Organising Low-Level Programs using Higher Types

Ulrich Schöpp
LMU Munich
Ulrich.Schoepp@ifi.lmu.de

Abstract

Type systems that allow control over low-level compilation details have been developed in the context of resource aware compilation, e.g. for circuit synthesis or for programming with logarithmic space. It was recently observed that some compilation techniques developed in this context, while motivated by capturing certain resource usage restrictions, are closely related to standard compilation techniques, such as CPS translation and defunctionalization. Previous results of this kind suggest to investigate type systems for resource aware compilation more generally with regard to their applicability to structuring general low-level languages, e.g. as used in compilers.

In this paper we study how ideas developed for the LOGSPACE type system INTML can be used for organising the programs in a first-order low-level language. We do so using a type system that simplifies and extends INTML, in particular with parametric polymorphism. The type system is simple but flexible, allowing the definition of combinators, such as for tail recursion, recursion, mutable variables or coroutines, in the language itself. We show that it supports separate compilation and consider its performance using an experimental implementation. For reasoning about the low-level programs represented in the type system, we develop an equational theory based on relational parametricity.

1. Introduction

Low-level programming languages are widely used, especially in compiler construction. Many compilers use a first-order intermediate language, typically in SSA form, as the input language to their backend. For implementing compilers and for developing techniques for reasoning about compiled code, it is useful to develop independent low-level languages that may serve as the target for new compiler front ends. Examples of such low-level languages are `C--` [12] or the language developed in the LLVM project [15].

The mathematical structure of a number of first-order low-level languages is being investigated in work on resource aware compilation for higher-order languages, e.g. for hardware synthesis [8] or for programming in logarithmic space [4]. In this context it was noticed that interactive models of computation in the style of game semantics [1, 11] are useful for organising low-level programs. The idea is to construct an interactive computation model from the programs in a low-level language and then use the structure of the model to build low-level programs and to reason about them. The structure of the models has guided the design of higher-order typed languages that can be compiled to low-level languages with strong resource constraints. For example, for hardware synthesis, one must compile to programs that cannot allocate any space and that do not have access to garbage collection.

While guided by interactive models of computation, it was recently noted [25] that some of this work on resource aware compilation method is connected to standard compilation techniques, such as CPS-translation and defunctionalization.

This suggests to think about the approaches to structuring low-level languages from resource aware compilation in a general context without resource constraints. For example, to enforce space bounds as in [4, 8], it is important to control the amount of stack space used by compiled programs. One may ask if such information is not useful in general, e.g. for optimisation, even without resource restrictions. The importance of supporting garbage collection in `C--` is emphasised in [12], and information about the stack is essential for keeping track of GC roots.

The starting point for this paper is one such approach, the LOGSPACE type system INTML [4, 24]. We study how ideas from INTML can be used in the context of general compilation. The INTML type system can be seen as an approach to organising the programs of a given first-order low-level language. The types specify the interfaces of fragments of low-level programs and the type system governs how these fragments may be assembled. For example, there is a function type $A \cdot X \multimap Y$ that represents low-level programs that expect to be linked with a program implementing interface X. Once they are linked, the resulting program then implements interface Y. The annotation A gives information about the stack shape whenever the module of type X is called.

Questions of how to organise fragments of low-level programs in such a way appear naturally in the compilation of higher programming languages. The translation of a program is often defined compositionally by induction on the structure of the program, so one must think about how to assemble the fragments into a whole program. A flexible language for organising the fragments should be useful for this task.

Perhaps more importantly, for proving properties of the compilation process one needs to develop methods for reasoning about low-level code fragments. This includes specifying code interfaces and the behaviour of code fragments. It seems that a calculus that explains administrative details such as interfaces, linking, inlining and the like should make such tasks easier.

In this paper study how the ideas from INTML can be applied to address such issues in the organisation of general low-level code without LOGSPACE constraints. We define a type system INT, which is closely based on INTML, and study it as a general approach of using higher-order types to organise low-level programs. The main new feature of INT over INTML is the addition of parametric polymorphism. The main technical contribution of this paper is the de-

PPDP'14, September 8-10, 2014, Canterbury, UK.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2947-7/14/09…$15.00.
http://dx.doi.org/10.1145/2643135.2643151

velopment of an equational theory for INT based on relational parametricity. The definition of INT makes further minor improvements over INTML. It adds a type $A \to X$ that is useful for working with tail calls in compilation. The presentation of the type system has been simplified: While INTML has two classes of terms, only one is needed in INT. This simplification brings INT closer to effect calculi [5, 16].

The particular new features of INT were motivated by an ongoing investigation of closure conversion with a subsystem of INT as the target language, see [26]. This work goes toward assessing how useful a higher-order typed language such as INT is for encoding higher languages and for proving the encoding correct.

The view that the structure identified by INT may be useful for reasoning about low-level programs is also supported by a connection to game semantics. The terms in INT can be understood as implementing a basic game semantics (INT is a syntactic presentation of what one obtains by applying the Int construction [13] to a syntactic model of a low-level language, see [4]). The many existing results on game semantics, such as [1, 11] show that it is possible to abstractly specify the behaviour of unknown program fragments arising from the compilation of expressive languages, such as PCF.

