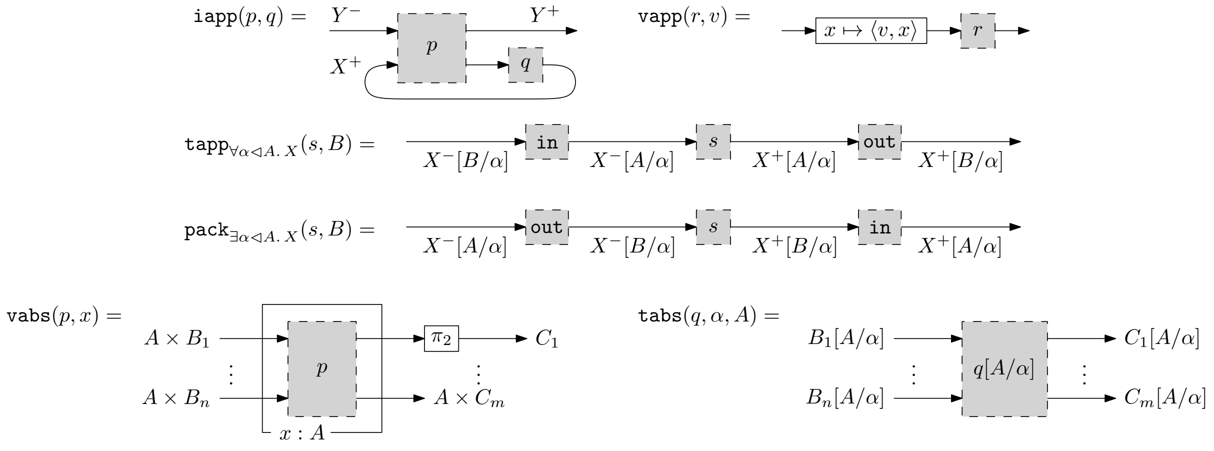


Figure 17: Combinators for the Interpretation of INT′

$$\begin{aligned}
[\![x]\!]_{\rho,\sigma,\phi} &= \sigma(x)\\
[\![s \otimes t]\!]_{\rho,\sigma,\phi} &= \langle [\![s]\!]_{\rho,\sigma,\phi}, [\![t]\!]_{\rho,\sigma,\phi}\rangle\\
[\![\mathtt{let}\ \langle x, y\rangle = s\ \mathtt{in}\ t]\!]_{\rho,\sigma,\phi} &= [\![t]\!]_{\rho,\sigma,\phi[x\mapsto\pi_1([\![s]\!]_{\rho,\sigma,\phi}),\, y\mapsto\pi_2([\![s]\!]_{\rho,\sigma,\phi})]}\\
[\![\lambda x : X.\, t]\!]_{\rho,\sigma,\phi} &= (d \in [\![X]\!]_\rho \mapsto [\![t]\!]_{\rho,\sigma,\phi[x\mapsto d]})\\
[\![s\ t]\!]_{\rho,\sigma,\phi} &= [\![s]\!]_{\rho,\sigma}([\![t]\!]_{\rho,\sigma,\phi})\\
[\![\mathsf{fn}\ x : A \Rightarrow t]\!]_{\rho,\sigma,\phi} &= (v \in [\![A]\!]_\rho \mapsto [\![t]\!]_{\rho,\sigma[x\mapsto v],\phi})\\
[\![t(v)]\!]_{\rho,\sigma,\phi} &= [\![t]\!]_{\rho,\sigma,\phi}(v[\sigma])\\
[\![\mathsf{return}(v)]\!]_{\rho,\sigma,\phi} &= v[\sigma]\\
[\![\mathtt{let}\ x = s\ \mathtt{in}\ t]\!]_{\rho,\sigma,\phi} &= \begin{cases}\bot & \text{if } [\![s]\!]_{\rho,\sigma,\phi} = \bot\\ [\![t]\!]_{\rho,\sigma[x\mapsto v],\phi} & \text{if } [\![s]\!]_{\rho,\sigma,\phi} = v\end{cases}\\
[\![\mathsf{case}\ v\ \mathsf{of}\ \mathtt{inl}(x) \Rightarrow s;\ \mathtt{inr}(x) \Rightarrow t]\!]_{\rho,\sigma,\phi} &= \begin{cases}[\![s]\!]_{\rho,\sigma[x\mapsto w],\phi} & \text{if } v[\sigma] = \mathtt{inl}(w)\\ [\![t]\!]_{\rho,\sigma[x\mapsto w],\phi} & \text{if } v[\sigma] = \mathtt{inr}(w)\end{cases}\\
[\![\Lambda\alpha.\, t]\!]_{\rho,\sigma,\phi} &= (B \mapsto [\![t]\!]_{\rho[\alpha\mapsto B],\sigma,\phi})\\
[\![t\ B]\!]_{\rho,\sigma,\phi} &= [\![t]\!]_{\rho,\sigma,\phi}([\![B]\!]_\rho)\\
[\![\mathsf{pack}(A, t)]\!]_{\rho,\sigma,\phi} &= ([\![A]\!]_\rho, [\![t]\!]_{\rho,\sigma,\phi})\\
[\![\mathtt{let}\ \mathsf{pack}(\alpha, x) = s\ \mathtt{in}\ t]\!]_{\rho,\sigma,\phi} &= [\![t]\!]_{\rho[\alpha\mapsto\pi_1([\![s]\!]_{\rho,\sigma,\phi})],\sigma,\phi[x\mapsto\pi_2([\![s]\!]_{\rho,\sigma,\phi})]}\\
[\![\mathtt{fix}_X]\!]_{\rho,\sigma,\phi} &= \text{fixed point combinator on } [\![X]\!]_\rho
\end{aligned}$$

