# Computation-by-Interaction for Structuring Low-Level Computation

**Ulrich Schöpp**


München 2014

# Computation-by-Interaction for Structuring Low-Level Computation

**Ulrich Schöpp**

Habilitationsschrift
Fakultät für Mathematik und Informatik
der Ludwig–Maximilians–Universität
München

vorgelegt von
Ulrich Schöpp

München, den 3. November 2014

# Abstract

In game semantics and related approaches to programming language semantics, computation is modelled by interaction dialogues. Such models of computation have been used to guide the implementation of programming languages. The idea is to implement interaction dialogues directly and thus realise computation by interaction. In this thesis we study computation-by-interaction as an approach to structuring low-level computation. We capture semantic structure of interactive computation in terms of a typed $\lambda$-calculus INT and study its use for organising low-level computation. We start by considering the practical application of INT as a language for low-level programming. Next we show that it allows fine-grained control over space usage by using it to characterise the complexity class of the functions computable in logarithmic space. We then show how INT can be used to translate functional languages with call-by-name and call-by-value evaluation strategy to a low-level language. We observe that the translation for call-by-name is closely related to standard compilation techniques, namely CPS-translation and defunctionalization. We use the translation of call-by-value as an example to illustrate the use of the structure identified by INT for non-trivial reasoning about low-level programs.

# Synopsis

This thesis gives an introduction to computation-by-interaction as an approach to structuring low-level computation. It summarises results of the following articles.

1. Ulrich Schöpp. Stratified Bounded Affine Logic for Logarithmic Space. In *Logic in Computer Science (LICS)*, 2007.
2. Ugo Dal Lago and Ulrich Schöpp. Functional Programming in Sublinear Space. In *European Symposium on Programming (ESOP)*, 2010.
3. Ugo Dal Lago and Ulrich Schöpp. Type Inference for Sublinear Space Functional Programming. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2010.
4. Ulrich Schöpp. Computation-by-Interaction with Effects. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2011.
5. Ulrich Schöpp. Organising Low-Level Programs using Higher Types. In *Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2014.
6. Ulrich Schöpp. Call-by-Value in a Basic Logic for Interaction. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2014.
7. Ulrich Schöpp. On the Relation of Interaction Semantics to Continuations and Defunctionalization. In *Logical Methods in Computer Science (LMCS)*, accepted, to appear.

   This is an extended version of the following conference article:
   Ulrich Schöpp. On Interaction, Continuations and Defunctionalization. In *Typed Lambda Calculi and Applications (TLCA)*, 2013

## Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise, and that this work has not been submitted for any other degree or professional qualification.

My own contribution to the co-authored papers 2. and 3. can roughly be specified as 75%.

## Acknowledgements

I would like to thank Martin Hofmann, Simone Martini and Hans-Jürgen Ohlbach for acting as academic mentors my habilitation process. I would like to thank Paul-André Melliès, Martin Hofmann and Simone Martini and for serving as referees for this thesis. Thanks are also due to Ugo Dal Lago, who coauthored two of the publications included in this thesis.

# Contents

# 1 Introduction

Modern programming languages are built on high-level abstractions that allow one to implement, control and analyse complex systems. While high-level abstractions have long been a focus of programming language theory, low-level computation has not received as much attention. However, recent advances in formal verification and certification have highlighted the need for a good understanding of the structure of low-level computation. The CompCert project has demonstrated that the development of a formally certified compiler for C is now within reach of formal methods (Leroy, 2009). In such developments one must account for all details of low-level implementation and prove their correctness formally. This requires an identification of the logical principles of low-level languages that are needed to implement high-level abstractions.

Formal certification does not stop at correctness. Resource usage is another essential aspect of computation. While formal certification of correctness has come within reach of existing methods, the analysis and certification of resource usage remains a challenging problem. Since resource usage is a property of low-level computation, its analysis requires a good understanding of how high-level abstractions are implemented by low-level programs.

The study of low-level computation has led to the realisation that low-level computation has interesting logical structure. Methods from logic and programming language theory are being applied to low-level languages in various ways. For example, types are being used in assembly language (Morrisett et al., 1998) and in order to guide the translation from high-level languages to machine code (Morrisett et al., 1999). Optimisation passes on low-level code are verified in formal logics (Zhao et al., 2013). The relation of code at various levels of abstraction is being captured using multi-language semantics (Perconti and Ahmed, 2014). There is work on resource analysis that connects reasoning about low-level programs with high-level reasoning, e.g. in the CerCo project (Amadio et al., 2013). This is by no means an exhaustive list, but it illustrates the use of logical tools in the context of low-level languages.

This thesis is about structural aspects of low-level computation. We study a simple low-level language with first-order data types and the ability to jump to fixed labels. It is modelled on the SSA-form low-level intermediate languages used by many compilers (Rastello, 2015; Appel, 1998). Such languages have a low abstraction level by design. They allow explicit control over the low-level details that are important for efficiency, e.g. memory management on stack and heap. Nevertheless, their uses in compilation exhibit patterns that suggest that it is useful to organise them using higher-order structure. The suggestion

is not to abstract from low-level details, as one would like to control these explicitly, but to capture patterns of low-level program construction and of reasoning about them.

Let us outline some of the structural patterns of low-level computation.

**Composition and Parameterisation** For many purposes it is useful to consider low-level programs as being composed of a number of program fragments. For instance, one may like to formulate a compiler for a higher programming language as a compositional translation. Each part of a source program is being translated to a low-level program fragment and these fragments are composed to make up the whole compiled program. To define such a translation, one defines a low-level program fragment for the primitive parts of the source language and explains how they can be composed to make up the whole compiled program.

But what is a low-level program fragment? It should be program code together with some information of how it can be completed. While it is possible to define fragments in an ad-hoc way, we argue that it is useful to capture a notion of program fragment systematically in terms of parameterisation of programs over programs. We consider low-level program fragments as low-level programs that are parameterised over the rest of the low-level program with which it is to be composed, see Chapter 3.

**Specification and Reasoning** Capturing how low-level programs can be composed also relates to the question of how to specify the behaviour of low-level program fragments. If the translation of (parts of) a high-level language produces only fragments of low-level programs, then it may not be immediately clear what it means for the translation to be correct, especially because low-level programs expose many implementation details. Ideally, specification and reasoning should be compositional, just as the translation. One possible approach to achieve this would be to require that a program fragment behaves correctly once it has been composed with suitable programs implementing the parts that are missing from the fragment. To make this idea precise, one needs to explain how programs can be composed and one needs to specify the allowed behaviour of the missing parts.

Notice that this natural approach quickly leads to tricky higher-order reasoning. The missing parts may themselves be parameterised by program fragments that may in fact be provided by the program fragment with which we combine them with. Parameterisation of this kind appears naturally in Chapters 6 and 7. To keep reasoning manageable, it appears to be essential to have a good formalisation of program parameterisation and composition.

**Code vs. Data** Another issue that is important for low-level computation is the distinction between code and data and the interplay between them. While in von Neumann architectures all code eventually becomes data, there is nevertheless a conceptual distinction between program code and data. For example, when writing a compiler one often wants to achieve separate compilation, i.e. that program modules can be compiled separately to low-level programs and that a full program can be obtained by linking the low-level programs. Separate compilation can be considered at various degrees of separation. A strong

form of separate compilation would be such that the compiler translates each module into a machine code file, which can then be linked with the system linker, perhaps even against code generated by a C compiler. A weaker form of separate compilation would be given by a compiler that produces a separate bytecode file for each source module. Such bytecode files can be executed using an interpreter, but it may not be possible to link them directly against machine code generated by a C compiler. There is a qualitative difference between the two kinds of separate compilation. In the first case, the compiler produces fully separate machine code. In the second case all modules are executed by the same code (the interpreter) and the separate compilation just produces separate data (the bytecode). To be able to express such differences, one needs to be able to distinguish conceptually between code and data.

A distinction between code and data also appears when one studies the fine-grained structure of the translation of higher-order languages into low-level languages. Minamide et al. (1996) introduce closure conversion, a key step in translating higher-order programs into first-order low-level programs, by: *'Closure conversion is a program transformation that achieves a separation between code and data.'* For questions of efficiency and resource usage, it is important to understand what is static code and what is data.

The interplay between code and data appears throughout this thesis, first in Chapter 3, where definitions are given, and then in Chapters 6 and 7 where translations of call-by-name and call-by-value are studied.

**Data Representation** Low-level computation exhibits issues that are often hidden at a higher level. At a low-level, encoding details are visible and one needs to take this into account in reasoning and specification, see Chapter 7 for an example. Other questions, such as efficient memory management, manifest themselves in low-level programs. It is interesting to study the logical principles underlying such issues.

In this thesis we argue that low-level computation can be organised using simple and natural higher-order structure. The emphasis is on the organisation of low-level computation, as opposed to the abstraction from low-level details. The aim is to allow explicit control of low-level details, while at the same time making use of higher-order constructs to control parameterisation, code composition and similar issues.

## 1.1 Computation-by-Interaction

In the work summarised in this thesis, we consider *computation-by-interaction* as a particular approach to structuring low-level computation. This approach is based on ideas from Game Semantics (Blass, 1992; Hyland and Ong, 1995; Abramsky et al., 2000) and related approaches, such as the Geometry of Interaction (Girard, 1989; Abramsky et al., 2002). At first sight, it is not obvious how these concepts are related to low-level computation. Identifying a relation between them is one of the contributions of this thesis; in Chapter 6 we outline a formal relation (Schöpp, 2014b) between an interactive semantics and standard techniques in compiler construction.

Game semantics is based on the idea of modelling computation by interaction dialogues. This approach goes back to the study of constructive models of logic (Lorenzen, 1961). The idea is to explain a logical sentence by how one may attack it in a dialogue and how it may be defended. A proof of a sentence is a strategy that explains how to defend against any possible attack. In programming language semantics (Hyland and Ong, 1995; Abramsky et al., 2000), types take the place of sentences and attacks become requests for information. Thus, programs of a certain type are interpreted as strategies that tell how to answer any possible request for information for this type.

One important aspect of game semantics is compositionality. The strategy for a composite program can be built from the strategies of its parts. To answer a request for information, one sends requests to the parts of the program and combines their answers to compute the answer of the initial request. This way of explaining the meaning of a program can be seen as letting the parts of the program enter into a dialogue.

This interactive explanation of programs moreover suggests an approach to their *implementation*. To compute the result of a program, it suffices to implement its strategy as a program that maps requests to answers. Compositionality can be used to guide such an implementation. To implement the strategy of a program, one implements the strategy of each of its parts and then composes the obtained programs into a single program that plays out their interaction. This is possible with very few assumptions about the computational model. Even restricted low-level languages are expressive enough to implement such computation by interaction.

This idea of using interactive semantics as an implementation technique, can be found in the literature in a number of forms. Interaction has been implemented using abstract machines (Mackie, 1995; Danos et al., 1996), hardware circuits (Ghica, 2007; Ghica and Smith, 2014), Offline Turing Machines (Dal Lago and Schöpp, 2010a, 2013), the $\pi$-calculus (Berger et al., 2001), distributed programs (Fredriksson and Ghica, 2013), quantum circuits (Yoshimizu et al., 2014), to name just a few. The wide range of examples illustrates that there are many ways of implementing interaction dialogues. In the $\pi$-calculus it may be implemented using message passing between communicating processes. In hardware circuits, strategies are implemented by static circuits, whose interaction is realised by connecting them with electrical wires.

In this thesis we look at computation-by-interaction as a technique for structuring low-level computation. That many instances of computation-by-interaction were developed independently illustrates that this approach identifies useful structure, and also that it should be useful to understand the underlying concepts in general and independently of the particular application. Instead of starting with certain interactive models and studying how they can be implemented by low-level programs, we start with a low-level language and ask what structure we can construct by constructing an interactive model from it. The idea is to use the structure of this interactive model as an approach to organising the low-level language.

The low-level language that we consider here is derived from SSA-form compiler intermediate languages, such as the one of Ziarek et al. (2008). In this language, interaction dialogues take the form of program traces. On should think of the implementation of a

strategy as a fragment of a goto program. To request information, one jumps to a certain point in this fragment. It will compute the answer and jump to an external return label to give the reply, the answer being encoded in the value of some variable.

## 1.2 Int-Construction

The plan is therefore to construct an interactive model from the low-level language and to study its structure. To construct such a model, we use the *Int-construction* of Joyal et al. (1996), see (Dal Lago and Schöpp, 2010a, 2013). It can be seen as a simple core of both Abramsky-Jagadeesan-Malacaria games (Abramsky et al., 2000) and also of the particle-style Geometry of Interaction (Abramsky et al., 2002).

The Int-construction generalises the construction of the integers from the semiring $(\mathbb{N}, +, 0)$ of natural numbers. Integers may be represented by pairs $(x^-, x^+)$ of natural numbers $x^- \in \mathbb{N}$ and $x^+ \in \mathbb{N}$, the intention being that the pair $(x^-, x^+)$ represents the integer $x^+ - x^-$. The order on integers may be defined using the order on natural numbers: $(x^-, x^+) \leq (y^-, y^+)$ if and only if $x^+ + y^- \leq x^- + y^+$. This order can be used to explain when two pairs represent the same integer. The set of integers are defined by taking a quotient that identifies such pairs,

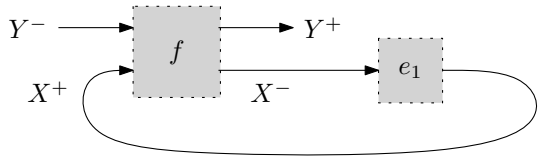
The Int-construction of Joyal et al. (1996) generalises this construction from natural numbers to traced monoidal categories. Let us outline one instance of this construction, which is particularly relevant for the purposes of this thesis. It is the instance where natural numbers are replaced by the category **Pfn** of sets and partial functions, where addition becomes disjoint union and where 0 becomes the empty set.

In this instance, pairs of natural numbers are replaced by pairs $(X^-, X^+)$ of two sets $X^-$ and $X^+$. One may think of such a pair as specifying the interface of an interactive entity. The elements of $X^-$ are the requests that one can send to the entity and the elements of $X^+$ are the possible answers.

What was an inequality between integers, now becomes a way of transforming one interface into another. A morphism from $(X^-, X^+)$ to $(Y^-, Y^+)$ is given by a partial function

$$f \colon X^+ + Y^- \to X^- + Y^+ \ .$$

Such a function explains how one can answer requests for interface $(Y^-, Y^+)$ if one already knows how to answer requests for interface $(X^-, X^+)$. Suppose we have a function $e_1 \colon X^- \to X^+$ that explains how to answer requests for interface $(X^-, X^+)$. Then $f$ allows us to construct a function $e_2 \colon Y^- \to Y^+$ that answers requests for interface $(Y^-, Y^+)$. The construction may be depicted as follows.




(1.1)

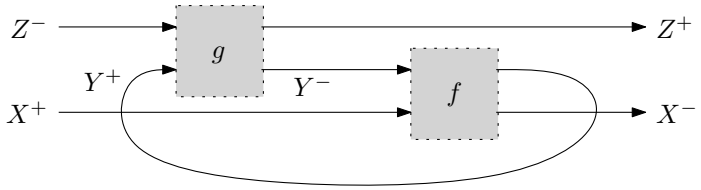
A concrete implementation in Standard ML can be given thus:

```
fun e2 y =
  let fun loop m =
        case f m of
          inr(y) => y
        | inl(x) => loop (inl(e1 x))
  in loop (inr(y)) end
```

Notice that $e_1$ may be invoked several times from $f$. One may think of $e_2$ as being implemented as an interactive process in which $f$ and $e_1$ engage in a dialogue.

A particular special case of morphisms are morphisms from $(\emptyset, \emptyset)$ to $(X^-, X^+)$. Such morphisms can be seen as entities that implement the interface $(X^-, X^+)$. Indeed, there exists a unique partial function $e_1 \colon \emptyset \to \emptyset$, so the morphism induces an implementation $X^- \to X^+$ as described just above.

The composition of two morphisms $f \colon (X^-, X^+) \to (Y^-, Y^+)$ and $g \colon (Y^-, Y^+) \to (Z^-, Z^+)$ is implemented as depicted below.




Composition is thus realised as an interactive process rather than a sequential one, which will be important in Chapter 5.

This outlines how the Int-construction organises **Pfn** into a model of interactive computation. The same idea can be used with a low-level language in place of **Pfn**. Then, sets become types and partial functions become low-level programs that implement a partial functions. But overall the idea is the same; one should just think of the functions in **Pfn** as being implemented in a low-level language.

The motivation for organising a low-level language into a model of interactive computation with the Int-construction is to identify useful structure. Such structure has been studied in the context of Game Semantics (Abramsky et al., 2000) and the Geometry of Interaction (Abramsky and Jagadeesan, 1994; Abramsky et al., 2002). It was shown that interactive models have rich structure, enough to give precise models of a wide range of programming languages, e.g. (Hyland and Ong, 2000; Abramsky et al., 2000; Laird, 2001; Abramsky et al., 2004; Murawski and Tzevelekos, 2013).

Let us briefly outline some of the basic structure that the Int-construction provides.

**Pairs** To work with pairs, one may use monoidal structure (Mac Lane, 1998) that is defined by $X \otimes Y = (X^- + Y^-, X^+ + Y^+)$. Informally, $X \otimes Y$ is the interface of an interactive entity that implements both the interface $X$ and the interface $Y$. The set $(X \otimes Y)^-$ of the possible queries for $X \otimes Y$ consists of the queries of both $X$ and $Y$. The possible answers for $(X \otimes Y)$ include the possible answers from both $X$ and $Y$. In particular, if we know how to answer queries for interface $X$, and also for interface $Y$, then we can also answer queries for the interface $X \otimes Y$.

**Functions** The Int-construction produces monoidal closed structure (Mac Lane, 1998), which is given by $X \multimap Y = (X^+ + Y^-, X^- + Y^+)$. The implementations of this interface may be thought of as an interactive kind of functions. A function from $X$ to $Y$ explains how one can transform any entity of type $X$ into one of $Y$. Implementations of the interface $X \multimap Y$ achieve this as described for morphisms above. Suppose we have an implementation of interface $X \multimap Y$, i.e. a function $f\colon X^+ + Y^- \to X^- + Y^+$ and an implementation $e_1\colon X^- \to X^+$ of interface $X$. Then we obtain an implementation of interface $Y$ by connecting $f$ and $e_1$ as shown in (1.1).

The definition of $X \multimap Y$ may also be described in game semantic terms. In Abramsky-Jagadeesan-Malacaria games (Abramsky et al., 2000), a play for a function starts with a request for the result, here an element of $Y^-$. The function may answer this request with an element of $Y^+$, or decide to query its argument with a query from $X^-$. In the latter case, the query may be answered by providing the answer in $X^+$ as a new 'request' to the function. The definition of $X \multimap Y$ by $(X^+ + Y^-, X^- + Y^+)$ is just right to allow this kind of interaction. The definition of $X \multimap Y$ here does not include any requirement on the order of messages. Game semantics imposes such requirements, e.g. by asking that answers are not given without preceding question. Here we focus on the implementation of interactive behaviour and do not impose such restrictions a priori.

The simple definitions of $\otimes$ and $\multimap$ identify already enough structure to interpret a linear higher-order $\lambda$-calculus, i.e. a functional programming language without duplication. Its terms are interpreted as morphisms in the Int-construction. Thus, one may use them to construct low-level programs that implement interaction strategies. We shall argue that this useful for working with low-level languages.

**Duplication** To interpret not just a linear $\lambda$-calculus, we need a construct for the duplication of interfaces. To this end, we may define $A \cdot X = (A \times X^-, A \times X^+)$ for any set $A$. The idea is that $A \cdot X$ is $X \otimes \cdots \otimes X$, where there is one copy of $X$ for each element of $A$. The elements of $A$ can be seen as the names of the copies. A request $\langle a, r\rangle \in A \times X^-$ means that $r$ is requested from the $a$-th copy of $X$. An answer would be given in the form $\langle a, s\rangle \in A \times X^+$.

This outlines some of the structure that one can naturally identify in the interactive universe obtained by applying the Int-construction to **Pfn**. Further structure will be defined using a typed $\lambda$-calculus in Chapter 3.

In this thesis we argue that the notion of computation-by-interaction identified by the Int-construction is useful for structuring low-level computation. It allows us to capture notions of parameterisation of programs over code and also over data, it identifies interfaces in a way that is useful for specification, it allows us to distinguish between code and data and to formalise the interaction between these two concepts. We argue that being able to capture these notions is useful for structuring low-level programming tasks. We show that it allows structured low-level programming (Chapter 4), that it is useful for programming with sublinear space (Chapter 5), that it is closely related to existing efficient compilation

techniques (Chapter 6), and that it makes available logical principles for reasoning about low-level programs (Chapter 7).

## 1.3 Overview

This thesis is intended to summarise the results of the publications listed in the Synopsis on page vii, to explain them in a common context and how they relate to each other. The aim is to motivate the results, to explain them informally and put them into relation. For technical details we refer to the original publications.

In Chapter 2, we begin by giving a definition of the low-level language.

In Chapter 3, we identify structure in the low-level language by organising it using a higher-order $\lambda$-calculus INT. The rest of the thesis can be seen as an investigation of INT.

In Chapter 4, we argue that INT captures useful a useful approach to low-level programming. We outline how the higher-order structure of INT can be used for programming in a way that still allows access to low-level details. We demonstrate this using an implementation of coroutines as an example.

In Chapter 5, we consider applications to resource usage analysis. We consider the space usage of programs and outline that INT provides a fine-grained way of controlling the space usage of higher-order programs. We show that this can be used to obtain an expressive characterisation of the functions computable in logarithmic space.

In Chapters 6 and 7, we consider the use of INT for the compilation of higher-order functional languages with call-by-name and call-by-value evaluation strategies.

In Chapter 6, we consider the compilation of a call-by-name functional language using INT. We outline that the resulting compilation method can be seen as a structured reconstruction of a call-by-name CPS-translation followed by a defunctionalization procedure. This shows that the approach of modelling computation by interaction allows a structured presentation of a defunctionalizing compilation method, including a simple correctness proof.

Chapter 7 is motivated by the relation of interactive computation and defunctionalization outlined in Chapter 6. It presents an implementation of call-by-value in INT. This implementation demonstrates what we gain from capturing interactive structure using INT; the proof of correctness provides a showcase for the logical principles identified by INT.

# 2 Low-Level Programs

For low-level programming, we use a simple first-order language. This language is similar to the SSA-form intermediate languages that are used in many compilers, see e.g. (Ziarek et al., 2008). The current formulation is based on that in Schöpp (2014a). Conceptually, it may be seen as a goto language, formulated for use in compilation.

Low-level programs are typed and work with values of the following first-order types.

$$\begin{array}{rrcl} \text{Value Types} & A, B & ::= & \alpha \mid \mathsf{nat} \mid \mathsf{unit} \mid A \times B \mid 0 \mid A + B \mid \mu\alpha.\, A \\ \text{Values} & v, w & ::= & x \mid n \mid * \mid \langle v, w\rangle \mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \end{array}$$

Low-level programs are built from *blocks* of the form $f\,(x\colon A) = b$, where $f$ is the block label, $x$ is a formal parameter, $A$ is the type of the formal parameter and $b$ is the body of the block, formed according to the following grammar. We sometimes hide the type annotation $A$ for better readability.

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

In this grammar $v$ ranges over values, $g$ over block labels and $op$ over primitive operation constants. The variables $x$ and $y$ are bound in the respective expressions. The set of operation constants that take arguments of type $A$ and return values of type $B$ is defined by a set $\mathit{Prim}(A, B)$. We use the following primitive operations.

$$\begin{array}{rcl} \{\mathsf{print}\} & \subseteq & \mathit{Prim}(\mathsf{nat}, \mathsf{unit}) \\ \{\mathsf{add}, \mathsf{sub}, \mathsf{mul}, \mathsf{div}\} & \subseteq & \mathit{Prim}(\mathsf{nat} \times \mathsf{nat}, \mathsf{nat}) \\ \{\mathsf{eq}, \mathsf{lt}\} & \subseteq & \mathit{Prim}(\mathsf{nat} \times \mathsf{nat}, \mathsf{unit} + \mathsf{unit}) \end{array}$$

Furthermore, we write short $\mathsf{G}$ for the type of binary lists $\mu\alpha.\, (\mathsf{unit} + \mathsf{unit}) \times \alpha$. Any closed value can be encoded as a value of this type. For any closed type $A$, we assume primitive operations $\mathsf{encode}_A \in \mathit{Prim}(A, \mathsf{G})$ and $\mathsf{decode}_A \in \mathit{Prim}(\mathsf{G}, A)$ that implement encoding and decoding. We assume them as primitive operations for technical convenience; it would also

be possible to implement them in the low-level language. We assume that these are all primitive operations.

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

For example, a program to compute the factorial can be written as follows.

$$\begin{aligned}
&fac\,(x\colon \mathsf{nat}) = facacc(\langle x, 1\rangle)\\
&facacc\,(z\colon \mathsf{nat}\times\mathsf{nat}) = \mathsf{let}\ \langle x, acc\rangle{=}z\ \mathsf{in}\\
&\qquad\qquad\qquad\qquad \mathsf{let}\ b{=}\mathsf{eq}(\langle x, 0\rangle)\ \mathsf{in}\\
&\qquad\qquad\qquad\qquad \mathsf{case}\ b\ \mathsf{of}\ \mathsf{inl}(u) \Rightarrow ret(acc)\\
&\qquad\qquad\qquad\qquad\qquad\quad ;\ \mathsf{inr}(v) \Rightarrow \mathsf{let}\ acc'{=}\mathsf{mul}(\langle x, acc\rangle)\ \mathsf{in}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{let}\ x'{=}\mathsf{sub}(\langle x, 1\rangle)\ \mathsf{in}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad facacc(\langle x', acc'\rangle)
\end{aligned}$$

The entry label is *fac* and the exit label is *ret*.

It is sometimes convenient to work with programs with more than one entry or exit label. A program with two entry labels $entry_1$ and $entry_2$ with argument types $A$ and $B$ respectively can be made into a program with a single entry label with argument type $A + B$ by adding the block

$$\begin{aligned}
entry\,(z\colon A + B) = \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}(x) &\Rightarrow entry_1(x)\\
;\ \mathsf{inr}(y) &\Rightarrow entry_2(x)\ .
\end{aligned}$$

Similarly, two exit labels $exit_1$ and $exit_2$ with argument types $A$ and $B$ respectively can be turned into a single one with argument type $A + B$ by adding blocks

$$exit_1\,(x\colon A) = exit(\mathsf{inl}(x)), \qquad exit_2\,(x\colon B) = exit(\mathsf{inr}(x))\ .$$

We shall use programs with more than one entry or exit label with the understanding that they are converted to programs with single labels as outlined.

## 2.1 Typing

Programs are typed in the canonical way.

**Values** The typing rules for values are shown below. In these rules, $\Sigma$ is a *value context*, which is a finite list of variable declarations $x\colon A$. As usual, each variable may be declared at most once in $\Sigma$.

$$\frac{x\colon A\ \text{in}\ \Sigma}{\Sigma \vdash x\colon A} \qquad \frac{}{\Sigma \vdash n\colon \mathsf{nat}} \qquad \frac{}{\Sigma \vdash *\colon \mathsf{unit}} \qquad \frac{\Sigma \vdash v\colon A \qquad \Sigma \vdash w\colon B}{\Sigma \vdash \langle v, w\rangle\colon A \times B}$$

$$\frac{\Sigma \vdash v\colon A}{\Sigma \vdash \mathsf{inl}(v)\colon A+B} \qquad \frac{\Sigma \vdash v\colon B}{\Sigma \vdash \mathsf{inr}(v)\colon A+B} \qquad \frac{\Sigma \vdash v\colon A[\mu\alpha.\, A/\alpha]}{\Sigma \vdash \mathsf{fold}(v)\colon \mu\alpha.\, A}$$

**Programs** To identify well-typed programs, we first define a judgement $\Sigma \mid \Phi \vdash b$ that identifies well-typed bodies of blocks. Therein, $\Phi$ is a *label context*, which is a list of declarations of the form $f\colon \neg A$, expressing that the block with label $f$ takes arguments of type $A$. For each label, $\Phi$ must contain at most one declaration.