Having noted the connection to game semantics, one may see the work in this paper as complementing existing work on using implementing game semantic models, such as [6, 17].

2. Low Level Language

The aim is to structure first-order computation at an abstraction level that corresponds to SSA-form intermediate languages used in many compilers. In this section we fix the syntax of the low-level language. It is similar to an intermediate language used in MLton [2] and is not far away from SSA-form intermediate languages, such as that of LLVM. The language may also be considered as a fragment of Levy's Jump-with-Argument [16] where continuations are not first-class values.

Value Types The low-level language is typed and works with values of the following first-order value types.

$$\text{Value types } A, B ::= \alpha \mid \mathsf{int} \mid \mathsf{unit} \mid A \times B \mid 0 \mid A + B \mid \mu\alpha.\, A$$
$$\text{Values } v, w ::= x \mid n \mid \langle\rangle \mid \langle v, w\rangle \mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \mid \mathsf{fold}(v)$$

Programs A low-level program consists of a set of *blocks*. Each block defines a unique *label* taken from an infinite set of $\mathcal{L}$ of labels.

A definition for a block with label f has the form $f(x{:}A)\{b\}$ where the body b is a term formed by the following grammar.

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

Here, v ranges over values, g over labels and op over primitive operation constants. The possible primitive operation constants are given by a set of constants $\mathit{Prim}(A, B)$. In this paper, we assume $\mathit{Prim}(\mathsf{int}, \mathsf{unit}) = \{\mathsf{print}\}$, $\mathit{Prim}(\mathsf{int} \times \mathsf{int}, \mathsf{int}) = \{\mathsf{add}, \mathsf{times}, \mathsf{div}\}$, $\mathit{Prim}(\mathsf{int} \times \mathsf{int}, \mathsf{unit} + \mathsf{unit}) = \{\mathsf{eq}, \mathsf{lt}\}$ and that all other *Prim*-sets are empty.

By definition, blocks are in A-normal form. The term $g(v)$ represents a jump with argument. It can only appear in tail position.

A *program* is a triple $p = (b, \mathit{entry}, \mathit{exit})$ where b is a set of blocks, $\mathit{entry} \in \mathcal{L}$ is an entry label and $\mathit{exit} \in \mathcal{L}$ is an exit label. The program must be such that there are no two blocks with the same label and that there is no block defining the exit label. For a program p, we write entry_p and exit_p for its entry and exit labels.

Operational Semantics The operational semantics of a program p can be defined by a simple small-step semantics. We define a relation $b_1 \overset{o}{\to}_p b_2$, where b_1 and b_2 are body terms and o is a sequence of closed values, which indicates the outputs performed by the print-operations.

This relation is defined by the following basic reductions (omitting straightforward cases for mul, div, eq and lt)

$$\begin{aligned} \mathsf{let}\ \langle x, y\rangle = \langle v, w\rangle\ \mathsf{in}\ b &\overset{\varepsilon}{\to}_p b[v/x, w/y] \\ \mathsf{case}\ \mathsf{inl}(v)\ \mathsf{of}\ \mathsf{inl}(x) \to b_1\ ;\ \mathsf{inr}(y) \to b_2 &\overset{\varepsilon}{\to}_p b_1[v/x] \\ \mathsf{case}\ \mathsf{inr}(w)\ \mathsf{of}\ \mathsf{inl}(x) \to b_1\ ;\ \mathsf{inr}(y) \to b_2 &\overset{\varepsilon}{\to}_p b_2[w/y] \\ \mathsf{case}\ \mathsf{fold}(v)\ \mathsf{of}\ \mathsf{fold}(x) \to b &\overset{\varepsilon}{\to}_p b[v/x] \\ \mathsf{let}\ x = \mathsf{print}(v)\ \mathsf{in}\ b &\overset{v}{\to}_p b[\mathsf{unit}/x] \\ \mathsf{let}\ x = \mathsf{add}(\langle v, w\rangle)\ \mathsf{in}\ b &\overset{\varepsilon}{\to}_p b[v + w/x] \\ f(v) \overset{\varepsilon}{\to}_p b[v/x] \text{ if } p \text{ contains } f(x{:}A)\{b\} & \end{aligned}$$

as well as the requirement that $b_1 \xrightarrow{o_1} b_2 \xrightarrow{o_2} b_3$ implies $b_1 \xrightarrow{o_1 o_2} b_3$. We write ε for the empty sequence and $o_1 o_2$ for concatenation.

We write short $p(v) \overset{o}{\to} w$ for $\mathit{entry}_p(v) \overset{o}{\to} \mathit{exit}_p(w)$.

In an actual implementation of the operational semantics, one would keep the values of variables in registers or on a stack, so that substitution need not be implemented as a syntactic operation.

2.1 Typing

The low-level language is typed in a standard way.