Figure 18: Denotational Semantics of INT′ Terms

closed values of base type, this means that the low-level program returns the correct result. The lemma is proved by induction on the derivation.

Lemma D.1. *If* Π *derives* $\Omega \mid \Gamma \mid \Phi \vdash t\colon X$*, then* $(\!|\Pi|\!) \approx_{\Omega|\Gamma|\Phi,X} [\![t]\!]$.

In the rest of this section, give an outline for the proof of this lemma, which uses the following substitution lemmas.

Lemma D.2 (Value Substitution). $\mathsf{vapp}(\mathsf{vabs}(p,x),v) = p[v/x]$.

Lemma D.3 (Type Substitution). *For all* $p\colon \vec{B} \to \vec{C}$ *whose free value variables all have a type not containing* α*, we have:*

$$\mathsf{tapp}_{\vec{B},\vec{C}}(\mathsf{tabs}(p,\alpha,B),A) = p[A/\alpha]$$

Proof of Lemma D.1. The proof goes by induction on the derivation Π, of which we show a few cases.

Case:

$$\frac{\Omega \mid \Gamma \vdash v\colon A}{\Omega \mid \Gamma \mid - \vdash \mathtt{return}(v)\colon [A]}$$

The program $(\!|\Pi|\!)$ is the block $(\langle\rangle \mapsto v)$. Suppose $\rho \in [\![\Omega]\!]$ and $\sigma \in [\![\Gamma]\!]_\rho$. Then, $\mathtt{iapp}((\!|\Pi|\!)[\rho][\sigma],\varepsilon)$ is $(\langle\rangle \mapsto v[\sigma])$.

By definition of the semantics, we have $[\![\mathtt{return}(v)]\!]_{\rho,\sigma,\phi} = v[\sigma]$, so we get $(\langle\rangle \mapsto v[\sigma]) \sim_{[A],\rho} v[\sigma]$, which is the same as $\mathtt{iapp}((\!|\Pi|\!)[\rho][\sigma],\varepsilon) \sim_{[A],\rho} [\![\mathtt{return}(v)]\!]_{\rho,\sigma,\phi}$. The required assertion follows by taking constant ω-chains.