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

$$\frac{\Sigma \vdash v\colon A \times B \qquad \Sigma, x\colon A, y\colon B \mid \Phi \vdash b}{\Sigma \mid \Phi \vdash \mathsf{let}\ \langle x, y\rangle{=}v\ \mathsf{in}\ b}$$

$$\frac{\Sigma \vdash v\colon A+B \qquad \Sigma, x\colon A \mid \Phi \vdash b_1 \qquad \Sigma, y\colon B \mid \Phi \vdash b_2}{\Sigma \mid \Phi \vdash \mathsf{case}\ v\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow b_1; \mathsf{inr}(y) \Rightarrow b_2}$$

$$\frac{\Sigma \vdash v\colon \mu\alpha.\, A \qquad \Sigma, x\colon A[\mu\alpha.\, A/\alpha] \mid \Phi \vdash b}{\Sigma \mid \Phi \vdash \mathsf{case}\ v\ \mathsf{of}\ \mathsf{fold}(x) \Rightarrow b}$$

$$\frac{\Sigma \vdash v\colon A}{\Sigma \mid \Phi, f\colon \neg A, \Psi \vdash f(v)}$$

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

We write $p\colon A \to B$ if $p$ is a program whose entry and exit labels accept values of type $A$ and $B$ respectively.

The typing judgement allows for programs with free value variables. This is useful for the construction of low-level programs. A program is *closed* if it is well-typed in the empty value context.

## 2.2 Operational Semantics

The operational semantics is defined for closed low-level programs. The operational semantics of a closed program $p$ is given by a relation $b_1 \xrightarrow{o}_p b_2$, which expresses that body term $b_1$ reduces to body term $b_2$, while giving the sequence of closed values $o$ as output using the print-operation. We write $\varepsilon$ for the empty sequence and $o_1 o_2$ for concatenation of $o_1$ and $o_2$.

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

$$\begin{aligned}
\mathsf{let}\ \langle x, y\rangle{=}\langle v, w\rangle\ \mathsf{in}\ b &\xrightarrow{\varepsilon}_p b[v/x, w/y] \\
\mathsf{case}\ \mathsf{inl}(v)\ \mathsf{of}\ \mathsf{inl}(x){\Rightarrow}b_1; \mathsf{inr}(y){\Rightarrow}b_2 &\xrightarrow{\varepsilon}_p b_1[v/x] \\
\mathsf{case}\ \mathsf{inr}(w)\ \mathsf{of}\ \mathsf{inl}(x){\Rightarrow}b_1; \mathsf{inr}(y){\Rightarrow}b_2 &\xrightarrow{\varepsilon}_p b_2[v/y] \\
\mathsf{case}\ \mathsf{fold}(v)\ \mathsf{of}\ \mathsf{fold}(x) \Rightarrow b &\xrightarrow{\varepsilon}_p b[v/x] \\
\mathsf{let}\ x{=}\mathsf{print}(v)\ \mathsf{in}\ b &\xrightarrow{v}_p b[\mathsf{unit}/x] \\
\mathsf{let}\ x{=}\mathsf{add}(\langle m, n\rangle)\ \mathsf{in}\ b &\xrightarrow{\varepsilon}_p b[m+n/x]
\end{aligned}$$

We omit the reductions for $\mathsf{sub}$, $\mathsf{mul}$, $\mathsf{div}$, $\mathsf{eq}$ and $\mathsf{lt}$. For $\mathsf{encode}_A$ and $\mathsf{decode}_A$, we choose for each closed value $v\colon A$ an encoding $\lceil v \rceil\colon \mathsf{G}$ and define:

$$\begin{aligned}
\mathsf{let}\ x{=}\mathsf{encode_A}(v)\ \mathsf{in}\ b &\xrightarrow{\varepsilon}_p b[\lceil v \rceil/x] \\
\mathsf{let}\ x{=}\mathsf{decode_A}(\lceil v \rceil)\ \mathsf{in}\ b &\xrightarrow{\varepsilon}_p b[v/x]
\end{aligned}$$

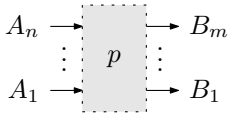
The only property of $\mathsf{encode}_A$ and $\mathsf{decode}_A$ that we will use is that encoding followed by decoding is the identity.

The operational semantics is a specification of the meaning of low-level programs. In an implementation on a machine, variables would typically be stored in registers or on the machine stack. In similar low-level languages like LLVM (Lattner and Adve, 2004), the register allocator is responsible for the storage of variables, so that their values will be stored in machine registers if possible and on the stack otherwise.

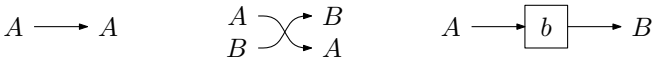
We consider two closed programs $p, q\colon A \to B$ *extensionally equal*, if they have the same observable effects and return the same values: Whenever $\mathit{entry}_p(v) \xrightarrow{o}_p b$ then $\mathit{entry}_q(v) \xrightarrow{o}_q b'$ for some $b'$, whenever $\mathit{entry}_p(v) \xrightarrow{o}_p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

## 2.3 Graphical Notation

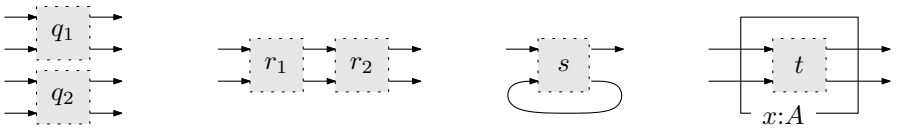
We use standard graphical notation (Selinger, 2011) for working with low-level programs. A program $p\colon A_1 + \cdots + A_n \to B_1 + \cdots + B_m$ is depicted as follows.




Basic programs are drawn as follows. The identity program $\mathit{id}_A\colon A \to A$ is shown on the left below. The evident swapping program $\mathit{swap}_{A,B} : B + A \to A + B$ appears in the middle. A single block $b$ is drawn by a solid box as shown on the right. For coherent isomorphisms, such as $A \times (B \times C) \to (A \times B) \times C$, we write just $\bullet$ instead of the box, as they are determined by the types.




The following figure shows constructions of low-level programs in terms of graphical notation.




For two programs $q_1 \colon A \to B$ and $q_2 \colon C \to D$, their vertical composition is the sum $q_1 + q_2 \colon A + C \to B + D$, which is the program obtained by renaming all labels in $q_1$ and $q_2$ so that no label appears in both programs. The resulting program is considered as a program with two entry and exit labels, from which we obtain a program with one entry and exit label as described above.

The horizontal composition of $r_1 \colon A \to B$ and $r_2 \colon B \to C$ stands for the sequential composition $r_2 \circ r_1 \colon A \to C$ and is defined similarly. Loops are defined by jumping from the exit to entry label.

We use a box notation as shown on the right in the figure. For any program $t \colon B \to C$ in context $\Sigma$, $x \colon A$, the box denotes a program $t \colon A \times B \to A \times C$ in context $\Sigma$. This program is defined by giving all blocks an additional argument of type $A$, which is bound to variable $x$. This value is passed on unchanged in between all blocks in $p$. The box may be explained so that it binds the $A$-part of an input to the variable $x$, executes the program and then retrieves the value from $x$ again when the program returns. If the variable $x$ does not appear free in program $t$, then we write just $A$ instead of $x{:}A$ as an annotation of the box.

The graphical notation is similar to proof nets for linear logic (Girard, 1996; Melliès, 2006).

# 3 A Calculus for Computation-by-Interaction

This chapter describes higher-order structure of low-level programs in terms of a $\lambda$-calculus INT. This calculus distinguishes between code and data and makes their interaction explicit. It captures forms of parameterisation of low-level programs, both over data and over other programs. It formalises value passing and also code composition by linking of separate low-level programs. These concepts are derived from the semantic structure of the Int-construction. While INT is a simple calculus, we argue in Chapters 4–7 that it identifies interesting structure of low-level computation.

In this chapter we introduce INT. This version of the calculus was first presented in (Schöpp, 2014c). It can be seen as a variant of INTML, studied in (Dal Lago and Schöpp, 2010a,b; Schöpp, 2011; Dal Lago and Schöpp, 2013), which is itself based on $\lambda$-calculi for linear logic, such as Dual Light Affine Logic of Atassi et al. (2006). It may also be understood as identifying core structure underlying Stratified Bounded Affine Logic (Schöpp, 2007).

## 3.1 The Calculus Int

The calculus INT is a typed $\lambda$-calculus with the following two classes of types.

$$\begin{array}{rrcl} \text{Value Types} & A, B & ::= & \alpha \mid \mathsf{nat} \mid \mathsf{unit} \mid A \times B \mid 0 \mid A + B \mid \mu\alpha.\, A \\ \text{Interface Types} & X, Y & ::= & TA \mid A \to X \mid A \cdot X \multimap Y \mid \forall\alpha.\, X \end{array}$$

The value types are exactly as in the low-level language. The interface types represent the interfaces of low-level programs. Terms of these types will represent programs that implement these interfaces. Thus, the terms of interface types represent static low-level program code with a particular interface.

We write short $X \multimap Y$ for $\mathsf{unit} \cdot X \multimap Y$.

We start with an informal explanation of interface types. The type $TA$ represents programs that may be started and that may return a value of type $A$. The type $A \to X$ represents programs that are parameterised over a value of type $A$. If we have a value of type $A$, then we can specialise any program of type $A \to X$ to obtain a program of type $X$. The type $A \cdot X \multimap Y$, on the other hand, allows for the parameterisation of programs over

programs. It represents programs that expect to be linked to a program of type $X$. When linked, they become a program of type $Y$. The value type $A$ in $A \cdot X \multimap Y$ will be explained below. We refer to it as a *subexponential* and it may be thought of as a generalisation of the exponential ! of Linear Logic (Girard, 1987). That is, one may understand the types of the form $A \cdot X \multimap Y$ much like the function spaces $!X \multimap Y$ in linear type systems, e.g. (Barber, 1996; Atassi et al., 2006).

More formally, we say that a *program with interface* $X$ is a low-level program of type $X^- \to X^+$, where $X^-$ and $X^+$ are value types that are defined as follows.

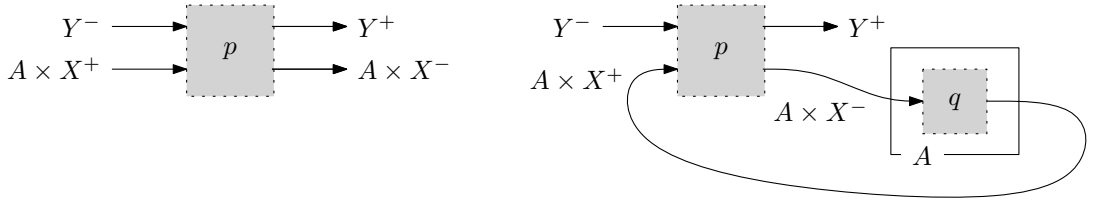
$$\begin{aligned} (TA)^- &= \mathsf{unit} & (TA)^+ &= A \\ (A \to X)^- &= A \times X^- & (A \to X)^+ &= X^+ \\ (A \cdot X \multimap Y)^- &= A \times X^+ + Y^- & (A \cdot X \multimap Y)^+ &= A \times X^- + Y^+ \\ (\forall \alpha.\, X)^- &= X^-[\mathsf{G}/\alpha] & (\forall \alpha.\, X)^+ &= X^+[\mathsf{G}/\alpha] \end{aligned}$$

Thus, a program with interface $X$ has an entry label with argument type $X^-$ and an exit label with argument type $X^+$. In working with these types, it is sometimes convenient to implicitly apply the isomorphisms $(\mathsf{unit} \times A) \cong A \cong (A \times \mathsf{unit})$ to simplify the presentation. In particular, we shall use the isomorphisms $(A \to TB)^- = A \times \mathsf{unit} \cong A$ and $(\mathsf{unit} \cdot X \multimap Y)^- = \mathsf{unit} \times X^+ + Y^- \cong X^+ + Y^-$ and $(\mathsf{unit} \cdot X \multimap Y)^+ = \mathsf{unit} \times X^- + Y^+ \cong X^- + Y^+$.

A program with interface $TA$ is a program with an entry label to start to computation and an exit label that returns the result of type $A$ of the computation.

The interface $A \to X$ differs from $X$ only in that the entry label expects an additional value of type $A$. It formalises parameterisation of programs over values.

The interface $A \cdot X \multimap Y$ formalises parameterisation of programs over programs. A program $p$ with interface $A \cdot X \multimap Y$ has the type shown on the left below. It represents a function in the sense that whenever $q$ is a program with interface $X$, then we obtain a program with interface $Y$ by linking the two as shown on the right below.




(3.1)

Note in particular that when $p$ jumps to $q$ with a value $\langle a, v \rangle : A \times X^-$, then $q$ will return with a value of the form $\langle a, w \rangle : A \times X^+$, i.e. $a$ is unchanged. The value $a$ can be used by $p$ like a callee-save value in low-level programming. The program $p$ does not have state and cannot store any data across a call to $q$. By choosing the type $A$ appropriately, any value that $p$ needs to preserve can be encoded in the value $a$, which is returned unchanged with any return value.

We call the type $A$ in $A \cdot X \multimap Y$ a subexponential annotation. One may think of $A \cdot X$ as a generalisation of the exponential $!X$ of linear logic. The instance $\mathsf{G} \cdot X$ may be used

as a stand-in for $!X$. For this instance, the type system for INT will support contraction. For other instances, such as $(\mathsf{unit} + \mathsf{unit}) \cdot X$, contraction is not available, however. There is similarity to the subexponentials of Nigam and Miller (2009).

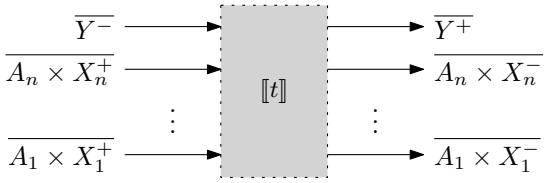
Finally, the type $\forall\alpha.\, X$ allows for data polymorphism. In low-level programs we use the type $\mathsf{G}$ to represent polymorphic data, as $\mathsf{G}$ can encode the values of any type. We substitute $\mathsf{G}$ for any occurrence of a type variable, so that polymorphic programs can process any kind of data. The advantage of using universal quantification over using $\mathsf{G}$ explicitly is that one can reason about programs using parametricity. Polymorphism will be essential in Chapter 7.

The terms and typing judgements of INT are defined in Figures 3.1-3.3. The typing judgements of INT have the form

$$\Sigma \mid \Gamma \vdash t \colon X \ ,$$

where $\Sigma$ is a value-context as in Section 2.1, and $\Gamma$ is an interface context of the form $x_1 \colon A_1 \cdot X_1, \ldots, x_n \colon A_n \cdot X_n$. This context $\Gamma$ expresses that each $x_i$ is a program with interface $X_i$. Each variable in $\Gamma$ appears under a subexponential, whose meaning is as for functions above. We write $B \cdot \Gamma$ for the context $x_1 \colon (B \times A_1) \cdot X_1, \ldots, x_n \colon (B \times A_n) \cdot X_n$.

The intention is that a term $\Sigma \mid \Gamma \vdash t \colon X$ represents a low-level program $[\![t]\!]$ with the following interface that is well-typed in context $\Sigma$.

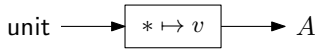



In this figure, we write $\overline{A}$ for the type obtained from $A$ by substituting $\mathsf{G}$ for all free type variables. In the following, we shall implicitly assume that in low-level programs all free type variables are replaced by $\mathsf{G}$. That is, we shall usually write just $A$ for $\overline{A}$.

The interface of program $[\![t]\!]$ should be understood like the interface of functions above. The intention is that eventually, for each variable $x_i \colon A_i \cdot X_i$, a program with interface $X_i$ will be connected, just like in $q$ is connected to $p$ in (3.1) above. Notice in particular that $[\![t]\!]$ can therefore assume that the value of type $A_i$ remains unchanged between a jump to $x_i$ and a return. All subexponentials will maintain this invariant.

In the rest of this section, we explain the typing rules of INT. The low-level program $[\![t]\!]$ is defined by induction on the typing derivation. For each typing rule we explain which low-level program it translates to.

The typing rules in Figure 3.1 give access to the basic computations of the low-level language in INT. Rule RET is the computation that just returns a value. Its conclusion is translated to the following program, in which we assume that all free type variables in $v$ are replaced with $\mathsf{G}$.

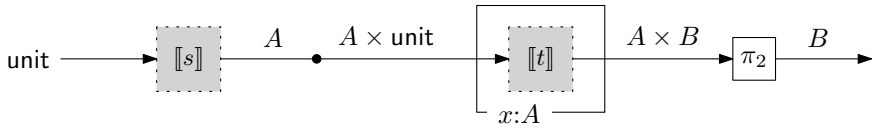


$$\textsc{ret}\ \frac{\Sigma \vdash v\colon A}{\Sigma \mid - \vdash \mathsf{return}\ v\colon TA} \qquad \textsc{bind}\ \frac{\Sigma \mid \Gamma \vdash s\colon TA \qquad \Sigma, x\colon A \mid \Delta \vdash t\colon TB}{\Sigma \mid \Gamma, A \cdot \Delta \vdash \mathsf{let}\ x{=}s\ \mathsf{in}\ t\colon TB}$$

$$\textsc{prim}\ \frac{\Sigma \vdash v\colon A \qquad p \in \mathit{Prim}(A, B)}{\Sigma \mid - \vdash p(v)\colon TB}$$

Figure 3.1: Computations

Rule BIND allows the composition of computations. The term $\mathsf{let}\ x{=}s\ \mathsf{in}\ t$ first runs $s$, binds the returned value of type $A$ to variable $x$ and then runs $t$. This is implemented by connecting the low-level programs for $s$ and $t$ as follows.




In this figure, the inputs and outputs for the contexts $\Gamma$ and $\Delta$ in the programs $[\![s]\!]$ and $[\![t]\!]$ are not shown. These inputs and outputs are just passed to the outside. With the box around $[\![t]\!]$, this explains why the conclusion of rule BIND has context $\Gamma$, $A \cdot \Delta$, as opposed to $\Gamma, \Delta$. By passing through the box, the type of the connections for $\Delta$ changes to the types for $A \cdot \Delta$ (up to coherent isomorphisms of the form $A \times (B \times X^-) \cong (A \times B) \times X^-$). Rule PRIM gives access to the primitive operations of the low-level language. Its low-level program is defined much like that for rule RET.

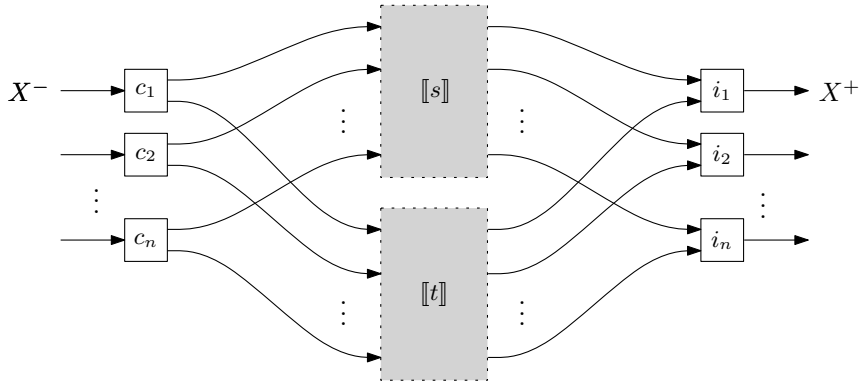
$$\times\mathrm{E}\ \frac{\Sigma \vdash v\colon A \times B \qquad \Sigma, x\colon A, y\colon B \mid \Gamma \vdash t\colon X}{\Sigma \mid \Gamma \vdash \mathsf{let}\ \langle x, y\rangle{=}v\ \mathsf{in}\ t\colon X}$$

$$+\mathrm{E}\ \frac{\Sigma \vdash v\colon A + B \qquad \Sigma, x\colon A \mid \Gamma \vdash t_1\colon X \qquad \Sigma, y\colon B \mid \Gamma \vdash t_2\colon X}{\Sigma \mid \Gamma \vdash \mathsf{case}\ v\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow t_1; \mathsf{inr}(y) \Rightarrow t_2\colon X}$$

$$\mu\mathrm{E}\ \frac{\Sigma \vdash v\colon \mu\alpha.\, A \qquad \Sigma, x\colon A[\mu\alpha.\, A/\alpha] \mid \Gamma \vdash t\colon X}{\Sigma \mid \Gamma \vdash \mathsf{case}\ v\ \mathsf{of}\ \mathsf{fold}(x) \Rightarrow t\colon X}$$

Figure 3.2: Value Eliminations

Figure 3.2 contains rules for the elimination of values in INT. We show the translation of $+\mathrm{E}$ to low-level programs:




This figure shows also the inputs and outputs for the contexts, as they are not just passed to the outside. The block labelled with $c_k$ is a case distinction over the value $v$, which can be computed using the variables in $\Sigma$. The block $i_k$ just forwards the values from each of the two inputs to its one output.

$$\textsc{ax}\ \frac{}{\Sigma \mid x\colon \mathsf{unit} \cdot X \vdash x\colon X} \qquad \textsc{contr}\ \frac{\Sigma \mid \Delta \vdash s\colon X \qquad \Sigma \mid \Gamma, x\colon A \cdot X, y\colon B \cdot X \vdash t\colon Y}{\Sigma \mid \Gamma, (A+B) \cdot \Delta \vdash \mathsf{copy}\ s\ \mathsf{as}\ x, y\ \mathsf{in}\ t\colon Y}$$

$$\textsc{weak}\ \frac{\Sigma \mid \Gamma \vdash t\colon X}{\Sigma \mid \Gamma, \Delta \vdash t\colon X} \qquad \textsc{exch}\ \frac{\Sigma \mid \Gamma, \Delta \vdash t\colon X}{\Sigma \mid \Delta, \Gamma \vdash t\colon X} \qquad \textsc{struct}\ \frac{\Sigma \mid \Gamma, x\colon A \cdot X \vdash t\colon Y}{\Sigma \mid \Gamma, x\colon B \cdot X \vdash t\colon Y}\ A \lhd B$$

$$\to\!\textsc{i}\ \frac{\Sigma, x\colon A \mid \Gamma \vdash t\colon X}{\Sigma \mid A \cdot \Gamma \vdash \mathsf{fn}\, x{:}A.\, t\colon A \to X} \qquad \to\!\textsc{e}\ \frac{\Sigma \mid \Gamma \vdash t\colon A \to X \qquad \Sigma \vdash v\colon A}{\Sigma \mid \Gamma \vdash t(v)\colon X}$$

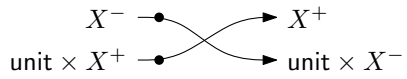
$$\multimap\!\textsc{i}\ \frac{\Sigma \mid \Gamma,\, x\colon A \cdot X \vdash t\colon Y}{\Sigma \mid \Gamma \vdash \lambda x{:}A{\cdot}X.\, t\colon A \cdot X \multimap Y} \qquad \multimap\!\textsc{e}\ \frac{\Sigma \mid \Gamma \vdash s\colon A \cdot X \multimap Y \qquad \Sigma \mid \Delta \vdash t\colon X}{\Sigma \mid \Gamma, A \cdot \Delta \vdash s\ t\colon Y}$$

$$\forall\textsc{i}\ \frac{\Sigma \mid \Gamma \vdash t\colon X}{\Sigma \mid \Gamma \vdash \Lambda\alpha.\, t\colon \forall\alpha.\, X}\ \alpha \text{ not free in } \Sigma, \Gamma \qquad \forall\textsc{e}\ \frac{\Sigma \mid \Gamma \vdash t\colon \forall\alpha.\, X}{\Sigma \mid \Gamma \vdash t\ A\colon X[A/\alpha]}$$

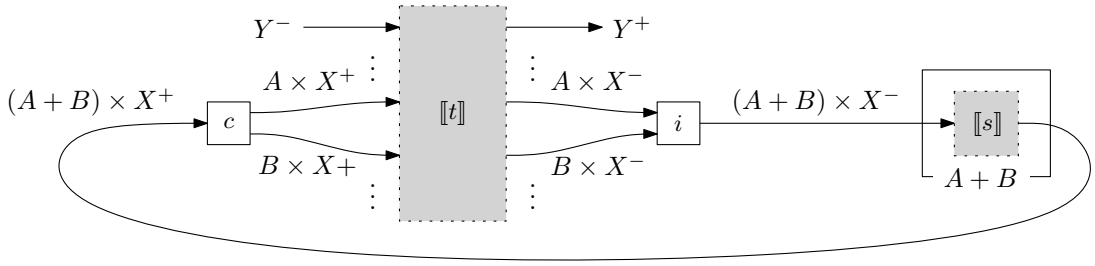
$$\textsc{direct}\ \frac{\Sigma \mid \Gamma \vdash t\colon X^- \to TX^+}{\Sigma \mid \Gamma \vdash \mathsf{direct}_X(t)\colon X}\ (*)$$

Figure 3.3: Structural and Logical Rules

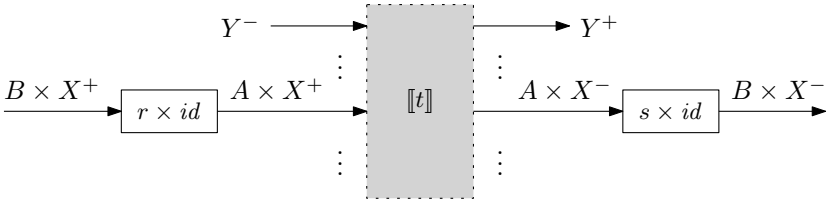
The rules for the new interface types of INT appear in Figure 3.3. Rule AX just forwards requests and answers to and from $x$. If one connects a program for $x$ as described above, then the variable rule will give back extensionally the same program.



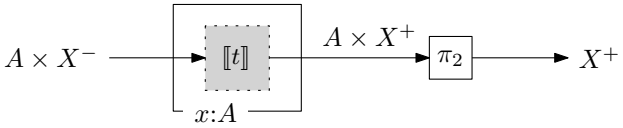

Rule COPY implements duplication by sharing. It is translated to the following low-level program, in which $c$ implements case distinction over the value of type $(A + B)$ in the input, and $i$ is an injection. Notice how a single copy of $[\![s]\!]$ serves the two variables $x$ and $y$ in the term $t$.



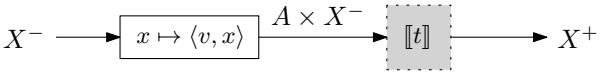

Rule STRUCT allows weakening of subexponentials. Its side condition $A \lhd B$ expresses that $A$ is a retract of $B$, which means that there exist low-level programs $s\colon A \to B$ (section) and $r\colon B \to A$ (retraction), such that $r \circ s = id$ holds. Rule STRUCT amounts to inserting $r$ and $s$ as shown below. It is sound, as the invariant for subexponentials is such that programs with interface $X$ are connected to $[\![t]\!]$ only so that they return any input value of type $B$ unchanged with the return value. When $[\![t]\!]$ sends a value of type $\langle a, x\rangle\colon A \times X^-$, the answer will thus have the form $\langle r(s(a)), y\rangle\colon A \times X^+$, i.e. $\langle a, y\rangle$, which shows that rule STRUCT preserves the invariant. We note that the semantic definition of $A \lhd B$ using a section-retraction pair is used only to obtain a flexible calculus. For specific applications, it is useful to choose syntactic approximations of this notion that are more tractable, e.g. for type inference, see (Dal Lago and Schöpp, 2010a,b) for possible choices.



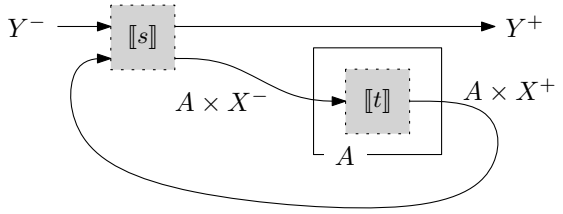

Rules $\to$I and $\to$E implement parameterisation of programs over values. Rule $\to$I binds the input value of type $A$ to variable $x$ and evaluates $[\![t]\!]$. When $[\![t]\!]$ returns, the value can safely be discarded.



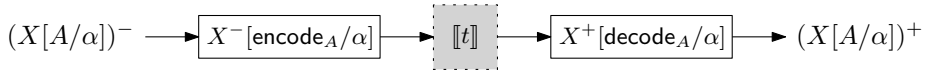

Rule $\to$E computes the value $v$ and provides it to $[\![t]\!]$.



Rules $\multimap$I and $\multimap$E implement parameterisation of programs over programs. Since the INT typing judgements already formalise such a parameterisation, rule $\multimap$I is essentially the identity. Only the interface of the program $[\![t]\!]$ is interpreted differently in that the inputs and outputs for the variable $x$ are now counted not towards the context, but toward the type $A \cdot X \multimap Y$. Rule $\multimap$E implements application by linking of function and argument as shown below.




The rules $\forall$I and $\forall$E implement value polymorphism by encoding into values of type $\mathsf{G}$. Rule $\forall$I is again the identity, as for$\multimap$I, as we have assumed that all type variables are substituted for by $\mathsf{G}$. The instantiation rule $\forall$E works by encoding any value of the instantiated type $A$ into a value of type $\mathsf{G}$ before sending it to $[\![t]\!]$ and by decoding them again in any output of $[\![t]\!]$.