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

$$\frac{x\colon A \text{ in } \Sigma}{\Sigma \vdash x\colon A} \qquad \frac{}{\Sigma \vdash n\colon \mathsf{int}} \qquad \frac{}{\Sigma \vdash \langle\rangle\colon \mathsf{unit}}$$

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

$$\frac{\Sigma \vdash v\colon A \qquad \Sigma \vdash w\colon B}{\Sigma \vdash \langle v, w\rangle\colon A \times B} \qquad \frac{\Sigma \vdash v\colon A[\mu\alpha.\, A/\alpha]}{\Sigma \vdash \mathsf{fold}(v)\colon \mu\alpha.\, A}$$

Programs To identify well-typed programs, we first define a judgement $\Sigma \mid \Phi \vdash b$ that identifies well-typed block terms. Therein Φ 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, Φ must contain at most one declaration.

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

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

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

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

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

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

We write $p\colon A \to B$ if p can be typed with a label context that assigns the entry label type $\neg A$ and the exit label type $\neg B$.

3. Notation

Two programs $p\colon A \to B$ and $q\colon B \to C$ can be composed to a program $q \circ p\colon A \to C$ essentially by renaming their labels so that no label appears in both programs, taking the union of the block definitions and by unifying the entry label of q with the exit label of p. All the definitions in this paper are invariant under bijective renaming of block labels.

For any program $p\colon A \to B$ and any type C, we write $p \times id\colon A \times C \to B \times C$ and $id \times p\colon C \times A \to C \times B$ for evident programs that pass on the value of type C unchanged.

We write Mem for a type of binary lists, that is $\mathsf{Mem} = \mu\alpha.\, (\mathsf{unit} + \mathsf{unit}) \times \alpha$. For each closed value type A one can write programs $\mathsf{encode}_A\colon A \to \mathsf{Mem}$ and $\mathsf{decode}_A\colon \mathsf{Mem} \to A$ satisfying $(\mathsf{decode}_A \circ \mathsf{encode}_A)(v) \xrightarrow{\varepsilon} v$ for any closed value $v\colon A$.

Given a value-type-with-hole $C[\cdot]$, that is a value type with zero or more occurrences of $\cdot$, we write $C[A]$ for the type obtained by replacing each $\cdot$ with A. The encoding and decoding programs can be lifted to $C[\mathsf{encode}_A]\colon C[A] \to C[\mathsf{Mem}]$ and $C[\mathsf{decode}_A]\colon C[\mathsf{Mem}] \to C[A]$ by induction on C.

Given any value type A, we write $\overline{A}$ for the type obtained from A by substituting Mem for all free type variables.

4. Types for Interfaces and Stack Analysis

We now define the type system INT for constructing and reasoning about low-level programs. It is closely based on INTML [4, 24]. Its types specify interfaces of low-level program fragments and the terms represent low-level programs that implement them. The aim here is not to define a full higher programming language, but to identify structure that helps with the compositional construction of programs and that allows reasoning about questions such as inlining of blocks. The language should still be a low level language, giving the user explicit control over low level details. Even though the proposed language is higher-order, it does not rely on garbage collection, much like [6]. We shall see that its types provide an analysis of the stack shape for any part of the program.

The type system INT is a typed λ-calculus that extends the value types from the low-level language with interface types that specify interfaces of low-level programs. The types are defined in Figure 1.

The idea is that an interface type X specifies an interface of low-level programs and the INT-terms of type X represent actual low-level programs that implement this interface.

The terms of type TA represent low-level programs that compute a value of type A. Computation is started by jumping to the entry label and the value of type A is returned with a jump to the exit label. We write TA rather than $[A]$ as in INTML to make explicit that programs have effects (here: printing and non-termination).

The type $A \to X$ allows for the parameterisation of programs over values. Terms of this type represent programs that differ from the ones represented by terms of type X only in that they expect to be passed an additional parameter of type A at the entry label.

Terms of type $A \cdot X \multimap Y$ represent programs that depend on some unimplemented module of type X. The low-level programs represented by this type are such that if one links them with a low-level program implementing interface type X, then one obtains a program implementing Y. The type A signifies that when jumping to the linked module of type X, the program passes a callee-save value of type A with the actual argument. This value is returned unchanged with any jump back from X. It is much like an annotation on stack usage; when calling the module of type X, a value of type A is put on the stack,[1] where it remains until the call returns.

[1] In an implementation, callee-save values can be held either in registers or in the stack. We use the term stack to include local storage both in machine registers and the actual machine stack.

The type $A \cdot X \multimap Y$ is a form of a linear function space. We emphasise, however, that linearity is not a restriction here. The type system can type as many programs as a simple type $X \to Y$ could. The linear decomposition of the function space does make the types more informative, however. We call A a *subexponential annotation*, the terminology being inspired by [19]. Subexponential annotations tell us the types of the values stored on the stack at any point during program execution.

Finally, the type $\forall\alpha.\, X$ allows for parametric polymorphism of value types.

Formally, the interface represented by an interface type X is defined by two value types X^- and X^+, defined inductively as shown below. A low-level implementation of interface X is then a low-level program of type $X^- \to X^+$.