Case:

$$\frac{\Omega \mid \Gamma,\, x\colon A \mid \Phi \vdash t\colon X}{\Omega \mid \Gamma \mid A \cdot \Phi \vdash \mathsf{fn}\, x \Rightarrow t\colon A \to X}$$

The induction hypothesis gives $(\!|\Pi_t|\!) \approx_{\Omega|\Gamma,x\colon A|\Phi,X} [\![t]\!]$, so there exist (p_i) and (f_i) with $(\!|\Pi_t|\!) = \bigsqcup p_i$ and $f = \bigsqcup f_i$ and $p_i \sim_{\Omega|\Gamma,x\colon A|\Phi,X} f_i$.

Assume $\rho \in [\![\Omega]\!]$, $\sigma \in [\![\Gamma]\!]_\rho$ and $\vec{p} \sim_{A\cdot\Phi} \phi$. By definition, $\vec{p}$ must have the form $A \cdot \vec{q}$, where $\vec{q} \sim_\Phi \phi$.

Let $v \in [\![A]\!]_\rho$. It suffices to show

$$\mathtt{iapp}(\mathsf{vabs}(p_i,x)[\rho][\sigma], A\cdot\vec{q}) \sim_{A\to X} (v \mapsto (f_i)_{\rho,\sigma[x\mapsto v],\phi}),$$

as we have $\bigsqcup_i \mathsf{vabs}(p_i,x) = \mathsf{vabs}((\!|\Pi_t|\!),x)$ and also

$$\begin{aligned}\bigsqcup_i (v \mapsto (f_i)_{\rho,\sigma[x\mapsto v],\phi}) &= (v \mapsto \bigsqcup_i (f_i)_{\rho,\sigma[x\mapsto v],\phi}) \\ &= (v \mapsto f_{\rho,\sigma[x\mapsto v],\phi}) \\ &= [\![\mathsf{fn}\, x \Rightarrow t]\!]_{\rho,\sigma,\phi}\end{aligned}$$

We have to show

$$\mathsf{vapp}(\mathtt{iapp}(\mathsf{vabs}(p_i,x)[\rho][\sigma], A\cdot\vec{q}),v) \sim_X (f_i)_{\rho,\sigma[x\mapsto v],\phi},$$

by definition of $\sim_{A\to X}$. We calculate:

$$\begin{aligned}&\mathsf{vapp}(\mathtt{iapp}(\mathsf{vabs}(p_i,x)[\rho][\sigma], A\cdot\vec{q}),v) \\ &= \mathsf{vapp}(\mathtt{iapp}(\mathsf{vabs}(p_i[\rho][\sigma],x), A\cdot\vec{q}),v) \\ &= \mathsf{vapp}(\mathsf{vabs}(\mathtt{iapp}(p_i[\rho][\sigma],q),x),v) \\ &= \mathtt{iapp}(p_i[\rho][\sigma, x\mapsto v],\vec{q})\end{aligned}$$

The hypothesis gives

$$\mathtt{iapp}(p_i[\rho][\sigma[x\mapsto v]],\vec{q}) \sim_X (f_i)_{\rho,\sigma[x\mapsto v],\phi},$$

which concludes this case.

Case:

$$\frac{\Omega \mid \Gamma \mid \Phi \vdash t\colon A \to X \qquad \Omega \mid \Gamma \vdash v\colon A}{\Omega \mid \Gamma \mid \Phi \vdash t(v)\colon X}$$

The induction hypothesis gives $(\!|\Pi_t|\!) \approx_{\Omega|\Gamma,x\colon A|\Phi,A\to X} [\![t]\!]$, so there exist $(p_i)_{i\geq 0}$ and $(f_i)_{i\geq 0}$ with $(\!|\Pi_t|\!) = \bigsqcup p_i$ and $f = \bigsqcup f_i$ and $p_i \sim_{\Omega|\Gamma,x\colon A|\Phi,A\to X} f_i$.

