Defunctionalisation as Typed Closure Conversion: Compositional Reasoning and Specification

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

Abstract. We study the problem of translating from call-by-value pcf to a typed 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 how such compositional reasoning can be supported in a realistic defunctionalisation method. We do so by decomposing defunctionalisation into a number of steps that each allows compositional reasoning. The main step is a typed closure conversion that translates from pcf to 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.

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 be able 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, for example, to improve compilation times, 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 and the understanding of how the
low-level code implements the source code. 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 [22], 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. A second theoretical motivation concerns 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 [4].

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 $(\star p)(\vec{v}, w)$ (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,
to define a calling convention and to link compiled low-level code with programs
written in C, for example.

Defunctionalisation is another well-known implementation method for higher-
order functional languages. Defunctionalisation 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 [11] implements a form of Geometry of Interaction, which can
be considered a variant of defunctionalisation [22]. 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 [5,26].

Defunctionalisation can be seen as a variant of closure conversion. The differ-
ence 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 \( \text{apply}(\langle t, \vec{v} \rangle, w) = \text{case } t \text{ of } \ldots \) that performs case distinction on the tag \( t \). Application of \( \langle t, \vec{v} \rangle \) to \( w \) thus becomes a call to \( \text{apply}(\langle t, \vec{v} \rangle, w) \).

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 \( \text{apply} \). 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 \( \text{apply}' \) that performs a case distinction only on these two possible cases, rather than all cases. Also, 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 \( \text{apply} \)-procedures and control flow information determines which one to use in application. Such an optimised defunctionalisation method is implemented in the MLton compiler, see [5].

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 higher-order function \( f : ((\text{int} \rightarrow \text{int}) \rightarrow \text{int}) \rightarrow \text{int} \) and the other an argument \( g : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \) for it. Suppose we want to perform the application of \( f \) \( g \) by linking the compiled low-level code, perhaps because the two modules have been compiled by different compilers or perhaps because \( g \) is implemented as a circuit on an FPGA. If they are to be complete compiled code, the two low-level modules must somehow contain \( \text{apply} \)-procedures for \( f \) and \( g \) respectively. Note that if \( f \) wants to apply its argument to some value, then it needs to invoke an \( \text{apply} \)-procedure for \( g \), which would be defined in the other module. Additionally, this code must also be able to invoke an \( \text{apply} \)-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 \( \text{apply} \)-procedures and one must provide an interface such that both modules can find and invoke the right \( \text{apply} \)-procedure in the other module.

To realise such linking of low-level modules, one must keep track of a number of low-level details. 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 impossible too keep track of such details, but it is not immediately clear how to do so and there are many details to manage. It is desirable to clarify these issues and to account for them in a systematic, clear and simple way.

A further issue is compositional specification and verification. In which sense does the low-level code for \( f : ((\text{int} \rightarrow \text{int}) \rightarrow \text{int}) \rightarrow \text{int} \) 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. [2,20] do not appear to apply directly in such a situation.
The point of this paper is that such issues of low-level modularity can be managed in a simple way by capturing defunctionalisation as an instance of a typed closure conversion method. The idea is to target a typed \( \lambda \)-calculus to control the composition of low-level code fragments and to specify their interfaces. Typed closure conversion can then be seen as a way of defining a compilation method that translates a whole program by linking the low-level code fragments produced by compiling its parts. The types specify how such code fragments interface with the rest of the program. Typed closure conversion makes explicit how low-level encoding choices in (possibly unknown) parts of the program affect other parts. We stress that the main point of the paper is not 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. [20].

1.1 Call-by-Value PCF

In this paper we take call-by-value \( \text{pcf} \) as a simple higher-order source language. Recall that this is a simply-typed functional programming language with types

\[
X, Y ::= \mathbb{N} | X \rightarrow Y
\]

We use a formulation with the following term syntax.

\[
s, t ::= x | \text{zero} | \text{succ}(t) | \text{pred}(t) | \text{ifz } s \text{ then } t_1 \text{ else } t_2 | s \ t | \text{fn } x \Rightarrow t | \text{fix } f x \Rightarrow t
\]

The term \( \text{fix } f x \Rightarrow t \) denotes a recursive function definition, in which the bound variable \( f \) of type \( X \rightarrow Y \) can be used to make recursive calls.

1.2 Typed Closure Conversion and Defunctionalisation

Typed closure conversion is a typed formulation of closure conversion. Minamide et al. [16] have shown how the translation that replaces an abstraction \( \text{fn } x \Rightarrow s \) with free variables \( \vec{x} \) by a pair \( \langle p, \vec{x} \rangle \), in which \( p \) identifies the code of the closed function \( \lambda \langle \vec{x}, x \rangle \). \text{closure-convert}(s) as a pointer (or similar), can be defined as a type-correct translation to a type-safe programming language. Application is implemented by a single procedure \( \text{apply}((p, \vec{v}), w) = (\ast p)(\vec{v}, x) \), which invokes the closed function identified by the pointer.

Above, we have outlined that to capture the essence of realistic defunctionalisation methods requires more than just to replace the pointer \( p \) by an abstract tag \( t \) that identifies the function. In realistic defunctionalisation methods, the tag alone is not enough to identify the function and control flow information must also be taken into account. Such methods produce not a single \text{apply}-procedure that performs case distinction on all possible tags, but many small \text{apply}-functions that branch only on the tags that are statically know to be possible at a particular call site. To translate an application, one needs the tag together with the information which of these small \text{apply}-functions to use.

In this paper we use a form of typed closure conversion from [24] for call-by-value that produces not just the translated program, but that also makes explicit compositionally how the many small \text{apply}-procedures are defined and
how they are to be linked together. The choice of apply-procedure at a call site is made explicit as part of closure conversion. In this way, the typed closure conversion captures the issues of code linking and composition that we would like to understand. We will see how it applies directly to low-level code. Making the definition and linking of apply-procedures explicit also allows more flexibility in the representation of functions. It is possible to use a different encoding for each abstraction in one program, e.g. by mixing heap-allocated closures with defunctionalisation.

Let us outline this typed closure conversion from [24] by example. It decomposes call-by-value pcf into a typed λ-calculus with the following types.

\[ X, Y ::= \text{exp}(A,B) \mid 1 \mid X \to Y \mid X \times Y \mid \forall \alpha. X \mid \exists \alpha. X \]

The types \( A \) and \( B \) in this grammar range over the low-level types of values that can appear in the final low-level programs. Formally, they are given by the following grammar (but it is not important to understand their details here):

\[ A, B ::= \alpha \mid \text{void} \mid \text{unit} \mid \text{int} \mid A \times B \mid A + B \mid A \cup B \mid \mu \alpha. A \]

The types \( X \) and \( Y \) themselves explain how computations can be composed. The basic type \( \text{exp}(A,B) \) should be thought of as the type of closed low-level programs that take as input a value of type \( A \) and that output a value of type \( B \). This type may also be thought of as representing strict functions from \( A \) to \( B \).

Let us outline by example how pcf terms of various types are decomposed into terms of these types.

- A term of base type \( N \) translates to a term of type \( \text{exp}(\text{unit}, \text{int}) \), i.e. a first-order computation that implements the term’s evaluation, which produces a number. For a term of base type, no additional code for apply or the like is needed.
- A term of type \( N \to N \) translates to one of type \( \exists \alpha. \text{exp}(\alpha \times \text{int}, \text{int}) \times \text{exp}(\text{unit}, \alpha) \). The part of type \( \text{exp}(\text{unit}, \alpha) \) represents the code that computes the function value. The part of type \( \text{exp}(\alpha \times \text{int}, \text{int}) \) represents the apply-procedure for the application of such an encoded function. The type \( \alpha \) that encodes functions is existentially quantified, so that the encoding of function values may be arbitrary. To encode an abstraction, one may use a type that can encode its free variables. To encode a term that may evaluate to more than one function value, one may choose a type that encodes both the function value and a tag identifying which it is. The application \( \text{exp}(\alpha \times \text{int}, \text{int}) \) then performs case distinction on the tag. It is also possible to use a pointer instead of a tag. In this case, the application \( \text{exp}(\alpha \times \text{int}, \text{int}) \) would directly invoke the pointed-to code.
- A term of type \( (N \to N) \to N \) translates to a term of type \( \exists \alpha. (\forall \beta. \text{exp}(\beta \times \text{int}, \text{int}) \to \text{exp}(\alpha \times \beta, \text{int})) \times \text{exp}(\text{unit}, \alpha) \). The part of type \( \text{exp}(\text{unit}, \alpha) \) is again the code to evaluate the term’s value. The other part is there for function application. It is more complicated this time because it also needs to be able to invoke the apply-procedure of the (yet unknown) argument.
function. Suppose we want to apply it to an argument encoded by a term of type $\exists \beta. \exp(\beta \times \text{int}, \text{int}) \times \exp(\text{unit}, \beta)$. By eliminating the existential quantifier, we obtain a type $B$, a term of type $\exp(B \times \text{int}, \text{int})$ for function application and a term of type $\exp(\text{unit}, B)$ to compute the function value. From $\forall \beta. \exp(\beta \times \text{int}, \text{int}) \rightarrow \exp(\alpha \times \beta, \text{int})$ we then obtain a term of type $\exp(\alpha \times B, \text{int})$ that implements function application. By sequencing the computations $\exp(\text{unit}, \alpha)$, $\exp(\text{unit}, B)$ and $\exp(\alpha \times B, \text{int})$, we obtain $\exp(\text{unit}, \text{int})$, which is the code that computes the result of function application.

In general, a closed call-by-value PCF term $t : X$ is translated to a term $TCC(t) : \mathcal{M}[X]_{\text{unit}}$, where $\mathcal{M}[X]_A := \exists \alpha. \mathcal{I}[X]_\alpha \times \exp(A, C[X]_\alpha)$. The type $C[X]_\alpha$ (code type) is the low-level type that encodes the value of the term $t$. The part $\exp(A, C[X]_\alpha)$ represents the code that computes the value of $t$. The part of type $\mathcal{I}[X]_\alpha$ (interface type) represents the code that may be needed to make use of the resulting value. If $X$ is a base type like $\text{N}$, then no code is needed. In this case, value is just a number, which is all we want. But if $X$ is a function type, then the code for the apply-procedure needs to be provided as outlined by example above. Formally, $C[X]_\alpha$ and $\mathcal{I}[X]_\alpha$ are defined by:

$$C[N]_\alpha := \text{int} \quad C[X \rightarrow Y]_\alpha := \alpha$$

$$\mathcal{I}[N]_\alpha := 1 \quad \mathcal{I}[X \rightarrow Y]_\alpha := \forall \beta. \mathcal{I}[X]_\beta \rightarrow \mathcal{M}[Y]_{\alpha \times C[X]_\beta}$$

To understand the definition of $\mathcal{I}[X \rightarrow Y]_\alpha$, it is instructive to try to spell out application: given terms of $\mathcal{M}[X \rightarrow Y]_{\text{unit}}$ and $\mathcal{M}[X]_{\text{unit}}$, define a term of type $\mathcal{M}[Y]_{\text{unit}}$, see also page 23.

The closure conversion method of Minamide et al. can be understood as a special case where everything to do with the types $\mathcal{I}[X]$ is handled implicitly using code pointers and jumps.

### 1.3 Typed Closure Conversion and Low-Level Programs

We have now outlined how typed closure conversion can organise a defunctionalising decomposition of call-by-value PCF by making making explicit the definition and linking of code for function application. It is not immediately clear, however, that this formal definition leads to an efficient low-level implementation of PCF. It may appear that this translation is just a translation from one higher-order lambda-calculus to another one. While the type $\exp(A, B)$ represents closed programs, the translation produces higher-order functions involving such types. The reader may be worried that the implementation of these functions induces runtime overhead or requires closure conversion itself.

In this paper we show how the typed closure conversion method can be instantiated to result in a realistic defunctionalisation method that produces efficient low-level programs. The basic idea is to consider a suitable low-level implementation of the lambda-calculus terms produced by typed closure conversion, so that, overall, one obtains a low-level defunctionalising implementation of PCF.
will show that it is possible to do so in a way that the overall translation takes into account global program information, such as control flow information and accounts for issues, such as the specification of low-level interfaces and linking. While the overall translation thus accounts for complicated low-level aspects, its correctness proof is not much more complicated than that of typed closure conversion alone.

The basic idea is to consider a suitable low-level implementation of the \(\lambda\)-calculus terms produced by typed closure conversion, so that the combined translation implements defunctionalisation. Already in [24] the calculus INT [23] is suggested as a possible replacement of the \(\lambda\)-calculus, as its terms have a direct low-level interpretation. However, just using INT instead of the \(\lambda\)-calculus is not sufficient for a number of reasons: (1) the implementation of quantifiers in [23] is naive and would result in closures being represented by encoding them in natural numbers rather than the more precise types used in defunctionalisation; (2) the translation would not take into account global information, e.g. about the control flow, that is needed for a precise choice of closure representation; (3) the INT type system restricts subexponentials to negative positions, which results in inefficient use of stack-allocated data; (4) the equational theory of INT [23] is too weak to support the correctness proof for typed closure conversion.