Here we write $B[\mathsf{encode}_A/\alpha]$ for the program of type $B[\mathsf{G}/\alpha] \to B[A/\alpha]$, which lifts $\mathsf{encode}_A$ in the context given by $B$.

It remains to explain rule DIRECT for direct definition. This rule allows one to define the program $[\![t]\!]$ for any type $X$ directly. The sequents in the premise and in the conclusion of rule DIRECT have isomorphic interfaces, so that $[\![t]\!]$ can be considered as a program with either interface. Rule DIRECT just allows us to interpret the term $t$ in a different way. If it has type $X^- \to TX^+$, then it can also be viewed as having type $X$. The point is that this allows us write higher-order programs that could otherwise not be written. Rule DIRECT has a side condition $(*)$ that will be defined at the end of Section 3.4.1.

This concludes the description of the typing rules of INT. We give example INT programs in the next chapters. With direct definition it is possible to define interesting higher-order combinators, e.g. for tail recursion, recursion, call/cc, block-scoped variables, coroutines, etc. (Dal Lago and Schöpp, 2010a, 2013; Schöpp, 2014c). Examples appear in Chapter 4. With such combinators, programming in INT is similar in spirit to approaches using Idealized Algol (Reynolds, 1997).

## 3.2 Related Work

### 3.2.1 Effect Calculi

Effect calculi, such as Call-by-Push-Value (CBPV) (Levy, 2004) or the Enriched Effect Calculus (EEC) (Egger et al., 2009) study the properties of computational effects in higher-

order computation. They feature a very similar distinction between values and computations as INT. The fragment of INT with the following types can be seen as a fragment of these calculi as well.

$$\begin{aligned} A, B &::= \alpha \mid \mathsf{nat} \mid \mathsf{unit} \mid A \times B \mid 0 \mid A + B \mid \mu\alpha.\, A \\ X, Y &::= TA \mid A \to X \end{aligned}$$

In CBPV and EEC, the types $X$ and $Y$ are called computation types.

One main conceptual difference between these effect calculi and INT is that the construction of INT is based on a low-level language with tail recursion. This assumption of tail recursion allows us to use the Int-construction and construct the function type $A \cdot X \multimap Y$, which does not appear in effect calculi. Note that the Int-construction makes essential use of tail recursion for the implementation of composition, as outlined in Section 3.1. The Enriched Effect Calculus features a linear function space $X \multimap Y$, but this appears to be different from the function space considered here. In this sense, INT is stronger than the above-cited effect calculi.

INT is also weaker than the effect calculi, in that there is no way to turn an interface type into a value type. In effect calculi it is very important that computations like $A \to X$ can be turned into values, which is not possible in INT. It should be possible to add such a feature to INT, e.g. by adding a value type $UX$ whose values encode programs with interface $X$. However, INT is intended to capture structure of low-level computation, e.g. for applications in compilation. Being able to be explicit about low-level details, such as the representation of closures, is one of the main reasons for using low-level languages. For this reason, we study INT without a built-in way of turning programs into values, as implementing such encodings is the very task of low-level languages.

The relation of INT to effect calculi can be clarified by studying the categorical structure underlying both approaches. This is done in (Schöpp, 2011), where the structure captured by INTML is described in categorical terms. In essence, INT can be considered as being constructed from effect calculi with tail recursion by an application of the Int-construction to the category of computations. This is explained in (Schöpp, 2011), and also in (Dal Lago and Schöpp, 2013, §3.7). While categorical structure has been very important for the development of INT, not least because it is based on the categorical Int-construction, for the presentation in this thesis we focus on the type theoretic formulation using the calculus INT and refer to loc. cit. for further information.

### 3.2.2 Tensorial Logic

Mellies' Tensorial Logic (Melliès, 2012) is a logic that conceptually appears to be very close to INT. Tensorial logic is a primitive linear logic of tensor and negation with formulae of the following form.

$$X, Y ::= A \mid X \otimes Y \mid \neg X \mid 1$$

If one considers programs in continuation-passing style in INT, then it is natural to consider the type $\bot^A := A \to T0$, which is the interface of programs that can be started

with a value of type $A$ and that never return. We have found that to implement call-by-value in continuation-passing style in INT, the fragment with the following interface types is useful, see (Schöpp, 2014a); the encoding of call-by-value is also outlined in Chapter 7.

$$X, Y \;::=\; \bot^A \;\mid\; X \multimap Y \;\mid\; \forall \alpha.\, X$$

If one adds a type $X \otimes Y$ (we have not done so only for simplicity), then one could restrict the function space to $X \multimap \bot^A$, i.e. to a form of negation. It appears that one obtains a type system that is very close to the work on Tensorial Logic.

While a formal relationship remains to be established, it appears that Tensorial Logic and such fragments of INT are related in many ways. For example, Melliès describes the use of many copies of $\bot$, similar to $\bot^A$ in INT, in his slides on Tensorial Logic with Algebraic Effects (Melliès, 2012), see also (Melliès, 2014). There is also a fundamental relation between Tensorial Logic and game semantics, as shown by Melliès (2012), which further suggests a close relation of the approaches.

### 3.2.3 Coeffect Calculi

Subexponentials in INT can be seen as a generalisation of the exponentials of linear logic. Such generalisations have been found useful for a number of different applications, which has motivated work to capture them by more general calculi (Petricek et al., 2013; Ghica and Smith, 2014; Brunel et al., 2014). This work is based on the observation that exponentials and their generalisations appear as comonadic notions of computation. In reference to the fact that monads capture computational effects, this work has coined the term coeffect for these notions of computation.

The type system INT, and its predecessor INTML, can be seen as particular coeffect type systems. We conjecture that $A \cdot X$ can be described as a parametric comonad in the sense of Katsumata (2014), which would make a connection precise. The details remain to be worked out in future work.

It is interesting to note that INT does not appear to be a direct instance of the coeffect type systems in loc. cit. The value contexts of INT and INTML interact with subexponentials and other coeffect type system do not have such value contexts. Investigating such issues is another interesting direction for further work.

### 3.2.4 IntML

Finally, we comment briefly on the relation of INT and INTML (Dal Lago and Schöpp, 2010a, 2013), on which INT is strongly based. First, the types $A \to X$ and $\forall \alpha.\, X$ are new in INT. These new types are useful in Chapter 7, for example. A second difference is that in INTML the terms are separated into two levels. There are terms for low-level programming and terms for interactive programming, the latter of which corresponds to the terms of INT. This choice of terms corresponds closely to the Int-construction. Low-level terms are used to write programs in a category $\mathbb{C}$ of computations and interactive terms

denote programs in INT($\mathbb{C}$), the category obtained from $\mathbb{C}$ by the Int-construction. The formulation in INT comes without a separate class of terms for writing low-level programs and is closer syntactically to effect calculi. We outline in Section 4.2 that low-level programs can nevertheless still be written in INT. For the purposes of this thesis, the differences between INT and INTML are mainly syntactic.

## 3.3 Operational Semantics

So far, the terms of INT have been given a meaning by translation to the low-level language. To justify the use of the $\lambda$-calculus in INT, we should convince ourselves that the calculus justifies a reasonable equational theory.

A first possible way to do this is to consider INT as a programming language, to specify an operational semantics for it, and to show that the translation to the low-level language correctly implements this operational semantics. This approach was worked out for INTML in (Dal Lago and Schöpp, 2010a, 2013). Here we outline the approach in terms of INT. For simplicity, we consider INT without the print-operation.

An operational semantics for this language may be defined by a binary reduction relation $\longrightarrow$ between well-typed closed terms. We define $\longrightarrow$ as the smallest relation that includes the following reductions

$$\begin{aligned}
\mathsf{add}(\langle m, n\rangle) &\longrightarrow m + n \\
\mathsf{eq}(\langle n, n\rangle) &\longrightarrow \mathsf{inl}(*) \\
\mathsf{eq}(\langle m, n\rangle) &\longrightarrow \mathsf{inr}(*) \text{ if } m \neq n \\
\mathsf{let}\ x{=}\mathsf{return}\ v\ \mathsf{in}\ t &\longrightarrow t[v/x] \\
\mathsf{let}\ \langle x, y\rangle{=}\langle v, w\rangle\ \mathsf{in}\ t &\longrightarrow t[v/x, w/y] \\
\mathsf{case}\ \mathsf{inl}(v)\ \mathsf{of}\ \mathsf{inl}(x){\Rightarrow}t; \mathsf{inr}(y){\Rightarrow}s &\longrightarrow t[v/x] \\
\mathsf{case}\ \mathsf{inr}(w)\ \mathsf{of}\ \mathsf{inl}(x){\Rightarrow}t; \mathsf{inr}(y){\Rightarrow}s &\longrightarrow s[w/y] \\
\mathsf{case}\ \mathsf{fold}(v)\ \mathsf{of}\ \mathsf{fold}(x) \Rightarrow t &\longrightarrow t[v/x] \\
\mathsf{copy}\ s\ \mathsf{as}\ x, y\ \mathsf{in}\ t &\longrightarrow t[s/x, s/y] \\
(\lambda x{:}A{\cdot}X.\, t)\ s &\longrightarrow t[s/x] \\
(\mathsf{fn}\, x{:}A.\, t)(v) &\longrightarrow t[v/x] \\
(\Lambda\alpha.\, t)\ A &\longrightarrow t[A/\alpha]
\end{aligned}$$

(analogous cases for the other primitive operations are omitted) and that is closed under the following congruence rules:

$$\frac{s \longrightarrow s'}{s\ t \longrightarrow s'\ t} \qquad \frac{t \longrightarrow t'}{s\ t \longrightarrow s\ t'} \qquad \frac{s \longrightarrow s'}{\mathsf{let}\ x{=}s\ \mathsf{in}\ t \longrightarrow \mathsf{let}\ x{=}s'\ \mathsf{in}\ t}$$

There are no general reduction rules for direct. Reductions for direct must be considered on a case-by-case basis for each instance.

For this operational semantics, we have the following correctness result: If $- \mid - \vdash s\colon X$ and $s \longrightarrow t$, then $- \mid - \vdash t\colon X$ and the low-level programs $[\![s]\!]$ and $[\![t]\!]$ are extensionally equal.

The proof is carried out in detail for INTML in (Dal Lago and Schöpp, 2013, Theorem 11). The new cases arising from the added types $A \to X$ and $\forall\alpha.\, X$ can be treated similarly.

## 3.4 Equational Theory

One motivation for studying INT is to use it as a tool for reasoning about low-level programs. To this end, we are interested not just in the reduction of closed terms, but also in equivalences of programs. We would like to identify an equational theory for INT that allows reasoning about low-level program equivalence in a useful way.

First we notice that for program equivalence, we should like to consider a relation coarser than extensional equality of low-level programs. For example, it is reasonable to expect the $\beta$-equality $(\lambda x{:}A{\cdot}X.\, t)\ s = t[s/x]$ to hold in general, not just for closed terms. However, if $t$ may have free variables, then $[\![(\lambda x{:}A{\cdot}X.\, t)\ s]\!]$ and $[\![t[s/x]]\!]$ may not be equal programs. Indeed, suppose that $x$ does not appear in $t$. Then, the code for $s$ in $[\![(\lambda x{:}A{\cdot}X.\, t)\ s]\!]$ would normally never be used and the $\beta$-equation would amount to reasonable dead code elimination. But if $s$ has free variables, then we can interact with the code for $s$ from the outside, by answering requests that have never been asked. In $[\![t[s/x]]\!]$, such requests will go unanswered, as the code for $s$ has been erased, while in $[\![(\lambda x{:}A{\cdot}X.\, t)\ s]\!]$ this code is still present and may react to certain requests. This means that the $\beta$-equation only expresses a reasonable form of low-level program equivalence, but not an equality of low-level programs.

In (Schöpp, 2011, 2014c,a) we work towards identifying an equational theory for INT that captures a useful notion of low-level program equivalence.

The equational theory for INT has $\beta\eta$-equations for terms in context. For $\multimap$-functions, these equations take the following form.

$$\multimap\beta\ \frac{\Sigma \mid \Gamma,\, x\colon A \cdot X \vdash t\colon Y \qquad \Sigma \mid \Delta \vdash s\colon X}{\Sigma \mid \Gamma, A \cdot \Delta \vdash (\lambda x{:}A{\cdot}X.\, t)\ s = t[s/x]\colon A \cdot X \multimap Y}$$

$$\multimap\eta\ \frac{\Sigma \mid \Gamma \vdash t\colon A \cdot X \multimap Y}{\Sigma \mid \Gamma \vdash (\lambda x{:}A{\cdot}X.\, t\ x) = t\colon A \cdot X \multimap Y}$$

All other equations are similarly to be understood in context. We state them here without contexts for brevity.

$$\mathsf{let}\ x{=}(\mathsf{return}\ v)\ \mathsf{in}\ t = t[v/x]$$
$$\mathsf{let}\ x{=}t\ \mathsf{in}\ \mathsf{return}\ x = t$$
$$(\mathsf{fn}\, x{:}A.\, s)(v) = s[v/x]$$

$$\begin{aligned}
\mathsf{fn}\, x{:}A.\, t(x) &= t \text{ if } x \text{ is not free in } t \\
(\Lambda\alpha.\, t)\ A &= t[A/\alpha] \\
\Lambda\alpha.\, t\ \alpha &= t \text{ if } \alpha \text{ is not free in } t \\
\mathsf{let}\ \langle x, y\rangle{=}\langle v, w\rangle\ \mathsf{in}\ t &= t[v/x, w/y] \\
\mathsf{let}\ \langle x, y\rangle{=}z\ \mathsf{in}\ t[\langle x, y\rangle/z] &= t \\
\mathsf{case}\ \mathsf{inl}(v)\ \mathsf{of}\ \mathsf{inl}(x){\Rightarrow}s; \mathsf{inr}(y){\Rightarrow}t &= s[v/x] \\
\mathsf{case}\ \mathsf{inr}(w)\ \mathsf{of}\ \mathsf{inl}(x){\Rightarrow}s; \mathsf{inr}(y){\Rightarrow}t &= t[w/y] \\
\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}(x){\Rightarrow}t[\mathsf{inl}(x)/z]; \mathsf{inr}(y){\Rightarrow}t[\mathsf{inr}(y)/z] &= t \\
\mathsf{case}\ \mathsf{fold}(v)\ \mathsf{of}\ \mathsf{fold}(x) \Rightarrow t &= t[v/x] \\
\mathsf{case}\ z\ \mathsf{of}\ \mathsf{fold}(x) \Rightarrow t[\mathsf{fold}(x)/z] &= t
\end{aligned}$$

The article (Schöpp, 2011) explains how to justify such equations for INTML. In essence, low-level programs are identified up to an extensional quotient. The idea is to define equality by induction on the type and to consider programs of function type as equal if they produce equal results when applied to equal arguments. An issue addressed in (Schöpp, 2011) is how to avoid such a quotient from being so strong that it excludes interesting direct definitions, such as a combinator for call/cc. In the definition of equality in (Schöpp, 2011), low-level programs are allowed to abort the computation, which weakens the quotient as desired, but still justifies the equational theory. Another possibility to achieve the same effect is to assume certain effects, such as output, in the low-level language, which is the approach taken in (Schöpp, 2014c).

Motivated by applications to call-by-value, see Chapter 7, in (Schöpp, 2014c,a) we further develop the equational theory to include also polymorphism and parametricity.

### 3.4.1 Relational Parametricity

For the application in Chapter 7, it is useful to restrict polymorphism in INT to be parametric and to appeal to parametricity in reasoning, as advocated by Wadler (1989). For example, suppose we have a term $t\colon \forall\alpha.\, \bot^\alpha \multimap \bot^\alpha$ (where $\bot^\alpha$ abbreviates $\alpha \to T0$), then $t\ A$ translates to a low-level program of type $0 + A \times \mathsf{unit} \to A \times \mathsf{unit} + 0$. By parametricity, we know that this program cannot inspect or modify the $A$-part of any input. If the program returns a value of type $A$, then this must have been the value that was supplied as input.

To formalise such parametricity reasoning, we follow the approach of using *relational parametricity* (Reynolds, 1983). In the rest of this section, we outline how relational parametricity can be defined for INT (Schöpp, 2014c,a).

A *value type relation* is given by a triple $(A, A', R)$ of two closed value types $A$ and $A'$ and a binary relation $R$ between the closed values of type $A$ and those of type $A'$. We write $R \subseteq A \times A'$ for the triple $(A, A', R)$. Depending on the application, we may allow for $R$ only *admissible* relations, see (Schöpp, 2014c).

A *type environment* $\rho$ is a mapping from type variables to value type relations. If $\sigma$ and $\sigma'$ are both mappings from type variables to closed value types, then we write $\rho \subseteq \sigma \times \sigma'$ if, for any $\alpha$, $\rho(\alpha)$ is a relation $R_\alpha \subseteq \sigma(\alpha) \times \sigma'(\alpha)$. We use the notation $A\sigma$ to denote the value type obtained from $A$ by replacing any free $\alpha$ with $\sigma(\alpha)$.

For each value type $A$ and type environment $\rho$, we define a relation $[\![A]\!]_\rho$ between the closed values of type $A\sigma$ and type $A\sigma'$ as usual:

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

The definition of relations can be extended to INT-types. For each $\rho \subseteq \sigma \times \sigma'$, we define a relation $[\![X]\!]_\rho$ between programs of type $\overline{X\sigma^-} \to \overline{X\sigma^+}$ and programs of type $\overline{X\sigma'^-} \to \overline{X\sigma'^+}$. In essence the definition follows the standard idea that related inputs must be mapped to related results.

For the base case, we define $p\ [\![TA]\!]_\rho\ p'$ to hold if and only if both programs have the same effects and produce related results, i.e. that the following conditions hold.

1. If $\mathit{entry}_p(*) \xrightarrow{o}_p t$ then $\mathit{entry}_{p'}(*) \xrightarrow{o}_{p'} t'$ for some $t'$.
2. If $\mathit{entry}_p(*) \xrightarrow{o}_p \mathit{exit}_p(v)$ then $\mathit{entry}_{p'}(*) \xrightarrow{o}_{p'} \mathit{exit}_{p'}(v')$ and $v\ [\![A]\!]_\rho\ v'$.
3. Conditions 1. and 2. hold with the roles of $p$ and $p'$ exchanged.

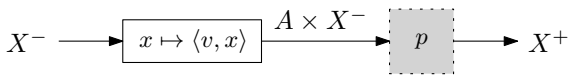
This definition is extended inductively to all interface types:

- $p\ [\![A \to X]\!]_\rho\ p'$ if and only if:

$$\forall v, v'.\ v\ [\![A]\!]_\rho\ v' \implies \mathsf{val}(p, v)\ [\![X]\!]_\rho\ \mathsf{val}(p', v') \ ,$$

  where $\mathsf{val}$ denotes value application:

$\mathsf{val}(p, v) =$

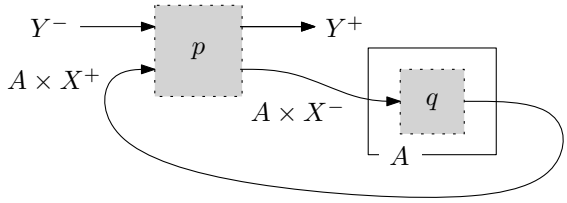



- $p\ [\![A \cdot X \multimap Y]\!]_\rho\ p'$ if and only if:

$$\forall p_1, p_1'.\ p_1\ [\![X]\!]_\rho\ p_1' \implies \mathsf{app}_{A\sigma}(p, p_1)\ [\![Y]\!]_\rho\ \mathsf{app}_{A\sigma'}(p', p_1') \ ,$$

  where $\mathsf{app}$ denotes linking application:

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

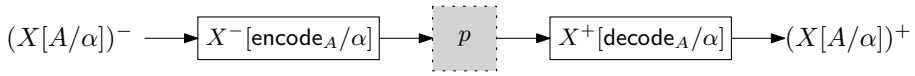



- $p \; [\![\forall\alpha.\, X]\!]_\rho \; p'$ if and only if:

$$\forall S \subseteq A \times A'.\; \mathsf{inst}_{(\forall\alpha.X)\sigma}(p, A) \; [\![X]\!]_{\rho[\alpha \mapsto S]} \; \mathsf{inst}_{(\forall\alpha.X)\sigma'}(p', A') \;\; ,$$

  where $\mathsf{inst}$ denotes type instantiation:

$\mathsf{inst}_{\forall\alpha.\, X}(p, A) =$




With these definitions, we define two terms $\Sigma \mid \Gamma \vdash s\colon X$ and $\Sigma \mid \Gamma \vdash t\colon X$ to be equal, if, after abstracting over all free variables, we obtain terms whose low-level programs are related by $[\![\forall\vec{\alpha}.\, \Sigma \to (\Gamma \multimap X)]\!]_\rho$, for any $\rho$. Here $\Gamma \multimap X$ denotes the type obtained by abstracting $X$ over all variables in $\Gamma$ using $\multimap$; and $\Sigma \to X$ denotes the type obtained by abstracting $X$ over all value types in $\Sigma$ using $\to$. Finally, $\vec{\alpha}$ is the list of free type variables in the type. More details appear in (Schöpp, 2014c,a).

Having defined relational parametricity, we can now explain the side condition $(*)$ on rule DIRECT. This side condition requires that the defined term $\mathsf{direct}_X(t)$ is equal to itself, i.e. that it respects the relation $[\![X]\!]_\rho$ for all $\rho$. Rule DIRECT is also the reason for restricting to admissible relations in the definition of parametricity in some situations. This restriction makes it easier to establish that terms defined by direct definition respect the relation, see (Schöpp, 2014c, Lemma 16).

## 3.5 Type Inference

We end the description of INT by briefly discussing type inference. The aim is to point out that the subexponential annotations of INT should not be seen as a restriction on typing, as is typical in linear type systems, but as adding low-level information. This is because subexponential annotations can be reconstructed for any term that is typeable without them (subject to a restriction on the types $X$ that are allowed in direct definition terms $\mathsf{direct}_X(t)$, see (Schöpp, 2014c, §6)).

A trivial way of finding subexponential annotation is to always use $\mathsf{G}$ for them, i.e. to restrict attention to typing sequents of the special form $\Sigma \mid x_1\colon \mathsf{G}\cdot X_1, \ldots, x_n\colon \mathsf{G}\cdot X_n \vdash t\colon Y$. There are rules, such as AX and $\multimap$E, whose conclusion does not have this form, even when all premises are of this form. However, with the encode and decode operations, it is not hard to see that we have $\mathsf{unit} \lhd \mathsf{G}$ and $(\mathsf{G} \times \mathsf{G}) \lhd \mathsf{G}$ and $(\mathsf{G} + \mathsf{G}) \lhd \mathsf{G}$. Therefore, we can use rule STRUCT to immediately bring the conclusion of any rule back into the special form.

This is one way to see that subexponentials are no restriction on typeability. In essence, this means that one can restrict one's attention to usual exponentials.

However, with many uses of $(\mathsf{G} \times \mathsf{G}) \lhd \mathsf{G}$ and $(\mathsf{G} + \mathsf{G}) \lhd \mathsf{G}$, the terms of INT translate to low-level programs that contain many encoding and decoding operations. This may not be desirable for reasons of efficiency. With subexponentials, encoding can often be avoided. The idea is to use a fresh type variable for each appearance of $\mathsf{G}$ and solve the resulting $\lhd$-constraints. Often one can choose the solution so that most constraints become trivial and that encoding and decoding operations are needed only in a few places.

In fact, the constraints that one obtains in this way are easy to solve. They are all of the form $A \lhd \alpha$, i.e. with a type variable as an upper bound. A set of such constrains can be solved simply by collecting all constraints with a common upper bound, say $A_1 \lhd \alpha, \ldots, A_n \lhd \alpha$, solving these by letting $\alpha := \mu\alpha.\, A_1 + \cdots + A_n$ and then repeating this until finished. This simple procedure is implemented in (Schöpp, 2012, 2014d) and is working well with practical examples (Dal Lago and Schöpp, 2010b; Schöpp, 2014c) in an experimental implementation.

# 4 Low-Level Programming with Higher Types

We have motivated INT as an approach to structuring low-level computation. In the following chapters we assess how useful the structure of INT is for low-level programming tasks. In this chapter we begin by considering the practical question of writing low-level programs in INT. This describes results that have appeared in (Schöpp, 2014c).

Although INT is a very simple calculus, with some syntactic sugar it can be used as a C-like low-level language. A few example programs can be found in an experimental implementation of a compiler (Schöpp, 2014d) from (a fragment of) INT to LLVM (Lattner and Adve, 2004), which in turn produces assembly code for a number of architectures. Here we outline how low-level programs can be written in INT and how the type system allows us to control low-level issues, such as being able to explicitly distinguish between calls and tail calls.

## 4.1 Recursion and Tail Recursion

The combination of higher-order functions and direct definition is essential for the expressiveness of INT. Let us outline how combinators for recursion and iteration can be defined using direct definition. A combinator for defining a function of type $\alpha \to T\beta$ by recursion can be given the following type.

$$\texttt{fix}\colon (\gamma\ \mathsf{list}) \cdot (\gamma \cdot (\alpha \to T\beta) \multimap (\alpha \to T\beta)) \multimap (\alpha \to T\beta)$$

Its argument is a step function of type $\gamma \cdot (\alpha \to T\beta) \multimap (\alpha \to T\beta)$, which it computes the fixed point of. The subexponential $\gamma$ in the type of the step function signifies that whenever the step function invokes its argument, then it passes along a value of type $\gamma$ that it expects to get back when the argument returns. To be able to return the right value, the `fix`-combinator keeps a list of $\gamma$-values (the call stack) in its subexponential, which explains the type of `fix`. The type $(\gamma\ \mathsf{list})$ is short-hand for the type $\mu\alpha.\, \mathsf{unit} + \gamma \times \alpha$ of lists.

We give the implementation of the fixed-point combinator below, but first discuss how it may be used. With some syntactic sugar, the Fibonacci function can, for example, be

written in INT simply as follows using the `fix`-combinator.

$$\texttt{fix}\ (\lambda f.\,\mathsf{copy}\ f\ \mathsf{as}\ f_1, f_2\ \mathsf{in}\ \mathsf{fn}\, x{:}\mathsf{nat}.\,\mathsf{if}\ x < 2\ \mathsf{then}\ \mathsf{return}\ 1\ \mathsf{else}\ f_1(x-1) + f_2(x-2)) \tag{4.1}$$

The step function in this definition can be given type

$$((\mathsf{nat} \times \mathsf{bool} \times \mathsf{nat}) + (\mathsf{nat} \times \mathsf{bool} \times \mathsf{nat})) \cdot (\mathsf{nat} \to T\mathsf{nat}) \multimap (\mathsf{nat} \to T\mathsf{nat})\ ,$$

where $\mathsf{bool}$ stands for $\mathsf{unit} + \mathsf{unit}$. The subexponential in this type arises because the argument function $f$ is used twice (hence the sum) and because at the points of both calls to the function, the local environment contains two additional numbers and one boolean (namely $x\colon \mathsf{nat}$, $(x < 2)\colon \mathsf{bool}$ and $(x-1)\colon \mathsf{nat}$ in the first call and $x\colon \mathsf{nat}$, $(x < 2)\colon \mathsf{bool}$ and $(x-2)\colon \mathsf{nat}$ in the second call).

Subexponentials in INT can be seen as an idealisation of the call-stack in machine language. In machine language, functions calls are usually implemented so that before the call all local data is put on the stack. The call instruction then pushes the return address on the machine stack and jumps to the function code. To return from the call, the callee pops the return address and jumps to it. At this point, the caller can recover its local data from the stack. Subexponentials implement calls similarly. When $f$ is invoked in the above example, the calling program constructs a value of type $((\mathsf{nat} \times \mathsf{bool} \times \mathsf{nat}) + (\mathsf{nat} \times \mathsf{bool} \times \mathsf{nat}))$, which contains local data and return address. If the value has the form $\mathsf{inl}(\dots)$, then computation should return to $f_1$, otherwise to $f_2$. This value is kept unchanged like on a stack until the call returns, when it is used to find the right point to return to and to restore the local environment. This can be seen as an abstract way of explaining the implementation of function calls using jumps and a stack (Schöpp, 2014c, §8).