Let $\rho \in [\![\Omega]\!]$, $\sigma \in [\![\Gamma]\!]_\rho$ be given and assume $\vec{q} \sim_\Phi \phi$. In order to show the required $\mathtt{iapp}(\mathsf{vapp}([\![t]\!][\rho][\sigma], v[\sigma]),\vec{q}) \approx_X [\![t(v)]\!]_{\rho,\sigma,\phi}$ it suffices to show the assertion

$$\mathtt{iapp}(\mathsf{vapp}(p_i[\rho][\sigma], v[\sigma]),\vec{q}) \sim_X (f_i)_{\rho,\sigma,\phi}(v[\sigma])$$

for all i.

The induction hypothesis gives

$$\mathtt{iapp}(p_i[\rho][\sigma],\vec{q}) \sim_{A\to X} (f_i)_{\rho,\sigma,\phi},$$

which implies

$$\mathsf{vapp}(\mathtt{iapp}(p_i[\rho][\sigma],\vec{q}), v[\sigma]) \sim_X (f_i)_{\rho,\sigma,\phi}(v[\sigma]),$$

by definition. We now note

$$\mathsf{vapp}(\mathtt{iapp}(p_i[\rho][\sigma],\vec{q}), v[\sigma]) = \mathtt{iapp}(\mathsf{vapp}(p_i[\rho][\sigma], v[\sigma]),\vec{q}),$$

so that we get $\mathtt{iapp}(\mathsf{vapp}(p_i[\rho][\sigma], v[\sigma]),\vec{q}) \sim (f_i)_{\rho,\sigma,\phi}(v[\sigma])$, which is just what we had to show.

Case:

$$\frac{\Omega, \alpha \lhd A \mid \Gamma \mid \Phi \vdash t\colon X}{\Omega \mid \Gamma \mid \Phi,\, \Psi \vdash \Lambda\alpha.\, t\colon \forall\alpha \lhd A.\, X}$$

This case follows from the substitution lemma, from which we get the equation $\mathsf{tapp}(\mathsf{tabs}(p,\alpha,A),B) = p[B/\alpha]$.

Case:

$$\frac{\Omega \mid \Gamma \mid \Phi \vdash t\colon \exists\alpha\lhd A.\, X \qquad \Omega,\, \alpha \lhd A \mid \Gamma \mid \Psi,\, x\colon X \vdash s\colon Y}{\Omega \mid \Gamma \mid \Phi,\, \Psi \vdash \mathtt{let}\ \mathtt{pack}(\alpha,x) = t\ \mathtt{in}\ s\colon Y}$$

Let $\rho \in [\![\Omega]\!]$, $\sigma \in [\![\Gamma]\!]_\rho$ be given and assume $\vec{q} \sim_\Phi \phi$. and $\vec{r} \sim_\Psi \psi$.

The induction hypothesis gives us both

$$\mathtt{iapp}(p_i[\rho][\sigma],\vec{q}) \sim_{\exists\alpha\lhd A.\,X} (f_i)_{\rho,\sigma,\phi}$$

and

$$\mathtt{iapp}(r_i[\rho[\alpha\mapsto B]][\sigma],(\vec{r},p')) \sim_Y (g_i)_{\rho[\alpha\mapsto B],\sigma,(\psi,f')}$$

whenever $p' \sim_X f'$ for some chains with $(\!|\Pi_t|\!) = \bigsqcup p_i$ and $[\![t]\!] = \bigsqcup f_i$ and $(\!|\Pi_s|\!) = \bigsqcup r_i$ and $[\![s]\!] = \bigsqcup g_i$.

The program $\mathtt{iapp}((\!|\mathtt{let}\ \mathtt{pack}(\alpha,x) = t\ \mathtt{in}\ s|\!)[\rho][\sigma],(\vec{q},\vec{r}))$ is the same as the program

$$\mathtt{iapp}((\!|s|\!)[\rho,\alpha\mapsto A][\sigma],(\vec{q},\mathtt{iapp}((\!|t|\!)[\rho][\sigma],\vec{r}))),$$

by definition. So it suffices to show

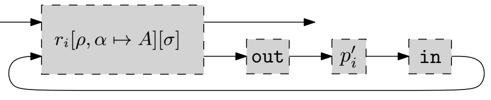
$$\begin{aligned}&\mathtt{iapp}(r_i[\rho,\alpha\mapsto A][\sigma],(\vec{q},\mathtt{iapp}(p_i[\rho][\sigma],\vec{r}))) \\ &\quad\sim (g_i)_{\rho,\sigma,(\psi,(f_i)_{\rho,\sigma,\phi})}.\end{aligned}$$