To address (1) one may try to avoid using quantifiers and reformulate typed closure conversion with particular choices of types. But this would weaken the logical specification in the types, e.g. if one wants to use parametricity reasoning, and would require a new correctness proof. For (2) one may consider to translate the program first and to perform global analysis only after translation directly on low-level programs. However, then we could not specify the interfaces of compiled low-level code parts compositionally. Performing a global analysis may change the data types in these interfaces.

In this paper we show how to address these issues in a simple way. We capture a defunctionalisation method as the composition of the following steps.

\[
\text{PCF} \xrightarrow{(i)} \text{PCF with annotated types} \xrightarrow{(ii)} \text{INT}^\prime \xrightarrow{(iii)} \text{low-level language}
\]

All global information about the input program is captured by type annotations, which are computed by type inference in step (i). This information is then used in an annotated typed closure conversion (ii) that and has a relatively simple correctness proof. Its target INT’ is a variant of INT with fewer typing restrictions, existential types, bounded polymorphism and a suitable equational theory. Step (iii) is an interactive interpretation.

Within the given space constraints, we cannot define all the translations (i)–(iii) and the intermediate languages in sufficient detail. We will therefore describe the composition of (ii) and (iii) as a single step in Sec. 3. In Sec. 5 we outline the factorisation into (ii) and (iii).

1.4 Related Work

Typed closure conversion appears in many typed approaches to compilation. For example, Morrissett et al. [17] present a type-preserving translation from
System F to typed assembly language. This approach has not only been applied to the compilation of whole programs. Perconti and Ahmed [19] have shown that a multi-language approach [1] can be used for the compilation of open programs, which means that program components are compiled and verified independently.

For defunctionalisation, we are not aware of similar work on the compilation of open programs. There are type-correct formulations of defunctionalisation. For example, Pottier and Gauthier [20] show how to use generalised algebraic data types to give a type-correct implementation of defunctionalisation for System F. Their correctness proof is based on establishing a simulation between the source program and the defunctionalised code. This is similar to other correctness argument, e.g. [2]. It seems that such arguments apply directly only when the whole source program is given. It is not clear how to apply them to the situation where one wants to link with code written in C, for example.

The work in this paper is related to implementations of computation by the Geometry of Interaction. Especially the Memoryful Geometry of Interaction [18] should be related. A main difference is that we focus more on aspects of efficient compilation, such as allowing good choices of closure representation and on a safe-for-space [25] translation. There may also be interesting connection points to approaches involving concurrency, such as the Geometry of Synchronization [6].

2 Low-Level Language

We begin by describing the simple low-level language, which is the final target language for the translation from $\text{pcf}$. The low-level language has a similar level of abstraction as compiler intermediate languages like $\text{LLVM}$. 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 show in Sec. 4 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, to implement tail recursion. It also allows us to consider unusual targets, e.g. for hardware synthesis.

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

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

Here, $\alpha$ ranges over an infinite set of type variables, void 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 $\text{LLVM}$. For example, the types are sufficient to represent tuples without overhead: a value of type $A \times B$ can be encoded simply as the concatenation of a value of type $A$ and one of type $B$. The low-level type system is not intended here to capture interesting correctness guarantees. In particular, the untagged union type $A \cup B$ makes the type system weak. While it would be possible to work without union types, they are useful in a number of
cases. We shall see examples, where control-flow information allows us to omit defunctionalisation tags completely, even when more than one function value is possible, see Sec. 3.1. An untagged union type is useful to handle such cases without overhead.

The low-level values are defined by:

\[
 v, w ::= x | () | n | (v, w) | \text{inl}_{A+B}(v) | \text{inr}_{A+B}(v) | \text{in}_{A,B}(v) | \text{fold}_A(v)
\]

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

Low-level programs are made up of blocks. A block has the form \(f(x: A) = b\), 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.

\[
 b ::= f(v) | \text{let } x = \text{op}(v) \text{ in } b | \text{let } \langle x, y \rangle = v \text{ in } b | \text{let } \text{in}_{A,B}(x) = v \text{ in } b
\]

Here, \(f, g, h\) and \(b\) range over block labels. A block therefore consists of a number of \text{let}-bindings and ends with a (possibly conditional) jump with argument to another block. In the first \text{let}-term, \text{op} ranges over primitive operations, such as \text{add}, \text{sub} and \text{mul}, which take an argument of type \text{int} \times \text{int} and produce a result of type \text{int}, as well as \text{eq}, which produces a result of type \text{unit} + \text{unit}, seen as a boolean. The other \text{let}-terms represent pattern matching operations. For example, the block \(f(x: A \times B) = \text{let } \langle x_1, x_2 \rangle = x \text{ in } g(x_1)\) 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, \((\text{let } \text{in}_{A,A\cup B}(x) = \text{in}_{B,A\cup B}(w) \text{ in } b)\) is undefined if \(A\) and \(B\) are not identical.

A program fragment is given by a set \(S\) of well-typed blocks together with a specified interface, which consists of a list of entry labels \(\text{entry} = (\text{entry}_i)_{i=1,\ldots,n}\) and a list of exit labels \(\text{exit} = (\text{exit}_i)_{i=1,\ldots,m}\). Each entry label must be defined in \(S\), while no exit label may be defined in it. For each block label there may be at most one definition in \(S\). We assume that the blocks in \(S\) are all well-typed. We write \(Q: A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m\) if entry label \(\text{entry}_i\) has argument type \(A_i\) and exit label \(\text{exit}_j\) has argument type \(B_j\).

We find it useful to define programs by control flow graphs. Blocks are depicted by 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, a fragment \(F\) with two blocks \(b_1: \text{entry}_1(x: \text{int}) = \text{case } v \text{ of } \ldots \Rightarrow \text{exit}_1(w); \ldots \Rightarrow f(r)\) and \(b_2: f(x: \text{int}) = \text{case } v \text{ of } \ldots \Rightarrow \text{entry}_1(w); \ldots \Rightarrow \text{exit}_2(r)\) would be depicted as follows.

```

int \[b_1\] \[\text{int}\] \[\Rightarrow\] \text{int} \[\rightarrow\] \text{int} \[\Rightarrow\] \text{int}
```
We depict a program fragment $P: A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ by a grey box with dashed lines as shown on the right below. The fragment $F$ from above would be shown as on the left below. The entry and exit labels are ordered from bottom to top. One should think of such a grey box as representing the control flow graph of the blocks in the fragment.

Each program may be transformed into a program with a single entry and exit label. From a program $P: A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$, one obtains a program $P': A_1 + \cdots + A_n \rightarrow B_1 + \cdots + B_m$ by adding case distinction and injection blocks. 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.

For a fragment $F: A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$, we write $C \cdot F$ for the fragment of type $(C \times A_1), \ldots, (C \times A_n) \rightarrow (C \times B_1), \ldots, (C \times B_m)$ that passes on unchanged the additional input value of type $C$. The new argument of type $C$ may be seen as a callee-save argument.

We consider two programs $P, Q: A \rightarrow B$ without free variables equal if they are equal extensionally. 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 $A \triangleleft B$ that formalises that any value of type $A$ can be encoded as a value of type $B$. We define $A \triangleleft B$ to hold if there exists two single-block (for simplicity) programs $\text{in}: A \rightarrow B$ and $\text{out}: B \rightarrow A$, such that their composition $b_2 \circ b_1: A \rightarrow A$ (obtained by taking the blocks from both programs and making the entry label of $\text{out}$ the exit label of $\text{in}$) is equal to the identity.

For working with the $\text{in}$ and $\text{out}$ programs from $A \triangleleft B$, it is convenient to use the notation $\text{let } y = \text{in}_{A \triangleleft B}(v) \text{ in } b_1$ and $\text{let } \text{in}_{A \triangleleft B}(x) = w \text{ in } b_2$ for definable blocks with the following behaviour. In the block $\text{let } y = \text{in}_{A \triangleleft B}(v) \text{ in } b_1$, the value $v$ must have type $A$. When executing this block, it encodes $v$ as an element of $B$, as the $\text{in}$ program would do, binds the result to $y$ and then executes $b_1$. In the block $\text{let } \text{in}_{A \triangleleft B}(x) = w \text{ in } b_2$, the value $w$ has type $B$. This block decodes $w$, as the $\text{out}$ program would do, binds the result to $x$ and then executes $b_2$.

### 3 Annotated PCF

The aim is now to translate PCF terms to low-level program fragments in a strongly compositional sense: the low-level translation of a term is obtained by combining the low-level translations of its parts. Each translated part has a well-defined low-level interface and correctness can be shown by verifying the parts individually. We use an annotated PCF type system to capture and control any global information that is needed in the translation.

Annotated PCF types are defined by the grammar

$$X, Y ::= A \cdot N \mid A \cdot (X \xrightarrow{C} B \cdot Y) ,$$

10
in which \( A \), \( B \) and \( C \) range over low-level types. We write just \( X \) for \( \text{unit} \cdot X \).

The \( \text{pcf} \) typing judgements are also annotated and now have the form \( x_1 : \mathcal{X}_1, \ldots, x_n : \mathcal{X}_n \vdash_D t : \mathcal{Y} \), where \( t \) is a normal \( \text{pcf} \) term, where the type \( \mathcal{X} \) and the \( \mathcal{X}_i \) are annotated source types, and where \( D \) is a low-level type.

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. Firstly, the annotation \( C \) on the function type is the low-level type used for closure representation, i.e. it is the low-level type that is used to represent the values of the function type.

All the other annotations give information about how the low-level implementation manages its data. The typed closure conversion in Sec. 1.2 produces two pieces of code for each term, one piece to evaluate the value of the term and the other (which may be vacuous) to implement the application for values of function type (the \( \text{apply} \)-procedures). The type annotations tell us how these pieces of code manage their data. The annotation \( B \) on the function type tells us that, in addition to the function value and argument value, the code for function application takes an additional argument of type \( B \), which it promises not to inspect and to return unchanged (see Sec. 3.2 below for details). We say that this code has a \textit{callee-save argument} of type \( B \). Such callee-save arguments are useful as the low-level language has neither state nor stack. Callee-save arguments can be understood as an explicit, typed formulation of the machine stack. The annotation \( D \) on the typing sequent states that the code for evaluating the term \( t \) comes with a callee-save argument of type \( D \), this code has an additional callee-save argument of type \( A \). The meaning of callee-save arguments will be described in detail in Sec. 3.2 below. For now, one may think of them as annotations that allow one to tell the types of all stack contents statically by looking at the types.

The type annotations are controlled by the typing rules in Fig. 1. We use the following notation. We write \( \mathcal{C}[\mathcal{A}] \) for the low-level type that encodes the value of type \( \mathcal{A} \). It is defined by \( \mathcal{C}[\text{unit}] = \text{int} \) and \( \mathcal{C}[A \cdot (\mathcal{X} \rightarrow B) \cdot \mathcal{Y}] = \mathcal{C} \). This definition extends to contexts: \( \mathcal{C}[] \) is defined by \( \mathcal{C}[] = \text{unit} \) and \( \mathcal{C}[\Delta, x : \mathcal{A}] = \mathcal{C}[\Delta] \times \mathcal{C}[\mathcal{A}] \). 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 \( \Gamma \). We further write \( A \cdot \Gamma \) for the context one obtains by replacing each declaration \( x : B \cdot X \) in \( \Gamma \) with \( x : (A \times B) \cdot X \).

The subtyping rule for \( \text{N} \) allows us to replace any \( E \cdot \text{N} \) by \( \text{unit} \cdot \text{N} \). This is justified, as the application code for values of type \( \text{N} \) is vacuous. We will therefore just write \( \text{N} \) for \( A \cdot \text{N} \).