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

In order to simplify the presentation in the rest of this paper, we make an exception to the above definition, letting $(A \to TB)^- = A$ instead of $A \times \mathsf{unit}$. In particular, a closed program implementing interface $A \to TB$ is just a low-level program with input type A and output type B.

The definition of $(A \cdot X \multimap Y)^-$ and $(A \cdot X \multimap Y)^+$ can be explained as follows: If a program jumps with a value of the form $\mathsf{inl}(\langle a, x\rangle)\colon (A \cdot X \multimap Y)^+$ to the exit label, then this means that it wants to pass the value x and callee-save value a to the entry label of the still unspecified module of type X. It expects the unspecified module to be connected to it such that whenever the unspecified module jumps to its exit label with value x', then this value is passed to the current program by jumping with value $\mathsf{inl}(\langle a, x'\rangle)\colon (A \cdot X \multimap Y)^-$ to the entry label. To link two modules implementing interfaces $A \cdot X \multimap Y$ and X, one essentially just needs to take the definitions from both programs and add blocks that implement the forwarding just described.

The types $A \to X$ and $\forall\alpha.X$ are new compared to previous work [4, 24]. Polymorphism is useful for reasoning about programs. If the interest is only in the construction of programs, then one could use $X[\mathsf{Mem}/\alpha]$ instead of $\forall\alpha.\, X$, but one would lose the parametricity information that values of type α cannot be inspected. The type $A \to X$ is useful in particular for modelling tail calls, e.g. for working with continuation blocks that expect to be passed a value. For this one may use the type $\neg A = A \to \bot$, where $\bot = T0$. With this definition, the INT type system can derive rules similar to those for low-level program blocks.

The typing judgements of INT have the form $\Sigma \mid \Gamma \vdash t\colon X$, where Σ is a value context as in the low-level language and Γ is an *interface context*, which is a list of declarations of the form $x_1\colon A_1 \cdot X_1, \dots, x_n\colon A_n \cdot X_n$. The intention is that t represents a low-level program implementing interface Y and that the variables $x_1, \dots, x_n$ parameterise this program over not yet known programs implementing interfaces $X_1, \dots, X_n$ respectively. The types $A_1, \dots, A_n$ specify the types of callee-save values that are passed along with jumps to the programs represented by the x_i.

The typing rules of INT are given in Figure 2. They make reference to the typing judgement for values $\Sigma \vdash v\colon A$, which is carried over unchanged from the low-level language. If Γ is $x_1\colon A_1 \cdot X_1, \dots, x_n\colon A_n \cdot X_n$, then we write $B \cdot \Gamma$ for the context $x_1\colon (B \times A_1) \cdot X_1, \dots, x_n\colon (B \times A_n) \cdot X_n$.

The contraction rules makes duplication explicit in the term, as this allows one to control sharing of terms explicitly. In rule STRUCT, $A \lhd B$ means that there exists a section-retraction pair between the types $\overline{A}$ and $\overline{B}$. That is, we require there to exist

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

Figure 1. INT Types

Computations

$$\textsc{ret}\ \frac{\Sigma \vdash v\colon A}{\Sigma \mid - \vdash v\colon TA} \qquad \textsc{bind}\ \frac{\Sigma \mid \Gamma \vdash s\colon TA \qquad \Sigma, x\colon A \mid \Delta \vdash t\colon TB}{\Sigma \mid \Gamma, A \cdot \Delta \vdash \mathsf{let}\ x = s\ \mathsf{in}\ t\colon TB} \qquad \textsc{prim}\ \frac{\Sigma \vdash v\colon A \qquad p \in \mathit{Prim}(A, B)}{\Sigma \mid - \vdash p(v)\colon TB}$$

Value Eliminations

$$\times\textsc{e}\ \frac{\Sigma \vdash v\colon A \times B \qquad \Sigma, x\colon A, y\colon B \mid \Gamma \vdash t\colon X}{\Sigma \mid \Gamma \vdash \mathsf{let}\ \langle x, y\rangle = v\ \mathsf{in}\ t\colon X} \qquad +\textsc{e}\ \frac{\Sigma \vdash v\colon A + B \qquad \Sigma, x\colon A \mid \Gamma \vdash t_1\colon X \qquad \Sigma, y\colon B \mid \Gamma \vdash t_2\colon X}{\Sigma \mid \Gamma \vdash \mathsf{case}\ v\ \mathsf{of}\ \mathsf{inl}(x) \to t_1\, ;\ \mathsf{inr}(y) \to t_2\colon X}$$

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

Structural and Logical Rules

$$\textsc{ax}\ \frac{}{\Sigma \mid x\colon \mathsf{unit} \cdot X \vdash x\colon X} \qquad \textsc{weak}\ \frac{\Sigma \mid \Gamma \vdash t\colon X}{\Sigma \mid \Gamma, \Delta \vdash t\colon X} \qquad \textsc{exch}\ \frac{\Sigma \mid \Gamma, \Delta \vdash t\colon X}{\Sigma \mid \Delta, \Gamma \vdash t\colon X} \qquad \textsc{direct}\ \frac{\Sigma \mid \Gamma \vdash t\colon X^- \to TX^+}{\Sigma \mid \Gamma \vdash \mathsf{direct}_X(t)\colon X}\ (*)$$