It is easy to modify the fixed-point combinator into a combinator for tail recursion. In tail recursion one does not need a call stack. If the recursive calls of the step function are all in tail position, then we may return any value returned from the step function as the final result of the whole recursion. As a result, recursive calls can never return and there is no need to store the call stack. This can be captured by a combinator for tail recursion, whose type differs from the one for recursion in that the call stack $\gamma\ \mathsf{list}$ is replaced by $\mathsf{unit}$.

$$\texttt{tailrec}\colon \mathsf{unit} \cdot (\gamma \cdot (\alpha \to T\beta) \multimap (\alpha \to T\beta)) \multimap (\alpha \to T\beta)$$

This combinator is implemented so that every time the step function invokes its argument, the value of type $\gamma$ is thrown away. If the step function returns at any point, then the returned value is not returned to the caller, but as the return value of the whole tail recursion.

For example, the factorial function can then be written as follows.

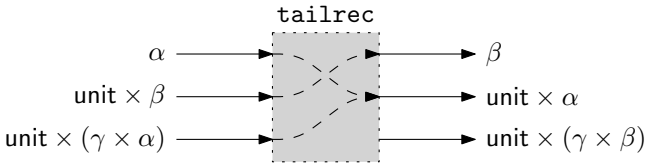
$$\texttt{tailrec}\ (\lambda f.\,\mathsf{fn}\, x{:}\mathsf{nat}.\,\mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{return}\ 1\ \mathsf{else}\ x * f(x-1))$$

One could replace `fix` with `tailrec` also in the above definition of the Fibonacci function as well, but since the recursive calls are not all in tail positions, this implementation would not produce correct results.

Let us now outline how direct definition can be used to define the combinators `fix` and `tailrec`. The term `tailrec` may be defined as shown below. Here we make the simplifying assumption that $(A \to TB)^-$ is $A$ rather than $A \times \mathsf{unit}$, as defined in general. This simplification can be justified by insertion of suitable coherent isomorphisms. Larger terms are shown in an informal programming language notation with some syntactic sugar for better readability.

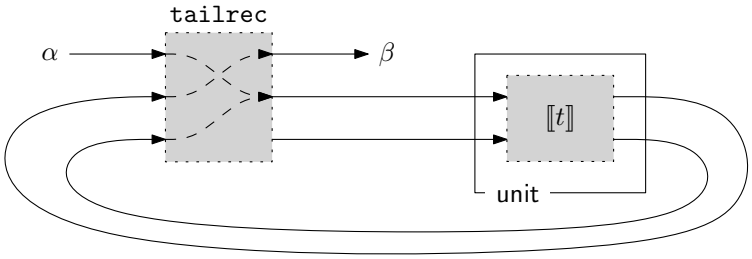
```
tailrec = direct(tailrecimp)
// tailrecrimp: unit × (γ × α + β) + α → T(unit × (γ × β + α) + β)
tailrecimp = fn x.
  case x of
  // Case: Start of computation with value a:α.
  // Invoke the step function with argument a.
  | inr(a) -> return inl(<*, inr(a)>)
  // Case: The step function returns value b: β.
  // Return b as the result of the whole tail recursion.
  | inl(<*, inr(b)>) -> return inr(b)
  // Case: The step function invokes its argument with value a:α
  //       and stack content g:γ.
  // Throw away g and restart the step function with argument a.
  | inl(<*, inl(g, a)>) -> return inl(<*, inr(a)>)
```

To give an idea of how this definition works in terms of the low-level language, notice that `tailrec` is defined as shown on the left below (up to distributiviy). A step function $t$ of type $(\gamma \cdot (\alpha \to T\beta) \multimap (\alpha \to T\beta))$ translates to a low-level program as shown on the right below.

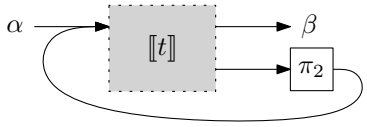




If one applies `tailrec` to the step function, then the low-level programs are connected thus:




A simple simplification procedure on low-level programs, e.g. as described in (Schöpp, 2014c, §5.1) and implemented in (Schöpp, 2014d), optimises this program to something like the following:




But this is how one would realise tail recursion in the low-level language by hand.

The combinator `fix` is implemented similarly, but instead of discarding the call stack of the step function, it is stored in a list. In this definition, we again simplify $(A \to TB)^-$ from $A \times \mathsf{unit}$ to $A$, and we use syntactic sugar for working with lists.

```
// fiximp: (γ list) × (γ × α + β) + α → T(unit × (γ × β + α) + β)
fix = direct(fiximp)
fiximp = fn x.
  case x of
  // Case: Start of recursion with argument a:α
  // Invoke the step function with argument a and empty call-stack.
  | inr(a) -> return inl(<Nil, inr(a)>)
  // Case: The step function returns b:β and the call-stack is empty.
  // Return b as the final value.
  | inl(<Nil, inr(b)>) -> return inr(b)
  // Case: The step function returns b:β and the call stack is not empty.
  // Pop the top element from the call stack and return it together with b
  // to the argument of the step function.
  | inl(<Cons(g, tl), inr(b)>) -> return inl(<tl, inl(g, b))>)
  // Case: The step function invokes its argument with value a:α and
  //       stack g:γ.
  // Push g on the call stack and invoke the step function with argument a.
  | inl(<l, inl(g, a)>) -> return inl(<Cons(g, l), inr(a)>)
```

That `tailrec` and `fix` are defined as higher-order combinators allows for their flexible use. For example, current C compilers optimise the naive recursive implementation of the Fibonacci function in (4.1) into a recursion with a single recursive call and a nested loop. This optimisation with a tail recursion nested inside a recursion can easily be expressed in INT with the combinators `tailrec` and `fix`:

```
fix (λfib.
  fn i:nat.
  tailrec (λtr.
      fn ⟨i, acc⟩:(nat × nat).
      let acc′=fib(i − 1) + acc in
      if i < 2 then acc else tr ⟨i − 2, acc′⟩
      ) ⟨i, 1⟩
  )
```

## 4.2 Other Combinators

By examining the definition of the tail recursion combinator, it is not hard to see that any low-level program can also be written in INT with little overhead. Suppose we have a low-level program with $n$ blocks with labels $f_1, \ldots, f_n$ and argument types $A_1, \ldots, A_n$. For each block we can define almost directly an INT term, such that the term for the $k$-th block has type

$$X_k \;:=\; (A_1 \to \bot) \multimap \ldots \multimap (A_n \to \bot) \multimap (A_k \to \bot) \;\; .$$

If, for example, if the $k$-th block is

$$f_k\,(x\colon A_k) \;=\; \mathsf{let}\ z{=}\mathsf{eq}(x,0)\ \mathsf{in}\ \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}(u){\Rightarrow} f_1(1); \mathsf{inr}(v){\Rightarrow} f_2(2) \;\; ,$$

then the corresponding INT term would be

$$\lambda f_1. \ldots . \lambda f_n.\, \mathsf{fn}\, x{:}A_k.\, \mathsf{let}\ z{=}\mathsf{eq}(x,0)\ \mathsf{in}\ \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}(u){\Rightarrow} f_1(1); \mathsf{inr}(v){\Rightarrow} f_2(2) \;\; .$$

To connect $k$ blocks written as INT-terms in this way, one can write a tail recursion combinator that takes arguments of types $X_1, \ldots, X_n$ and that has return type $A_i \to TA_j$, where $i$ is the index of the entry block and $j$ is the index of the exit block. This means that when moving to a language like INT one does not lose intensional expressiveness.

These are just a few examples of combinators that can be written using direct definition. Other examples are callcc (Dal Lago and Schöpp, 2010a, 2013) or a combinator newvar for block scoped state (Dal Lago and Schöpp, 2013; Schöpp, 2014c). The fixed point combinator defined above can in fact be given the following more general type, for any $X$:

$$\mathtt{fix}\colon (\alpha\ \mathsf{list}) \cdot (\alpha \cdot X \multimap X) \multimap X \;\; .$$

A first evaluation using an experimental implementation of INT (Schöpp, 2014d) suggests that the implementation of programs using directly defined combinators can be used to obtain efficient programs, see (Schöpp, 2014c).

Larger example programs, such as a simple raytracer and a program to compute the digits of Euler's number, can be found as part of an experimental implementation of a compiler for INT (Schöpp, 2014d).

## 4.3 Coroutines

To illustrate how INT allows control over low-level programming details, we show how to implement coroutines in it. Coroutines are a form of cooperative multi-tasking. We consider here the case where two processes are executed as coroutines. The idea is to run the first process until it yields control to the other process, then to stop the first process and run the other one until it yields again to the first process, and so on, until one of the processes terminates.

In INT, the two process may be given the following types $X_1 := \delta_1 \cdot (\alpha \to T\beta) \multimap T\gamma$ and $X_2 := \delta_2 \cdot (\beta \to T\alpha) \multimap \alpha \to T\gamma$. The first process of type $X_1$ gets a function $\mathit{yield}\colon \alpha \to T\beta$ as its first argument. The intention is that when $\mathit{yield}(v)$ is evaluated, the process sends the value $v$ to the other process and suspends its computation until the other process yields with some value $w$, which will be the return value of the call $\mathit{yield}(v)$. The second process gets an analogous yield function as argument. It moreover gets a value $\alpha$, because computation starts with the first process, so that the second process can only be started when the first process yields.

For example, the following term $\mathit{proc}_1$ is a program that counts up a variable $x$ from 0 in a loop and that stores in a variable $y$ the sum $1 + 2 + \cdots + x$. In each loop iteration it yields $y$ to the other process, which prints $y$ and yields back to the other process.

$$
\begin{aligned}
\mathit{proc}_1 = \ & \lambda \mathit{yield}.\ \texttt{tailrec}\ (\lambda l.\, \mathsf{fn}\, \langle x, y\rangle{:}\mathsf{nat}. \\
& \qquad \mathsf{let}\ x'{=}x+1\ \mathsf{in} \\
& \qquad \mathsf{let}\ y'{=}y+x\ \mathsf{in} \\
& \qquad \mathsf{let}\ z{=}\mathit{yield}(y)\ \mathsf{in} \\
& \qquad l(\langle x', y'\rangle) \\
& )(\langle 0, 0\rangle) \\
\mathit{proc}_2 = \ & \lambda \mathit{yield}.\ \mathsf{fn}\, y{:}\mathsf{nat}.\, \texttt{tailrec}\ (\lambda l.\, \mathsf{fn}\, y{:}\mathsf{nat}. \\
& \qquad \mathsf{print}(y); \\
& \qquad \mathsf{let}\ y'{=}\mathit{yield}(*)\ \mathsf{in} \\
& \qquad l(y') \\
& )(y)
\end{aligned}
$$

The aim is now to implement a combinator that allows us to execute such programs as coroutines.

The main difficulty in implementing coroutines is to make efficient use of space. It is possible to implement coroutines naively using recursion, for example, but the call-stack would grow with each yield, leading to a space leak. Hence, one needs some way of accessing the state of the processes in order to be able to suspend them and to restart them at the place where they left off. In INT the administrative details can be taken care of using subexponentials.

A combinator for coroutines may be defined by direct definition in INT.

$$\texttt{corout}\colon (\mathsf{unit} + \delta_2) \cdot X_1 \multimap \delta_1 \cdot X_2 \multimap T\gamma$$

The term ($\texttt{corout}$ $\mathit{proc}_1$ $\mathit{proc}_2$) is then a program that runs the two above processes as coroutines. It is not hard to see that there is no space leak coming from the implementation of coroutines, see also the next chapter. The implementation of `corout` can be given as follows.

```
corout = direct(coroutimp)
coroutimp = fn x.
  case x of
   // Start of computation: Start first process and use inl(): unit + δ2 in the
   // subexponential to indicate that the second process has not been started.
  | inr(*) -> return inl(<inl(*), inr(*)>)
   // The first process returns a value c: γ:
   // Return c as the result of the whole term.
  | inl(<_, inr(c)>) -> return inr(inr(c))
   // The first process invokes its yield function and the subexponential
   // contains inl(): unit + δ2, meaning that the second process has not
   // been started: Start the second process with value a: α and
   // save the own stack content d1: δ1 in the subexponential.
  | inl(<inl(*), inl(<d1, a>)>) -> return inr(inl(<d1, inr(a)>))
   // The first process invokes its yield function and the subexponential
   // contains inr(d2): unit + δ2, meaning that the second process has been run
   // before and its stack content when it yielded was d2:
   // Restart the second process with value a: α,
   // putting the own stack content d1: δ1 in the subexponential.
  | inl(<inr(d2), inl(<d1, a>)>) -> return inr(inl(<d1, inl(<d2, a>)>))
   // The second process invokes its yield function:
   // Resume computation of the first process with the subexponential d1: δ1;
   // put own stack content as inr(d2): unit + δ2 in the subexponential.
  | inr(inl(<d1, inl(<d2, b>)>)) -> return inl(<inr(d2), inl(<d1, b>)>)
   // The second process returns a value c: γ:
   // Return c as the result of the whole term.
  | inr(inl(<_, inr(c)>)) -> return inr(inr(c))
```

These examples illustrate that programs that need complicated access to the stack / local environment can be written using the simple higher-order structure identified by INT. In low-level languages, such as LLVM, tail calls and access to the stack are supported using additional built-in language primitives. In INT these can be constructed using the core language constructs.

Further information on practical issues, such as the performance of compiled code and the optimisation of low-level code can be found in (Schöpp, 2014c). Further example programs appear in the experimental implementations (Schöpp, 2014d, 2012), see also (Dal Lago and Schöpp, 2010b) for INTML examples.

# 5 Sublinear Space Bounds

The concept of computation-by-interaction appears naturally in the study of programming language aspects of sublinear space complexity classes. In this field of study the question is how one may design higher programming languages that guarantee that all their programs can be evaluated within the bounds of certain sublinear space complexity classes, such as LOGSPACE. One asks how typical programming constructs must be restricted in order to remain within classes such as LOGSPACE.

An interactive view of computation appears naturally when computing with sublinear space bounds. With such strong space bounds it is typical that certain values need to be re-computed many times, as there is not enough space to store them in memory. Intermediate values that are too large to fit in memory cannot be stored as a whole at all. Instead, only small parts of them are computed as they are needed. Such on-demand re-computation is naturally captured using concepts from interactive computation models.

Giving a systematic account of on-demand re-computation strategies that appear in sublinear space algorithms, in particular the one by Møller-Neergaard (Møller Neergaard, 2004), was my main initial motivation for studying the structure of computation by interaction (Schöpp, 2006, 2007). In further work with Ugo Dal Lago (Dal Lago and Schöpp, 2010a,b, 2013) it became clear that the structure of computation by interaction is relevant to sublinear space computation on a very fundamental level, even before one considers on-demand re-computation for saving space.

Sublinear space complexity classes are defined using Offline Turing Machines rather than standard Turing Machines. Offline Turing Machines have a read-only input tape, work tapes and a write-only output tape. The head on the output tape is further restricted in that it may only move in one direction. The move from Turing Machines to Offline Turing Machines is necessary for capturing sublinear space; if one could write on the input tape, for example, then one would already have linear space at disposal.

Turing Machines correspond to functions from strings to strings. If we consider $\Sigma^*$ as a low-level type, then their implementation in INT would simply appear as functions of type

$$\Sigma^* \to T\Sigma^* \quad .$$

While Offline Turing Machines are also often presented as machines that transform strings to strings, the computational intention is different. This becomes most clear when looking at the composition of two machines. To compose two Turing Machines, one makes the output tape of the first machine the input tape of the second machine. On any input,

one then just runs the machines one after the other. The first machine writes its output on the input tape of the second machine and then the second machine computes the result.

The composition of Offline Turing Machines is defined differently. This is essential for sublinear space computation, as there is not enough space to record the output of the first machine. To compute the composition of two Offline Turing Machines, one first starts the second machine and lets it run until it wants to read a character from its input tape. Only then does one start the first machine to compute only this character (all other output is discarded). When the character is computed, one resumes the computation of the second machine and continues thus until all output is produced. This implementation of the composition of Offline Turing Machines is quite different from that for Turing Machines and has an interactive flavour.

The interactive nature of Offline Turing Machines can be captured naturally by the INT type

$$(\mathsf{nat} \to T\Sigma) \multimap (\mathsf{nat} \to T\Sigma) \ .$$

Input and output word are represented as functions from numbers (positions in the string) to characters (the character at the given position). INT terms of the above type translate to low-level programs of type $\mathsf{nat} + \Sigma \to \Sigma + \mathsf{nat}$. Importantly, the composition of two such functions in INT implements just the interactive composition of Offline Turing Machines outlined above.

The different ways of modelling Turing Machines and Offline Turing Machines illustrate the value of being able to explicitly distinguish between code and data in INT. In Turing Machines the input is given by writing a value on the input tape, while for Offline Turing Machines it is given by connecting a program that computes characters on demand.

The main contribution of (Dal Lago and Schöpp, 2010a,b), building on earlier work in Schöpp (2006, 2007), is to show that by using the above representation of Offline Turing Machines in INT, one obtains a simple way of capturing the complexity class LOGSPACE by a higher-order programming language. In essence, the approach is simply to represent sublinear space algorithms as terms of the above type $(\mathsf{nat} \to T\Sigma) \multimap (\mathsf{nat} \to T\Sigma)$. To construct such terms, one can of course use the higher-order structure of INT, which immediately suggests an approach to designing a higher-order programming language for sublinear space programming.

For higher-order programming with logarithmic space, it then suffices to identify a fragment of INT where the space usage of all such terms remains within logarithmic bounds. For the whole of INT it is not easy to prove precise space bounds. We shall see in Chapter 6 that the whole of PCF can be embedded into INT, which means that establishing precise space bounds in general is likely to be difficult.

However, in this chapter we show that it is not hard to identify a fragment of INT that allows space analysis in a way that is suitable for establishing logarithmic space bounds. We outline this fragment of INT here and state the obtained complexity results. While the fragment is very simple, practical experiments (see Dal Lago and Schöpp, 2010b) suggest that it nevertheless allows convenient higher-order programming within logarithmic space.

## 5.1 Finitary Int

For programming with bounded space, we use a finitary fragment of INT, which is built over a restriction of the low-level language to finite value types. We remove recursive types and restrict the type nat to represent numbers of a certain fixed bit-width. With the restriction to a low-level language with finite types, general polymorphism cannot be implemented as outlined in Chapter 3.

The finitary fragment of INT, which we call $\text{INT}_{fin}$, therefore has the following types and all the terms that can be formed with these types. We assume that only the primitive operations for nat are present, i.e. we consider the fragment without effects for simplicity.

$$\begin{array}{rrcl} \text{Value Types} & A, B & ::= & \alpha \mid \mathsf{nat} \mid \mathsf{unit} \mid A \times B \mid 0 \mid A + B \\ \text{Computation Types} & X, Y & ::= & TA \mid A \to X \mid A \cdot X \multimap Y \end{array}$$

The translation from $\text{INT}_{fin}$ to the low-level language is exactly as before. However, the reduction of low-level programs is now modified so that values of type nat have fixed bit-width. We say that a low-level program is *evaluated using $k$-bit numbers* if the values of type nat are $k$-bit numbers and any operation that would produce a value greater that the maximum value $2^k - 1$ is being mapped to $2^k - 1$. For example, when evaluated using 2-bit numbers, the addition $\mathsf{add}(2, 2)$ would produce the result 3 $(= 2^2 - 1)$.

For the finitary fragment $\text{INT}_{fin}$, the following result is immediate.

**Theorem 1.** *Any term $\Sigma \mid \Gamma \vdash t \colon X$ in $\text{INT}_{fin}$ translates to a low-level program whose evaluation using $k$-bit numbers can be implemented to use space $O(k)$.*

A corresponding result was proved in (Dal Lago and Schöpp, 2010a, 2013). The theorem holds because the types appearing in the low-level program $[\![t]\!]$ can be seen as fixed 'polynomials' in nat, so that computation in $[\![t]\!]$ becomes computation with a constant number of nat-values. Any reasonable implementation of the low-level language will have the claimed space usage behaviour.

While this is a very simple result, it nevertheless shows that $\text{INT}_{fin}$ can be used for higher-order programming with very limited space. While $\text{INT}_{fin}$ is a fragment of INT, it still supports higher-order functions and, with the exception of full recursion, all the combinators mentioned in the last chapter, e.g. `tailrec`, `callcc`, `newvar` and `corout`, are still available. The INT-type system is expressive enough to allow us to distinguish full recursion, which cannot be allowed in a finitary system, from tail recursion, which is standard in programming with logarithmic space.

If $k$ is fixed, then we obtain a functional language for programming in constant space. This may be interesting for applications in hardware synthesis, as are being studied in the Geometry of Synthesis (Ghica, 2007). It is interesting to note that $\text{INT}_{fin}$ is more expressive than the fragment of Idealized Algol used in the Geometry of Synthesis (Ghica, 2007), see (Dal Lago and Schöpp, 2013, §4.3) for details. This makes it interesting to consider $\text{INT}_{fin}$ also in connection to hardware synthesis. To this end, Franz (2012) has shown in his B.Sc. thesis that INTML can be used for hardware synthesis. However, the

results in (Franz, 2012) are based on an unoptimised implementation and the experiments in (Franz, 2012) show that further optimisations are needed.

## 5.2 Logarithmic Space

Here we explain how the complexity class LOGSPACE of the functions computable in logarithmic space can be captured using $\text{INT}_{fin}$.

We consider the implementation of the functions from words to words with logarithmic space usage. Let us write $\Sigma$ for the finite alphabet over which the words are formed (one may assume w.l.o.g. that $\Sigma$ is $\{0,1\}$). The space available to the program may of course depend on the length of the input word. In $\text{INT}_{fin}$ we model this dependency by varying the bit-width of the numbers. For an input word of length $n$, we allow the program to use $\lceil \log(n) \rceil$-bit numbers. This corresponds to approaches in descriptive complexity theory (Immerman, 1999) and finite model theory (Ebbinghaus and Flum, 1995), where in logics with counting, all number variables have range $\{1, \ldots, n\}$.

With this choice of bit-width, words of length $n$ can be represented in $\text{INT}_{fin}$ by functions of type $\mathsf{nat} \to T\Sigma_\square$, where $\Sigma_\square$ is a type that can encode the set $\Sigma \cup \{\square\}$ of the alphabet and a blank symbol. The blank symbol $\square$ is needed to allow words that are shorter than the range of $\mathsf{nat}$. In order to allow words of polynomial length in the length of the input word, we define the representation of words as follows:

$$\mathit{Word}^i \;:=\; \underbrace{\mathsf{nat} \times \cdots \times \mathsf{nat}}_{i \text{ times}} \to T\Sigma_\square$$

The type $\mathit{Word}^i$ can represent words of length up to $n^i$. The tuple $\mathsf{nat} \times \cdots \times \mathsf{nat}$ in its domain can encode numbers up to $n^i$, encoded as tuples and ordered lexicographically.

We say that a closed $\text{INT}_{fin}$-term $t$ of type $\mathit{Word}^i$ *represents* a word $w \in \Sigma^*$ if and only if it translates to a low-level program $[\![t]\!] \colon (\mathsf{nat} \times \cdots \times \mathsf{nat}) \to \Sigma_\square$ that on input $k$ outputs the $k$-th character of $w$ if $k < |w|$ and $\square$ if $k \geq |w|$. This definition makes reference to low-level programs. An equivalent definition could also be given purely in terms of INT and the operational semantics from Section 2.2, see (Dal Lago and Schöpp, 2010a, 2013).

For any word $w \in \Sigma^*$, it is possible to write down a closed $\text{INT}_{fin}$-term $r(w)$ of type $\mathit{Word}^1$ that represents $w$, provided that numbers have at least $\lceil \log |w| \rceil$ bits. The term $r(w)$ simply consists of a huge case distinction. Its size is as least linear in the word $w$.

Next we define how partial functions $f \colon \Sigma^* \to \Sigma^*$ from words to words are represented. In essence, we use the type for Offline Turing Machines outlined above. With the restriction of the bit-width of numbers, we must be careful to also allow for polynomial size increase. Therefore, we consider terms of type $A \cdot \mathit{Word}^1 \multimap \mathit{Word}^j$ for arbitrary $A$ and $j$.

We say that a closed term $t$ of type $A \cdot \mathit{Word}^1 \multimap \mathit{Word}^j$ *represents* a partial function $f \colon \Sigma^* \to \Sigma^*$ if, for any $w \in \Sigma^*$, the term $t\, r(w)$ represents $f(w)$ when integers are $\lceil \log |w| \rceil$ bits wide.

With these definitions, we get the following characterisation of the functions computable in logarithmic space.

**Theorem 2.** *If a term of type $A \cdot Word^1 \multimap Word^j$, for any $A$ and $j$, represents a partial function $f\colon \Sigma^* \to \Sigma^*$, then $f$ is computable in logarithmic space. Moreover, any partial function computable in logarithmic space is represented by some such term.*

The soundness part of the theorem, that any term of type $A \cdot Word^1 \multimap Word^j$ represents a partial function computable in logarithmic space, follows immediately from the definition and the above Theorem 1, see (Dal Lago and Schöpp, 2010a, 2013).

Notice that an efficient compilation method is essential in this Theorem. If one implements $\text{INT}_{fin}$ naively using term reduction, then the output of a function $t\colon A \cdot Word^1 \multimap Word^j$ may be obtained by reducing $(t\ r(w))(0)$, $(t\ r(w))(1)$, etc., but this would give us a linear space algorithm at best.

For the completeness part of the theorem, it is straightforward to show that any LOGSPACE Offline Turing Machine can be implemented in $\text{INT}_{fin}$, see (Dal Lago and Schöpp, 2010a, 2013).

More interesting than extensional LOGSPACE-completeness is to consider the question whether natural LOGSPACE algorithms can be implemented in $\text{INT}_{fin}$. This question was considered for INTML in (Dal Lago and Schöpp, 2010b). First, one would like to compute not just on strings, but on structured data, such as graphs. Graphs can be represented much like words using a higher-order representation. The edge relation can be represented by a function of type $\mathsf{nat} \times \mathsf{nat} \to T\mathsf{bool}$, for example. With such an encoding, we have shown as a case study in (Dal Lago and Schöpp, 2010b) that a typical LOGSPACE graph algorithm – an acyclicity test in an undirected graph – can be represented naturally in INTML. $\text{INT}_{fin}$ can be considered a variant of INTML.

Even the complicated algorithm by Møller-Neergaard (Møller-Neergaard, 2004, §3.2) of implementing course-of value recursion by computational amnesia can be programmed. This algorithm provided the first motivation for studying an interactive computation model in the context of logarithmic space computation. The implementation of a program for this algorithm was developed with Ugo Dal Lago in INTML and can be found among the examples of the experimental implementation of INTML (Schöpp, 2012). The final program is quite similar to Møller-Neergaard's implementation in Standard ML (Møller-Neergaard, 2004, Figure 3.4). Where Møller-Neergaard's program uses tail recursion and exceptions, the INTML program uses combinators for tail recursion and call/cc. It demonstrates the expressiveness of INTML and $\text{INT}_{fin}$. While Møller-Neergaard had to prove LOGSPACE-soundness of his program by hand, here the type system guarantees soundness.

The characterisation of LOGSPACE by INTML and $\text{INT}_{fin}$ has its origin in the Stratified Bounded Affine Logic (SBAL) of (Schöpp, 2007). Stratified Bounded Affine Logic is a logic based on Bounded Linear Logic (Girard et al., 1992). It captures LOGSPACE much like INTML and $\text{INT}_{fin}$, i.e. logarithmic space bounds are obtained by interpretation in an interactive model. The most important difference is that SBAL contains a universal quantifier, which is restricted to bounded quantification in order to ensure logarithmic space bounds. This quantifier is different from the quantifier in INT, where it would correspond to a quantifier over interface types rather than value types. In SBAL the universal quantifier is used for impredicative representation of data in the tradition of System F (Girard et al.,

1989). Thus, one may think of INTML and $\text{INT}_{fin}$ as simplifications of SBAL without interface type polymorphism, where data types are instead assumed as primitives.

## 5.3 Type Inference

While the finitary fragment $\text{INT}_{fin}$ of INT still allows many programs to be expressed, the type inference procedure for INT that was outlined in Section 3.5 does not directly restrict to $\text{INT}_{fin}$. For INT we have seen that the $\lhd$-constraints that arise in type inference can always be solved using $\mathsf{G}$ or recursive types, neither of which is available in $\text{INT}_{fin}$.

Without recursive types, the typing constraints cannot always be solved, i.e. there exist simply-typed $\lambda$-terms that cannot be typed in $\text{INT}_{fin}$. A concrete example appears in (Schöpp, 2014b, Example 8.6) and illustrates that while $\text{INT}_{fin}$ can still type complicated higher-order terms, it cannot type all of them. The Kierstead term

$$t \;=\; \lambda g.\, \mathsf{copy}\ g\ \mathsf{as}\ g_1, g_2\ \mathsf{in}\ g_1\ (\lambda x.\, g_2\ (\lambda y.\, x))$$

is often used as an example to show the need for justification pointers in Hyland-Ong games (Hyland and Ong, 2000). This term can be typed in $\text{INT}_{fin}$ and be given type $(\mathsf{unit} + \alpha) \cdot (\alpha \cdot (\alpha \cdot X \multimap X) \multimap X) \multimap X$ for any $X$. A typing derivation in INTML can be found in (Dal Lago and Schöpp, 2010a, §2.3). There it is also shown that the other Kierstead term of the same order can also be typed and that it is possible to define a term that distinguishes them.