Let us sketch the meaning of the annotations of rule \((\text{APP})\). To evaluate \( s \cdot t \) in call-by-value, one first evaluates \( s \), then \( t \) and then invokes the right application code with the obtained function and argument values. The conclusion of \((\text{APP})\) states that the evaluation code offers a callee-save value of type \( U \). While we evaluate \( s \), we must remember this value together with the values of the variables that we need to evaluate \( t \). For this we use the callee-save argument of type

\[
\begin{array}{c}
\text{Subtype} \quad \frac{\Gamma, x : \mathcal{C} \vdash_A t : \mathcal{D}}{\Gamma, y : \mathcal{E} \vdash_A t : \mathcal{F}} \quad \frac{\Gamma \vdash_B t : \mathcal{G}}{\Gamma \vdash_A t : \mathcal{H}} \quad \frac{\mathcal{A} \leq \mathcal{B}}{\mathcal{C} \leq \mathcal{D}} \quad \frac{\mathcal{A} \triangleleft \mathcal{B}}{\mathcal{C} \triangleleft \mathcal{D}}
\end{array}
\]

\[
\begin{array}{c}
\text{Var} \quad \frac{x : \mathcal{A} \vdash x : \mathcal{A}}{\Gamma, x : \mathcal{A} \vdash x : \mathcal{A}} \quad \frac{\Gamma \vdash_A t : \mathcal{A}}{\Gamma, x : \mathcal{A} \vdash t : \mathcal{A}}
\end{array}
\]

\[
\begin{array}{c}
\text{W} \quad \frac{\Gamma \vdash_A t : \mathcal{A}}{\Gamma, x : \mathcal{A} \vdash t : \mathcal{A}} \quad \frac{\Gamma, y : \mathcal{A} \vdash t : \mathcal{A}}{\Gamma, x : \mathcal{A}, y : \mathcal{B} \vdash t : \mathcal{A}}
\end{array}
\]

\[
\begin{array}{c}
\text{C} \quad \frac{\Gamma, x : \mathcal{A}, y : \mathcal{B} \vdash t : \mathcal{C}}{\Gamma, z : (\mathcal{A} + \mathcal{B}) \vdash t[z/x, z/y] : \mathcal{C}}
\end{array}
\]

\[
\begin{array}{c}
\text{Succ} \quad \frac{t_A \vdash_B t : \mathcal{A} \leq \mathcal{B}}{t_A \vdash_B \text{succ}(t) : \mathcal{A} \leq \mathcal{B}} \quad \frac{t_A \vdash_B t : \mathcal{A} \leq \mathcal{B}}{t_A \vdash_B \text{pred}(t) : \mathcal{A} \leq \mathcal{B}}
\end{array}
\]

\[
\begin{array}{c}
\text{If} \quad \frac{\Gamma, \Delta_1, \Delta_2 \vdash_A \text{ifz } s \text{ then } t_1 \text{ else } t_2 : \mathcal{A}}{\Gamma, \Delta_1, \Delta_2 \vdash_A \text{if } s \text{ then } t_1 \text{ else } t_2 : \mathcal{A}}
\end{array}
\]

\[
\begin{array}{c}
\text{APP} \quad \frac{\Gamma \vdash_U \mathcal{F} \vdash_A \mathcal{G} \vdash_Y \mathcal{H}}{\Gamma, \Delta \vdash_U \mathcal{F} \vdash_Y \mathcal{H} : \mathcal{A} \vdash_U \mathcal{G} \vdash_Y \mathcal{H} : (\mathcal{A} \vdash_U \mathcal{G} \vdash_Y \mathcal{H})}
\end{array}
\]

\[
\begin{array}{c}
\text{Fix} \quad \frac{\Gamma, t : (\mathcal{X} \vdash_U \mathcal{Y}) \vdash \mathcal{X} \vdash_U \mathcal{Y}}{H \vdash \mathcal{F} \vdash \mathcal{H} : \mathcal{F} \vdash \mathcal{H} : \mathcal{E} \vdash (\mathcal{H} \vdash \mathcal{G}) \vdash (\mathcal{H} \vdash \mathcal{G})}
\end{array}
\]

Subtyping

\[
\begin{array}{c}
\mathcal{A} \leq \mathcal{B} \quad \mathcal{X} \leq \mathcal{Y} \quad \mathcal{Z} \leq \mathcal{W} \quad \mathcal{C} \leq \mathcal{D}
\end{array}
\]

Joining Interfaces

\[
\begin{array}{c}
\mathcal{X} \leq \mathcal{Y} \quad \mathcal{X} = \mathcal{Y} \quad \mathcal{X} = \mathcal{Y}
\end{array}
\]

\[
\begin{array}{c}
\mathcal{A} \cdot \mathcal{X} \vdash \mathcal{A} \cdot \mathcal{X} \vdash \mathcal{A} \cdot \mathcal{X}
\end{array}
\]

Fig. 1. Annotated PCF Typing
$U \times C[\Delta]$ in the left premise of $(\text{APP})$. While we evaluate 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 $s$.

One part of the type system that deserves comment are the rules for joining interfaces. The joining judgement is used in rule $(\text{IF})$ 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 $X_1$ and $X_2$ are both $N \rightarrow_B N$. This means that both branches return a function that is encoded without defunctionalisation tag. In this case $X$ must be $N^{1+1} \rightarrow_B N$, i.e. the result has a tag of a single bit. So joining amounts to tagging in this case. It is explained in more detail in Sec. 3.2.

3.1 Typing Examples

One derives $f : \text{unit} \cdot (X \downarrow_B N), x : X \vdash_A f x : N$ using $(\text{APP})$ and $(\text{VAR})$. From two instances of this, one derives using $(\text{IF})$ and $(\text{C})$ the judgement

$$f : (\text{unit} + \text{unit}) \cdot (X \downarrow_B N), x_1 : X, x_2 : X \vdash_A \text{if} ... \text{then } f x_1 \text{ else } f x_2 : N.$$

The annotation $\text{unit} + \text{unit}$ says the that application code for $f$ has a callee-save argument of this type. This means that while we apply the function $f$, we can store a value of this type and are guaranteed to have it returned unchanged. This value of type $\text{unit} + \text{unit}$ is used as an abstract form of the return address on the stack. It encodes whether we should return to the left or the right copy of $f$.

In the example, $x_1$ and $x_2$ have the same type, which is somewhat restrictive. If $X$ is a function type $B \cdot (N \downarrow_E N)$, then this will mean that both variables denote functions represented with exactly the same closure representation, for example. This constraint can be weakened using subtyping. To derive the same judgement with $x_1 : X_1$ and $x_2 : X_2$, it suffices to take $X_i$ to be $B_i \cdot (N \downarrow_{E_i} N)$, where $E_i, C_i$ and $B_i$ are such that $B \triangleleft B_i$ and $E \triangleleft E_i$ and $C_i \triangleleft C$ 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 $\text{unit} + \text{unit}$, 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 [5]. 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
A × C \llbracket x : X_1, \ldots, x_n : X_n \rrbracket
I/llbracket X_n/rrbracket+
A ... I/llbracket X_1/rrbracket–
.........
I/llbracket Y/rrbracket– I/llbracket Y/rrbracket+
/llparenthesisΠ /rrparenthesis

Another source of tagging is case distinction of higher type. The following example, which uses an externally defined addition function \texttt{plus}, defines a function by case distinction.

\[
f : \mathbb{N} \xrightarrow{C} \mathbb{N}, \ y : \mathbb{N} \vdash 0 \ldots \text{then \ } f \text{ \ else \ fn } x \Rightarrow \texttt{plus } x \ y : \mathbb{N} \xrightarrow{C + (\text{unit } \times \text{int})} \mathbb{N}
\]

The function \( f \) is encoded using type \( C \). The function \( \texttt{fn } x \Rightarrow \texttt{plus } x \ y \) is encoded by the tuple of its free variables of type \( C[y : \mathbb{N}] = \text{unit } \times \text{int} \). The two possible results of the case distinction are represented using the sum type \( C + (\text{unit } \times \text{int}) \).

For a slightly larger example, consider the following term \( \text{step} \):

\[
\text{fn } f \Rightarrow \text{fn } x \Rightarrow \text{let } x_1 = \text{pred}(x) \text{ in let } x_2 = \text{pred(pred}(x)) \text{ in}
\]

\[
\text{ifz } x_1 \text{ then } 1 \text{ else ifz } x_2 \text{ then } 1 \text{ else plus } (f \ x_1) (f \ x_2)
\]

It can be used to define a Fibonacci function by \( \text{fix } f \Rightarrow \text{step } f \ x \). It can be given type \( \text{step } : A \cdot ((B + B) \cdot (N \xrightarrow{E} S \mathbb{N})) \xrightarrow{C} B \cdot (N \xrightarrow{E} D \mathbb{N}) \) where the annotation \( S \) is \((D \times \text{int } \times E) \cup (D \times \text{int})\). 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 \( \text{int } \times E \), while the second time it contains the \( \text{int-value} f \ x_1 \).

### 3.2 Direct Translation to Low-Level Programs

We next outline a direct translation from annotated \texttt{PCF} to the low-level language, which we shall then identify as an instance of the typed closure conversion from Sec. 1.2. The type annotations are enough to fully specify the interface of low-level code fragments. We associate to each annotated type \( X \) two low-level types \( \mathcal{I}[X]^- \text{ and } \mathcal{I}[X]^+ \) as follows.

\[
\mathcal{I}[A \cdot \mathbb{N}]^- = A \times \text{void} \quad \mathcal{I}[A \cdot (X \xrightarrow{C_B} Y)]^- = A \times (\mathcal{I}[X]^+ + \mathcal{I}[Y]^- + B \times (C \times \mathcal{C}[A]\mathcal{C}[X]))
\]

\[
\mathcal{I}[A \cdot \mathbb{N}]^+ = A \times \text{void} \quad \mathcal{I}[A \cdot (X \xrightarrow{C_B} Y)]^+ = A \times (\mathcal{I}[X]^+ + \mathcal{I}[Y]^+ + B \times \mathcal{C}[Y])
\]

The direct translation goes by induction on typing derivations. It maps a derivation \( \Pi \) of \( x_1 : X_1, \ldots, x_n : X_n \vdash_A t : Y \) to a low-level program fragment \( \langle \Pi \rangle \) with the following interface.

\[
A \times C[x_1 : X_1, \ldots, x_n : X_n] \rightarrow A \times C[Y]
\]

\[
\mathcal{I}[Y]^– \rightarrow \mathcal{I}[Y]^+ \text{ and } \mathcal{I}[X_i]^+ \rightarrow \mathcal{I}[X_i]^– \text{ for } i = 1, \ldots, n.
\]

To explain the low-level interface of \( \langle \Pi \rangle \), we must explain how low-level programs fragments represent \texttt{PCF} values. A \texttt{PCF} value of type \( X \) is represented
by a pair of low-level value of type $C[[\mathcal{X}]]$ and a low-level program fragment of type $\mathcal{I}[\mathcal{X}]^- \rightarrow \mathcal{I}[\mathcal{X}]^+$. For values of type $A \cdot N$, this amounts to just a low-level value of type int, as $\mathcal{I}[A \cdot N]^-$ and $\mathcal{I}[A \cdot N]^+$ are empty types.

A value of type $A \cdot (\mathcal{X} \subseteq_R B \mathcal{Y})$ is represented by a low-level value $f : C$ and a code fragment $F : \mathcal{I}[A \cdot (\mathcal{X} \subseteq_R B \mathcal{Y})]^- \rightarrow \mathcal{I}[A \cdot (\mathcal{X} \subseteq_R B \mathcal{Y})]^+$. The code fragment $F$ is the code for function application. It must have the following properties:

- The fragment must satisfy the callee-save invariant that there exists a fragment $F'$ such that $F = A \cdot F'$. In other words, the first component $a : A$ of any input to $F$ must not be inspected and be returned unchanged.
- The fragment $F$ must implement application in the following sense. Suppose we have a value of type $\mathcal{X}$, represented by a value $x : C[[\mathcal{X}]]$ and a fragment $X : \mathcal{I}[\mathcal{X}]^- \rightarrow \mathcal{I}[\mathcal{X}]^+$. Link $X$ to $F$ as in the diagram below, in which we have used distributivity for $A$ in the type of $F$.

Then the following must hold for all values $a : A$ and $b : B$. If we jump with argument $\langle a, b, (f, x) \rangle$ to the topmost entry label, then the module will compute the function application of $f$ to $x$ (and perform possible effects if we allow them) and then jump with argument value $\langle a, b, y \rangle$ to the topmost exit label, where $y$ is the value encoding the result of function application. Both $a$ and $b$ must be treated as callee-save arguments; they must not be inspected and be returned unchanged. Since we have a value $a : A$, we can define single-block fragments of type $\mathcal{I}[\mathcal{Y}]^- \rightarrow A \times \mathcal{I}[\mathcal{Y}]^-$ and $A \times \mathcal{I}[\mathcal{Y}]^+ \rightarrow \mathcal{I}[\mathcal{Y}]^+$. By connecting them to the entry and exit labels with types $A \times \mathcal{I}[\mathcal{Y}]^- \rightarrow \mathcal{I}[\mathcal{Y}]^-$ and $A \times \mathcal{I}[\mathcal{Y}]^+$ in the above program, we obtain a fragment $Y : \mathcal{I}[\mathcal{Y}]^- \rightarrow \mathcal{I}[\mathcal{Y}]^+$. The value $y$ and this fragment $Y$ must then represent the result of the function application of $f$ to $x$.

With these definitions, we can explain the meaning of (III). Suppose we have values of types $X_1, \ldots, X_n$, represented by low-level values $x_1, \ldots, x_n$ and fragments $X_i : \mathcal{I}[X_i]^- \rightarrow \mathcal{I}[X_i]^+$ for $i = 1, \ldots, n$. Suppose we connect these fragments to (III) like we have connected $X$ to $F$ above (only without the need to apply $A \cdot (-)$). Then we have a code fragment that can evaluate the value of the term $t$. If we jump with $\langle a, v \rangle$ to the topmost entry point, where $v$ is the tuple of the low-level values for $x_1, \ldots, x_n$, then the program will compute the value of the term $y : C[[\mathcal{Y}]]$ of the term, and jump with $\langle a, y \rangle$ to the topmost exit point. By invariant, the program is required to treat $a$ as a callee-save argument. The value $y : C[[\mathcal{Y}]]$ and the fragment $Y : \mathcal{I}[\mathcal{Y}]^- \rightarrow \mathcal{I}[\mathcal{Y}]^+$ obtained from (III) by making the topmost entry and exit labels private then represent the value of the term $t$. 