From $\mathtt{iapp}(p_i[\rho][\sigma],\vec{r}) \sim_{\exists\alpha\lhd A.\,X} (f_i)_{\rho,\sigma,\phi}$ we get $(f_i)_{\rho,\sigma,\phi} = (B, f'_i)$ and $\mathtt{iapp}(p_i[\rho][\sigma],\vec{r}) = \mathtt{pack}(p'_i,B)$ and $p'_i \sim_{X,\rho[\alpha\mapsto B]} f'_i$.

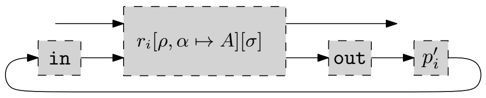
The hypothesis for s yields $\mathtt{iapp}(r_i[\rho[\alpha\mapsto B]][\sigma],(\vec{r},p'_i)) \sim_Y (g_i)_{\rho[\alpha\mapsto B],\sigma,(\psi,f'_i)}$. Now observe

$$\begin{aligned}&\mathtt{iapp}(r_i[\rho,\alpha\mapsto A][\sigma],(\vec{r},\mathtt{pack}(p'_i,B))) \\ &\quad= \mathtt{iapp}(r_i[\rho,\alpha\mapsto B][\sigma],(\vec{r},p'_i)),\end{aligned}$$

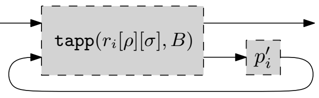
which can be seen using the following transformations and the above substitution lemma.



=



=



We obtain that $\mathtt{iapp}(r_i[\rho, \alpha \mapsto A][\sigma], (\vec{r}, \mathtt{pack}(p'_i, B)))$ is $\sim_Y$-related to $(g_i)_{\rho[\alpha\mapsto B],\sigma,(\psi,f'_i)}$. Because we also have $\mathtt{iapp}(p_i[\rho][\sigma], \vec{r}) = \mathtt{pack}(p'_i, B)$, this is just as required. □

D.4 Typed Closure Conversion from Annotated PCF to Int'

We spell out in detail the case for (APP) in the definition of $TCC'(\Pi)$ that is outlined in the main text.

$$\text{APP}\ \frac{\Gamma \vdash_{U\times C[\![\Delta]\!]} s\colon \mathcal{X} \xrightarrow{C}_U \mathcal{Y} \qquad \Delta \vdash_{U\times C} t\colon \mathcal{X}}{\Gamma, \Delta \vdash_U s\, t\colon \mathcal{Y}}$$

To simplify the notation, we consider an equivalent definition of the translation, where a typing derivation Π of $x_1\colon \mathcal{X}_1, \dots, x_k\colon \mathcal{X}_k \vdash_S t\colon Y$ in annotated PCF (Fig. 1) is translated to an INT′-derivation $TCC'(\Pi)$ of the following INT′ judgement:

$$\Omega \mid - \mid x_1\colon \mathcal{I}[\![\mathcal{X}_1]\!]_{\alpha_1}, \dots, x_k\colon \mathcal{I}[\![\mathcal{X}_k]\!]_{\alpha_k} \vdash TCC'(t)\colon \mathcal{M}[\![\mathcal{Y}]\!]_{S,C}$$

with

$$\Omega = \alpha_1 \lhd \mathcal{B}[\![\mathcal{X}_1]\!], \dots, \alpha_k \lhd \mathcal{B}[\![\mathcal{X}_k]\!]$$
$$C = \mathtt{unit} \times C[\![\mathcal{X}_1]\!]_{\alpha_1} \times \cdots \times C[\![\mathcal{X}_k]\!]_{\alpha_k}$$

(We have used an alternative definition in the main text, as the INT′ type system had to be omitted.)