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

$$\to\textsc{i}\ \frac{\Sigma, x\colon A \mid \Gamma \vdash t\colon X}{\Sigma \mid A \cdot \Gamma \vdash \mathsf{fn}\, x{:}A.\, t\colon A \to X} \qquad \multimap\textsc{i}\ \frac{\Sigma \mid \Gamma,\, x\colon A \cdot X \vdash t\colon Y}{\Sigma \mid \Gamma \vdash \lambda x{:}A{\cdot}X.\, t\colon A \cdot X \multimap Y} \qquad \forall\textsc{i}\ \frac{\Sigma \mid \Gamma \vdash t\colon X}{\Sigma \mid \Gamma \vdash \Lambda\alpha.\, t\colon \forall\alpha.\, X}\ \alpha \text{ not free in } \Sigma, \Gamma$$

$$\to\textsc{e}\ \frac{\Sigma \mid \Gamma \vdash t\colon A \to X \qquad \Sigma \vdash v\colon A}{\Sigma \mid \Gamma \vdash t(v)\colon X} \qquad \multimap\textsc{e}\ \frac{\Sigma \mid \Gamma \vdash s\colon A \cdot X \multimap Y \qquad \Sigma \mid \Delta \vdash t\colon X}{\Sigma \mid \Gamma, A \cdot \Delta \vdash s\, t\colon Y} \qquad \forall\textsc{e}\ \frac{\Sigma \mid \Gamma \vdash t\colon \forall\alpha.\, X}{\Sigma \mid \Gamma \vdash t\, A\colon X[A/\alpha]}$$

Figure 2. Typing Rules for INT

programs $s\colon \overline{A} \to \overline{B}$ and $r\colon \overline{B} \to \overline{A}$, such that for any value v of type A we have $s(v) \xrightarrow{\varepsilon} w$ and $r(w) \xrightarrow{\varepsilon} v$ for some value w.

The semantic definition of $\lhd$ requires just what is needed for rule STRUCT to be sound. For the rest of this paper, it would be sufficient to take a simpler syntactic approximation for $\lhd$. The following results remain true if one takes $\lhd$ to be the smallest relation that satisfies $C[A \times \mathsf{unit}] \lhd C[A]$, $C[\mathsf{unit} \times A] \lhd C[A]$, $C[A] \lhd C[A + B]$, $C[B] \lhd C[A + B]$ and $C[A[\mu\alpha.\, A/\alpha]] \lhd C[\mu\alpha.\, A]$ for all A, B and $C[\cdot]$. This relation is easily decidable (note that we do not even need transitivity).

Rule DIRECT represents an extension mechanism allowing one to define new combinators in the language itself. It has a side condition $(*)$ that will be defined in Section 9.

It is possible to define an operational semantics for INT, along the lines of the definitions in [4]. In this paper we define an equational theory, which can be used to justify such a definition. The $\beta\eta$-equations for the types $\to$, $\multimap$ and $\forall$ appear in Lemma 15.

5. Translation

INT programs are translated to low-level programs by a straightforward compositional translation. One may see each INT term as a direct representation of a low-level program. Questions that would normally require work in the translation, such as how to manage the stack, have been moved to the type system, where we must now decide when to apply rule STRUCT, for example.

The translation is based on the translation of INTML to first-order programs. The reader may wish to consult [4, 24] for a more detailed description of the translation for INTML.

The translation is also related to approaches such as Context Semantics [9] or the Geometry of Interaction [17], but also to compilation techniques such as CPS-translation and defunctionalization, see [25].

To define the translation, we extend $(-)^-$ and $(-)^+$ to interface contexts by defining $(-)^- = (-)^+ = 0$ and $(\Gamma, x\colon A \cdot X)^- = \Gamma^- + (A \times X^-)$ and $(\Gamma, x\colon A \cdot X)^+ = \Gamma^+ + (A \times X^+)$. We identify the value context $x_1\colon A_1, \dots, x_n\colon A_n$ with the type $((\dots(\mathsf{unit} \times A_1) \times \dots) \times A_{n-1}) \times A_n$.

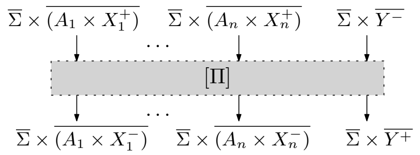
A derivation Π of $\Sigma \mid \Gamma \vdash t\colon Y$ is translated to a program

$$[\![\Pi]\!]\colon \overline{\Sigma} \times (\overline{\Gamma^+} + \overline{Y^-}) \to \overline{\Gamma^-} + \overline{Y^+}\ .$$

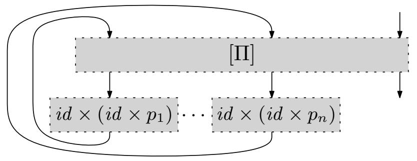
The meaning of the interface of this program is best explained by looking at its definition. To define $[\![\Pi]\!]$, we construct a control flow graph $[\Pi]$ with specified lists of entry points and exit points. Formally, a control flow graph is a directed graph whose nodes are low-level program blocks. There is an edge from $f(x{:}A)\{b\}$ to $g(y{:}B)\{b'\}$ if b contains a subterm of the form $g(v)$. We label this edge with the type B of v. Both entry points and exit points for the control flow graph are just labels. An entry point into such a graph is just the label of a block defined in some node. An exit point is a label that not defined in any node, but that may be jumped to.