15
3.3 Translation

The translation from annotated PCF to the low-level language goes by induction on the typing derivation. We spell out representative cases, beginning with (VAR).

\[
(VAR) \quad A \times (\text{unit} \times C[X]) \xrightarrow{[\Pi]} A \times C[X]
\]

\[
I[X]^- \xrightarrow{[\Pi]} I[X]^+ \xrightarrow{[\Pi]} I[X]^-
\]

In the translation of (VAR), the block named 1 is defined by:

\[
\text{entry}_1(z: A \times (\text{unit} \times C[X])) = \text{let } \langle a, z' \rangle = z \text{ in } \text{let } \langle u, x \rangle = z' \text{ in } \text{exit}_1(\langle a, x \rangle)
\]

It implements the mapping \(1: \langle a, (\cdot), x \rangle \mapsto \langle a, x \rangle\) and just extracts the value of the variable from the context. Since the labels are clear from the control flow graph, we shall define such blocks using similar mappings as short notation.

\[
(FN) \quad F \times C[\Gamma] \xrightarrow{[\Pi]} F \times C[\Gamma]
\]

\[
E \times (A \times (\text{unit} \times C[X])) \xrightarrow{[\Pi]} E \times (A \times C[Y])
\]

\[
E \times I[X]^+ \xrightarrow{[\Pi]} E \times I[Y]^+ \xrightarrow{[\Pi]} E \times I[X]^-
\]

\[
I[E \cdot \Gamma]^+ \xrightarrow{[\Pi]} I[E \cdot \Gamma^-]
\]

The translation of rule (FN) is such that the closure is represented by the tuple of the values of the variables in \(\Gamma\). Thus, block 1 is the identity. It returns the tuple of variables in \(\Gamma\) as the term’s value. The next three entry and exit wires are the three summands (multiplied out) in \(I[\Pi E \cdot (X \xrightarrow{C[\Gamma]} Y)]^-\) and \(I[\Pi E \cdot (X \xrightarrow{C[\Gamma]} Y)]^+\). The entry point with type \(E \times (A \times (\text{unit} \times C[X]))\) is the entry point for function application. 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 is defined to jump to the code to evaluate \(t\). The result will be returned at the exit point of type \(E \times (A \times C[Y])\). Thus, function application for \(\text{fn } x \Rightarrow t\) is defined by jumping to the evaluation code for \(t\). Finally, the blocks 2 : \(\langle \langle e, x \rangle, y \rangle \mapsto \langle e, (x, y) \rangle\) and 3 : \(\langle e, (x, y) \rangle \mapsto \langle (e, x), y \rangle\) are there to bring the callee-save arguments in the context in the right form.

The translation of rule (APP) implements the description of the typing rule given above.

\[
(APP) \quad U \times C[\Gamma, \Delta] \xrightarrow{[\Pi]} U \times C[\Gamma]
\]

\[
I[\Gamma]^+ \xrightarrow{[\Pi]} I[\Delta]^+ \xrightarrow{[\Pi]} I[\Gamma]^-
\]
To evaluate the application, one first evaluates \( s \), which is what block 1: \( \langle u, c \rangle \mapsto \langle \langle u, \pi_A(c) \rangle, \pi_T(c) \rangle \) does. In its definition we write \( \pi_T \) and \( \pi_A \) for the projections from \( C[I, \Delta] \) to \( C[I] \) and \( C[\Delta] \) respectively. Block 1 thus puts the values in \( \Delta \) on in a callee-save argument for later use and jumps to the code to evaluate \( s \). Block 2: \( \langle \langle u, d \rangle, f \rangle \mapsto \langle \langle u, f \rangle, d \rangle \) receives the function value \( f \). It retrieves the values in \( \Delta \) from the callee-save argument (by invariant, \( d \) must be \( \pi_A(c) \)), puts the function value \( f \) in a callee-save argument for later use and jumps to the code to evaluate \( t \). The result \( x \) is passed to block 3: \( \langle \langle u, f \rangle, x \rangle \mapsto \langle () \rangle \). This block now constructs the pair \( \langle f, x \rangle \) of function value and argument and jumps with it to the application code for the function \( s \). The result is passed to block 8, which returns it. This and the remaining blocks are defined by 4, 6, 8: \( \langle () \rangle \mapsto x \) and 5, 7: \( y \mapsto () \). They connect the code provided by \( t \) to \( s \) appropriately for application, as outlined above.

As a representative example of a structural rule, we show the case for rule (c). In this case, we define \( \langle IL_3 \rangle \) to be the following program fragment.

\[
\begin{align*}
A \times (C[I] \times C[X]) & \rightarrow T[I]^+ \rightarrow A \times C[I] \rightarrow T[I]^+
\end{align*}
\]

\[
\begin{align*}
(B + C) \times T[X]^+ & \rightarrow 3 \rightarrow (B + C) \times T[X]^- \rightarrow T[I]^+ \\
I[I]^+ & \rightarrow 2 \rightarrow T[I]^- \\
\end{align*}
\]

Block 1: \( \langle a, \langle c, z \rangle \rangle \mapsto \langle a, \langle \langle c, z \rangle, z \rangle \rangle \) takes as input a callee-save value \( a \) and the values of the variables in the context. It duplicates the value of \( z \) and passes the resulting value to the evaluation code for \( t \). 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 the entry point belonging to \( z \) by blocks 3: \( \langle b, v \rangle \mapsto \langle \text{inl}(b), v \rangle \) and \( 3' : \langle c, v \rangle \mapsto \langle \text{inr}(c), v \rangle \). In the callee-save argument, these blocks record whether the jump came from \( x \) or \( y \). When the code for \( z \) returns by a jump to its exit label, block 2: \( \langle e, v \rangle \mapsto \text{case } e \text{ of } \text{inl}(b) \Rightarrow \text{exit}_\text{top}(\langle b, v \rangle); \text{inr}(c) \Rightarrow \text{exit}_\text{bottom}(\langle c, v \rangle) \) 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 in callee-save arguments. 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 an implementation on a machine with pointers.

The implementation of \( (\Pi) \) is such that the evaluation code first evaluates \( s \) and then, depending on the result, jumps either to the evaluation code for \( t_1 \) or \( t_2 \).
This is implemented by defining 1: \(\langle a, c \rangle \mapsto (\langle a, \pi_{\Delta_1}(c), \pi_{\Delta_2}(c) \rangle, \pi_I(c))\) and 2: \(\langle (a, c_1, c_2), n \rangle \mapsto \text{let } b = \text{eq}(n, 0) \text{ in case } b \text{ of inl}(y) \Rightarrow \text{exit_top}(\langle a, c_1 \rangle); \text{inr}(y) \Rightarrow \text{exit_bot}(\langle a, c_2 \rangle)\).

In addition, a fragment of type \(\mathcal{I}[\mathcal{X}]^- \rightarrow \mathcal{I}[\mathcal{X}]^+\) must be provided for the result value. Depending on whether \(t_1\) or \(t_2\) was evaluated, this fragment should behave like the fragment provided by \(t_1\) or that by \(t_2\). To implement this, fragment 3 code encodes in the result value just enough information so that we can identify which code we need to jump to. 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 \(t_1: \mathcal{N} \xrightarrow{C_1} \mathcal{A} \mathcal{N}\) and \(t_2: \mathcal{N} \xrightarrow{C_2} \mathcal{A} \mathcal{N}\), then ifz ... then \(t_1\) else \(t_2\) will have type \(\mathcal{N} \xrightarrow{C_1 + C_2} \mathcal{A} \mathcal{N}\). In this case, fragment 3 maps \(\langle a, c_1 \rangle\) on the topmost input to \(\langle a, \text{inl}(c_1) \rangle\) on its output and \(\langle a, c_2 \rangle\) on the other input to \(\langle a, \text{inl}(c_2) \rangle\). If \(\mathcal{X}\) is \(\mathcal{N} \xrightarrow{C_1 + C_2} \mathcal{A} \mathcal{N}\), then fragment 4 only has one non-vacuous entry point of type \(\mathcal{A} \cdot ((C_1 + C_2) \times \text{int})\) (for function application). It is defined by \(\langle a, (c, n) \rangle \mapsto \text{case } c \text{ of inl}(c_1) \Rightarrow \text{exit_top}(\langle a, (c_1, n) \rangle); \text{inr}(c_2) \Rightarrow \text{exit_bot}(\langle a, (c_2, n) \rangle)\).

The fragment 5 is the identity in this case; it just passes on the returned value of the function (an integer). The general definition of fragments 3, 4 and 5 may be found in the appendix.

The remaining cases of the translation can be found in Appendix B.1. The following correctness result is a consequence of Theorem 2 in Sec. 6. While its statement pertains only to closed terms of base type, it is obtained by showing properties of open terms of higher type compositionally.

**Corollary 1 (Correctness).** Suppose \(\Pi\) derives \(\vdash_A t: B \cdot N\). Then \(t\) reduces to a value \(v\) in a standard call-by-value operational semantics if and only if we have \(\langle \Pi \rangle: \langle a, () \rangle \mapsto \langle a, v \rangle\) for any closed low-level value \(a: A\).

## 4 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 a given PCF program.

A type inference algorithm can be developed using a standard approach. We first eliminate the subtyping rules by closing all other rules under subtyping, see Appendix B.2. The resulting rules are all ordinary PCF rules with type annotations.
We can therefore use type inference for PCF and only need to decorate PCF typing derivations with annotations. We choose a fresh type variable for each annotation and solve the side-conditions. All side-conditions have the form $A \triangleleft \alpha$, i.e. with a type variable as upper bound, and can be solved for one type variable at a time [23,9]. To solve for $\beta$, one gathers all constraints with $\beta$ as an upper bound, say $B_1 \triangleleft \beta, \ldots, B_n \triangleleft \beta$. If $\beta$ 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.

The outlined inference algorithm gives us the following result. In it we write $|\cdot|$ for the function that removes all type annotations, e.g. $|A \cdot (N \xrightarrow{C} B N)| = N \rightarrow N$.

**Theorem 1.** If $\Gamma \vdash t : X$ in the PCF type system, then there exist a derivable sequent $\Gamma_1 \vdash t_1 : X_1$ in annotated PCF, such that $\Gamma = |\Gamma_1|$, $t = |t_1|$ and $X = |X_1|$.

It is important to note that the algorithm implements just one of many ways of solving constraints. Different solutions correspond to different choices of managing low-level details. We can simply take all the annotations to be $\omega := \mu \alpha. \text{unit} + \alpha$, in which case we would obtain a standard Geometry of Interaction interpretation of PCF, see e.g. [14]. Using a type of trees, one obtains an implementation reminiscent of abstract machines [15,10]. 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 $\text{Stack}$ with $(\text{Stack} \times A) \triangleleft \text{Stack}$ for all $A$ (implemented by push and pop). Then, one may use $\text{Stack}$ for all callee-save annotations, i.e. all typing sequents have the form $\Gamma \vdash_{\text{Stack}} t : X$. In the end, what is being passed around is just a stack pointer. But for performance reasons, one may not want to put values with short liveness ranges on the stack. To this end, the type system allows one to use types other than $\text{Stack}$ in select places. Such choices without modifying the translation to low-level code, simply by controlling type inference.

Type annotation inference can also be understood as simple space usage analysis, as the annotated type system gives information about the space usage of low-level programs. For example, one may be interested in constant space programs, e.g. for embedded applications or for hardware synthesis [11]. A sufficient condition for restricting to constant space programs is that the $\triangleleft$-constraints can be solved without recursive types. This rules out (fix) for its side condition. However, one may restrict oneself to tail recursion on first-order types with the typing rule below. Note that the type system allows us to specify that $f$ appears in a tail position by using $\text{unit} \cdot (-)$ in its type.

\[
\text{tailfix} \quad \frac{\Gamma, f : \text{unit} \cdot (N \xrightarrow{C} t N), x : N \vdash A \vdash t : N}{(E \times A) \cdot \Gamma \vdash_E \text{tailfix} f x \Rightarrow t : E \cdot (N \xrightarrow{C} t N)}
\]

It is also possible to modify rule (fix) to capture recursion with bounded depth $n$. This may be done by taking $H$ in (fix) to be $E + (E \times G) + (E \times G \times G) + \cdots + (E \times G^n)$ instead of solving $H \triangleleft (E + (H \times G))$. 