However, the application of $t$ to the following term $s$ cannot be typed in $\text{INT}_{fin}$ anymore.

$$s \;=\; \lambda f.\, \mathsf{copy}\ f\ \mathsf{as}\ f_1, f_2\ \mathsf{in}\ f_1\ (f_2\ (\lambda y.\, y))$$

The term $s$ has a type of the form $(\mathsf{unit} + \beta) \cdot (\beta \cdot Y \multimap Y) \multimap Y$, which to form the application $t\ s$ would would need to be unified with $\alpha \cdot (\alpha \cdot X \multimap X) \multimap X$. To this end we would need to unify $\mathsf{unit} + \beta$ with $\alpha$ and $\beta$ with $\alpha$, i.e. we would need to solve $(\mathsf{unit} + \beta) = \beta$, which requires recursive types. The application $s\ t$ cannot be typed in $\text{INT}_{fin}$. In INT, however, recursive types can be used to type it.

For the use of $\text{INT}_{fin}$ as a programming language for logarithmic space, one would therefore like to know whether or not a given program can be typed in it. One would like types to be found quickly by a type inference algorithm. The possibilities for type inference were studied with Ugo Dal Lago for INTML; the results are reported in (Dal Lago and Schöpp, 2010b).

The general definition of $\lhd$ by section-retraction-pairs is suitable for the definition of INT as a flexible type system. This definition makes type inference hard, however[1]. One can think of a strategy of solving the $\lhd$-constraints for INT as a strategy for organising the stack space of the compiled programs. It is perhaps not surprising that this is difficult in a finitary system. Therefore, it is reasonable to consider the type inference problem for approximations of the relation $\lhd$ that, while incomplete, are still useful for programming

[1] I conjecture that general type inference is undecidable.

in $\text{INT}_{fin}$. In (Dal Lago and Schöpp, 2010b) we have considered the question of automatic type inference for a number of possible choices of approximations of $\lhd$. We show in (Dal Lago and Schöpp, 2010b) that for many natural syntactic approximations of $\lhd$, the type inference problem is nevertheless still at least NP-hard.

For use in practice, we have identified a simple heuristic in (Dal Lago and Schöpp, 2010b) that captures a useful form of fast type inference. Examining the type inference procedure outlined in Section 3.5, one can see that for type inference it is sufficient to solve constraints of the form $(A_1 \lhd \alpha) \wedge \cdots \wedge (A_n \lhd \alpha)$. A simple heuristic to solve such constraints is to simply try to unify $\alpha$ with $A_1 + \cdots + A_n$ and reject if this does not succeed. This simple approach is implemented in (Schöpp, 2014d) and for all the examples mentioned in this section and in (Dal Lago and Schöpp, 2010a,b), types can be inferred with it.

# 6 Call-by-Name, Continuations and Defunctionalization

We have seen that the interactive structure of low-level programs identified by INT is useful for capturing programs with sublinear space usage. In this application, INT has been considered as a programming language in which one writes programs by hand. Programs in low-level languages are very often written automatically by compilers, however. In the following two chapters, we study how useful the structure identified by INT is for translating higher-order source languages to the low-level language.

In this chapter we begin by showing how to translate a PCF-like language with call-by-name evaluation strategy to INT. The translation is very simple and the equational theory of INT makes a correctness almost proof immediate. Nevertheless, it implements a translation from PCF to the low-level language.

Defining a sound and efficient translation from a higher-order language like PCF to the first-order low-level language directly is not trivial. One possibility is to use CPS-translation (Hofmann and Streicher, 1997) to enforce the call-by-name evaluation strategy and then to use defunctionalization (Reynolds, 1972) to implement the resulting higher-order program in the first-order low-level language. The result of (Schöpp, 2014b) is that the simple translation obtained using INT can in fact be understood as CPS-translation followed by a flow-based defunctionalization method. This shows that the interactive structure identified by INT allows us to give a simple account of a compilation method that uses sophisticated standard compilation techniques, namely CPS-translation (Hofmann and Streicher, 1997) and flow-based defunctionalization (Banerjee et al., 2001).

## 6.1 Source Language

Consider the following variant of PCF as a source language, which we want to compile into the low-level language.

$$\textsc{var}\ \frac{}{x\colon X \vdash x\colon X} \qquad \textsc{unit}\ \frac{}{\vdash *\colon 1}$$

$$\textsc{weak}\ \frac{\Gamma \vdash t\colon Y}{\Gamma, x\colon X \vdash t\colon Y} \qquad \textsc{exch}\ \frac{\Gamma, y\colon Y, x\colon X, \Delta \vdash t\colon Z}{\Gamma, x\colon X, y\colon Y, \Delta \vdash t\colon Z}$$

$$\textsc{contr}\,\frac{\Gamma, x_1\colon X, x_2\colon X \vdash t\colon Y}{\Gamma, x\colon X \vdash t[x/x_1, x/x_2]\colon Y}$$

$$\to\!\textsc{i}\,\frac{\Gamma,\, x\colon X \vdash t\colon Y}{\Gamma \vdash \lambda x{:}X.\, t\colon X \to Y} \qquad \to\!\textsc{e}\,\frac{\Gamma \vdash s\colon X \to Y \qquad \Delta \vdash t\colon X}{\Gamma, \Delta \vdash s\ t\colon Y}$$

$$\textsc{const}\,\frac{}{\vdash n\colon \mathbb{N}} \qquad \textsc{add}\,\frac{\Gamma \vdash s\colon \mathbb{N} \qquad \Delta \vdash t\colon \mathbb{N}}{\Gamma, \Delta \vdash s + t\colon \mathbb{N}}$$

$$\textsc{if}\,\frac{\Gamma \vdash s\colon \mathbb{N} \qquad \Delta_1 \vdash t\colon \mathbb{N} \qquad \Delta_2 \vdash u\colon \mathbb{N}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{if0}(s,t,u)\colon \mathbb{N}}$$

$$\textsc{fix}\,\frac{}{\vdash \mathsf{fix}_X\colon (X \to X) \to X}$$

Call-by-name evaluation may be formalised by the following reduction rules on closed well-typed terms.

$$\begin{aligned}
(\lambda x{:}X.\, s)\ t &\longrightarrow_{\mathsf{cbn}} s[t/x] \\
m + n &\longrightarrow_{\mathsf{cbn}} r \text{ if } r \text{ is the sum of } m \text{ and } n \\
\mathsf{if0}(0, s, t) &\longrightarrow_{\mathsf{cbn}} s \\
\mathsf{if0}(n, s, t) &\longrightarrow_{\mathsf{cbn}} t \text{ if } n > 0 \\
\mathsf{fix}_X\ t &\longrightarrow_{\mathsf{cbn}} t\ (\mathsf{fix}_X\ t) \\
u[s/x] &\longrightarrow_{\mathsf{cbn}} u[t/x] \text{ if } s \longrightarrow_{\mathsf{cbn}} t
\end{aligned}$$

## 6.2 Translation to Int

This call-by-name source language has a straightforward translation into INT. It is almost the identity. Essentially, we can consider each source term directly as an INT term; we just need to add suitable subexponential annotations.

This can be made precise using a relation $\rightsquigarrow$ that relates source types to INT types and source terms to INT terms. It is a relation rather than a translation function, as the choice of subexponential annotations is not unique. The relation $\rightsquigarrow$ is defined to be the least relation satisfying the following conditions.

- Types:

$$\begin{aligned}
1 &\rightsquigarrow T\mathsf{unit} \\
\mathbb{N} &\rightsquigarrow T\mathsf{nat} \\
(X \to Y) &\rightsquigarrow (A \cdot X' \multimap Y') \text{ if } X \rightsquigarrow X' \text{ and } Y \rightsquigarrow Y'
\end{aligned}$$

- Terms:

$$
\begin{aligned}
x &\rightsquigarrow x \\
s\ t &\rightsquigarrow s'\ t' \text{ if } s \rightsquigarrow s' \text{ and } t \rightsquigarrow t' \\
\lambda x{:}X.\, t &\rightsquigarrow \lambda x{:}A{\cdot}X'.\, t' \text{ if } X \rightsquigarrow X' \text{ and } t \rightsquigarrow t' \\
* &\rightsquigarrow \mathsf{return}\ * \\
n &\rightsquigarrow \mathsf{return}\ n \\
s + t &\rightsquigarrow \mathit{add}\ s'\ t' \text{ if } s \rightsquigarrow s' \text{ and } t \rightsquigarrow t' \\
\mathsf{if0}(s,t,u) &\rightsquigarrow \mathit{if}\ s'\ t'\ u' \text{ if } s \rightsquigarrow s',\ t \rightsquigarrow t' \text{ and } u \rightsquigarrow u' \\
\mathsf{fix}_X &\rightsquigarrow \mathtt{fix} \\
t[x/x_1, x/x_2] &\rightsquigarrow \mathsf{copy}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ t' \text{ if } t \rightsquigarrow t'
\end{aligned}
$$

  where

$$
\begin{aligned}
\mathit{add} = {} & \lambda x{:}\mathsf{unit}{\cdot}(T\mathsf{nat}).\, \lambda y{:}\mathsf{nat}{\cdot}(T\mathsf{nat}). \\
& \quad \mathsf{let}\ v{=}x\ \mathsf{in}\ \mathsf{let}\ w{=}y\ \mathsf{in}\ \mathsf{let}\ z{=}\mathsf{add}(\langle v, w\rangle)\ \mathsf{in}\ \mathsf{return}\ z \\
\mathit{if} = {} & \lambda x{:}\mathsf{unit}{\cdot}(T\mathsf{nat}).\, \lambda y{:}((\mathsf{unit}+\mathsf{unit})\times\mathsf{nat}){\cdot}(T\mathsf{nat}).\, \lambda z{:}((\mathsf{unit}+\mathsf{unit})\times\mathsf{nat}){\cdot}(T\mathsf{nat}). \\
& \quad \mathsf{let}\ v{=}x\ \mathsf{in}\ \mathsf{let}\ \mathsf{eq}(\langle x, 0\rangle){=}b\ \mathsf{in}\ \mathsf{case}\ b\ \mathsf{of}\ \mathsf{inl}(\_){\Rightarrow}t'; \mathsf{inr}(\_){\Rightarrow}u'
\end{aligned}
$$

  and where `fix` is the fixed-point combinator from Chapter 4.

The translation is extended to contexts in the evident way.

This translation has the property that whenever $\Gamma \vdash t\colon X$ is derivable in the PCF-like source language, then there exist $\Gamma'$, $t'$ and $X'$ with $\Gamma \rightsquigarrow \Gamma'$, $t \rightsquigarrow t'$ and $X \rightsquigarrow X'$, such that $- \mid \Gamma' \vdash t'\colon X'$ is derivable in INT. Context $\Gamma'$, term $t'$ and $X'$ may be computed from the derivation of $\Gamma \vdash t\colon X$ by first translating these to INT using a fresh type variable for each subexponential annotation and by inserting a copy-term for each use of the contraction rule. The missing subexponentials can then be constructed using type inference as outlined in Section 3.5.

It is interesting to look at the translation with INT's distinction between code and data in mind. Functions in the source language are translated directly to $\multimap$-functions in INT. This means that in the translation of a function application $s\ t$, the function $s$ and its argument $t$ are translated to separate code that is linked together to implement the application. Abstraction is realised by parameterisation of the low-level code for the body over a code module for the argument.

Correctness of the translation with respect to call-by-name reduction follows almost directly from the equational theory for INT. Only for fixed points do we have to show $\mathsf{fix}\ f = f\ (\mathsf{fix}\ f)$. But this can be shown directly on the low-level programs using a bisimulation-style argument.

We have defined the translation from source to INT using a relation to allow for different choices of subexponentials, e.g. for proving space bounds as outlined in the previous

chapter. If one is interested in a translation *function*, then one can choose $\mathsf{G}$ for all subexponentials. With this choice, one may consider the translation as a variant of an interpretation of the source language in Abramsky-Jagadeesan-Malacaria games (Abramsky et al., 2000). The low-level programs may be seen as implementations of the plays in a game semantic interpretation of the source language. Game semantic message passing becomes a jump to the recipient in the low-level language, much like in Levy's Jump-with-Argument calculus (Levy, 2004). As an implementation of a game semantics, the implementation of the source language using INT is related to other approaches to compilation using game semantics, e.g. (Ghica, 2007; Fredriksson and Ghica, 2013), and Geometry of Interaction, e.g. (Mackie, 1995; Fredriksson and Ghica, 2012).

It is natural to ask how a compilation of the source language based on an implementation of interaction dialogues relates to more traditional compilation techniques, e.g. (Appel, 1992). When I first used game semantic dialogues for the implementation for the space-efficient evaluation of functional programs (Schöpp, 2006, 2007), I thought that this implementation method was useful for controlling space usage, but I did not expect it to produce otherwise efficient low-level programs. The result of (Schöpp, 2014b) is that the compilation of the source language using INT is in fact very close to an efficient compilation method using CPS-translation and defunctionalization. We outline this result in the rest of this section.

## 6.3 Relation to Continuations and Defunctionalization

The main result of (Schöpp, 2014b) is that the implementation of call-by-name using INT can be understood as call-by-name CPS-translation followed by a defunctionalization procedure. Both CPS-translation and defunctionalization are well-known techniques for compiler construction.

We sketch the relation between the two translations by explaining how they translate the simple source term $\lambda x{:}\mathbb{N}.\, 1 + x$. The text in this section expands the Introduction of (Schöpp, 2014b) and summarises the results of this article, where technical details can be found.

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

**CPS-translation** The call-by-name CPS-translation of Hofmann and Streicher (1997) translates the source term $\lambda x{:}\mathbb{N}.\, 1 + x$ to the term $\lambda\langle x, k\rangle.\, (\lambda k.k\ 1)\ (\lambda u.\, x\ (\lambda n.\, k\ (u + n)))$ of type $\neg(\neg\neg\mathbb{N} \times \neg\mathbb{N})$, where we write $\neg A$ for $A \to \bot$. This term defines a function that takes as argument a pair $\langle x, k\rangle$ of a continuation $k\colon \neg\mathbb{N}$ that accepts the result and a variable $x\colon \neg\neg\mathbb{N}$ that supplies the function argument. To obtain the actual function argument it applies $x$ to the continuation $\lambda n.\, k\ (u + n)$ to ask for the actual argument to be thrown into this continuation.

**Defunctionalization** Defunctionalization (Reynolds, 1972) translates this higher-order term into a first-order program. The basic idea of defunctionalization is to give each function a name and to pass around not the function itself, but only its name and the values of its free variables. To this end, each $\lambda$-abstraction is named with a unique label: $\lambda^{l_1}\langle x, k\rangle.\,(\lambda^{l_2}k.k\ 1)\ (\lambda^{l_3}u.\,x\ (\lambda^{l_4}n.\,k\ (u+n)))$. The whole term defines the function named with label $l_1$. As it does not have free variables, it can be represented simply by the label $l_1$. The function with label $l_3$ has free variables $x$ and $k$ and is represented by the label together with the values of $x$ and $k$, which we write as $l_3(x, k)$.

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

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

This definition should be understood as the recursive definition of a function *apply* with two arguments. The definition is untyped, as in Reynold's original definition of defunctionalization (Reynolds, 1972).

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

$$\begin{aligned}\mathit{apply}(l, a) = \mathsf{case}\ l\ \mathsf{of}\ &\ldots\\ \mid l_5 &\Rightarrow \mathit{apply}(a, 42)\\ \mid l_6 &\Rightarrow \mathtt{print}(n)\end{aligned}$$

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

This outlines a naive defunctionalization method for translating a higher-order language into a first-order language with (tail) recursion. This method can be improved in various ways. The above *apply*-function performs a case distinction on the function name each time it is invoked. However, using control flow analysis it is often possible to determine the label, i.e. the function name, in the first argument of each appearance of *apply* statically. With this information, it is possible to avoid the case distinction on the function name and jump directly to the code for the respective case. This means that one may define one function $\mathit{apply}_l$ for each label $l$ and replace the jump to $\mathit{apply}(l(x), a)$, which involves a

case distinction, by a direct jump to $apply_l(l(x), a)$. In fact, one may go further and remove the label $l$ from the first argument, passing just $x$ instead of $l(x)$.

**Defunctionalization using Control-Flow Information** A defunctionalization procedure that takes into account control flow information in this way was introduced by Banerjee et al. (2001).

Control flow information can be added to the term by annotating applications. Instead of the standard application $s\ t$, we write $s@_l t$ for the application in which we know that the function $s$ evaluates to a function with label $l$, i.e. to a term of the form $\lambda^l x.u$. In general, one would need to annotate applications with more than one label to account for the possibility that $s$ may evaluate to functions with different labels, depending on data. For linear $\lambda$-terms, such as our example, a single label suffices. In the example term, applications can be annotated as follows:

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

For example, the application $k@_{l_3}1$ in this term expresses that only the abstraction with label $l_3$ can flow to the variable $k$ there.

Control flow annotations in the terms can be controlled using the type system. To this end, the function type $X \to Y$ is annotated with a label $X \xrightarrow{l} Y$ and the rules for abstraction and application are modified so that the type system correctly tracks control flow information:

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

Such type-based approaches to control-flow analysis are standard, see e.g. (Nielson et al., 2005). Writing $\neg_l X$ for $X \xrightarrow{l} \bot$, we get the following type for the above term:

$$\lambda^{l_1}\langle x, k\rangle.\, (\lambda^{l_2} k.k@_{l_3}1)@_{l_2}(\lambda^{l_3} u.\, x@_{l_5}(\lambda^{l_4} n.\, k@_{l_6}(u+n))) \colon \neg_{l_1}(\neg_{l_5}\neg_{l_4}\mathbb{N} \times \neg_{l_6}\mathbb{N})$$

Now, if we take into account the control flow information in the defunctionalization of our example, then we can avoid case distinction completely. We get:

$$\begin{aligned} apply_{l_1}(\langle l_1(x), k\rangle) &= apply_{l_2}(\langle l_2, l_3(x,k)\rangle) & apply_{l_2}(\langle l_2, k\rangle) &= apply_{l_3}(\langle k, 1\rangle) \\ apply_{l_3}(\langle l_3(x,k), u\rangle) &= apply_{l_5}(\langle x, l_4(k,u)\rangle) & apply_{l_4}(\langle l_4(k,u), n\rangle) &= apply_{l_6}(\langle l_6, u+n\rangle) \end{aligned} \tag{6.1}$$

These definitions can be understood as block definitions in the low-level language. They only represent a fragment of a low-level program, however. The blocks $apply_{l_5}$ and $apply_{l_6}$ must still be defined, one must identify an entry block with which to start the computation, and one must designate an exit label. If we apply the above example term to the argument $\langle \lambda^{l_5} k.\, k@_{l_4}42,\ \lambda^{l_6} n.\, \texttt{print}(n)\rangle$, then we get the missing definitions

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

and as the entry and exit labels we can choose $apply_{l_1}$ and $apply_{l_6}$ respectively.

A crucial point is that the control flow annotations on the type allow us to identify an interface of entry labels and exit labels for the definitions obtained by defunctionalization. This means that from defunctionalization we obtain not just a set of block definitions, but a complete low-level program, a definition of which also requires a choice of entry and exit labels. For example, the type

$$\neg_{l_1}(\neg_{l_5}\neg_{l_4}\mathbb{N} \times \neg_{l_6}\mathbb{N})$$

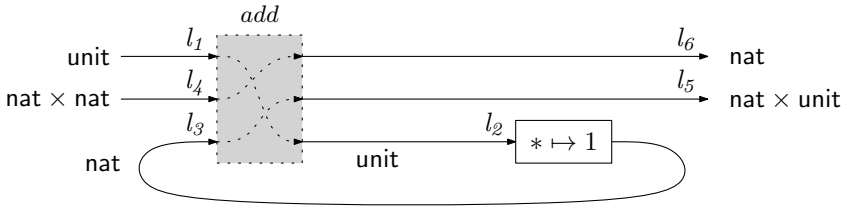
of the above program expresses that the program defines blocks $apply_{l_1}$ and $apply_{l_4}$ and may jump to external blocks $apply_{l_5}$ and $apply_{l_6}$. Since the type comes from the CPS-translation of the function type $\mathbb{N} \to \mathbb{N}$, we can assign meaning to the labels. The label $l_1$ represents the initial request to compute the function. The function may return its result by applying the function with label $l_6$. Labels $l_5$ and $l_4$ are for requesting the function argument and for providing it respectively.

**Interactive Interpretation** Let us now compare the low-level program obtained by CPS-translation and flow-based defunctionalization to the low-level program that we obtain by interpretation in INT. The source term $\lambda x{:}\mathbb{N}.\, 1 + x$ may be translated to the INT term $(\lambda x{:}\mathsf{nat}\cdot(T\mathsf{nat}).\, add\ (\mathsf{return}\ 1)\ x)$ of type $\mathsf{nat}\cdot(T\mathsf{nat}) \multimap T\mathsf{nat}$. Notice first that by definition we have:

$$\begin{aligned}(\mathsf{nat} \cdot T\mathsf{nat} \multimap T\mathsf{nat})^- &= (\mathsf{nat} \times \mathsf{nat}) + \mathsf{unit}\\ (\mathsf{nat} \cdot T\mathsf{nat} \multimap T\mathsf{nat})^+ &= (\mathsf{nat} \times \mathsf{unit}) + \mathsf{nat}\end{aligned}$$

This means that the low-level program obtained from the INT term may be considered as having two input labels (one for each summand) and two exit labels. The entry label with type $\mathsf{unit}$ plays the same role as $l_1$ above. The other entry label corresponds to $l_3$. The exit labels of type $\mathsf{nat}$ and $\mathsf{nat} \times \mathsf{unit}$ correspond to $l_6$ and $l_5$ respectively.

The low-level program arising from the term $(\lambda x{:}\mathsf{nat}\cdot(T\mathsf{nat}).\, add\ (\mathsf{return}\ 1)\ x)$ may be depicted as follows:




The sub-program *add* maps $*$ on the topmost input to $*$ on the bottom-most output, $\langle m, n\rangle$ on the middle input to $m + n$ on the topmost output and $n$ on the lowermost input to $\langle n, *\rangle$ on the middle output. The definition of *add* as an INT term in the definition of $\rightsquigarrow$ on page 49 implements these mappings.

With some simplification of the internal structure of *add*, the above interactive program can be written as follows.

$$\begin{aligned} apply_{l_1}(*) &= apply_{l_2}(*) & apply_{l_2}(*) &= apply_{l_3}(1)\\ apply_{l_3}(n) &= apply_{l_5}(\langle n, *\rangle) & apply_{l_4}(\langle m, n\rangle) &= apply_{l_6}(m+n)\end{aligned} \tag{6.2}$$

The block labels correspond to the indicated positions in the diagram.

Comparing this implementation of the interactive interpretation with the result of CPS-translation and defunctionalization (6.1) shows that both translation produce very similar results. In particular, function abstraction and application are implemented very similarly.

### 6.3.1 Relating the Translations

We now have two translations of the call-by-name source language to the low-level language. The translation by interpretation in INT was derived from the semantic structure of the Int-construction, which underlies the Geometry of Interaction and Game Semantics. The other translation is based on CPS-translation and defunctionalization, which are both standard techniques in the compilation of programming languages.

The contribution of (Schöpp, 2014b) is to make precise a relation between the two translation methods. The main results in this paper are summarised in the rest of this section. The relation of the two translation is considered for various fragments of the source language, as this allows one to study various aspects in isolation.

**Core Fragment** First we may concentrate just on how functions, i.e. $\lambda$-abstraction and application, are implemented by both translations. To this end, consider the following basic linear fragment of the source language having just abstraction and application.

$$\textsc{var}\ \frac{}{x\colon X \vdash x\colon X} \qquad \textsc{unit}\ \frac{}{\vdash *\colon 1}$$

$$\textsc{weak}\ \frac{\Gamma \vdash t\colon Y}{\Gamma, x\colon X \vdash t\colon Y} \qquad \textsc{exch}\ \frac{\Gamma, y\colon Y, x\colon X, \Delta \vdash t\colon Z}{\Gamma, x\colon X, y\colon Y, \Delta \vdash t\colon Z}$$

$$\to\textsc{i}\ \frac{\Gamma,\ x\colon X \vdash t\colon Y}{\Gamma \vdash \lambda x{:}X.\, t\colon X \to Y} \qquad \to\textsc{e}\ \frac{\Gamma \vdash s\colon X \to Y \qquad \Delta \vdash t\colon X}{\Gamma, \Delta \vdash s\ t\colon Y}$$

For this fragment, the two translations produce essentially the same low-level programs. Of course, in the translation of INT to the low-level language there are implementation details that may be treated in many ways. For example, the program *add* for addition above may be implemented in many ways. The article (Schöpp, 2014b) considers a reasonable choice of such details. The result (Schöpp, 2014b, Proposition 6.3) is that the two translations, CPS-translation followed by flow-based defunctionalization and interpretation in INT, can be set up so that they produce results that are identical only up to applications of the isomorphism $\mathsf{unit} \times A \cong A$. This correspondence includes open terms and terms of higher type.

**Linear Fragment** Next we extend the fragment of the source language to include a type of natural numbers.

$$\textsc{const}\ \frac{}{\vdash n\colon \mathbb{N}} \qquad \textsc{add}\ \frac{\Gamma \vdash s\colon \mathbb{N} \qquad \Delta \vdash t\colon \mathbb{N}}{\Gamma, \Delta \vdash s + t\colon \mathbb{N}}$$

$$\textsc{if}\ \frac{\Gamma \vdash s\colon \mathbb{N} \qquad \Delta_1 \vdash t\colon \mathbb{N} \qquad \Delta_2 \vdash u\colon \mathbb{N}}{\Gamma, \Delta_1, \Delta_2 \vdash \mathsf{if0}(s, t, u)\colon \mathbb{N}}$$

With this extension, the two translations do not produce identical results anymore. However, the two translations still produce programs of the same shape. One may see that the control-flow graphs of the low-level programs produced by both translations are isomorphic (Schöpp, 2014b, Proposition 7.2). This is illustrated by the example from the previous section. The control flow graphs of the programs in (6.1) and (6.2) are the same, even though the programs are not identical.

The isomorphism of control-flow graphs is only a weak correspondence result, of course. For closed terms of type $\mathbb{N}$, the result of (Schöpp, 2014b, Theorem 7.4) implies that the computation in both programs takes the same path in the control flow graph (and returns the same result, of course). Formally this means that if

$$\mathit{apply}_{l_1}(v_1)\mathit{apply}_{l_2}(v_2)\mathit{apply}_{l_3}(v_3)\ldots\mathit{apply}_{l_n}(v_n)$$

is a trace in the program obtained by CPS-translation and defunctionalization, then the program obtained using interpretation in INT has a trace of the form

$$\mathit{apply}_{l_1}(v'_1)\mathit{apply}_{l_2}(v'_2)\mathit{apply}_{l_3}(v'_3)\ldots\mathit{apply}_{l_n}(v'_n)$$

and vice versa. This means that the execution of both programs jumps to the same blocks in the same order. The values $v_i$ and $v'_i$ are not identical. One can however say that $v'_i$ simplifies $v_i$ in the sense that for each occurrence of a natural number in $v'_i$ we can find a corresponding occurrence of the same number in $v_i$. A precise statement of the relation of the values may be found in (Schöpp, 2014b, Theorem 7.4).

For instance, in the example from the previous section, the program (6.1) obtained using defunctionalization has a trace

$$\mathit{apply}_{l_1}(\langle *, *\rangle)\ \mathit{apply}_{l_2}(\langle *, \langle *, *\rangle\rangle)\ \mathit{apply}_{l_3}(\langle\langle *, *\rangle, 1\rangle)\ \mathit{apply}_{l_5}(\langle *, \langle *, 1\rangle\rangle)\ \ldots$$

The corresponding trace in the interactive program (6.2) is

$$\mathit{apply}_{l_1}(*)\ \mathit{apply}_{l_2}(*)\ \mathit{apply}_{l_3}(1)\ \mathit{apply}_{l_5}(\langle 1, *\rangle)\ \ldots$$

The values are different but contain the same $\mathsf{nat}$-values.