In a graph, we depict entry points by edges without a source node and exit points as edges without a target node.

If Π is a derivation of $\Sigma \mid \Gamma \vdash t \colon Y$ and Γ has the form $x_1 \colon A_1 \cdot X_1, \dots, x_n \colon A_n \cdot X_n$, then the entry and exit points of $[\Pi]$ can be depicted as follows. They are labelled with their respective argument type.


$\overline{\Sigma} \times \overline{(A_1 \times X_1^+)}$
$\overline{\Sigma} \times \overline{(A_n \times X_n^+)}$
$\overline{\Sigma} \times \overline{Y^-}$
$[\Pi]$
$\overline{\Sigma} \times \overline{(A_1 \times X_1^-)}$
$\overline{\Sigma} \times \overline{(A_n \times X_n^-)}$
$\overline{\Sigma} \times \overline{Y^+}$

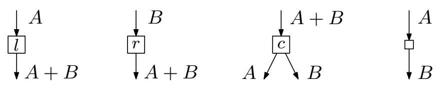
This interface formalises the parameterisation of t over unimplemented modules of types $X_1, \dots, X_n$. The intention of the interface of $[\Pi]$ is that eventually low-level programs $p_1, \dots, p_n$ implementing the interfaces $X_1, \dots, X_n$ will be connected to $[\Pi]$, as shown below, and that the resulting program then implements interface Y.



The program $[\![\Pi]\!]$ differs from $[\Pi]$ only in that the $n+1$ entry and exit labels are packed into a single label repectively. It is obtained from $[\Pi]$ by adding a new entry block and $n+1$ exit blocks. The entry block the performs case distinction over the $n+1$ summands of the input an jumps to the corresponding entry label in $[\Pi]$. Similarly, the exit blocks perform an injection and jump to a single freshly chosen exit label.

The definition of $[\Pi]$ proceeds by induction on the derivation Π. We show representative cases of the definition of the control flow graph of $[\Pi]$ in Figure 3. In the graphs in this figure, the incoming and outgoing edges are ordered as from left to right as shown above. In the figures, white boxes represent graph nodes, i.e. program blocks, and gray dotted boxes stand for sub-graphs whose entry and exit points are connected as shown. For the nodes and subgraphs in the figure we use the following notation:

- Nodes labelled with l, r and c stand for program blocks for the two injections into a coproduct and for case distinction.



  Concretely, c stands for a low-level program block of the form $c(x{:}A + B)\{\mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}(y) \to l(y)\,;\, \mathsf{inr}(z) \to r(z)\}$, where l and r are the labels of the blocks to which the two outgoing wires connect. Nodes labelled with l and r represent the blocks $l(x{:}A)\{o(\mathsf{inl}(x))\}$ and $r(x{:}A)\{o(\mathsf{inr}(x))\}$ respectively, where o is the respective label of the block to which the outgoing wire connects. Small unlabelled boxes are used for coherent isomorphisms arising from the laws for associativity, symmetry and the units for $\times$ and $+$, as well as the distributivity law $A \times (B + C) \cong A \times B + A \times C$.

- For any value $\Sigma \vdash v \colon A$, we write $[v]$ for the evident program of type $\overline{\Sigma} \to \overline{\Sigma} \times \overline{A}$.

- If p is a control flow graph with entry points of types $\Sigma \times A_1, \dots, \Sigma \times A_n, \Sigma \times A$ and exit points of type $\Sigma \times B_1, \dots, \Sigma \times B_m, \Sigma \times B$, then we write $C \cdot p$ for a graph with entry points of types $(\Sigma \times C) \times A_1, \dots, (\Sigma \times C) \times A_n, \Sigma \times (C \times A)$ and exit points of types $(\Sigma \times C) \times B_1, \dots, (\Sigma \times C) \times B_m, \Sigma \times (C \times B)$, which just passes on unchanged the value of type C and otherwise behaves like p. It can be defined by adding a new parameter of type C to each block and passing it on unchanged in any jump.

- For better readability we do not write the operation $\overline{(-)}$ in the figure. It is assumed to be applied to *all* value types in the figure.

The translation is given in Figure 3. The case for $\multimap$E implements the linking $[\Pi_s]$ with $[\Pi_t]$ as described in the explanation of $(A \cdot X \multimap Y)^-$ and $(A \cdot X \multimap Y)^+$ above. The program for rule $\multimap$I constructs a program implementing $A \cdot X \multimap Y$. In the premise of $\multimap$I there is a variable x. The control flow graph $[\Pi_t]$ has an exit point for jumping to the (as yet connected) program denoted by this variable and an entry point to which this program is meant to return. The translation of rule $\multimap$I is such that if one links a program to the result using $\multimap$E, then this program is connected to the entry and exit points for x.