19
5 Factoring the Translation

While the translation from annotated PCF to the low-level language may be defined and understood directly, the amount of low-level detail in it makes reasoning about it is complicated. In the presence of recursion, reasoning at the level of detail of Sec. 3.2 seems to be very unwieldy. Next we outline how this translation can be seen as an instance of typed closure conversion and that its correctness proof need not be much more complicated than that for typed closure conversion alone.

5.1 Organising Low-Level Programs

The structure that is needed for manipulating low-level program fragments in reasoning may be captured using ideas from interaction semantics. For example, the composition of fragments \( F \) and \( X \) in Sec. 3.2 is captured by the application in the calculus \( \text{int} \) from [23]. We will therefore use a variant of \( \text{int} \) as a target for the typed closure conversion outlined in Sec. 1.2. However, as outlined in Sec. 1.3, \( \text{int} \) is not quite sufficient for capturing defunctionalisation by typed closure conversion.

To address these issues, we extend \( \text{int} \) to \( \text{int}' \) and redevelop the equational theory. The types of \( \text{int}' \) are defined by the following grammar, in which \( A \) ranges over low-level types.

\[
X,Y ::= [A] \mid A \to X \mid I \mid X \otimes Y \mid X \multimap Y \mid A \cdot X \mid \forall \alpha \triangleleft A. X \mid \exists \alpha \triangleleft A. X
\]

Compared to \( \text{int} \), 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.

Rather than presenting the full type system of \( \text{int}' \) in detail here, we explain its terms and types informally, so that one can understand how \( \text{int}' \) is used as a target for typed closure conversion and how it relates to the low-level language. The terms and typing rules for \( \text{int} \) are adapted from [23]; we refer to Appendix C.1 for details.

The reader may think of \( \text{int}' \) as a sub-system of System F. Indeed, \( \text{int}' \) has all the standard terms from System F, in particular abstraction \( \lambda x : X.t \), 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 \( \forall \alpha \triangleleft A. X \) and \( \exists \alpha \triangleleft A. X \) 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 [12]. Indeed, contraction of a variable \( x : (A + B) \cdot X \) gives \( x_1 : A \cdot X \) and \( x_2 : B \cdot X \). Existential and universal quantification is restricted to range over low-level types – type variables can only appear in low-level types. If one ignores the bounds, then this part of the \( \text{int}' \) type system is just a variant of System F with explicit contraction like in Linear Logic. We note that linearity does not restrict the set of typeable terms – it is only used to keep track of low-level annotations on types.

20
In addition to these types and terms, 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 closed low-level program fragment of with an entry label of type unit and an exit label of type \(A\). The type \(A \rightarrow X\) represents a call-by-value function space that takes as argument low-level values of type \(A\). In particular, \(A \rightarrow [B]\) represents low-level program fragments with an entry of type \(A\) and an exit of type \(B\). In INT’, there is a special kinds of abstraction and applications for functions of type \(A \rightarrow X\) and there are monadic terms (return and let) for working with terms of type \([A]\). Assume, for example, that there is a constant succ: int \(\rightarrow\) [int] for incrementation. Then the term \((\text{fn } x \Rightarrow \text{let } y = \text{succ}(x) \text{ in return}(x, y))\) has type int \(\rightarrow [\text{int} \times \text{int}]\), for example.

While being a sub-system of System F, INT’ can also be seen as a language for organising low-level program fragments. The types of INT’ may be understood as interfaces of low-level program fragments. For each INT’ type \(Z\) we define two low-level types \(Z^-\) and \(Z^+\) that define a low-level interface:

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c}
Z & I & A \rightarrow X & X \otimes Y & X \rightarrow Y & A \cdot X & \exists \alpha < A.X & \forall \alpha < A.X \\
\hline
Z^- & \text{void} & \text{unit} & A \times X^- & X^- + Y^- & X^- + Y^+ & A \times X^- & X^-[A/\alpha] & X^-[A/\alpha] \\
Z^+ & \text{void} & A & X^+ & X^+ + Y^+ & X^- + Y^+ & A \times X^+ & X^+[A/\alpha] & X^+[A/\alpha] \\
\end{array}
\]

For any INT’-type \(Z\), we call the low-level program fragments of type \(Z^- \rightarrow Z^+\), i.e. with an entry label of type \(Z^-\) and an exit label of type \(Z^+\), also low-level programs with interface \(Z\).

The INT’-terms of type \(X\) can be regarded as representing low-level program fragments with interface \(X\). For example, a term of type \([A]\) thus represents a low-level program fragment of type unit \(\rightarrow A\).

The type \(X \rightarrow Y\) can be understood as a type that explains low-level program linking. A term \(f\) of type \(X \rightarrow Y\) represents a low-level program fragment \(p : X^- + Y^- \rightarrow X^- + Y^+\). 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 a term \(g\) of type \(X\), representing a low-level program fragment \(q : X^- \rightarrow X^+\). Then function application \(f \, g\) is a term of type \(Y^-\) corresponding to the fragment with interface \(Y\) shown below.

In contrast, the type \(A \rightarrow X\) captures value passing. A term \(f\) of type \(A \rightarrow X\) represents a low-level program fragment \(r : A \times X^- \rightarrow X^+\). 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 \(v : A\). Then we can construct the program fragment shown below, which represents the application \(f(v)\) of type \(X\).

\[
\begin{array}{c}
X^- \xrightarrow{x \mapsto (v, x)} A \times X^- \\
\hline
X^+ \\
\end{array}
\]
In the low-level interpretation, the type $A \cdot X$ captures callee-save arguments. Terms of this type correspond to programs $A \times X^- \rightarrow A \times X^+$ that do not inspect the input value of type $A$ and just pass it on unchanged.

Quantification is realised by low-level programs by using the upper bound in place of the type variable. The figure below shows how from a program $s$ that corresponds to a term of type $\forall \alpha \exists \beta \cdot X \cdot (X \rightarrow \text{unit})$ one gets to a program corresponding to a term of type $X[B/\alpha]$, for any $B < A$. In this figure, the program fragments $\text{in}$ and $\text{out}$ are derived from the corresponding program from $B < A$. In most cases, they will be just coercions.

$$
\begin{array}{c}
\xrightarrow{X^-[B/\alpha]} \text{in} \quad \xrightarrow{X^-[A/\alpha]} \quad \xrightarrow{X^+[A/\alpha]} \quad \xrightarrow{X^+[B/\alpha]} \text{out}
\end{array}
$$

For existential types, a program $s$ corresponding to a term of type $X[B/\alpha]$ is made into a program for $\exists \alpha \cdot A \cdot X$ that encodes the concrete type $B$ in the bound type $A$, as shown below.

$$
\begin{array}{c}
\xrightarrow{X^-[A/\alpha]} \text{in} \quad \xrightarrow{X^-[B/\alpha]} \quad \xrightarrow{X^+[B/\alpha]} \quad \xrightarrow{X^+[A/\alpha]} \text{out}
\end{array}
$$

This outlines the terms and types of $\text{int}'$ and their relation to the low-level language.

Equational reasoning for $\text{int}'$ differs from that for $\text{int}$ in that it allows one to equate terms of different type. The intention 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 $A \cdot X$, such a generalisation appears to be necessary to allow a useful reasoning. We formalise it by defining a logical relation $\sim_{X, \rho} [t]$ that relates the low-level program $[t]$ obtained from an $\text{int}'$-term $t$ to a domain-theoretic interpretation $[[t]]$ of $t$. The domain-theoretic interpretation is a standard interpretation of $\lambda$-calculus that ignores low-level annotations. 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 for equational reasoning we may use the domain-theoretic interpretation $[[t]]$, i.e. to ignore low-level annotations, and still obtain useful results about low-level programs. This formalises the intention that the type annotations in $\text{int}'$ represent implementation details, but do not affect the meaning of programs. We refer to Appendix [C.1] for the details, which require a more detailed definition of $\text{int}'$.

### 5.2 Typed Closure Conversion from Annotated PCF to Int'

We can now define typed closure conversion from annotated PCF to $\text{int}'$. For each annotated PCF type $\mathcal{X}$, we define low-level types $\mathcal{B}[\mathcal{X}]$ (bound type) and $\mathcal{C}[\mathcal{X}]_\alpha$ (code type) and an $\text{int}'$-type $\mathcal{I}[\mathcal{X}]_\alpha$ (interface type). We write $\mathcal{C}[\mathcal{X}]$ for $\mathcal{C}[\mathcal{X}]_{\mathcal{B}[\mathcal{X}]}$ and $\mathcal{I}[\mathcal{X}]$ for $\mathcal{I}[\mathcal{X}]_{\mathcal{B}[\mathcal{X}]}$.

- $\mathcal{B}[B \cdot N] = \text{unit}$
- $\mathcal{C}[B \cdot N]_\alpha = \text{int}$
- $\mathcal{I}[B \cdot N]_\alpha = B \cdot I$
- $\mathcal{B}[B \cdot (\mathcal{X} \rightarrow_s \mathcal{Y})] = A$
- $\mathcal{B}[B \cdot (\mathcal{X} \rightarrow_s \mathcal{Y})]_\alpha = A$
- $\mathcal{C}[B \cdot (\mathcal{X} \rightarrow_s \mathcal{Y})]_\alpha = \alpha$
- $\mathcal{I}[B \cdot (\mathcal{X} \rightarrow_s \mathcal{Y})]_\alpha = B \cdot \left( \forall \beta \cdot \mathcal{B}[\mathcal{X}] \rightarrow \mathcal{D}[\mathcal{X}, \alpha \times \mathcal{C}[\mathcal{Y}]_\beta] \right)$

These definitions capture the intended meaning of these types and their interpretation. For instance, the definition for the unit type $\text{unit}$ ensures that it maps to the unit type $\text{int}'$. Similarly, the definition for the function type $\mathcal{X} \rightarrow_s \mathcal{Y}$ ensures that it maps to the function type $\mathcal{I}[\mathcal{X}] \rightarrow \mathcal{I}[\mathcal{Y}]$ in $\text{int}'$. These definitions are designed to preserve the meaning of programs while allowing for a useful equational reasoning framework.
where $\mathcal{M}[\mathcal{Y}]_{S,D} = \exists \gamma \triangleleft \mathcal{B}[\mathcal{Y}] \cdot \mathcal{I}[\mathcal{Y}]_{\gamma} \otimes S \cdot (D \rightarrow [\mathcal{C}[\mathcal{Y}],\gamma])$.

The closure representation appears in the upper bounds of existential types. In the low-level implementation, the bound variable is replaced by its upper bound. Subexponentials capture the right callee-save invariants. Notice that the type $\mathcal{M}[\mathcal{Y}]_{S,D}$ would not be allowed in $\text{INT}$. In $\text{INT}$ we could not separate the callee-save arguments for the evaluation code and the application code for a function. This would lead to more space allocation than needed.

It now remains to define the translation of annotated $\text{PCF}$ terms and to show correctness. The point of this paper is that even though the translation accounts for global program analysis in the choice of low-level details, the correctness proof is not much harder than for the simple typed closure conversion in [24]. We can follow the lines of [24], where correctness is shown using an approach of Benton and Hur [3]. The main difficulty was to identify suitable invariants that capture realistic defunctionalisation as annotations of typed closure conversion and to verify that they are respected by the translation.

The translation from annotated $\text{PCF}$ to $\text{INT}'$ is defined by induction on typing derivations. A derivation $\Pi$ of the typing judgement $x_1 : X_1 \ldots, x_k : X_k \vdash S : Y$ in annotated $\text{PCF}$ is translated to an $\text{INT}$ term $\text{TCC}'(\Pi)$ of the following type:

$$\forall \alpha_1 \triangleleft \mathcal{B}[X_1], \ldots, \alpha_k \triangleleft \mathcal{B}[X_k]. \mathcal{I}[X_1]_{\alpha_1} \circ \ldots \circ \mathcal{I}[X_k]_{\alpha_k} \circ \mathcal{M}[\mathcal{Y}]_{S,C}[x_1,\ldots,x_k]$$

The direct translation from Sec. 3.3 can be seen as a (slightly simplified) direct description of $\text{TCC}'(\Pi)$. The definition of $\text{TCC}'(\Pi)$ is such that if one removes low-level type annotations, then one obtains a translation that is equal, though not identical, to that in [24]. To define a translation to $\text{INT}'$ with the claimed type annotations, in some cases the translation needs to be changed to improve the low-level implementation.

Consider, for example, the case where $\Pi$ ends with rule (APP). Write $\Pi_s$ and $\Pi_t$ for the sub-derivations of the two premises. The translation $\text{TCC}'(\Pi)$ may be written as

$$\text{TCC}'(\Pi) := \text{let } \text{pack}(\alpha, (f, \text{eval } f)) = \text{TCC}'(\Pi_s) \text{ in } \text{let } \text{pack}(\beta, (x, \text{eval } x)) = \text{TCC}'(\Pi_t) \text{ in } \text{let } \text{pack}(\rho, (y, \text{apply})) = f \beta x \text{ in } \text{pack}(\rho, (y, \text{eval } y))$$

as in [24], where

$$\text{eval} := \text{fn } (\vec{x}, \vec{y}) \Rightarrow \text{let } v_f = \text{eval } f(\vec{x}) \text{ in } \text{let } v_x = \text{eval } x(\vec{y}) \text{ in } \text{apply}(v_f, v_x)$$

The term $\text{eval}$ first computes function value, then argument value, and then invokes the application code for the function. The code for function application is obtained by linking the code $f$ for the function with the code $x$ for the argument.