**Simple Types** If we further extend the source language with the contraction rule,

$$\textsc{contr}\ \frac{\Gamma, x_1 \colon X, x_2 \colon X \vdash t \colon Y}{\Gamma, x \colon X \vdash t[x/x_1, x/x_2] \colon Y}$$

then it becomes harder to establish a relation between the two translations. This is because defunctionalization now needs more than the very simple control-flow analysis outlined in the previous section. To account for this source language, applications need to be annotated with more than one label, i.e. instead of just $s@_l t$ we need annotations such as $s@_{\{l_1,\ldots,l_n\}} t$. This annotation states that a function with any of the labels from $\{l_1, \ldots, l_n\}$ could flow to $s$. In defunctionalization, one now needs to perform a case distinction at runtime over which label actually reaches the application:

$$\begin{aligned} apply_{\{l_1,\ldots,l_n\}}(\langle f, a\rangle) = \mathsf{case}\ f\ \mathsf{of}\ & l_1(\vec{x}) \Rightarrow apply_{l_1}(\langle f, a\rangle) \\ \mid\ & \ldots \\ \mid\ & l_n(\vec{x}) \Rightarrow apply_{l_n}(\langle f, a\rangle) \end{aligned}$$

Perhaps surprisingly, another consequence of adding contraction is that defunctionalization now needs recursive types in the low-level language, see (Schöpp, 2014b, Example 8.1) for an example.

In (Schöpp, 2014b) we are still able to show that the two translations produce programs with isomorphic control-flow graphs also with contraction. While this is a weak correspondence result, it nevertheless shows that the appearance of case distinction in defunctionalization, as shown above, is related to the case distinction that is performed in the implementation of the contraction rule of INT, as defined on page 20. Details can be found in (Schöpp, 2014b, §8).

## 6.4 Further Directions

To summarise, we have outlined that, for call-by-name, compilation by implementation of interaction dialogues turns out to be close to methods of compiler construction, such as defunctionalization. This is an interesting connection, as the two approaches have been investigated from quite different angles. Computation-by-interaction is rooted in the formal semantics of programming languages. In Game Semantics and the Geometry of Interaction the focus has been on answering theoretical questions, such as how to characterise observational equivalence of expressive programming language like PCF by construction of fully abstract interactive models, see e.g. (Hyland and Ong, 1995; Abramsky et al., 2000; Nickau, 1994; Murawski and Tzevelekos, 2013). Research on defunctionalization, on the other hand, has focused more on practical aspects of programming language implementation (Banerjee et al., 2001; Cejtin et al., 2000). In fact, although Reynolds introduced defunctionalization in 1972 (Reynolds, 1972), a proof of its correctness only appeared more than 25 years later (Nielsen, 2000). Nevertheless, it has been observed that there are interesting connections of defunctionalization to other concepts, see e.g. (Danvy, 2006, 2008).

One may hope that the identification of a connection between interactive models of computation and defunctionalization will help to transfer existing methods between the two areas. In one direction, this may help to develop a better theoretical understanding of the properties of defunctionalization. This may be useful in formal verification efforts, for example. In the other direction, a transfer of methods may lead to an improved understanding of the operational properties of game semantics and how results on game semantics apply to compiled programs. This may be useful for other applications of computation-by-interaction.

# 7 Call-by-Value

The observation that a straightforward interpretation of call-by-name PCF in INT corresponds closely to well-known compilation techniques such as defunctionalization makes it natural to ask if a similar correspondence holds for call-by-value source languages as well. To assess this question, we first need to study how call-by-value can be implemented using INT.

In this chapter, we develop an embedding of a call-by-value source into INT. This embedding allows us to showcase the use of the higher-order structure and the equational theory of INT. We show how to translate a call-by-value language into INT and how the structure of INT can be used to prove the correctness of the resulting compilation. The correctness proof makes essential use of the equational theory of INT, including relational parametricity. These results are the content of (Schöpp, 2014a).

By composing the translation from a call-by-value source language to INT with the translation from INT to the first-order low-level language, one obtains a direct translation from the call-by-value source language to the first-order low-level language. While we are not yet in the position to prove a formal result, we note that this translation bears a striking resemblance to existing defunctionalizing compilation techniques, such as (Cejtin et al., 2000). To the best of our knowledge, a correctness proof for the translation in (Cejtin et al., 2000) does not appear in the literature, and does not appear to be an easy exercise. The results in this chapter suggest that factoring such translations through languages like INT may offer an approach to obtaining such proofs.

In the following we explain the translation of call-by-value by using the following simply-typed $\lambda$-calculus as source language. Compared to PCF, case distinction and recursion are missing in this source language. How to handle case distinction is explained in (Schöpp, 2014a). We have not considered recursion formally yet, but expect it to be possible to include it.

$$\textsc{var}\ \frac{}{x\colon X \vdash x\colon X}$$

$$\textsc{weak}\ \frac{\Gamma \vdash t\colon Y}{\Gamma, x\colon X \vdash t\colon Y} \qquad \textsc{exch}\ \frac{\Gamma, y\colon Y, x\colon X, \Delta \vdash t\colon Z}{\Gamma, x\colon X, y\colon Y, \Delta \vdash t\colon Z}$$

$$\textsc{contr}\ \frac{\Gamma, x_1\colon X, x_2\colon X \vdash t\colon Y}{\Gamma, x\colon X \vdash t[x/x_1, x/x_2]\colon Y}$$

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

$$\text{CONST}\,\frac{}{\vdash n\colon \mathbb{N}} \qquad \text{ADD}\,\frac{\Gamma \vdash s\colon \mathbb{N} \qquad \Delta \vdash t\colon \mathbb{N}}{\Gamma, \Delta \vdash s + t\colon \mathbb{N}}$$

The values of this language are defined by the following grammar, in which $n$ ranges over natural number constants.

$$v, w \ ::= \ x \ \mid \ n \ \mid \ \lambda x{:}X.\, t$$

The reduction relation is defined on closed well-typed terms by:

$$\begin{aligned}
(\lambda x{:}X.\, s)\ v &\longrightarrow_{\mathsf{cbv}} s[v/x] \\
s\ t &\longrightarrow_{\mathsf{cbv}} s'\ t \text{ if } s \longrightarrow_{\mathsf{cbv}} s' \\
(\lambda x{:}X.\, s)\ t &\longrightarrow_{\mathsf{cbv}} (\lambda x{:}X.\, s)\ t' \text{ if } t \longrightarrow_{\mathsf{cbv}} t' \\
m + n &\longrightarrow_{\mathsf{cbv}} r \text{ if } r \text{ is the sum of } m \text{ and } n \\
s + t &\longrightarrow_{\mathsf{cbv}} s' + t \text{ if } s \longrightarrow_{\mathsf{cbv}} s' \\
n + t &\longrightarrow_{\mathsf{cbv}} n + t' \text{ if } t \longrightarrow_{\mathsf{cbv}} t'
\end{aligned}$$

There are many ways to implement this call-by-value source calculus in INT. One could implement an abstract machine, for example. Since a main interest here is in compilation, we consider an implementation with a high amount of code separation, much like in the translation of call-by-name in Chapter 6. The translation of an application $s\ t$ will be such that $s$ and $t$ are compiled to separate low-level programs that only need to be linked together in order to obtain a program for the application. Such a separation is desirable for separate compilation, for example.

## 7.1 CPS-translation

Since we have already explained how to translate the call-by-name source language to INT, an immediate idea of accounting for call-by-value would be to translate call-by-value to call-by-name. A call-by-value CPS-translation (Plotkin, 1975) may be used for this purpose. However, without further refinements, the translation from source language to low-level language obtained in this way would be inefficient with regard to space usage.

Let us outline the CPS-translation, why it is inefficient, and how the efficiency problems can be solved. A simple way of presenting a call-by-value CPS-translation is by using a continuation monad. We first recall the translation with the simply-typed $\lambda$-calculus as target language and then consider it with INT as target.

With the simply-typed $\lambda$-calculus as target language, the continuation monad may be defined by

$$\mathit{Cont}(X) = (X \to \bot) \to \bot \ ,$$

where $\bot$ is an arbitrary fixed type. It is well-known that $\mathit{Cont}(-)$ is a strong monad and we may define combinators

$$\begin{aligned} \eta &\colon X \to \mathit{Cont}(X) \\ \mathit{bind2} &\colon \mathit{Cont}(X) \to \mathit{Cont}(Y) \to (X \to Y \to \mathit{Cont}(Z)) \to \mathit{Cont}(Z) \end{aligned}$$

by $\eta(t) = \lambda k.\, k\ t$ and $\mathit{bind2}(s,t,u) = \lambda k.\, s\ (\lambda x.\, t\ (\lambda y.\, u\ x\ y\ k))$. The combinator *bind2* integrates both monad multiplication and strength. We use it instead of the latter two, as we have not introduced pair types in the simply-typed $\lambda$-calculus or INT (for the sake of simplicity).

With these combinators, the CPS-translation of the call-by-value $\lambda$-calculus can be formulated as follows. Types are translated by

$$\begin{aligned} [\![\mathbb{N}]\!] &= \mathbb{N} \\ [\![X \to Y]\!] &= [\![X]\!] \to \mathit{Cont}([\![Y]\!]) \end{aligned}$$

and this definition is extended to contexts: $[\![x_1 \colon X_1, \ldots, x_n \colon X_n]\!] = x_1 \colon [\![X_1]\!], \ldots, x_n \colon [\![X_n]\!]$.

A typing judgement $\Gamma \vdash t \colon X$ is translated to a judgement $[\![\Gamma]\!] \vdash \mathsf{cps}(t) \colon \mathit{Cont}([\![X]\!])$ by induction on the derivation as follows:

- Rule VAR:

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

$$\Downarrow$$

$$\frac{}{x \colon [\![X]\!] \vdash \eta(x) \colon \mathit{Cont}([\![X]\!])}$$

- Rule $\to$E:

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

$$\Downarrow$$

$$\frac{[\![\Gamma]\!] \vdash \mathsf{cps}(s) \colon \mathit{Cont}([\![X \to Y]\!]) \qquad [\![\Delta]\!] \vdash \mathsf{cps}(t) \colon \mathit{Cont}([\![X]\!])}{[\![\Gamma]\!], [\![\Delta]\!] \vdash \mathit{bind2}(\mathsf{cps}(s), \mathsf{cps}(t), \lambda f.\, \lambda x.\, f\ x) \colon \mathit{Cont}([\![Y]\!])}$$

- Rule $\to$I:

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

$$\Downarrow$$

$$\frac{[\![\Gamma]\!],\, x \colon [\![X]\!] \vdash \mathsf{cps}(t) \colon \mathit{Cont}([\![Y]\!])}{[\![\Gamma]\!] \vdash \eta(\lambda x.\, \mathsf{cps}(t)) \colon \mathit{Cont}([\![X \to Y]\!])}$$

- Rule CONST:

$$\frac{}{\vdash n\colon \mathbb{N}}$$

$$\Downarrow$$

$$\frac{}{\vdash \eta(n)\colon \mathit{Cont}([\![\mathbb{N}]\!])}$$

- Rule ADD:

$$\frac{\Gamma \vdash s\colon \mathbb{N} \qquad \Delta \vdash t\colon \mathbb{N}}{\Gamma, \Delta \vdash s + t\colon \mathbb{N}}$$

$$\Downarrow$$

$$\frac{[\![\Gamma]\!] \vdash \mathsf{cps}(s)\colon \mathit{Cont}([\![\mathbb{N}]\!]) \qquad [\![\Delta]\!] \vdash \mathsf{cps}(t)\colon \mathit{Cont}([\![\mathbb{N}]\!])}{[\![\Gamma]\!], [\![\Delta]\!] \vdash \mathit{bind2}(\mathsf{cps}(s), \mathsf{cps}(t), \lambda m.\, \lambda n.\, \eta(m+n))\colon \mathit{Cont}([\![\mathbb{N}]\!])}$$

- Each structural rule is translated to a corresponding instance of the same structural rule. We show the case for weakening and omit the analogous cases for exchange and contraction.

$$\frac{\Gamma \vdash t\colon Y}{\Gamma, x\colon X \vdash t\colon Y}$$

$$\Downarrow$$

$$\frac{[\![\Gamma]\!] \vdash \mathsf{cps}(t)\colon \mathit{Cont}([\![Y]\!])}{[\![\Gamma]\!],\ x\colon [\![X]\!] \vdash \mathsf{cps}(t)\colon \mathit{Cont}([\![Y]\!])}$$

This outlines a standard monadic formulation of a call-by-value CPS-translation.

Let us now consider what happens when one uses not the $\lambda$-calculus but INT as a target for this translation. We have shown in Section 3.5 that INT can type all simply-typed terms. If we choose $T\mathsf{nat}$ for the base type $\mathbb{N}$, then we may simply interpret $\mathsf{cps}(t)$ as a term of INT. We obtain the following translation of source terms to terms that can be typed in INT.

$$\begin{aligned}
\mathsf{cps}(x) &= \eta(x)\\
\mathsf{cps}(s\ t) &= \mathit{bind2}(\mathsf{cps}(s), \mathsf{cps}(t), \lambda f.\, \lambda x.\, f\ x)\\
\mathsf{cps}(\lambda x{:}X.\, t) &= \eta(\lambda x.\, \mathsf{cps}(t))\\
\mathsf{cps}(n) &= \eta(\mathsf{return}\ n)\\
\mathsf{cps}(s+t) &= \mathit{bind2}(\mathsf{cps}(s), \mathsf{cps}(t), \lambda m.\, \lambda n.\, \eta(\mathsf{let}\ x{=}m\ \mathsf{in}\ \mathsf{let}\ y{=}n\ \mathsf{in}\ \mathsf{return}\ x+y))
\end{aligned}$$

It is reasonable to ask if this translation induces an efficient compilation of the call-by-value source language to the low-level language.

Let us outline how the resulting translation would implement the source language by low-level programs. To this end, we first consider the types of the resulting INT term. Up to subexponential annotations and the replacement of $\mathbb{N}$ by $T\mathsf{nat}$, the INT types are the same as the simple types spelled out above. In essence, these types only need to be

annotated with suitable subexponentials. Thus, in the translation to INT each appearance of continuation monad $(X \to \bot) \to \bot$ is replaced by a type of the form

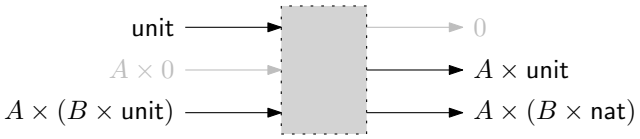
$$A \cdot (B \cdot X \multimap \bot) \multimap \bot \ .$$

In INT we choose $\bot := T0$, which is the interface of programs that accept a single request $*$ and that cannot answer.

A closed value of type $\mathbb{N}$ is translated to a closed INT term of type

$$A \cdot (B \cdot T\mathsf{nat} \multimap \bot) \multimap \bot \ .$$

It represents a low-level program with the following interface (up to distributivity):




The topmost input means 'Please compute the value.' The output of type $A \times \mathsf{unit}$ means 'Ok, the value is computed and ready.' We do not know anything of the value of type $A$, other than that we need it to ask for the computed number. The input of type $A \times (B \times \mathsf{unit})$ allows us to request the computed value. If we pass $\langle a, \langle b, * \rangle\rangle$ to this input, where $a$ is the value returned before, then we get $\langle a, \langle b, n \rangle\rangle$, where $n$ is the number that is the value of the whole term.

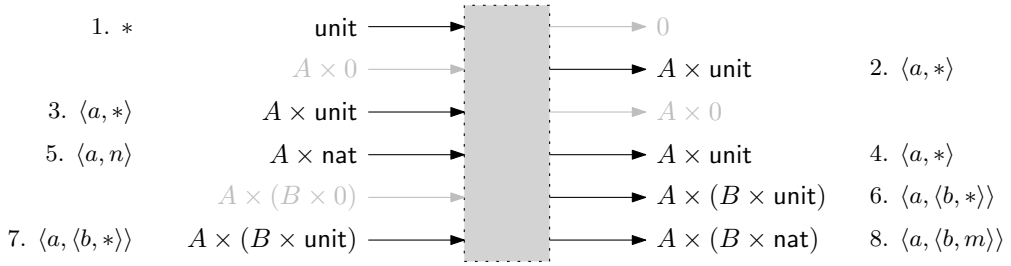
Thus, in the type $A \cdot (B \cdot T\mathsf{nat} \multimap \bot) \multimap \bot$ the rightmost $\bot$ is where we pass a request to compute the value, the other $\bot$ is where we receive the acknowledgement (formally as a request), and the $T\mathsf{nat}$ allows us to request the actual value.

It is interesting that the value of the source term is encoded abstractly in the subexponential. In order to map $\langle a, \langle b, * \rangle\rangle$ to $\langle a, \langle b, n \rangle\rangle$, it must be possible to compute $n$ from $a$. The value $a$ is thus an abstract representation of the value of the term. With the ports corresponding to $T\mathsf{nat}$, the program provides a way of accessing this abstract value. This is like in object oriented programming, where object state is private and can only be accessed using public methods.

With a second example, we illustrate how functions are implemented by low-level programs. A closed term of type $\mathbb{N} \to \mathbb{N}$ may be translated to a closed INT term of type

$$A \cdot ((B \cdot (T\mathsf{nat} \multimap \bot) \multimap T\mathsf{nat} \multimap \bot) \multimap \bot) \multimap \bot \ .$$

For illustration, we consider here the special case where all subexponentials except $A$ and $B$ are $\mathsf{unit}$. In general, the other function spaces could have non-trivial subexponentials as well. The type of the low-level program and a typical sequence of messages is depicted in the following diagram (again up to distributivity).




The messages are placed next to the input port over which they are passed to the program, respectively the output port over which they are returned. The number in front of the message value is a sequence number. The computation can be read as follows: Message number 1 represents a request to compute the function value. The reply is message number 2, which signals that the function value is ready. Then we can ask to apply the thus computed function by passing message 3 to the program (the value $a$ therein must be as in message 2). The program will typically reply with message 4 to ask for the function argument. The argument value $n$ is supplied using message 5. Then, the program applies the function to the argument $n$ and when the result is computed it signals that the result is ready by replying with message 6. While the result of the function application can be computed from the value $b$ in message 6, we do not know how the program has encoded the result in this value. Using message 7 we can request the result of the function application, which will finally be returned in message 8.

The examples illustrate that, in principle, the CPS-translation into INT gives rise to a reasonable implementation of call-by-value. It can still be simplified, e.g. so that values of base type are returned right away rather than having to be requested. Such simplifications are not difficult to make. However, the sequence in which the function value is computed and how the application is realised, is reasonable for the implementation of call-by-value. It is interesting that subexponentials automatically give rise to a representation similar to closures. In essence, the subexponential for a function will be the tuple of its free value variables.

However, upon further analysis, it becomes apparent that the translation does not treat memory in an efficient way. All computed values are represented using subexponentials and these are never deallocated. Informally, values in subexponentials are deallocated only when a function call returns. But continuations never return, so no value is ever deallocated. Consider for example the source term

$$\mathsf{let}\ x{=}5\ \mathsf{in}\ \mathsf{let}\ y{=}x+1\ \mathsf{in}\ \mathsf{let}\ z{=}y+4\ \mathsf{in}\ z+3\ ,$$

where ($\mathsf{let}\ x{=}s\ \mathsf{in}\ t$) abbreviates $(\lambda x.t)\ s$, i.e. the term is sugared notation for the term $(\lambda x.\,(\lambda y.\,(\lambda z.\,z+3)\ (y+4))\ (x+1))\ 5$). If we translate this term as described above, then we obtain an INT program of type

$$(\mathsf{nat} \times \mathsf{nat} \times \mathsf{nat}) \cdot (B \cdot T\mathsf{nat} \multimap \bot) \multimap \bot\ .$$

If we ask the resulting low-level program to compute the value by sending the request $*$ to the rightmost occurrence of $\bot$, the program will reply with $\langle\langle 6, 10, 13\rangle, *\rangle$ in the other

occurrence of $\bot$ to signal that the result is ready. The subexponential contains all intermediate results, including the ones for $x$ and $y$, which are not needed anymore. Indeed, in a long series of let-terms, all intermediate values will be kept. Such space usage behaviour is undesirable in the compilation of call-by-value. Implementations of call-by-value should be *safe for space* (Appel, 1992; Shao and Appel, 2000), which is a requirement that values should be discarded as soon as they go out of scope.

In order to address this issue of space inefficiency, we need to consider the deallocation of values. The stack-like memory management afforded to us by subexponentials is too simple. In order to be able to deallocate values when they are not needed anymore, more direct control over the stored values is desirable.

## 7.2 Explicit Manipulation of State

To implement call-by-value efficiently, we would like to be able to control not only which values are computed when, but also how long they are stored in the resulting low-level program. One way of achieving this is to make the state of all stored values explicit in the program, so that it is possible to deallocate values explicitly. In the above CPS-translation to INT, values are not mentioned explicitly. For example, the encoding of functions as values of subexponentials remains implicit. In the following section we modify the CPS-translation so that values are fully explicit.

### 7.2.1 The Linear Case

Let us describe in detail first the linear case, where the source language does not have contraction. In this case we do not need the subexponentials of INT, which simplifies types and makes it easier to spell out low-level programs explicitly. We will show in the following section how to account for contraction.

We use the following refinement of the continuation monad, in which we write $\bot^A$ for $A \to \bot$ to make the definition more readable.

$$Cont_{A,B}(X) \;=\; \forall\alpha.\,(X \multimap \bot^{B\times\alpha}) \multimap \bot^{A\times\alpha}$$

This definition can be seen as adding information to the continuation monad $Cont(X) = (X \multimap \bot) \multimap \bot$. We have $Cont(X)^- \cong X^- + \mathsf{unit}$ and $Cont(X)^+ \cong X^+ + \mathsf{unit}$. As outlined above, an input of type $\mathsf{unit}$ can be understood as the request to compute the value, while an output of type $\mathsf{unit}$ signals completion of the computation. Above we have seen that subexponentials can be used to add information to the reply implicitly. Here we add information explicitly. We have $Cont_{A,B}(X)^- \cong X^- + A\times\overline{\alpha}$ and $Cont_{A,B}(X)^+ \cong X^+ + B\times\overline{\alpha}$ (recall the notation $\overline{(-)}$ for replacing all free type variables by $\mathsf{G}$). The initial request takes a value of type $A$ (which will represent the values of the free variables of the term to be computed) and returns a value of type $B$ (which will be the actual computed value rather than just the signal that it is ready).

We return to the low-level interpretation of $Cont_{A,B}(X)$ at the end of this section. For now one may think of the terms of type $Cont_{A,B}(X)$ as programs that take a value of type $A$, that in response return a value of type $B$ and that then give us access to a program of interface type $X$. One may understand *Cont* as a parameterised monad (Atkey, 2009). It was identified from the definitions of (Schöpp, 2014a) in discussions with Shin-ya Katsumata.

We use the following combinators for working with $Cont_{A,B}(X)$.

$$\begin{aligned}
\eta &\colon X \multimap Cont_{A,A}(X) \\
\mathit{bind2} &\colon Cont_{A,B}(X) \multimap Cont_{C,D}(Y) \multimap (X \multimap Y \multimap Cont_{B\times D,E}(Z)) \multimap Cont_{A\times C,E}(Z) \\
(-) \lessdot (=) &\colon (A' \to TA) \multimap Cont_{A,B}(X) \multimap Cont_{A',B}(X) \\
(-) \gtrdot (=) &\colon Cont_{A,B}(X) \multimap (B \to TB') \multimap Cont_{A,B'}(X)
\end{aligned}$$

Informally, $\eta$ and *bind2* have the same meaning as before. The effect of $\mathit{bind2}(s,t,u)$ is to first execute $s$ and $t$ to obtain $x$ and $y$, and then to execute $(u\ x\ y)$. Only now the state is made explicit. We start with a value of type $A \times C$. Using the $A$-part, we can execute $s$ to obtain a value of type $B$. With the $C$-part, we can execute $t$ and obtain a value of type $D$. We thus get a value of type $B \times D$, using which we can start $(u\ x\ y)$ to get some value of type $E$, which is our result value.

For $f\colon A' \to TA$ and $t\colon Cont_{A,B}(X)$, we have $f \lessdot t\colon Cont_{A',B}(X)$, which amounts to a pre-composition of $t$ with $f$. Similarly, we write $t \gtrdot g$ for post-composition of $t$ with $g$. Definitions of these combinators are given below. Here we first show how they can be used to refine the CPS-translation.

The translation makes an explicit distinction between data and code. For any source type $X$, we define a *value type* $\mathcal{C}[\![X]\!]$ that is used to represent the values of type $X$. Second, we define an *interaction type* $[\![X]\!]$. These types are defined as follows.

$$\begin{aligned}
\mathcal{C}[\![\mathbb{N}]\!] &= \mathsf{nat} & [\![\mathbb{N}]\!] &= \bot^0 \\
\mathcal{C}[\![X \to Y]\!] &= \mathsf{G} & [\![X \to Y]\!] &= [\![X]\!] \multimap Cont_{\mathsf{G}\times\mathcal{C}[\![X]\!],\mathcal{C}[\![Y]\!]}([\![Y]\!])
\end{aligned}$$

Values of function type are represented as values of $\mathsf{G}$. They are intended to be opaque, i.e. they are not meant to be inspected. To be able to use opaque function values, they will each come with a program with interface $[\![X \to Y]\!]$ that makes it possible to apply a function given by an opaque value to any possible argument. To execute such a program with interface $[\![X \to Y]\!]$, we must connect to it a program of type $[\![X]\!]$ and we must provide a value of type $\mathsf{G} \times \mathcal{C}[\![X]\!]$. The value is the pair of the function value and the value of an argument. The program $[\![X]\!]$ must be supplied so that the program can make use of the argument value. This value may be opaque to the program itself.

For natural numbers we have $[\![\mathbb{N}]\!] = \bot^0$. A program with this type is vacuous, having no inputs and no output. This is because numbers are already fully specified by their values in $\mathcal{C}[\![\mathbb{N}]\!]$, so there is no need to provide a program for accessing them.

These definitions are extended to contexts as follows: $\mathcal{C}[\![\Gamma]\!]$ is the value type defined by

$$\mathcal{C}[\![\mathrm{empty}]\!] = \mathsf{unit}\ , \qquad \mathcal{C}[\![\Gamma,\, x\colon X]\!] = \mathcal{C}[\![\Gamma]\!] \times \mathcal{C}[\![X]\!]\ ,$$

and $[\![\Gamma]\!]$ is the context defined by

$$[\![\text{empty}]\!] = \text{empty} \ , \qquad [\![\Gamma,\, x\colon X]\!] = [\![\Gamma]\!],\, x\colon \mathsf{unit} \cdot [\![X]\!] \ .$$

With these definitions, we can define a refined CPS-translation that maps a source typing judgement $\Gamma \vdash t\colon X$ to the INT typing judgement $[\![\Gamma]\!] \vdash \mathsf{cps}(t)\colon \mathit{Cont}_{\mathcal{C}[\![\Gamma]\!],\mathcal{C}[\![X]\!]}(\mathcal{C}[\![X]\!])$. To execute $\mathsf{cps}(t)$, we therefore need to supply a value of type $\mathcal{C}[\![\Gamma]\!]$, which is the tuple of the free variables of $t$, as well as a program of type $[\![\Gamma]\!]$ to allow the term to make use of these values. We get a value of type $\mathcal{C}[\![X]\!]$ and access to a program with interface $[\![X]\!]$, using which we can use the value.

The translation is defined by induction on the source typing derivation.