The translation of rules $\forall$I and $\forall$E is based on ideas from [23]. The interpretation of $\forall$I is the identity, as Mem has already been substituted for all variables. The translation of rule $\forall$E is such that any value of the substituted type A is encoded into a value of type Mem before it is passed to $[\Pi_s]$ and decoded when it is received from there.

The interpretation of DIRECT is the identity (note $(X^- \to TX^+)^- = X^-$ and $(X^- \to TX^+)^+ = X^+$). In $\mathsf{direct}(t)$, the term t explicitly defines the low-level program that $\mathsf{direct}(t)$ should translate to. Examples are given in Section 7.

In rule STRUCT, s and r are a section retraction pair witnessing $A \lhd B$. We shall see in Section 9 that its choice does not matter.

5.1 Simplification and Optimisation

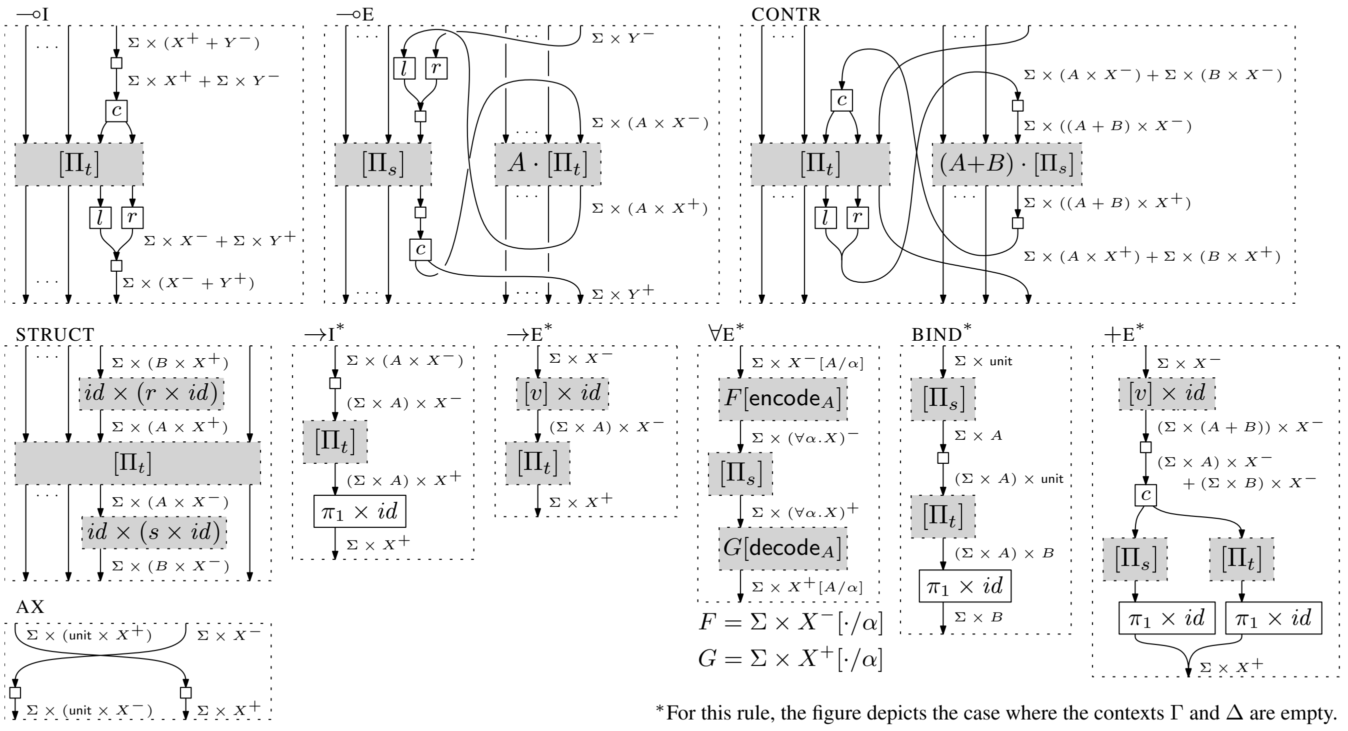
The low-level programs generated by the translation typically contain many unnecessary operations, arising from coherent isomorphisms, for example. However, such inefficiencies can be removed by optimisations on the low-level language.

There is a large amount of existing work on the optimisation of first-order compiler intermediate languages, that we may build on. In this paper we show that a very simple simplification procedure, which has been used in the experimental evaluation in Section 10, can be quite effective already.