Here we use a different implementation of $\text{eval}$ that defines the same function, but that has a better low-level implementation. The low-level implementation of $\text{eval}$ above is not optimal with respect to its space usage. It stores the variables in $\vec{x}$ until $\text{apply}(v_f, v_x)$ returns, even though they could be disposed of earlier. Hence we use a more space efficient implementation of $\text{eval}$, see Appendix C.4.
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 \((\texttt{if})\), which was restricted to base types in \cite{24} for simplicity. To do so, one defines an \(\texttt{int}\) term \(\texttt{join}: \mathcal{I}[[X_1]]_{A_1} \otimes \mathcal{I}[[X_2]]_{A_2} \rightarrow \mathcal{I}[[X]]_{A_1 + A_2}\) whenever \(X_1 \uplus X_2 = X\). 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 \((\texttt{if})\) in Sec. 3.3 that join the interfaces of \(\langle t_1 \rangle\) and \(\langle t_2 \rangle\) into a single one.

By following the development of \cite{24} and using Lemma \[\text{1}\] from the appendix, we obtain:

**Theorem 2.** Suppose \(\Pi \vdash_A t: B \cdot N\). Then \(t\) reduces to a value \(v\) in a standard call-by-value operational semantics if and only if we have \(\langle \text{TCC}'(\Pi) \rangle: \langle a, () \rangle \mapsto \langle a, v \rangle\) for any closed low-level value \(a: A\).

The proof of this result uses logical relations to relate \(\text{PCF}\) to the \(\text{INT}'\) translation, as in \cite{24}, and to relate \(\text{INT}'\) terms to low-level programs using the logical relation outlined above, see Lemma \[\text{1}\]. These relations can also be used to give a specification of the low-level program fragments obtained by translating (possibly open) term 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. Only upper bounds of the closure types appear in the low-level interfaces.

### 6 Conclusion

The use of global program information in defunctionalisation presents a difficulty for compositional reasoning about compiled low-level code. Not only does such information affect low-level encoding choices, it is also visible in the interfaces of produced code fragments. We have solved this difficulty by developing a new method for compositional reasoning about defunctionalisation by capturing it in terms of typed closure conversion. By managing the low-level details in \(\text{INT}'\), 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 \(\text{PCF}\) captures the essential information needed in the typed closure conversion to \(\text{INT}'\). In a translation from \(\text{PCF}\) 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
<-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. The typed closure conversion makes it
easy to use different closure representation in different places.

The annotated PCF type system should also be of interest for resource usage
control. We believe that it can be used to characterise LOGSPACE along the lines
of [9]. A restriction to constant space may be useful for hardware synthesis [11].
To certify meaningful space bounds for higher-order functional programs, it
suffices bound the depth of the recursive types that appear in type inference.
The main issue is bounding recursion depth, which may be approached using
existing methods, e.g. [13,7,8].

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. For a meaningful performance
evaluation of the produced code, the implementation is currently too simple (it
uses a naive garbage collector and and does not implement the optimisation
of using the machine stack, as described in Sec. [4]. One may nevertheless look
at a few example PCF-programs to check that the generated machine code is
reasonable. For tail-recursive programs, the implementation produces code that
does not access the heap (only values of properly recursive low-level types are
stored on the heap), so that the simplicity of the runtime is not important. The
raytracing example from folder Tests is tail-recursive and the implementation
produces a program that runs in 1.6s without inlining and 0.6s with inlining on
an x64 Intel i7-4770 machine. OCaml and SML versions of the same example
take 1.7s and 2.6s when compiled with ocamlopt 4.02.3 and MLton 20100608
respectively. This is likely mostly due to the quality of the LLVM-backend. 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. [4] is currently
not implemented, so that each recursive call incurs heap allocation and copying.
Rather than implementing such optimisations (which would not be difficult), we
currently intend to focus on possible new applications of the approach, e.g. for
formal verification or as a call-by-value approach to hardware synthesis.

Acknowledgments: I thank the reviewers of an earlier version for their feedback.
References

A PCF and Low-Level Language

For reference, we include the typing rules for our variant of PCF in Fig. 2.

For low-level programs, the typing rules for values are shown in Fig. 3. In these rules, $\Gamma$ 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 $\Gamma \vdash \Phi \vdash b$ that identifies well-typed bodies of blocks. Therein, $\Phi$ 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, $\Phi$ must contain at most one declaration. The typing rules for this judgement are defined in Fig. 4.

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

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 $b_1 \rightarrow_P b_2$, which expresses that body term $b_1$ reduces to body term $b_2$. It is defined to be the smallest relation transitive relation satisfying $f (v) \rightarrow_P b[v/x]$ if $p$ contains a block definition $f (x : A) = b$, and such that the following basic transitions hold.

\[
\begin{align*}
\text{let } & (x, y) = (v, w) \text{ in } b \rightarrow_P b[v/x, w/y] \\
\text{case inl} (v) \text{ of inl} (x) & \Rightarrow b_1; \text{inr} (y) \Rightarrow b_2 \rightarrow_P b_1[v/x] \\
\text{case inr} (w) \text{ of inl} (y) & \Rightarrow b_1; \text{inr} (y) \Rightarrow b_2 \rightarrow_P b_2[v/y] \\
\text{let } \text{fold}_{\mu A} (x) = \text{fold}_{\mu A} (v) \text{ in } b \rightarrow_P b[v/x] \\
\text{let } \text{in}_{A, B} (x) = \text{in}_{A, B} (v) \text{ in } b \rightarrow_P b[v/x]
\end{align*}
\]
Γ, x: A ⊢ x: A
Γ ⊢ (): unit
Γ ⊢ n: int
Γ ⊢ ⟨v, w⟩: A × B

Γ ⊢ v: A
Γ ⊢ inl_{A+B}(v): A + B

Γ ⊢ v: A
Γ ⊢ inr_{A+B}(v): A + B

Γ ⊢ v: A, i ∈ \{1, 2\}
Γ ⊢ \mu α. A(v): \mu α. A

Fig. 3. Typing of Low-Level Values

Γ ⊢ v: A
op ∈ Prim(A, B)
Γ, x: B | Φ ⊢ b
Γ | Φ ⊢ let x = op(v) in b

Γ ⊢ v: A × B
Γ, x: A, y: B | Φ ⊢ b
Γ | Φ ⊢ let ⟨x, y⟩ = v in b

Γ ⊢ v: A + B
Γ, x: A | Φ ⊢ b
Γ, y: B | Φ ⊢ b
Γ | Φ ⊢ case v of inl(x) ⇒ b_1; inr(y) ⇒ b_2

Γ ⊢ v: \mu α. A
Γ, x: A[\mu α. A/α] | Φ ⊢ b
Γ | Φ ⊢ let fold_{\mu α. A}(x) = v in b

Γ ⊢ v: A ∪ B
Γ, x: A | Φ ⊢ b
Γ | Φ ⊢ let in_{A, B}(x) = v in b

Γ | Φ ⊢ v: A
Γ | Φ, f: ¬A, Ψ ⊢ f(v)

Fig. 4. Typing of Low-Level Blocks

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

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

B Annotated PCF

B.1 Translation

We give the missing cases for the translation from Sec. 3.3.

– Case (zero): \( \langle \Pi \rangle \) must have type \( \text{void} + A × \text{unit} → \text{void} + A × \text{int} \). As \( \text{void} \) is the empty type, it suffices to give a program of type \( A × \text{unit} → A × \text{int} \), which we define by \( \langle a, () \rangle → \langle a, 0 \rangle \).
Case (succ): Write $Π_i$ for the derivation of the premise of this rule. After simplification as in the case for (zero), $⟨Π_i⟩$ amounts to a program $A × \text{unit} \rightarrow A × \text{int}$ that computes the value of $t$. We obtain $⟨Π⟩$ by appending to $⟨Π_i⟩$ 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. 3.3, but it remains to define the fragments 3, 4 and 5. To define them, we use the side condition $X_1 \uparrow X_2 = X$ from the premise of (if). For any derivation $Π$ of $X_1 \uparrow X_2 = X$, we define three programs $b_H: C[X_1] + C[X_2] \rightarrow C[X]$ and $\text{in}_H: Z[X_1]^- + Z[X_2]^- \rightarrow Z[X]^-$ and $\text{out}_H: Z[X]^+ \rightarrow Z[X_1]^+ + Z[X_2]^+$.

Then we take $A \cdot b_H$, $\text{in}_H$ and $\text{out}_H$ for 3, 4 and 5, respectively.

The programs $b_H$, $\text{in}_H$ and $\text{out}_H$ are defined by induction on $Π$. In the base case $A \cdot Ν \uparrow A \cdot Ν = A \cdot Ν$, the programs $\text{out}_H$ and $\text{in}_H$ are vacuous, as their interfaces are void. For $b_H$ we choose the program $(x \mapsto \text{case } x \text{ of } \text{inl}(y) \Rightarrow y; \text{inl}(z) \Rightarrow z)$. In the induction case, $Π$ ends with the joining rule for functions. In this case, we let $b_H$ be the identity and define the other two programs in Fig. 5. Blocks 1, 5 and 6 perform case distinction over $D_1 + D_2$ or $B_1 + B_2$. Programs 2, 3 and 4 are $b$, $\text{out}$ and $\text{in}$ from the induction hypothesis for $Y_1 \uparrow Y_2 = Y$.

Cases (subl) and (subr): To translate these rules, we define, for any derivation $Π$ of $X \leq Y$, low-level programs $\text{in}_H: Z[Y]^+ \rightarrow Z[X]^-$ and $\text{out}_H: Z[X]^+ \rightarrow Z[Y]^+$. The definition goes by induction on the derivation $Π$. The base case for $Ν$ is trivial, as the interface $Z[Ν]$ consists of empty types. If $Π$ ends with the subtyping rule for functions, then we write $Π_X$ and $Π_Y$ for the
subderivation deriving its two premises and the desired programs as in Fig. 6.

![Figure 6. Translation of Subtyping](image)

- Case (fix): The implementation of recursion is shown in Fig. 7. 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.

### B.2 Algorithmic Formulation of Typing Rules

The algorithmic typing rules used in the proof of Theorem 1 are shown in Fig. 8. There, we write $\Gamma \triangleright \Delta$ if $\Gamma$ has the form $x_1 : A_1 \cdot X_1, \ldots, x_n : A_n \cdot X_n$ and $\Delta$ has the form $x_1 : B_1 \cdot X_1, \ldots, x_n : B_n \cdot X_n$ and we have $A_i \triangleright B_i$ for $i = 1, \ldots, n$. This restricted form of subtyping is enough for type inference.
Interface of translation of premise $\Gamma, f : G \cdot (\mathcal{X} \xrightarrow{C}\mathcal{Y}), x : \mathcal{X} \vdash t : \mathcal{Y}$:

Translation of conclusion $H \cdot \Gamma \vdash f \text{ fix } x \Rightarrow t : E \cdot (\mathcal{X} \xrightarrow{C}\mathcal{Y})$:

Fig. 7. Translation of Recursion
Fig. 8. Algorithmic Annotated PCF typing
C Factoring the Translation

C.1 Organising Low-Level Programs

We define the type system for \( \mathbf{int}' \) and define the logical relation that relates \( \mathbf{int}' \) terms to low-level programs.

The \( \mathbf{int}' \) typing judgement has the form \( \Omega \mid \Gamma \mid \Phi \vdash t : X \), where \( \Omega = \alpha_1 \triangleleft A_1, \ldots, \alpha_n \triangleleft A_n \) is a context declaring type variables and their bounds, where \( \Gamma = y_1 : B_1, \ldots, y_m : B_m \) is a context declaring value variables, and where \( \Phi = x_1 : X_1, \ldots, x_k : X_k \) 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 \( \mathbf{int}' \) types. A derivation \( \Pi \) of the typing judgement \( \Omega \mid \Gamma \mid \Phi \vdash t : X \) represents a low-level program, called \( L_\Pi \) in a slight abuse of notation, of the following type.

The program \( L_\Pi \) may contain free occurrences of the type variables from \( \Omega \) and the value variables from \( \Gamma \). 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 \( X_{i-} \to X_{i+}, \ldots, X_{k-} \to X_{k+} \) are connected to it using \( \text{iapp} \) from the figure below. These fragments may come from closed terms, for example. Then one obtains a fragment of type \( Y^- \to Y^+ \) that corresponds to \( t \) with the free variables \( x_1, \ldots, x_k \) bound to the fragments.