- Rule VAR:

$$\frac{}{x\colon X \vdash x\colon X}$$
$$\Downarrow$$
$$\frac{}{x\colon \mathsf{unit} \cdot [\![X]\!] \vdash \eta(x)\colon \mathit{Cont}_{\mathcal{C}[\![X]\!],\mathcal{C}[\![X]\!]}(\mathcal{C}[\![X]\!])}$$

- Rule $\to$I: In an abstraction, the tuple of the values of the variables in the context $\Gamma$ is used as the function value. Since functions are represented using type $\mathsf{G}$, this tuple is encoded using $\mathsf{encode}$. Of course, when the function is applied, this encoded value needs to be decoded again. This idea is implemented as follows:

$$\frac{\Gamma,\, x\colon X \vdash t\colon Y}{\Gamma \vdash \lambda x{:}X.\, t\colon X \to Y}$$
$$\Downarrow$$
$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{[\![\Gamma]\!],\, x\colon \mathsf{unit} \cdot [\![X]\!] \vdash [\![t]\!]\colon \mathit{Cont}_{\mathcal{C}[\![\Gamma]\!]\times\mathcal{C}[\![X]\!],\mathcal{C}[\![Y]\!]}(\mathcal{C}[\![Y]\!])}{[\![\Gamma]\!],\, x\colon \mathsf{unit} \cdot [\![X]\!] \vdash (\mathsf{decode} \times \mathit{id}) \lessdot [\![t]\!]\colon \mathit{Cont}_{\mathsf{G}\times\mathcal{C}[\![X]\!],\mathcal{C}[\![Y]\!]}(\mathcal{C}[\![Y]\!])}}{[\![\Gamma]\!] \vdash \lambda x.\, (\mathsf{decode} \times \mathit{id}) \lessdot [\![t]\!]\colon [\![X]\!] \multimap \mathit{Cont}_{\mathsf{G}\times\mathcal{C}[\![X]\!],\mathcal{C}[\![Y]\!]}(\mathcal{C}[\![Y]\!])}}{[\![\Gamma]\!] \vdash \eta(\lambda x.\, (\mathsf{decode} \times \mathit{id}) \lessdot [\![t]\!])\colon \mathit{Cont}_{\mathcal{C}[\![\Gamma]\!],\mathcal{C}[\![\Gamma]\!]}([\![X]\!] \multimap \mathit{Cont}_{\mathsf{G}\times\mathcal{C}[\![X]\!],\mathcal{C}[\![Y]\!]}(\mathcal{C}[\![Y]\!])))}}{[\![\Gamma]\!] \vdash \eta(\lambda x.\, (\mathsf{decode} \times \mathit{id}) \lessdot [\![t]\!]) \gtrdot \mathsf{encode}\colon \mathit{Cont}_{\mathcal{C}[\![\Gamma]\!],\mathsf{G}}(\underbrace{([\![X]\!] \multimap \mathit{Cont}_{\mathsf{G}\times\mathcal{C}[\![X]\!],\mathcal{C}[\![Y]\!]}(\mathcal{C}[\![Y]\!]))}_{[\![X\to Y]\!]})}$$

- Rule $\to$E: In the case for application, the term $\mathit{split}_{\Gamma,\Delta}$ is an implementation of the canonical isomorphism $[\![\Gamma, \Delta]\!] \to [\![\Gamma]\!] \times [\![\Delta]\!]$.

$$\frac{\Gamma \vdash s\colon X \to Y \qquad \Delta \vdash t\colon X}{\Gamma, \Delta \vdash s\ t\colon Y}$$
$$\Downarrow$$
$$\frac{\dfrac{[\![\Gamma]\!] \vdash \mathsf{cps}(s)\colon \mathit{Cont}_{\mathcal{C}[\![\Gamma]\!],\mathsf{G}}(\overbrace{([\![X]\!] \multimap \mathit{Cont}_{\mathsf{G}\times\mathcal{C}[\![X]\!],\mathcal{C}[\![Y]\!]}(Y))}^{[\![X\to Y]\!]}) \qquad [\![\Delta]\!] \vdash \mathsf{cps}(t)\colon \mathit{Cont}_{\mathcal{C}[\![\Delta]\!],\mathcal{C}[\![X]\!]}(\mathcal{C}[\![X]\!])}{[\![\Gamma]\!], [\![\Delta]\!] \vdash \mathit{bind2}\ \mathsf{cps}(s)\ \mathsf{cps}(t)\ (\lambda f.\, \lambda x.\, f\ x))\colon \mathit{Cont}_{\mathcal{C}[\![\Gamma]\!]\times\mathcal{C}[\![\Delta]\!],\mathcal{C}[\![Y]\!]}(\mathcal{C}[\![Y]\!])}}{[\![\Gamma]\!], [\![\Delta]\!] \vdash \mathit{split}_{\Gamma,\Delta} \lessdot (\mathit{bind2}\ \mathsf{cps}(s)\ \mathsf{cps}(t)\ (\lambda f.\, \lambda x.\, f\ x))\colon \mathit{Cont}_{\mathcal{C}[\![\Gamma,\Delta]\!],\mathcal{C}[\![Y]\!]}(\mathcal{C}[\![Y]\!])}$$

- Rule WEAK: The translation of weakening now involves explicit discarding of the value of the weakened variable in the translated program.

$$\frac{\Gamma \vdash t\colon Y}{\Gamma,\, x\colon X \vdash t\colon Y}$$

$$\Downarrow$$

$$\frac{[\![\Gamma]\!] \vdash \mathsf{cps}(t)\colon [\![Y]\!]}{[\![\Gamma]\!], x\colon \mathsf{unit} \cdot [\![X]\!] \vdash \pi_1 \lessdot \mathsf{cps}(t)\colon [\![Y]\!]}$$

The term $\pi_1 \lessdot \mathsf{cps}(t)$ explicitly discards the value of the variable $x$, which does not appear in term $t$. This addresses the issue with the simple CPS-translation that values were never deallocated.

- Rule CONST: In the translation of constants and addition, we write $\star$ for the canonical term of type $\bot^0$.

$$\frac{}{\vdash n\colon \mathbb{N}}$$

$$\Downarrow$$

$$\frac{\dfrac{}{\vdash \eta\,\star\colon \mathit{Cont}_{\mathsf{unit},\mathsf{unit}}(\bot^0)}}{\vdash (\eta\,\star) \gtrdot \mathsf{const}_n\colon \underbrace{\mathit{Cont}_{\mathsf{unit},\mathsf{nat}}(\bot^0)}_{\mathit{Cont}_{\mathcal{C}[\![\mathrm{empty}]\!],\mathcal{C}[\![\mathbb{N}]\!]}([\![\mathbb{N}]\!])}}$$

- Rule ADD:

$$\frac{\Gamma \vdash s\colon \mathbb{N} \qquad \Delta \vdash t\colon \mathbb{N}}{\Gamma, \Delta \vdash s + t\colon \mathbb{N}}$$

$$\Downarrow$$

$$\frac{\dfrac{[\![\Gamma]\!] \vdash \mathsf{cps}(s)\colon \mathit{Cont}_{\mathcal{C}[\![\Gamma]\!],\mathsf{nat}}(\bot^0) \qquad [\![\Delta]\!] \vdash \mathsf{cps}(t)\colon \mathit{Cont}_{\mathcal{C}[\![\Delta]\!],\mathsf{nat}}(\bot^0)}{[\![\Gamma]\!], [\![\Delta]\!] \vdash \mathit{bind2}\ \mathsf{cps}(s)\ \mathsf{cps}(t)\ (\lambda i.\, \lambda j.\, (\eta\,\star) \gtrdot \mathsf{add})\colon \mathit{Cont}_{\mathcal{C}[\![\Gamma]\!]\times\mathcal{C}[\![\Delta]\!],\mathsf{nat}}(\bot^0)}}{[\![\Gamma]\!], [\![\Delta]\!] \vdash \mathit{split}_{\Gamma,\Delta} \lessdot (\mathit{bind2}\ \mathsf{cps}(s)\ \mathsf{cps}(t)\ (\lambda i.\, \lambda j.\, (\eta\,\star) \gtrdot \mathsf{add}))\colon \mathit{Cont}_{\mathcal{C}[\![\Gamma,\Delta]\!],\mathcal{C}[\![\mathbb{N}]\!]}([\![\mathbb{N}]\!])}$$

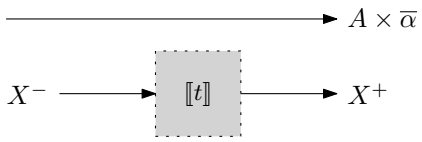
### 7.2.2 Low-Level Interpretation

Let us look at what the refined translation amounts to in terms of low-level programs. We first give concrete definitions of $\eta$, *bind2*, $\lessdot$ and $\gtrdot$, spell out the low-level programs that they define and give a concrete example for the complete translation from source language to low-level language.

The definition of $\eta\colon X \multimap \mathit{Cont}_{A,A}(X)$ is given by

$$\eta = \lambda x{:}X.\, \Lambda\alpha.\, \lambda k{:}\bot^{A\times\alpha}.\, k\ x$$

It is defined such that $\eta\, t$ translates (with simplification) to the following low-level program:



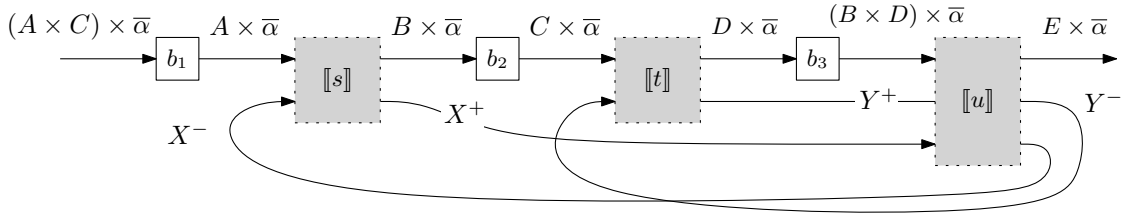

To define the other combinators, we first define a term $f^* \colon \bot^B \multimap \bot^A$ for any closed $f \colon A \to TB$. In fact it is possible to define it by a combinator of type $(A \to TB) \multimap (\bot^B \multimap \bot^A)$ using direct definition. It is interesting to note that direct is needed to define $f^*$. One may try to define it by $\lambda k.\, \mathsf{fn}\, x{:}A.\, \mathsf{let}\ y{=}f\ x\ \mathsf{in}\ k\ y$, but then $f^*$ would have type $(A \times B) \cdot \bot^B \multimap \bot^A$, rather than $\bot^B \multimap \bot^A$.

With this proviso, the binding combinator can be defined as follows.

$$\mathit{bind2} = \lambda s.\, \lambda t.\, \lambda u.\, \Lambda\alpha.\, \lambda k.\, f_1^*(s\ (C \times \alpha)\ (\lambda x.\, f_2^*(t\ (B \times \alpha)\ (\lambda y.\, f_3^*(u\ x\ y\ \alpha\ k)))))$$

where $f_1 = \mathsf{fn}\, \langle\langle a, c\rangle, z\rangle.\, \mathsf{return}\ \langle a, \langle c, z\rangle\rangle$, $f_2 = \mathsf{fn}\, \langle b, \langle c, z\rangle\rangle.\, \mathsf{return}\ \langle c, \langle b, z\rangle\rangle$ and finally $f_3 = \mathsf{fn}\, \langle d, \langle b, z\rangle\rangle.\, \mathsf{return}\ \langle\langle b, d\rangle, z\rangle$.

This combinator connects low-level programs $s$, $t$ and $u$ in the following way; compare this to the informal explanation of *bind2* above.



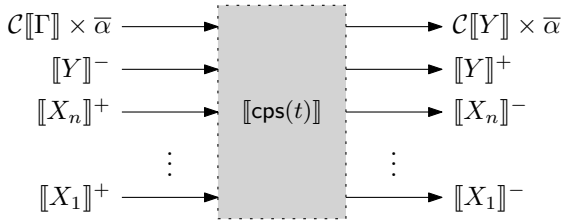

As in Chapter 3, we do not show any ports corresponding to free variables in $s$, $t$ and $u$, which are just passed to the outside. The blocks $b_1$, $b_2$ and $b_3$ are given by:

$$\begin{aligned} b_1\ (\langle\langle a, c\rangle, z\rangle) &= \langle a, \mathsf{encode}(\langle c, z\rangle)\rangle \\ b_2\ (\langle b, u\rangle) &= \mathsf{let}\ \langle c, z\rangle{=}\mathsf{decode}(u)\ \mathsf{in}\ \langle c, \mathsf{encode}(\langle b, z\rangle)\rangle \\ b_3\ (\langle d, u\rangle) &= \mathsf{let}\ \langle b, z\rangle{=}\mathsf{decode}(u)\ \mathsf{in}\ \langle\langle b, d\rangle, z\rangle \end{aligned}$$

Finally, the combinators for pre- and post-composition are defined as follows:

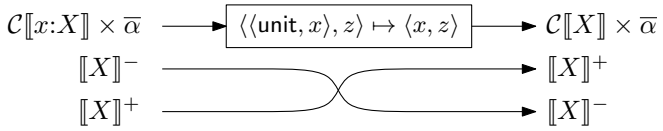
$$f \lessdot t = \Lambda\alpha.\, \lambda k.\, f^*(t\ \alpha\ k) \qquad t \gtrdot g = \Lambda\alpha.\, \lambda k.\, t\ \alpha\ \lambda y.\, g^*(k\ y)$$


With these definitions of the combinators, we can now explain what the CPS-translation amounts to in terms of low-level programs. A source typing sequent $\Gamma \vdash t \colon Y$ is translated to the INT sequent $[\![\Gamma]\!] \vdash \mathsf{cps}(t) \colon \mathit{Cont}_{\mathcal{C}[\![\Gamma]\!], \mathcal{C}[\![Y]\!]}([\![Y]\!])$, which translates to a low-level program with the following interface.

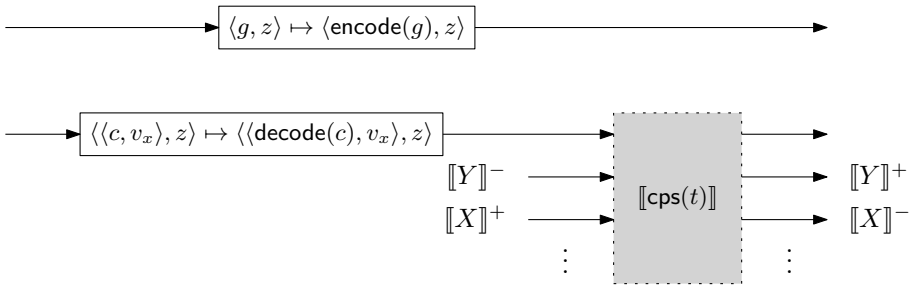



If one spells out the above translation and applies a few immediate simplifications, then one obtains the following translation.

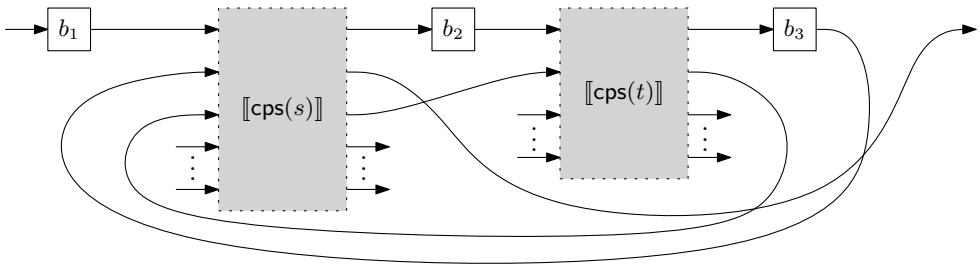
- Case variable:




- Case abstraction:

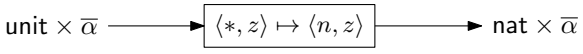



- Case application:

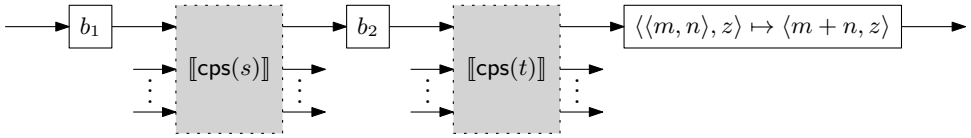



The blocks $b_1$, $b_2$ and $b_3$ are defined as for *bind2* above.

- Case number constant $n$:

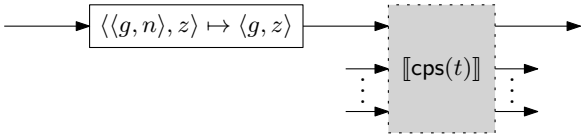



- Case addition:




The blocks $b_1$ and $b_2$ are again defined as for *bind2* above.

- Case weakening:




To give an idea of how this translation from source to low-level language implements call-by-value, we spell out as an example the translation of the term

$$(\lambda f.\, f\ 1)\ (\lambda x.\, y + x)\ .$$

We spell out the translation fully using the syntax of the low-level language (as opposed to just showing the graphical notation) in order to illustrate that the translation is quite similar to defunctionalization. In the concrete syntax, we must give names to the entry and exit labels of blocks (these names are hidden in the graphical notation). We shall use suggestive names and choose the labels so as to indicate from which subterm of the original term they originate. These subterms are numbered as follows:

$$((\lambda f.\, (f^5\ 1^6)^7)^8\ (\lambda x.\, (y^2 + x^1)^3)^4)^9$$

Next, we give the translation of each subterm of this term and build the translation of the whole program step by step.

- $x\colon \mathbb{N} \vdash x\colon \mathbb{N}$ translates to the program with one block:

$$\mathit{eval}_1\ (\langle\langle *, v_x\rangle, z\rangle\colon \mathcal{C}[\![x\colon \mathbb{N}]\!] \times \overline{\alpha})\ =\ \mathit{ret}_1(v_x, z)$$

  Its entry label is $\mathit{eval}_1$ and the exit label is $\mathit{ret}_1$.

- $y\colon \mathbb{N} \vdash y\colon \mathbb{N}$ translates to the program with one block:

$$\mathit{eval}_2\ (\langle\langle *, v_y\rangle, z\rangle\colon \mathcal{C}[\![y\colon \mathbb{N}]\!] \times \overline{\alpha})\ =\ \mathit{ret}_2(v_y, z)$$

  Its entry label is $\mathit{eval}_2$ and the exit label is $\mathit{ret}_2$.

- $y\colon \mathbb{N},\ x\colon \mathbb{N} \vdash y + x\colon \mathbb{N}$ translates to the program with the blocks from the translation of $x\colon \mathbb{N} \vdash x\colon \mathbb{N}$ and $y\colon \mathbb{N} \vdash y\colon \mathbb{N}$, in addition to

$$\begin{aligned}
&\mathit{eval}_3\ (\langle\langle\langle *, v_y\rangle, v_x\rangle, z\rangle\colon \mathcal{C}[\![y\colon \mathbb{N}, x\colon \mathbb{N}]\!] \times \overline{\alpha})\ =\ \mathit{eval}_2(\langle\langle *, v_y\rangle, \mathsf{encode}(\langle\langle *, v_x\rangle, z\rangle))\\
&\mathit{ret}_2\ (\langle n, u\rangle\colon \mathsf{nat} \times \overline{\alpha})\ =\ \mathsf{let}\ \langle\langle *, v_x\rangle, z\rangle{=}\mathsf{decode}(u)\ \mathsf{in}\ \mathit{eval}_1(\langle\langle *, v_x\rangle, \mathsf{encode}(n, z)\rangle)\\
&\mathit{ret}_1\ (\langle m, u\rangle\colon \mathsf{nat} \times \overline{\alpha})\ =\ \mathsf{let}\ \langle n, z\rangle{=}\mathsf{decode}(u)\ \mathsf{in}\ \mathit{ret}_3(\langle m+n, z\rangle)
\end{aligned}$$

  The single entry label is $\mathit{eval}_3$ and the single exit label is $\mathit{ret}_3$.

- $y\colon \mathbb{N} \vdash \lambda x.\, y+x\colon \mathbb{N} \to \mathbb{N}$ translates to the program with the blocks from $y\colon \mathbb{N},\ x\colon \mathbb{N} \vdash y + x\colon \mathbb{N}$ and the following additional blocks.

  $$\begin{aligned}
  &\mathit{eval}_4\ (\langle\langle *, v_y\rangle, z\rangle\colon \mathcal{C}[\![y\colon \mathbb{N}]\!] \times \overline{\alpha}) \ =\ \mathit{ret}_4(\langle \mathsf{encode}(\langle *, v_y\rangle), z\rangle)\\
  &\mathit{apply}_4\ (\langle\langle c, v_x\rangle, z\rangle\colon (\mathsf{G} \times \mathsf{nat}) \times \overline{\alpha}) \ =\ \mathsf{let}\ \langle *, v_y\rangle{=}\mathsf{decode}(c)\ \mathsf{in}\ \mathit{eval}_3(\langle\langle\langle *, v_y\rangle, v_x\rangle, z\rangle)\\
  &\mathit{ret}_3\ (\langle n, z\rangle\colon \mathsf{nat} \times \overline{\alpha}) \ =\ \mathit{applyret}_4(\langle n, z\rangle)
  \end{aligned}$$

  The list of entry labels is $\mathit{apply}_4, \mathit{eval}_4$, and the list of exit labels is $\mathit{applyret}_4, \mathit{ret}_4$.

- $f\colon \mathbb{N} \to \mathbb{N} \vdash f\colon \mathbb{N} \to \mathbb{N}$ translates to the program with the following blocks:

  $$\begin{aligned}
  &\mathit{eval}_5\ (\langle\langle *, v_f\rangle, z\rangle\colon \mathcal{C}[\![f\colon \mathbb{N} \to \mathbb{N}]\!] \times \overline{\alpha}) \ =\ \mathit{ret}_5(\langle\langle *, v_f\rangle, z\rangle)\\
  &\mathit{apply}_5\ (\langle\langle v_f, n\rangle, z\rangle\colon \mathsf{nat} \times \overline{\alpha}) \ =\ \mathit{apply}_f(\langle\langle v_f, n\rangle, z\rangle)\\
  &\mathit{applyret}_f\ (\langle n, z\rangle\colon \mathsf{nat} \times \overline{\alpha}) \ =\ \mathit{applyret}_5(\langle n, z\rangle)
  \end{aligned}$$

  The list of entry labels is $\mathit{applyret}_f, \mathit{eval}_5$, and the list of exit labels is $\mathit{apply}_f, \mathit{ret}_5$.

- $\vdash 1\colon \mathbb{N}$ translates to the program with a single block

  $$\mathit{eval}_6\ (\langle *, z\rangle\colon \mathcal{C}[\![\mathrm{empty}]\!] \times \overline{\alpha}) \ =\ \mathit{ret}_6(\langle 1, z\rangle)$$

  and a single entry label $\mathit{eval}_6$ and a single exit label $\mathit{ret}_6$.

- $f\colon \mathbb{N} \to \mathbb{N} \vdash f\ 1\colon \mathbb{N}$ translates to the program with the blocks from both the source terms $f\colon \mathbb{N} \to \mathbb{N} \vdash f\colon \mathbb{N} \to \mathbb{N}$ and $\vdash 1\colon \mathbb{N}$ and

  $$\begin{aligned}
  &\mathit{eval}_7\ (\langle\langle *, v_f\rangle, z\rangle\colon \mathcal{C}[\![f\colon \mathbb{N} \to \mathbb{N}]\!] \times \overline{\alpha}) \ =\ \mathit{eval}_5(\langle\langle *, v_f\rangle, \mathsf{encode}(\langle *, z\rangle)\rangle)\\
  &\mathit{ret}_5\ (\langle v_1, u\rangle\colon \mathsf{G} \times \overline{\alpha}) \ =\ \mathsf{let}\ \langle *, z\rangle{=}\mathsf{decode}(u)\ \mathsf{in}\ \mathit{eval}_6(\langle *, \mathsf{encode}(\langle v_1, z\rangle)\rangle)\\
  &\mathit{ret}_6\ (\langle v_2, u\rangle\colon \mathsf{G} \times \overline{\alpha}) \ =\ \mathsf{let}\ \langle v_1, z\rangle{=}\mathsf{decode}(u)\ \mathsf{in}\ \mathit{apply}_5(\langle\langle v_1, v_2\rangle, z\rangle)\\
  &\mathit{applyret}_5\ (\langle n, z\rangle\colon \mathsf{nat} \times \overline{\alpha}) \ =\ \mathit{ret}_7(\langle n, z\rangle)
  \end{aligned}$$

  The list of entry labels is $\mathit{applyret}_f, \mathit{eval}_7$, and the list of exit labels is $\mathit{apply}_f, \mathit{ret}_7$.

- $\vdash \lambda f.\, f\ 1\colon (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ translates to the program with the blocks from the source term $f\colon \mathbb{N} \to \mathbb{N} \vdash f\ 1\colon \mathbb{N}$ together with the following blocks.

  $$\begin{aligned}
  &\mathit{eval}_8\ (\langle *, z\rangle\colon \mathcal{C}[\![\mathrm{empty}]\!] \times \overline{\alpha}) \ =\ \mathit{ret}_8(\langle \mathsf{encode}(*), z\rangle)\\
  &\mathit{apply}_8\ (\langle\langle c, v_f\rangle, z\rangle\colon (\mathsf{G} \times \mathsf{G}) \times \overline{\alpha}) \ =\ \mathsf{let}\ v{=}\mathsf{decode}(c)\ \mathsf{in}\ \mathit{eval}_7(\langle\langle v, v_f\rangle, z\rangle)\\
  &\mathit{ret}_7\ (\langle n, z\rangle\colon \mathsf{nat} \times \overline{\alpha}) \ =\ \mathit{applyret}_8(\langle n, z\rangle)
  \end{aligned}$$

  The list of entry labels is $\mathit{applyret}_f, \mathit{apply}_8, \mathit{eval}_8$.
  The list of exit labels is $\mathit{apply}_f, \mathit{applyret}_8, \mathit{ret}_8$.

- Finally, $y\colon \mathbb{N} \vdash (\lambda f.\, f\ 1)\ (\lambda x.\, y + x)\colon \mathbb{N}$ translates to the program with the blocks from both $\vdash \lambda f.\, f\ 1\colon (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ and $y\colon \mathbb{N} \vdash \lambda x.\, y + x\colon \mathbb{N} \to \mathbb{N}$ as well as the following blocks.

  $$\begin{aligned}
  &\mathit{eval}_9\ (\langle\langle *, v_y\rangle, z\rangle\colon \mathcal{C}[\![y\colon \mathbb{N}]\!] \times \overline{\alpha}) \ =\ \mathit{eval}_8(\langle *, \mathsf{encode}(\langle\langle *, v_y\rangle, z\rangle)\rangle)\\
  &\mathit{ret}_8\ (\langle v_1, u\rangle\colon \mathsf{G} \times \overline{\alpha}) \ =\ \mathsf{let}\ \langle\langle *, v_y\rangle, z\rangle{=}\mathsf{decode}(u)\ \mathsf{in}\ \mathit{eval}_4(\langle\langle *, v_y\rangle, \mathsf{encode}(\langle v_1, z\rangle)\rangle)\\
  &\mathit{ret}_4\ (\langle v_2, u\rangle\colon \mathsf{G} \times \overline{\alpha}) \ =\ \mathsf{let}\ \langle v_1, z\rangle{=}\mathsf{decode}(u)\ \mathsf{in}\ \mathit{apply}_8(\langle\langle v_1, v_2\rangle, z\rangle)\\
  &\mathit{applyret}_8\ (\langle n, z\rangle\colon \mathsf{nat} \times \overline{\alpha}) \ =\ \mathit{ret}_9(\langle n, z\rangle)\\
  &\mathit{apply}_f\ (\langle\langle v_f, v_x\rangle, z\rangle\colon (\mathsf{G} \times \mathsf{nat}) \times \overline{\alpha}) \ =\ \mathit{apply}_4(\langle\langle v_f, v_x\rangle, z\rangle)\\
  &\mathit{applyret}_4\ (\langle n, z\rangle\colon \mathsf{nat} \times \overline{\alpha}) \ =\ \mathit{applyret}_f(\langle n, z\rangle)
  \end{aligned}$$

  It has a single entry label $\mathit{eval}_9$ and a single exit label $\mathit{ret}_9$.

To see that the low-level program correctly implements the source term, let us evaluate its value with the value 3 for $y$. We get the following trace of jumps to block labels.