In the case for (APP) the induction hypothesis derives

$$\Omega \mid - \mid \mathcal{I}[\![\Gamma]\!]_\Omega \vdash TCC'(s)\colon \mathcal{M}[\![\mathtt{unit} \cdot (\mathcal{X} \xrightarrow{C}_U \mathcal{Y})]\!]_{U\times C[\![\Delta]\!], C[\![\Gamma]\!]_\Omega}$$

and

$$\Theta \mid - \mid \mathcal{I}[\![\Delta]\!]_\Theta \vdash TCC'(t)\colon \mathcal{M}[\![\mathcal{X}]\!]_{U\times C, C[\![\Delta]\!]_\Theta}.$$

Both $TCC'(s)$ and $TCC'(t)$ are terms of existential type. Unpacking them and the pairs they contain gives us type variables $\alpha \lhd C$ and $\beta \lhd \mathcal{B}[\![\mathcal{X}]\!]$ and module variables

$$\begin{aligned} f&\colon \mathtt{unit} \cdot \left(\forall \beta \lhd \mathcal{B}[\![\mathcal{X}]\!].\, \mathcal{I}[\![\mathcal{X}]\!]_\beta \multimap \mathcal{M}[\![Y]\!]_{U,\alpha\times C[\![\mathcal{X}]\!]_\beta}\right)\\ e_f&\colon (U \times C[\![\Delta]\!]_\Theta) \cdot (C[\![\Gamma]\!]_\Omega \to [\alpha])\\ x&\colon \mathcal{I}[\![\mathcal{X}]\!]_\beta\\ e_x&\colon (U \times C) \cdot C[\![\Delta]\!]_\Theta \to [C[\![\mathcal{X}]\!]_\beta] \end{aligned}$$

The term $(f\ \beta\ x)$ can be given type $\mathcal{M}[\![Y]\!]_{U,\alpha\times C[\![\mathcal{X}]\!]_\beta}$. We define the required term of type $\mathcal{M}[\![Y]\!]_{U,C[\![\Gamma,\Delta]\!]_{\Omega,\Theta}}$ to be

$$\mathtt{let}\ \mathtt{pack}(\rho, \langle y, a\rangle) = f\ \beta\ x\ \mathtt{in}\ \mathtt{pack}(\rho, \langle y, e\rangle)$$

for a suitable term $e\colon U \cdot (C[\![\Gamma, \Delta]\!]_{\Omega,\Theta} \to [C[\![\mathcal{Y}]\!]])$.

We want to define e to be a function that takes an argument of type $C[\![\Gamma, \Delta]\!]_{\Omega,\Theta}$, then first extracts $C[\![\Gamma]\!]_\Omega$ and $C[\![\Delta]\!]_\Theta$-components and then evaluates e_f and e_x in this order. The result can then be used to invoke a, which returns the desired value of type $C[\![\mathcal{Y}]\!]$. While we evaluate e_f, we must keep $C[\![\Delta]\!]_\Theta$ for later use. Then, when we evaluate e_x, we must keep the value returned by e_f, so that at the end we have both this value and the value returned by e_x. We implement this as follows.

Using strength and composition constants, one can define a term $\mathtt{seq}_{A,B,C,D}$ for space-efficient sequential composition of two functions with the following type.

$$B \cdot (A \to [C]) \multimap C \cdot (B \to [D]) \multimap (A \times B) \to [C \times D]$$

(The strength gives terms $t_1 : B \cdot (A \to [C]) \multimap (B \times A \to [B \times C])$ and $t_2 : C \cdot (B \to [D]) \multimap (C \times B \to [C \times D])$. One then defines swapping functions $A \times B \to [B \times A]$ and $B \times C \to [C \times B]$ and uses the composition combinator.)