The typing rules for \( \mathbf{int}' \) are shown in Figs. 9–14. This variant of \( \mathbf{int}' \) 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 \( \triangleleft \)-constraints. The rules make reference to a judgement \( \Omega \vdash A \triangleleft B \), which we define to mean that \( A' \triangleleft B' \) holds, where \( A' \) and \( B' \) are the closed types obtained from \( A \) and \( B \) by replacing each type variable from \( \Omega \) with its upper bound.

The translation from \( \mathbf{int}' \) to the low-level language is defined directly by induction on the typing derivation. We refer to \[21][23\] 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. For
example, the rule for application $st$ is interpreted using iapp from the figure above, that for $t(v)$ is interpreted using vapp, that for $tB$ is interpreted using tapp, and the introduction for existential types is interpreted using pack. The interpretation of the abstraction $\lambda x: X.t$ is essentially the identity and determined by the types. The abstractions $fnx \Rightarrow t$ and $\Lambda \alpha.t$ are interpreted using the following combinators vabs and tabs.

$$
\text{vabs}(p, x) = \begin{cases} 
A \times B_1 & \rightarrow \quad p

\vdots

A \times B_n & \rightarrow \quad A \times C_m

x : A
\end{cases}
$$

$$
\text{tabs}(q, \alpha, A) = \begin{cases} 
B_1[A/\alpha] & \rightarrow \quad q[A/\alpha]

\vdots

B_n[A/\alpha] & \rightarrow \quad C_m[A/\alpha]
\end{cases}
$$

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 $f(z) = \text{let } (x, y) = z \text{ in } g((x, v))$ 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 $\alpha$ in $LtM$(the interpretation of the derivation of the right-hand premise) and then connecting the program $LtM$.

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

$$
\text{comp}_{A,B,C} : \text{unit} \cdot (A \rightarrow [B]) \rightarrow \text{unit} \cdot (B \rightarrow [C]) \rightarrow (A \rightarrow [C])
$$

(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 unit), a strength

$$
\text{str}_{A,B,C} : B \cdot (A \rightarrow [C]) \rightarrow (A \times B \rightarrow [C \times B])
$$

and a fixed-point combinator

$$
\text{fix}_{X,A,B} : B \cdot (A \cdot X \rightarrow X) \rightarrow X
$$

where $\text{unit} + (B \times A) \ll B$.

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.
Fig. 9. Typing Rules for \( \text{int}' \): Axiom and Structural Rules

\[
\begin{align*}
\Omega \mid \Gamma \vdash x \colon X & \quad \Omega \mid \Gamma \mid \Phi \vdash t \colon Y \quad \Omega \mid \Gamma \mid \Phi, \Psi, \Psi' \vdash t \colon Z \\
\Omega \mid \Gamma \vdash x \colon X & \quad \Omega \mid \Gamma \mid \Phi, x \colon X \vdash t \colon Y \\
\Omega \mid \Gamma \mid \Phi, \Psi, \Psi' \vdash t \colon Z
\end{align*}
\]

Fig. 10. Typing Rules for \( \text{int}' \): Basic Computations

\[
\begin{align*}
\Gamma \vdash v \colon A & \quad \Omega \mid \Gamma \mid \Phi \vdash [A] \quad \Omega \mid \Gamma, x \colon A \mid \Psi \vdash t \colon [B] \\
\Omega \mid \Gamma \mid \Phi \vdash \text{return}(v) \colon [A] & \quad \Omega \mid \Gamma \mid \Phi, A \cdot \Psi \vdash \text{let } x = s \text{ in } t \colon [B] \\
\Gamma \vdash v \colon A + B & \quad \Omega \mid \Gamma, x \colon A \mid \Phi \vdash s \colon X \quad \Omega \mid \Gamma, y \colon B \mid \Phi \vdash t \colon X \\
\Omega \mid \Gamma \mid \Phi \vdash \text{case } v \text{ of } inl(x) \Rightarrow s; \text{ inr}(y) \Rightarrow t \colon X
\end{align*}
\]

Fig. 11. Typing Rules for \( \text{int}' \): Value Passing

\[
\begin{align*}
\Omega \mid \Gamma \mid \Phi \vdash t \colon X & \quad \Omega \mid \Gamma, x \colon A \mid \Phi \vdash t \colon X \\
\Omega \mid \Gamma \mid \Phi \vdash \text{fn } x \Rightarrow t \colon A \rightarrow X & \quad \Omega \mid \Gamma \mid \Phi \vdash \lambda x \colon X. t \colon X \rightarrow Y \\
\Omega \mid \Gamma \mid \Phi, x \colon A \cdot B \mid \Phi \vdash x \colon (A \times B) \cdot X \vdash t \colon Y
\end{align*}
\]

Fig. 12. Typing Rules for \( \text{int}' \): Subexponentials

\[
\begin{align*}
\Omega \mid \Gamma \mid \Phi, x \colon A \cdot X, y \colon B \cdot X \vdash t \colon Y & \quad \Omega \mid \Gamma \mid \Phi, z \colon (A + B) \cdot X \vdash t[z/x, z/y] \colon Y \\
\Omega \mid \Gamma \mid \Phi \vdash s \colon X \rightarrow Y & \quad \Omega \mid \Gamma \mid \Phi \vdash s \colon X \otimes Y \\
\Omega \mid \Gamma \mid \Phi \vdash s \colon X \otimes Y & \quad \Omega \mid \Gamma \mid \Phi \vdash \text{let } (x, y) = s \text{ in } t \colon Z
\end{align*}
\]

Fig. 13. Typing Rules for \( \text{int}' \): Functions and Pairs
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 \texttt{INT'} correctly implements its terms when one ignores the type annotations.

\section{Denotational Semantics}

The meaning of terms is defined by a standard domain theoretic semantics. We interpret \texttt{INT'} types by $\omega$-cpos and terms by continuous functions.

A \emph{type environment} $\rho$ is a mapping from type variables to closed low-level types. Each low-level type is interpreted as the ordinary set \(\llbracket A \rrbracket_{\rho}\) consisting of the set of closed values of type \(A[\rho]\), where \((-)[\rho]\) is the type substitution that replaces $\alpha$ with $\rho(\alpha)$.

\texttt{INT'} types are interpreted as $\omega$-cpos. We define an $\omega$-cpo $\llbracket X \rrbracket_{\rho}$ for each type $X$ inductively as follows:

\begin{align*}
\llbracket A \rrbracket_{\rho} &= \bot \\
\llbracket A \cdot X \rrbracket_{\rho} &= \llbracket X \rrbracket_{\rho} \\
\llbracket X \rightarrow Y \rrbracket_{\rho} &= \llbracket Y \rrbracket_{\rho} \\
\llbracket \forall \alpha \triangleleft A. X \rrbracket_{\rho} &= \prod_{B \text{ type}} \llbracket X \rrbracket_{\rho[\alpha \rightarrow B]} \\
\llbracket A \rightarrow X \rrbracket_{\rho} &= \llbracket X \rrbracket_{\rho[A]} \\
\llbracket \exists \alpha \triangleleft A. X \rrbracket_{\rho} &= \sum_{B \text{ type}} \llbracket X \rrbracket_{\rho[\alpha \rightarrow B]} 
\end{align*}

Here, $\Rightarrow$ denotes the $\omega$-cpo of continuous functions, $\prod$ and $X^A$ denote products and $\Sigma$ 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 $\rho$, a value environment $\sigma$ and a module environment $\phi$. A value environment assigns meaning to value variables. A variable $y$ of type $A$ is mapped to an element of $\llbracket A \rrbracket_{\rho}$. A module environment maps \texttt{INT'} variables to their denotation; a variable of type $x$ is mapped to an element of $\llbracket X \rrbracket_{\rho}$. The interpretation of a term $t$ of type $X$ is then
C.3 Relating Implementation and Denotation

Define a family of relations \( \sim_{X,\rho} \) between low-level programs and elements of \( \llbracket X \rrbracket_\rho \) by induction on the type \( X \):

- \( p \sim_{\{A\},\rho} d \) iff: \( p : \text{unit} \to A[\rho] \) and \( p : () \to v \) iff \( d = [v] \).
- \( p \sim_{A \times X,\rho} d \) iff: there exists \( q \) with \( A \cdot q \) 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 \( p_2 \sim_{Y,\rho} d_2 \).
- \( p \sim_{X \times Y,\rho} f \) iff: whenever \( q \sim_{X,\rho} d \), then \( \text{inr}(p, q) \sim_{Y,\rho} f(d) \).
- \( p \sim_{A \times X,\rho} f \) iff: for all \( v \in \llbracket A \rrbracket_\rho \), we have \( \text{vapp}(p, v) \sim_{X,\rho} f(v) \).
- \( p \sim_{A \cdot A,\rho} f \) iff: we have \( \text{tapp}_X(p, B) \sim_{X,\rho} f(B) \) for all \( B \triangleleft A \).
- \( p \sim_{\exists x : A \times X,\rho} f \) iff: there exist \( B, q, d \), such that \( f = (B, d) \) and \( p = \text{pack}(q, B) \) and \( q \sim_{X,\rho} f \).

This definition is extended to terms in context in a logical way. Let \( \Omega \) be the set of all type environments \( \rho \), such that \( (\alpha \triangleleft B) \in \Omega \) implies \( \rho(\alpha) \triangleleft B \). If \( \Gamma \) is the value context \( y_1 : B_1, \ldots, y_n : B_n \), then \( \llbracket \Gamma \rrbracket_\rho \) is a the set of environments \( \sigma \) that map \( y_i \) to \( \llbracket B_i \rrbracket_\rho \), for \( i = 1, \ldots, n \). If \( \Phi \) is \( x_1 : X_1, \ldots, x_n : X_n \), and \( \phi \) is an environment mapping each \( x_i \) to \( \llbracket X_i \rrbracket_\rho \), and if we have \( p_i \sim_{X_i,\rho} \phi(x_i) \) for all \( i = 1, \ldots, n \), then we write \( \bar{p} \sim_{\Phi,\rho} \phi \). With this notation, we extend \( \sim \) to the denotations of terms as follows. Write \( q \sim_{\Omega \upharpoonright \Gamma,\Phi,\rho, X} f \) if, whenever \( \rho \in \Omega \), \( \sigma \in \llbracket \Gamma \rrbracket_\rho \), and \( \bar{p} \sim_{\Phi,\rho, \phi} \), then we have \( \text{iapp}(\bar{p}[\rho][\sigma], \bar{p}) \sim_{X,\rho} f_{\rho,\sigma,\phi} \).

Finally, in order to account for recursion, we close the relation under limits. We write \( p \approx_{\Omega \upharpoonright \Gamma,\Phi,\rho, X} f \) if there exist \( \omega \)-chains \( (p_i)_{i \geq 0}, (f_i)_{i \geq 0} \) with \( p = \bigsqcup_{i \geq 0} p_i \) and
Lemma 1. If $\Pi$ derives $\Omega \mid \Gamma \mid \Phi \vdash t : X$, then $\langle \Pi \rangle \approx_{\Omega \mid \Gamma \mid \Phi,X} [t]$.

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

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

Lemma 3 (Type Substitution). For all $p : \tilde{B} \to \tilde{C}$, whose free value variables all have a type not containing $\alpha$, we have $\text{tapp}_{\tilde{B}, \tilde{C}}(\text{tabs}(p, \alpha, B), A) = p[A/\alpha]$.

Proof (Proof of Lemma 2). The proof goes by induction on the derivation $\Pi$, of which we show a few cases.

- Case:

$$
\frac{\Omega \mid \Gamma \vdash v : A}{\Omega \mid \Gamma \vdash \text{return}(v) : [A]}
$$

The program $\langle \Pi \rangle$ is the block $((\lambda \to v))$. Suppose $\rho \in \lbrack \Omega \rbrack$ and $\sigma \in \lbrack \Gamma \rbrack_{\rho}$. Then, $\text{iapp}(\langle \Pi \rangle[\rho][\sigma], \varepsilon)$ is $((\lambda \to v[\sigma]))$.

By definition of the semantics, we have $\lbrack \text{return}(v) \rbrack_{\rho, \sigma, \phi} = v[\sigma]$, so we get $((\lambda \to v[\sigma])) \sim_{[A], \rho} v[\sigma]$, which is the same as $\text{iapp}(\langle \Pi \rangle[\rho][\sigma], \varepsilon) \sim_{[A], \rho} \lbrack \text{return}(v) \rbrack_{\rho, \sigma, \phi}$. The required assertion follows by taking constant $\omega$-chains.

- Case:

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

The induction hypothesis gives $\langle \Pi_0 \rangle \approx_{\Omega \mid \Gamma, x : A \cdot \Phi,X} [t]$, so there exist $(p_i)$ and $(f_i)$ with $\langle \Pi_0 \rangle = \bigcup_i p_i$ and $f = \bigcup_i f_i$ and $p_i \sim_{\Omega \mid \Gamma, x : A \cdot \Phi,X} f_i$.