$1 : \mathit{eval}_9(\langle\langle *, 3\rangle, z\rangle)$
$2 : \mathit{eval}_8(\langle *, \mathsf{encode}(\langle\langle *, 3\rangle, z\rangle)\rangle)$
$3 : \mathit{ret}_8(\langle \mathsf{encode}(*), \mathsf{encode}(\langle\langle *, 3\rangle, z\rangle)\rangle)$
$4 : \mathit{eval}_4(\langle\langle *, 3\rangle, \mathsf{encode}(\langle \mathsf{encode}(*), z\rangle)\rangle)$
$5 : \mathit{ret}_4(\langle \mathsf{encode}(\langle *, 3\rangle), \mathsf{encode}(\langle \mathsf{encode}(*), z\rangle)\rangle)$
$6 : \mathit{apply}_8(\langle\langle \mathsf{encode}(*), \mathsf{encode}(\langle *, 3\rangle)\rangle, z\rangle)$
$7 : \mathit{eval}_7(\langle\langle *, \mathsf{encode}(\langle *, 3\rangle)\rangle, z\rangle)$
$8 : \mathit{eval}_5(\langle\langle *, \mathsf{encode}(\langle *, 3\rangle)\rangle, \mathsf{encode}(\langle *, z\rangle)\rangle)$
$9 : \mathit{ret}_5(\langle\langle *, \mathsf{encode}(\langle *, 3\rangle)\rangle, \mathsf{encode}(\langle *, z\rangle)\rangle)$
$10 : \mathit{eval}_6(\langle *, \mathsf{encode}(\langle \mathsf{encode}(\langle *, 3\rangle), z\rangle)\rangle)$
$11 : \mathit{ret}_6(\langle 1, \mathsf{encode}(\langle \mathsf{encode}(\langle *, 3\rangle), z\rangle)\rangle)$
$12 : \mathit{apply}_5(\langle\langle \mathsf{encode}(\langle *, 3\rangle), 1\rangle, z\rangle)$
$13 : \mathit{apply}_f(\langle\langle \mathsf{encode}(\langle *, 3\rangle), 1\rangle, z\rangle)$
$14 : \mathit{apply}_4(\langle\langle \mathsf{encode}(\langle *, 3\rangle), 1\rangle, z\rangle)$
$15 : \mathit{eval}_3(\langle\langle\langle *, 3\rangle, 1\rangle, z\rangle)$
$16 : \mathit{eval}_2(\langle\langle *, 3\rangle, \mathsf{encode}(\langle *, 1\rangle, z)\rangle)$
$17 : \mathit{ret}_2(\langle 3, \mathsf{encode}(\langle *, 1\rangle, z)\rangle)$
$18 : \mathit{eval}_1(\langle\langle *, 1\rangle, \mathsf{encode}(3, z)\rangle)$
$19 : \mathit{ret}_1(\langle 1, \mathsf{encode}(3, z)\rangle)$
$20 : \mathit{ret}_3(\langle 4, z\rangle)$
$21 : \mathit{applyret}_4(\langle 4, z\rangle)$
$22 : \mathit{applyret}_f(\langle 4, z\rangle)$
$23 : \mathit{applyret}_5(\langle 4, z\rangle)$
$24 : \mathit{ret}_7(\langle 4, z\rangle)$
$25 : \mathit{applyret}_8(\langle 4, z\rangle)$
$26 : \mathit{ret}_9(\langle 4, z\rangle)$

The example illustrates that the low-level program can be simplified quite a lot. The low-level language is suitable for such simplifications. If one extends the very simple optimisation pass described in (Schöpp, 2014c) and implemented in (Schöpp, 2014d) with knowledge about encode and decode, then one can expect to simplify the above program to:

$$\mathit{eval}_9\ (\langle\langle *, v_y\rangle, z\rangle\colon \mathcal{C}[\![y\colon \mathbb{N}]\!] \times \overline{\alpha}) \ =\ \mathit{ret}_9(\langle v_y + 1, z\rangle)$$

### 7.2.3 Contraction

So far we have only described the translation for the linear fragment of the source language. To account for contraction, it is possible to use subexponentials.

The contraction rule

$$\textsc{contr}\,\frac{\Gamma, x_1\colon X, x_2\colon X \vdash t\colon Y}{\Gamma, x\colon X \vdash t[x/x_1, x/x_2]\colon Y}$$

can then be translated to a derivation of the form:

$$\frac{\dfrac{[\![\Gamma]\!], x_1\colon A_1 \cdot [\![X]\!], x_2\colon A_2 \cdot [\![X]\!] \vdash \mathsf{cps}(t)\colon \mathit{Cont}_{(\mathcal{C}[\![\Gamma]\!]\times\mathcal{C}[\![X]\!])\times\mathcal{C}[\![X]\!],\mathcal{C}[\![Y]\!]}(\mathcal{C}[\![Y]\!])}{[\![\Gamma]\!], x_1\colon A_1 \cdot [\![X]\!], x_2\colon A_2 \cdot [\![X]\!] \vdash \mathit{dup} \lessdot \mathsf{cps}(t)\colon \mathit{Cont}_{\mathcal{C}[\![\Gamma]\!]\times\mathcal{C}[\![X]\!],\mathcal{C}[\![Y]\!]}(\mathcal{C}[\![Y]\!])}}{[\![\Gamma]\!], x\colon (A_1 + A_2) \cdot [\![X]\!] \vdash \mathsf{copy}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ (\mathit{dup} \lessdot \mathsf{cps}(t))\colon \mathit{Cont}_{\mathcal{C}[\![\Gamma]\!]\times\mathcal{C}[\![X]\!],\mathcal{C}[\![Y]\!]}(\mathcal{C}[\![Y]\!])}$$

In this derivation, *dup* is the canonical term that duplicates the value of type $\mathcal{C}[\![X]\!]$. The appearances of *Cont* in this derivation should be understood to be annotated appropriately with subexponentials (e.g. by type inference).

Consider for example the source term $(\lambda f.\,(f\ 1) + (f\ 2))\ (\lambda x.\,y + x)$. The low-level program for it will have only a single copy of the code for the application of the function $(\lambda x.\,y + x)$, which the program jumps to twice. The blocks in the program for $(\lambda x.\,y + x)$ however have an additional argument of type $\mathsf{unit}+\mathsf{unit}$ (or similar, depending on the choice of typing derivation) arising from the subexponentials. This value remembers whether the term is evaluated for the first time or the second time. It determines where to return the result value of the application to.

Concretely, if we label subterms as follows $((\lambda f.\,(f\ 1)^1 + (f\ 2)^2)^3\ (\lambda x.\,y + x)^4)^5$, then the relevant parts of the blocks for evaluating terms 1 and 2 and for applying function 4 are:

$$\begin{aligned}
&\mathit{eval}_1\,(\langle\langle *, v_f\rangle, \cdots\rangle\colon \mathsf{unit} \times \overline{\alpha}) \;=\; \mathit{apply}_4(\langle \mathsf{inl}(*), \langle\langle v_f, 1\rangle, \cdots\rangle\rangle)\\
&\mathit{eval}_2\,(\langle\langle *, v_f\rangle, \cdots\rangle\colon \mathsf{unit} \times \overline{\alpha}) \;=\; \mathit{apply}_4(\langle \mathsf{inr}(*), \langle\langle v_f, 2\rangle, \cdots\rangle\rangle)\\
&\mathit{apply}_4(\langle c, \langle\langle e, v_x\rangle, \cdots\rangle\rangle\colon (\mathsf{unit} + \mathsf{unit}) \times ((\mathsf{G} \times \mathsf{nat}) \times \overline{\alpha})) \;=\\
&\qquad\qquad \mathsf{let}\ \langle *, v_y\rangle{=}\mathsf{decode}(e)\ \mathsf{in}\ \mathit{applyret}_4(\langle c, \langle v_y + v_x, \cdots\rangle\rangle)\\
&\mathit{applyret}_4\,(\langle c, \langle n, \cdots\rangle\rangle\colon (\mathsf{unit} + \mathsf{unit}) \times (\mathsf{nat} \times \overline{\alpha})) \;=\; \mathsf{case}\ c\ \mathsf{of}\ \mathsf{inl}(_) \Rightarrow \mathit{ret}_1(\langle n, \cdots\rangle)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ;\ \mathsf{inr}(_) \Rightarrow \mathit{ret}_2(\langle n, \cdots\rangle)
\end{aligned}$$

...

The block $\mathit{eval}_1$ implements the evaluation of the term $(f\ 1)$. To evaluate this term, the program jumps to label $\mathit{apply}_4$ with function value $v_f$ and argument 1 and further the value $\mathsf{inl}(*)$, which indicates that the call came from the first use of $f$. In the block $\mathit{eval}_2$, which evaluates $(f\ 2)$, the value $\mathsf{inr}(*)$ is used instead. The return block $\mathit{applyret}_4$ now has a case distinction that returns the result of the application of the function the right caller.

Note, however, that while the function is used twice, its value is computed only once. To evaluate the whole term, first the subterm 3 is evaluated, then subterm 4. Only then does the program jump to $\mathit{eval}_1$ with the computed function value $v_f$. This function value remains unchanged in the rest of the computation. It is passed to $\mathit{apply}_4$ twice.

### 7.2.4 What do we gain?

We have outlined how a translation of a call-by-value source language to the low-level language may be defined by a CPS-translation into INT. Of course, it is possible to define such a translation directly without going through INT. For example, in (Cejtin et al., 2000) a defunctionalizing translation from source to low-level language is presented directly. However, it is not obvious how to prove the correctness of such a direct translation.

By factoring the translation through INT, we identify logical structure in the translation that can be used to show its correctness. In Chapter 3 we have defined an equational theory for INT using relational parametricity, which captures non-trivial reasoning on the level of low-level programs. We have noted in Section 3.3 that even the $\beta$-equations for $\multimap$ capture non-trivial reasoning. These equations were enough to account for the encoding of a call-by-name source language. The translation of call-by-value is an example that illustrates the utility of relational parametricity.

For the translation that has been outlined in this section, it is not obvious how to prove correctness. Consider the translation of a $\lambda$-abstraction. A source term $\Gamma \vdash \lambda x{:}X.\, t\colon X \to Y$ is translated to a term of type

$$Cont_{\mathcal{C}[\![\Gamma]\!],\mathsf{G}}([\![X]\!] \multimap Cont_{\mathsf{G}\times\mathcal{C}[\![X]\!],\mathcal{C}[\![Y]\!]}([\![Y]\!]))\ .$$

Notice that there are two occurrences of $\mathsf{G}$. In the outermost occurrence of *Cont*, we first compute the function value as a value $v_f$ of type $\mathsf{G}$. Then, any time we want to apply the function encoded by this value to an argument value $v_x\colon \mathcal{C}[\![X]\!]$, we pass the pair $\langle v_f, v_x\rangle\colon \mathsf{G}\times\mathcal{C}[\![X]\!]$ to the inner occurrence of *Cont*. As an input to the inner occurrence of *Cont*, we should pass only the value $v_f$ that has been returned by the outermost occurrence. We consider the value $v_f$ as an opaque abstract value that may not be inspected or modified and that must be passed to the inner occurrences of *Cont* unchanged. The above type does not enforce such a correspondence.

In a proof of correctness of the translation, we must prove that such a correspondence between the two occurrences of $\mathsf{G}$ is respected. In (Schöpp, 2014a), we show how to do this using relational parametricity. Roughly, the idea is to change the type to

$$\exists\alpha.\, Cont_{\mathcal{C}[\![\Gamma]\!],\alpha}([\![X]\!] \multimap Cont_{\alpha\times\mathcal{C}[\![X]\!],\mathcal{C}[\![Y]\!]}([\![Y]\!]))$$

and use parametricity to show that this enforces the required invariants. Another option might be to enforce suitable invariants globally, but here we use relational parametricity.

## 7.3 Correctness using Parametricity

The aim is now to integrate the information hiding needed for the correctness argument in the CPS-translation by means of parametric polymorphism. To control the scope of the quantifiers, it is convenient to move to an equivalent formulation of the CPS-translation. There is a one-to-one correspondence between functions of type $A \to ((B \to \bot) \to \bot)$ and functions of type $(B \to \bot) \to (A \to \bot)$. Hence, in the CPS-translation of the simply-typed

$\lambda$-calculus we may replace $[\![X \to Y]\!] = [\![X]\!] \to \mathit{Cont}([\![Y]\!])$ by $[\![X \to Y]\!] = \mathcal{K}Y \to \mathcal{K}X$, where $\mathcal{K}X = [\![X]\!] \to \bot$ is the type of continuations for type $X$. The CPS-translation of terms changes only in the order of arguments. In this variant, the continuation is always the first argument.

A similar modification can be made to the CPS-translation with explicit state. In this form it is then suitable for adding information hiding using polymorphic quantification. We outline the resulting transformation without subexponentials for the linear source language. To cover the full source language, one only needs to allow subexponentials, as above.

The type of continuations is an interface type in INT and has the following form.

$$\mathcal{K}_\alpha(X) = \forall\varphi.\, [\![X]\!]_\varphi \multimap \bot^{(\mathcal{C}[\![X]\!]_\varphi \times \alpha)}$$

To fully apply a continuation, one needs to supply a type $\varphi$, a program of type $[\![X]\!]_\varphi$ and a value of type $\mathcal{C}[\![X]\!]_\varphi$. The type $\varphi$ is the type that is used to represent the private/opaque part of the value that is put into the continuation. The value of type $\mathcal{C}[\![X]\!]_\varphi$ represents the actual value that is thrown into the continuation. Finally, the program $[\![X]\!]_\varphi$ allows the continuation to make use of the value. After all, the value that is thrown into the continuation is encoded using the type $\varphi$ that the continuation knows nothing about. To make use of this value, the continuation can use the program of type $[\![X]\!]_\varphi$.

The CPS-translation of types is then defined as follows.

$$\begin{aligned} \mathcal{C}[\![\mathbb{N}]\!]_\varphi &= \mathsf{nat} & [\![\mathbb{N}]\!]_\varphi &= \bot^0 \\ \mathcal{C}[\![X \to Y]\!]_\varphi &= \varphi & [\![X \to Y]\!]_\varphi &= \forall\alpha.\, \mathcal{K}_\alpha(Y) \multimap \mathcal{K}_{\varphi\times\alpha}(X) \end{aligned}$$

The parameter $\varphi$ is the value type whose values can be considered private in the encoding of the values. The continuation monad becomes:

$$\mathit{Cont}_\gamma(X) = \forall\alpha.\, \mathcal{K}_\alpha(X) \multimap \bot^{(\gamma\times\alpha)} \quad .$$

With these definitions, the above CPS-translation can be adapted so that the source typing judgement $x_1 \colon X_1, \ldots, x_n \colon X_n \vdash t \colon Y$ is translated to

$$x_1 \colon [\![X_1]\!]_{\varphi_1}, \ldots, x_n \colon [\![X_n]\!]_{\varphi_n} \vdash \mathsf{cps}(t) \colon \mathit{Cont}_{\mathcal{C}[\![X_1]\!]_{\varphi_1} \times \cdots \times \mathcal{C}[\![X_n]\!]_{\varphi_n}}(Y)$$

in INT. For each free variable $x_i$, we choose a fresh type variable $\varphi_i$ here.

Such a translation is presented in (Schöpp, 2014a). In this paper, the translation is presented using a basic logic for interaction, which can be seen as a fragment of INT that allows one to focus on the issues of the translation. This fragment is close to Tensorial Logic, as outlined in Section 3.2.2. The resulting translation into this fragment does not mention $\mathsf{encode}$ and $\mathsf{decode}$ terms anymore. These encodings and decodings are now captured by the use of universal quantification. The description from the previous section can be seen as an explicit description of the obtained implementation.

The correctness theorem of (Schöpp, 2014a, Corollary 1) states that, for any closed source term of base type $\vdash t \colon \mathbb{N}$ and any natural number $n$, if $t \longrightarrow^*_{\mathsf{cbv}} n$ then $[\![\mathsf{cps}(t)\ \mathsf{unit}]\!]$ is, to isomorphism, a program of type $\mathsf{unit} \to \mathsf{nat}$ that maps $*$ to $n$. The proof makes use of parametricity, see (Schöpp, 2014a, Lemma 9).

## 7.4 Further Directions

One interesting direction for further work is to compare the translation of call-by-value to practical applications of defunctionalization in compilation, e.g. in MLton (Cejtin et al., 2000). Above we have observed that the result of the translation using INT resembles that of a call-by-value defunctionalization. One goal might be to make this observation precise.

Even before establishing a formal correspondence, one may consider the similarity to defunctionalization as first evidence that a translation using INT may also produce efficient low-level programs. From the work on MLton, call-by-value defunctionalization is known to produce efficient programs. It would be an interesting direction for practical work to ascertain if this is the case for a translation using INT too. In this context, the approach of using computation-by-interaction may also bring new a toolkit to work on problems with defunctionalizing compilation, e.g. to achieve useful (partial) separate compilation.

For practical purposes, the translation using INT may need to be optimised. For example, the use of $\mathsf{G}$, $\mathsf{encode}$ and $\mathsf{decode}$ in the translation of universal quantifiers is quite simple-minded. It may perhaps be improved by using union types in place of $\mathsf{G}$, so that $\mathsf{encode}$ and $\mathsf{decode}$ become merely injection into and projection out of a union type. By avoiding encoding and decoding, one may hope to obtain more efficient low-level programs. One possibility of tracking the details of union types may be to introduce bounded quantifiers to the INT type system, i.e. have types of the form $\forall\alpha \lhd (\beta_1 \cup \cdots \cup \beta_n).\, X$ or similar.

The certification of space bounds and the space analysis of call-by-value functional programs would be another a natural direction for further work. We have seen that INT makes all space usage fully visible.

In a more theoretical direction, we should like to clarify the relation of the interactive implementation to game semantic models of call-by-value (Honda and Yoshida, 1999; Abramsky and McCusker, 1997). Discussions with Nikos Tzevelekos exposed a striking similarity of the implementation of call-by-value in INT with the dialogues of call-by-value games. However, the details remain to be worked out.

Finally, we note that computational effects from the low-level language can be lifted to call-by-value source language. Indeed, note that $\bot^A$ is defined by $A \to T0$ and thus wraps the effects of the low-level language, much like in the work of Melliès (2012). For instance, a term $x\colon \mathbb{N} \vdash \mathsf{print}(x)\colon \mathbb{N}$ may be translated to $\lambda k.\, (\mathsf{fn}\, x{:}\mathsf{nat}.\, \mathsf{let}\ u{=}\mathsf{print}(x)\ \mathsf{in}\ \mathsf{return}\ x)^* k$ of type $((\bot^0 \multimap \bot^{\mathsf{nat}}) \multimap \bot^{\mathsf{nat}}$. This interprets $\mathsf{print}(x)$ so that $x$ is printed when the term is evaluated. Operations for other effects that may be present in the low-level language may similarly be lifted. While the definition of the translation of call-by-value into INT was motivated mainly by the issue of space usage outlined above, it should be interesting to compare this approach to the work of Hoshino et al. (2014) on effects in the Geometry of Interaction.

# 8 Conclusion

We have studied computation-by-interaction as an approach to structuring low-level computation. We have defined a calculus INT that uses higher types to organise low-level computation, and we have assessed the structure of low-level programs that is identified in this way. We have outlined the practical application of INT for low-level programming, we have used it to characterise the complexity class of the functions computable in logarithmic space, and we have shown how it can decompose call-by-name and call-by-value. While INT was derived from mathematical constructions in the context of game semantics, we have shown that it is related to defunctionalization, a standard technique in the compilation of higher-order languages to low-level languages. Moreover, in the translation of call-by-value, we have seen that the structure of INT guides the identification of logical principles that are useful for reasoning about low-level programs.

This work may serve as the basis for further work in a number of directions.

First, there is the connection between game semantics, low-level languages and compiler construction. The results described in Chapter 6 show that the plays of game semantics are closely related to machine code traces, at least for a call-by-name translation. The results from Chapter 7 indicate that a similar correspondence may also hold for the call-by-value case. This raises the question to what extent game semantics may be used as a theory of compiler construction. For example, in game semantics it is well-known how to specify open terms and their possible behaviour. Recent work on the verification of separate compilation (Beringer et al., 2014) shows a striking resemblance to such methods. It should be worth to investigate to what extent a transfer of techniques can solve existing problems in these areas.

Since work on game semantics has concentrated on theoretical aspects, while the development of compilation techniques has been motivated by practical concerns, a transfer of techniques may well be interesting. The focus on structure, proof techniques and compositionality in work on game semantics may make the methods developed there useful for compiler verification. On the other hand, the applications of computation-by-interaction mentioned in the Introduction were based on implementations of game semantics. The knowledge from compiler construction could be useful when it comes to the efficient implementation of such approaches.

These directions all lead to the long-term goal of identifying the mathematical structure that allows the construction of correct-by-construction compilers from mathematical components. Identifying the low-level structure, e.g. for the details of memory management or

for efficient cache usage, appears to be a challenging problem.

Another difficult problem for further work is to extend the focus from correctness to encompass analysis and certification of resource usage. As a very small first step, one may consider the application of the fine structure of INT towards this goal. For example, the characterisation of space complexity classes from Chapter 5 and the translation of call-by-value from Chapter 7 suggest an approach to establishing space bounds on call-by-value programs, e.g. for hardware synthesis or sublinear space computation as before. The monadic description of the translation from call-by-value to INT suggests a formulation as an effect type system that makes value types explicit.

# Bibliography


Samson Abramsky and Radha Jagadeesan. New foundations for the geometry of interaction. *Information and Computation*, 111(1):53–119, 1994.

Samson Abramsky and Guy McCusker. Call-by-value games. In *Computer Science Logic, CSL 1997*, volume 1414 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1997.

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

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

Samson Abramsky, Dan R. Ghica, Andrzej S. Murawski, C.-H. Luke Ong, and Ian D. B. Stark. Nominal games and full abstraction for the nu-calculus. In *Logic in Computer Science, LICS 2004*, pages 150–159. IEEE, 2004.

Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. Certified complexity (CerCo). In Ugo Dal Lago and Ricardo Peña, editors, *Foundational and Practical Aspects of Resource Analysis, FOPARA 2013*, volume 8552 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN 0-521-41695-7.

Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

Vincent Atassi, Patrick Baillot, and Kazushige Terui. Verification of ptime reducibility for system F terms via dual light affine logic. In Zoltán Ésik, editor, *Computer Science Logic, CSL 2006*, volume 4207 of *Lecture Notes in Computer Science*, pages 150–166, 2006.

Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009.

Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, TACS 2001*, pages 420–447, 2001.

Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, The University of Edinburgh, 1996.

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

Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. Verified compilation for shared-memory C. In Zhong Shao, editor, *European Symposium on Programming, ESOP 2014*, volume 8410 of *Lecture Notes in Computer Science*, pages 107–127. Springer, 2014.

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

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In Zhong Shao, editor, *European Symposium on Programming, ESOP 2014*, volume 8410 of *Lecture Notes in Computer Science*, pages 351–370. Springer, 2014.

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In Gert Smolka, editor, *European Symposium on Programming, ESOP 2000*, pages 56–71, 2000.

Ugo Dal Lago and Ulrich Schöpp. Functional programming in sublinear space. In Andrew D. Gordon, editor, *European Symposium on Programming, ESOP 2010*, pages 205–225, 2010a.

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

Ugo Dal Lago and Ulrich Schöpp. Computation by interaction for space bounded functional programming. Submitted to Information and Computation, 2013. URL `http://www2.ifi.lmu.de/~schoepp/intml.pdf`.

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

Olivier Danvy. Refunctionalization at work. In Tarmo Uustalu, editor, *Mathematics of Program Construction, MPC 2006*, volume 4014 of *Lecture Notes in Computer Science*, page 4. Springer, 2006.

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

Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite model theory*. Perspectives in Mathematical Logic. Springer, 1995. ISBN 978-3-540-60149-4.

Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. Enriching an effect calculus with linear types. In Erich Grädel and Reinhard Kahle, editors, *Computer Science Logic, CSL 2009*, volume 5771 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2009.

Sebastian Franz. Erweiterung eines Compilers für funktionale Programme um Hardwaresynthese. BSc Thesis, Ludwig-Maximilians-Universität München, 2012.

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

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

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

Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In Zhong Shao, editor, *European Symposium on Programming, ESOP 2014*, volume 8410 of *Lecture Notes in Computer Science*, pages 331–350. Springer, 2014.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

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

Jean-Yves Girard. Proof-nets: The parallel syntax for proof-theory. In Aldo Ursini and Paulo Agliano, editors, *Logic and Algebra (Pontignano, 1994)*, Lecture Notes in Pure and Applied Mathematics, pages 97–124. CRC Press, 1996.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989. ISBN 0-521-37181-3.

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

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

Kohei Honda and Nobuko Yoshida. Game-theoretic analysis of call-by-value computation. *Theor. Comput. Sci.*, 221(1-2):393–456, 1999.

Naohiko Hoshino, Koko Muroya, and Ichiro Hasuo. Memoryful geometry of interaction: from coalgebraic components to algebraic effects. In *Computer Science Logic – Logic in Computer Science, CSL-LICS 2014*. ACM, 2014.

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

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

Neil Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999. ISBN 978-1-4612-6809-3.

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

Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Principles of Programming Languages, POPL '14*, pages 633–646. ACM, 2014.

James Laird. A fully abstract game semantics of local exceptions. In *Logic in Computer Science, 2001*, pages 105–114. IEEE, 2001.

Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, CGO 2004*, pages 75–88. IEEE, 2004.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52 (7):107–115, 2009.

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

Paul Lorenzen. Ein dialogisches Konstruktivitätskriterium. *Infinitistic Methods*, 1961.

Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, 2nd edition, 1998.

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

Paul-André Melliès. Functorial boxes in string diagrams. In Zoltán Ésik, editor, *Computer Science Logic, CSL 2006*, volume 4207 of *Lecture Notes in Computer Science*, pages 1–30, Berlin, Heidelberg, 2006. Springer.

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

Paul-André Melliès. Tensorial logic with algebraic effects. Slides from workshop *Logic and interactions*, Luminy, 2012.

Paul-André Melliès. Local states in string diagrams. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi, RTA-TLCA 2014*, volume 8560 of *Lecture Notes in Computer Science*, pages 334–348. Springer, 2014.

Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Principles of Programming Languages, POPL 1996*, pages 271–283. ACM, 1996.

P. Møller-Neergaard. *Complexity Aspects of Programming Language Design*. PhD thesis, Brandeis University, 2004.

Peter Møller Neergaard. A functional language for logarithmic space. In Wei-Ngan Chin, editor, *Asian Symposium on Programming Languages and Systems, APLAS 2004*, volume 3302 of *Lecture Notes in Computer Science*, pages 311–326. Springer, 2004.

J. Gregory Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In Xavier Leroy and Atsushi Ohori, editors, *Types in Compilation, TIC 1998*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52. Springer, 1998.

J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

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

Hanno Nickau. Hereditarily sequential functionals. In Anil Nerode and Yuri Matiyasevich, editors, *Logical Foundations of Computer Science, LFCS 1994*, volume 813 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 1994.

Lasse R. Nielsen. A denotational investigation of defunctionalization. Technical report, BRICS, 2000. URL `http://brics.dk/RS/00/47/BRICS-RS-00-47.pdf`.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis (2. corr. print)*. Springer, 2005. ISBN 978-3-540-65410-0.

Vivek Nigam and Dale Miller. Algorithmic specifications in linear logic with subexponentials. In *Principles and Practice of Declarative Programming, PPDP 2009*, pages 129–140. ACM, 2009.

James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In Zhong Shao, editor, *European Symposium on Programming, ESOP 2014*, volume 8410 of *Lecture Notes in Computer Science*, pages 128–148. Springer, 2014.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: unified static analysis of context-dependence. In *International Conference on Automata, Languages, and Programming - Volume Part II, ICALP 2013*, 2013.

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

Fabrice Rastello, editor. *SSA-based Compiler Design*, 2015. Springer. To appear. Draft available from: `http://ssabook.gforge.inria.fr/latest/`.

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

John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

John C. Reynolds. The essence of ALGOL. In Peter W. O'Hearn and Robert D. Tennent, editors, *ALGOL-like Languages, Volume 1*, pages 67–88. Birkhauser Boston Inc., Cambridge, MA, USA, 1997. ISBN 0-8176-3880-6.

Ulrich Schöpp. Space-efficient computation by interaction. In Zoltán Ésik, editor, *Computer Science Logic, CSL 2006*, volume 4207 of *Lecture Notes in Computer Science*, pages 606–621. Springer, 2006.

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

Ulrich Schöpp. Computation-by-interaction with effects. In Hongseok Yang, editor, *Asian Symposium on Programming Languages and Systems, APLAS 2011*, volume 7078 of *Lecture Notes in Computer Science*, pages 305–321. Springer, 2011.

Ulrich Schöpp. Experimental interpreter for IntML. `http://www.github.com/uelis/intml`, 2012.

Ulrich Schöpp. Call-by-value in a basic logic for interaction. In Jacques Garrigue, editor, *Asian Symposium on Programming Languages and Systems, APLAS 2014*, volume 8858 of *Lecture Notes in Computer Science*, pages 428–448. Springer, 2014a.

Ulrich Schöpp. On the relation of interaction semantics to continuations and defunctionalization. *Logical Methods in Computer Science*, 2014b. accepted, to appear. Electronic version available from `http://arxiv.org/pdf/1410.4980`.

Ulrich Schöpp. Organising low-level programs using higher types. In *Principles and Practice of Declarative Programming, PPDP 2014*, New York, NY, 2014c. ACM. to appear.

Ulrich Schöpp. Experimental compiler from INT to LLVM. `http://www.github.com/uelis/intc`, 2014d.

Peter Selinger. A survey of graphical languages for monoidal categories. In *New Structures for Physics*, volume 813 of *Lecture Notes in Physics*, pages 289–355. Springer, 2011.

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

Philip Wadler. Theorems for free! In *Functional programming languages and computer architecture, FPCA 1989*, pages 347–359, 1989.

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

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of ssa-based optimizations for LLVM. In Hans-Juergen Boehm and Cormac Flanagan, editors, *Programming Language Design and Implementation, PLDI 2013*, pages 175–186. ACM, 2013.

Lukasz Ziarek, Stephen Weeks, and Suresh Jagannathan. Flattening tuples in an SSA intermediate representation. *Higher-Order and Symbolic Computation*, 21(3):333–358, 2008.