Using this term, we can define a suitable term for the sequential composition of e_f and e_x using the rules for subexponentials (note: $\alpha \lhd C$):

$$\cfrac{\cfrac{\cfrac{\vdots}{e_f\colon C[\![\Delta]\!] \cdot Z_1,\ e_x\colon \alpha \cdot Z_2 \vdash \mathtt{seq}\ e_f\ e_x\colon Z_3}}{e_f\colon C[\![\Delta]\!] \cdot Z_1,\ e_x\colon C \cdot Z_2 \vdash \mathtt{seq}\ e_f\ e_x\colon Z_2}}{e_f\colon (U \times C[\![\Delta]\!]) \cdot Z_1,\ e_x\colon (U \times C) \cdot Z_2 \vdash \mathtt{seq}\ e_f\ e_x\colon U \cdot Z_3}$$

In this derivation, we use the abbreviations $Z_1 = A \to [\alpha]$, $Z_2 = C[\![\Delta]\!]_\Theta \to [C[\![X]\!]]$ and $Z_3 = A \times C[\![\Delta]\!]_\Theta \to [\alpha \times C[\![X]\!]]$.

This shows that the types of e_f and e_x are sufficient to define a term implementing sequential evaluation of e_x after e_f with type $U \cdot (C[\![\Gamma, \Delta]\!]_{\Omega,\Theta} \to [\alpha \times C[\![X]\!]_\beta])$. Since the variable a has type $U \cdot (\alpha \times C[\![\Gamma]\!]_\Omega \to [C[\![\mathcal{Y}]\!]])$, which means that we can complete the definition of e with the desired type.

We end with a brief outline of Theorem 8.1.

THEOREM 8.1. *Suppose Π derives $\vdash_A t\colon B \cdot \mathbb{N}$. Then t reduces to a value v in a standard call-by-value operational semantics if and only if we have $(\![TCC'(\Pi)]\!)\colon \langle a, ()\rangle \mapsto \langle a, v\rangle$ for any closed low-level value $a\colon A$.*

The prove it, one first shows that $TCC'(\Pi)$ is correct with respect to the equational theory induces by the denotational semantics of INT′. This means that for reasoning one can completely ignore low-level annotations, and that all equations used in [17] are available in this sense. This means that one can carry out the equational reasoning of [17] unchanged: One defines two logical relations $\leq_\mathcal{X}$ and $\geq_\mathcal{X}$ by induction on the type $\mathcal{X}$, such that $TCC'(\Pi) \leq_\mathcal{X} [\![t]\!]$ expresses that $TCC'(\Pi)$ implements only behaviour that is found in the denotational semantics $[\![t]\!]$ of t, and $TCC'(\Pi) \geq_\mathcal{X} [\![t]\!]$ expresses that $TCC'(\Pi)$ implements at least all behaviour found in $[\![t]\!]$. To deal with the more general (IF), e.g. in $\leq_\mathcal{X}$, one shows that $(A_i, a_i, c_i) \leq_{\mathcal{X}_i} f_i$ for $i = 1, 2$ implies $(A_1 + A_2, \mathtt{join}(a_1 \otimes a_2), d_i) \leq_\mathcal{X} f_i$, where $d_1 = \mathtt{inl}(c_1)$ and $d_2 = \mathtt{inr}(c_2)$ and where $\mathtt{join}\colon \mathcal{I}[\![\mathcal{X}_1]\!]_{A_1} \otimes \mathcal{I}[\![\mathcal{X}_2]\!]_{A_2} \multimap \mathcal{I}[\![\mathcal{X}]\!]_{A_1+A_2}$ is the term used in the interpretation of (IF). The reasoning goes as in [17]. Using Lemma D.1, one then concludes from $TCC'(\Pi) \leq_\mathcal{X} [\![t]\!]$ and $TCC'(\Pi) \geq_\mathcal{X} [\![t]\!]$ that, for any closed a, we have $(\![TCC'(\Pi)]\!)\colon \langle a, ()\rangle \mapsto \langle a, v\rangle$ if and only if $[\![t]\!] = \lfloor v \rfloor$. The result then follows from soundness and adequacy of the denotational semantics of PCF.