Assume $\rho \in \lbrack \Omega \rbrack$, $\sigma \in \lbrack \Gamma \rbrack_{\rho}$ and $\tilde{p} \sim_{A \cdot \Phi \phi}$ $\tilde{q}$. By definition, $\tilde{p}$ must have the form $A \cdot \tilde{q}$, where $\tilde{q} \sim_{\alpha \phi}$. Let $v \in \lbrack A \rbrack_{\rho}$. It suffices to show

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

as we have $\bigcup_i \text{vabs}(p_i, x) = \text{vabs}(\langle \Pi_0 \rangle, x)$ and also $\bigcup_i (v \mapsto (f_i)_{\rho,\sigma[x\mapsto v],\phi}) = (v \mapsto \bigcup_i (f_i)_{\rho,\sigma[x\mapsto v],\phi}) = (v \mapsto f_{\rho,\sigma[x\mapsto v],\phi}) = \lbrack \text{fn} x \Rightarrow t \rbrack_{\rho,\sigma,\phi}$. We have to show

$$
\text{vapp}(\text{iapp}(\text{vabs}(p_i, x)[\rho][\sigma], A \cdot \tilde{q}), v) \sim_{X} (f_i)_{\rho,\sigma[x\mapsto v],\phi},
$$
by definition of $\sim_{A\to X}$. We calculate:

$$vapp(iapp(vabs(p_i, x)[\rho][\sigma], A \cdot \bar{q}), v) = vapp(iapp(vabs(p_i[\rho][\sigma][x], A \cdot \bar{q}), v) = vapp(iapp(p_i[\rho][\sigma][q], x), v) = iapp(p_i[\rho][\sigma, x \mapsto v], \bar{q})$$

The hypothesis gives

$$iapp(p_i[\rho][\sigma[x \mapsto v]], \bar{q}) \sim X (f_i)_{\rho, \sigma[x \mapsto v], \phi},$$

which concludes this case.

- Case:

$$\begin{array}{c}
\Omega \mid \Gamma \mid \Phi \vdash t: A \to X \\
\Omega \mid \Gamma \mid \Phi \vdash t(v): X
\end{array}$$

The induction hypothesis gives $\langle H_0 \rangle \approx \Pi^{t|\Gamma, x: A|\Phi, A \to X \ [t]}$, so there exist $(p_i)_{i \geq 0}$ and $(f_i)_{i \geq 0}$ with $\langle H_0 \rangle = \bigcup p_i$ and $f = \bigcup f_i$ and $p_i \sim \Pi^{t|\Gamma, x: A|\Phi, A \to X} f_i$. Let $\rho \in \Pi^{|\Omega|}$, $\sigma \in |\Gamma|_\rho$ be given and assume $\bar{q} \overset{\phi}{\sim} \phi$. In order to show the required $iapp(vapp([t][\rho][\sigma], v|\sigma]), \bar{q}) \approx X ([t(v)]_{\rho, \sigma, \phi}$ it suffices to show the assertion

$$iapp(vapp(p_i[\rho][\sigma], v|\sigma], \bar{q}) \sim X (f_i)_{\rho, \sigma, \phi(v|\sigma)}$$

for all $i$.

The induction hypothesis gives

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

which implies

$$vapp(iapp(p_i[\rho][\sigma], \bar{q}, v|\sigma]) \sim X (f_i)_{\rho, \sigma, \phi(v|\sigma)},$$

by definition. We now note

$$vapp(iapp(p_i[\rho][\sigma], \bar{q}, v|\sigma]) = iapp(vapp(p_i[\rho][\sigma], v|\sigma], \bar{q}),$$

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

- Case:

$$\begin{array}{c}
\Omega, \alpha \triangleleft A \\
\Gamma | \Phi \vdash t: X
\end{array}$$

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

- Case:

$$\begin{array}{c}
\Omega | \Gamma | \Phi \vdash t: \exists \alpha \triangleleft A \\
\Omega | \Gamma | \Psi, x: X \vdash s: Y
\end{array}$$

This case follows from the substitution lemma, from which we get the equation $tapp(tabs(p, \alpha, A), B) = p[B/\alpha]$.
Let $\rho \in \Omega$, $\sigma \in [\Gamma]$, and assume $q \sim \varphi \phi$ and $r \sim \varphi \psi$.

The induction hypothesis gives us both

$$\text{iapp}(p_i[\rho][\sigma], q) \sim \exists \alpha \in A \cdot X \cdot (f_i)_\rho, \sigma, \phi$$

and

$$\text{iapp}(r_i[\rho[\alpha \rightarrow B]][\sigma], (r', p')) \sim Y \cdot (g_i)_{\rho[\alpha \rightarrow B], \sigma, \psi, f'}$$

whenever $p' \sim_X f'$ for some chains with $\langle \Pi_s \rangle = \bigcup p_i$ and $\|s\| = \bigcup f_i$ and $\langle \Pi_g \rangle = \bigcup g_i$.

The program $\text{iapp}((\text{let} \ pack(\alpha, x) = t \ \text{in} \ \psi)[\rho][\sigma], (q, r))$ is the same as the program

$$\text{iapp}((s)[\rho, \alpha \rightarrow A][\sigma], (q, \text{iapp}(\psi)[\rho][\sigma], r'))$$

by definition. So it suffices to show

$$\text{iapp}(r_i[\rho[\alpha \rightarrow A]][\sigma], (q, \text{iapp}(p_i[\rho][\sigma], r')))$$

$$\sim (g_i)_{\rho, \sigma, \psi, (f_i)_\rho, \sigma, \phi}.$$

From $\text{iapp}(p_i[\rho][\sigma], r') \sim \exists \alpha \in A \cdot X \cdot (f_i)_\rho, \sigma, \phi$ we get $(f_i)_\rho, \sigma, \phi = (B, f_i')$ and $\text{iapp}(p_i[\rho][\sigma], r') = \text{pack}(p_i', B)$ and $p_i' \sim_X \rho[\alpha \rightarrow B] f_i'$.

The hypothesis for $s$ yields $\text{iapp}(r_i[\rho[\alpha \rightarrow B]][\sigma], (r', p_i')) \sim Y \cdot (g_i)_{\rho[\alpha \rightarrow B], \sigma, \psi, f'}$.

Now observe

$$\text{iapp}(r_i[\rho[\alpha \rightarrow A]][\sigma], (r', \text{pack}(p_i', B)))$$

$$= \text{iapp}(r_i[\rho[\alpha \rightarrow B]][\sigma], (r', p_i')),$$

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

![Diagram](image)

We obtain $\text{iapp}(r_i[\rho[\alpha \rightarrow B]][\sigma], (r', \text{pack}(p_i', B))) \sim Y \cdot (g_i)_{\rho[\alpha \rightarrow B], \sigma, \psi, f'}$. Because we also have $\text{iapp}(p_i[\rho][\sigma], r') = \text{pack}(p_i', B)$, this is just as required.

### C.4 Typed Closure Conversion from Annotated PCF to Int'

We spell out in detail the case for (APP) in the definition of $TCC'\langle II \rangle$ that is outlined in the main text.
We implement this as follows.

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

\[
\Omega \vdash \Pi[\mathcal{X}_1/\alpha_1, \ldots, \mathcal{X}_k/\alpha_k] \vdash \mathcal{TCC'}(t) : \mathcal{M}[\mathcal{Y}]_{S,C}
\]

with

\[
\begin{align*}
\Omega &= \alpha_1 < B[\mathcal{X}_1], \ldots, \alpha_k < B[\mathcal{X}_k] \\
C &= \text{unit} \times C[\mathcal{X}_1/\alpha_1, \ldots, \times C[\mathcal{X}_k/\alpha_k]
\end{align*}
\]

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

In the case for \((\text{APP})\) the induction hypothesis derives

\[
\Omega \vdash \Pi[\mathcal{X}/\beta] \vdash \mathcal{TCC'}(s) : \mathcal{M}[\mathcal{Y}]_{U \times \mathcal{X}, C}[\mathcal{X}, \Theta]
\]

and

\[
\Theta \vdash \Pi[\mathcal{X}/\Theta] \vdash \mathcal{TCC'}(t) : \mathcal{M}[\mathcal{Y}]_{U \times \mathcal{X}, C}[\mathcal{X}, \Theta].
\]

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

\[
\begin{align*}
f : \text{unit} \cdot (\forall \beta < B[\mathcal{X}]. C[\mathcal{X}, \beta] \rightarrow \mathcal{M}[\mathcal{Y}]_{U, \alpha \times C}[\mathcal{X}, \beta]) \\
e_f : (U \times C[\mathcal{X}, \beta] \cdot \mathcal{C}[\mathcal{Y}]_{U, \alpha \times C}[\mathcal{X}, \beta]) \\
x : \Pi[\mathcal{X}/\beta] \\
e_x : (U \times C[\mathcal{X}, \beta] \cdot \mathcal{C}[\mathcal{X}, \beta])
\end{align*}
\]

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

\[
\text{let pack}(\rho, \langle y, a \rangle) = f \beta x \text{ in pack}(\rho, \langle y, c \rangle)
\]

for a suitable term \( c : U \cdot (C[\mathcal{X}, \beta] \times \Theta \rightarrow [\Theta]) \).

We want to define \( e \) to be a function that takes an argument of type \( C[\mathcal{X}, \beta] \), then first extracts \( C[\mathcal{X}, \beta] \), \( C[\mathcal{Y}]_\beta \)-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 \( \mathcal{C}[\mathcal{Y}]_\beta \). While we evaluate \( e_f \), we must keep \( \mathcal{C}[\mathcal{Y}]_\beta \) 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 \( \text{seq}_{A,B,C,D} \) for space-efficient sequential composition of two functions with the following type.

\[
B \cdot (A \rightarrow [C]) \rightarrow C \cdot (B \rightarrow [D]) \rightarrow (A \times B) \rightarrow [C \times D]
\]
(Using the strength twice gives terms type \( B \cdot (A \rightarrow [C]) \rightarrow (B \times A \rightarrow [B \times C]) \) and \( C \cdot (B \rightarrow [D]) \rightarrow (C \times B \rightarrow [C \times D]) \). One then defines swapping functions \( A \times B \rightarrow [B \times A] \) and \( B \times C \rightarrow [C \times B] \) 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: \( \alpha \leq C \)):

\[
\begin{align*}
\frac{e_f : C[\Delta] \cdot Z_1, e_x : \alpha \cdot Z_2 \vdash \text{seq } e_f e_x : Z_3}{e_f : (U \times C[\Delta]) \cdot Z_1, e_x : (U \times C) \cdot Z_2 \vdash \text{seq } e_f e_x : U \cdot Z_3}
\end{align*}
\]

In this derivation, we use the abbreviations \( Z_1 = A \rightarrow [\alpha], Z_2 = C[\Delta] \rightarrow [\alpha \times C[X]] \) and \( Z_3 = A \times C[\Delta] \rightarrow [\alpha \times C[X]] \).

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 \( U \cdot (C[\Gamma, \Delta] \rightarrow [\alpha \times C[X]]) \). Since \( a \) has type \( U \cdot (\alpha \times C[\Gamma] \rightarrow [C[Y]]) \), which means that we can complete the definition of \( e \) with the desired type.

We end with a brief outline of Theorem 2.

**Theorem 2.** Suppose \( II \) derives \( \vdash a : B \cdot \mathbb{N} \). Then \( t \) reduces to a value \( v \) in a standard call-by-value operational semantics if and only if, for any any closed low-level value \( a : A \), we have \( \langle TCC'({II}) : \langle a, () \rangle \rightarrow (a, v) \).

The prove it, one shows that \( TCC'({II}) \) implements the denotation of \( t \) in a denotational semantics correctly, as in \( [24] \). To this end, one defines two logical relations \( \leq_X \) and \( \geq_X \) by induction on the type \( X \), such that \( TCC'({II}) \leq_X [t] \) expresses that \( TCC'({II}) \) implements only behaviour that is found in the denotational semantics \( [t] \) of \( t \), and \( TCC'({II}) \geq_X [t] \) expresses that \( TCC'({II}) \) implements at least all behaviour found in \( [t] \). To deal with the more general (ir), e.g. in \( \leq_X \), one shows that \( (A_i, a_i, c_i) \leq_{X_i} f_i \) for \( i = 1, 2 \) implies \( (A_1 + A_2, \text{join}(a_1 \otimes a_2), d_i) \leq_X f_i \), where \( d_1 = \text{inl}(c_1) \) and \( d_2 = \text{inr}(c_2) \) and where \( \text{join} : Z[X_1]_{A_1} \otimes Z[X_2]_{A_2} \rightarrow Z[X]_{A_1 + A_2} \) is the term used in the interpretation of (ir). Using Lemma 1, one concludes from \( TCC'({II}) \leq_X [t] \) and \( TCC'({II}) \geq_X [t] \) that, for any closed \( a \), we have \( \langle TCC'({II}) : \langle a, () \rangle \rightarrow (a, v) \) if and only if \( [t] = [v] \). The result then follows from soundness and adequacy of the denotational semantics of PCF.