First we apply constant propagation whenever possible, that is, we always assume that the low-level programs are reduced with respect to the following reduction rules on block terms.

$$\begin{array}{rcl}
\mathsf{let}\ \langle x, y\rangle = \langle v, w\rangle\ \mathsf{in}\ b & \Rightarrow & b[v/x, w/y] \\
\mathsf{case}\ \mathsf{inl}(v)\ \mathsf{of}\ \mathsf{inl}(x) \to b_1\,;\, \mathsf{inr}(y) \to b_2 & \Rightarrow & b_1[v/x] \\
\mathsf{case}\ \mathsf{inr}(w)\ \mathsf{of}\ \mathsf{inl}(x) \to b_1\,;\, \mathsf{inr}(y) \to b_2 & \Rightarrow & b_2[w/y] \\
\mathsf{case}\ \mathsf{fold}(v)\ \mathsf{of}\ \mathsf{fold}(x) \to b & \Rightarrow & b[v/x]
\end{array}$$

Second, we join the many small blocks generated by the translation, which may create new redexes for the above reduction rules. The blocks generated by the above translation are such that case distinction appears only at the end of blocks, i.e. each block consists of a series of let-definitions and ends with a direct jump $f(v)$ or a conditional jump $\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}(x) \to g_1(v_1)\,;\, \mathsf{inr}(y) \to g_2(v_2)$. For a block that ends in a direct jump, we rewrite with $f(v) \Rightarrow b[v/x]$ if the program contains a definition $f(x{:}A)\{b\}$. If the resulting block again ends in a direct jump, then we repeat this. To avoid infinite unrolling of loops, we do not apply this reduction further if a function symbol f has appeared more than once (or a fixed number of times if some unrolling is desired). For blocks with indirect jumps, we apply the same routine to both branches, but with the restriction that at the end both branches must be of the form $g(w)$ for some g.



⊸I
$\Sigma \times (X^+ + Y^-)$
$\Sigma \times X^+ + \Sigma \times Y^-$
c
$[\Pi_t]$
l
r
$\Sigma \times X^- + \Sigma \times Y^+$
$\Sigma \times (X^- + Y^+)$
⊸E
$\Sigma \times Y^-$
$[\Pi_s]$
$A \cdot [\Pi_t]$
$\Sigma \times (A \times X^-)$
$\Sigma \times (A \times X^+)$
$\Sigma \times Y^+$
CONTR
$\Sigma \times (A \times X^-) + \Sigma \times (B \times X^-)$
$\Sigma \times ((A+B) \times X^-)$
$(A{+}B) \cdot [\Pi_s]$
$\Sigma \times ((A+B) \times X^+)$
$\Sigma \times (A \times X^+) + \Sigma \times (B \times X^+)$
STRUCT
$\Sigma \times (B \times X^+)$
$id \times (r \times id)$
$\Sigma \times (A \times X^+)$
$\Sigma \times (A \times X^-)$
$id \times (s \times id)$
$\Sigma \times (B \times X^-)$
→I*
$\Sigma \times (A \times X^-)$
$(\Sigma \times A) \times X^-$
$(\Sigma \times A) \times X^+$
$\pi_1 \times id$
$\Sigma \times X^+$
→E*
$\Sigma \times X^-$
$[v] \times id$
$(\Sigma \times A) \times X^-$
$\Sigma \times X^+$
∀E*
$\Sigma \times X^-[A/\alpha]$
$F[\mathsf{encode}_A]$
$\Sigma \times (\forall\alpha.X)^-$
$\Sigma \times (\forall\alpha.X)^+$
$G[\mathsf{decode}_A]$
$\Sigma \times X^+[A/\alpha]$
$F = \Sigma \times X^-[\cdot/\alpha]$
$G = \Sigma \times X^+[\cdot/\alpha]$
BIND*
$\Sigma \times \mathsf{unit}$
$\Sigma \times A$
$(\Sigma \times A) \times \mathsf{unit}$
$(\Sigma \times A) \times B$
$\Sigma \times B$
+E*
$(\Sigma \times (A+B)) \times X^-$
$(\Sigma \times A) \times X^- + (\Sigma \times B) \times X^-$
AX
$\Sigma \times (\mathsf{unit} \times X^+)$
$\Sigma \times X^-$
$\Sigma \times (\mathsf{unit} \times X^-)$
$\Sigma \times X^+$

*For this rule, the figure depicts the case where the contexts Γ and Δ are empty.

Figure 3. Translation of Selected Rules

6. Inference of Subexponential Annotations

Subexponential annotations contain useful information about the stack usage of programs as well as their interface, but from the typing rules it is not immediately clear that subexponential annotations can be found for any term that would be typeable without them. It is not hard to see that subexponentials are no restriction on typeability here, however. One trivial choice is to use Mem for all subexponentials, i.e. to use only functions of type $\mathsf{Mem} \cdot X \multimap Y$ and interface contexts of the form $x_1\colon \mathsf{Mem} \cdot X_1, \dots, x_n\colon \mathsf{Mem} \cdot X_n$. By using rule STRUCT with $\mathsf{unit} \lhd \mathsf{Mem}$, $(\mathsf{Mem} \times \mathsf{Mem}) \lhd \mathsf{Mem}$ and $(\mathsf{Mem} + \mathsf{Mem}) \lhd \mathsf{Mem}$, it is possible to maintain this special form of subexponentials in any derivation. The following example shows how to bring the conclusion of an instance of the application rule back into this form by using STRUCT twice.

$$\dfrac{\dfrac{\dfrac{\Sigma \mid \Gamma \vdash s\colon \mathsf{Mem} \cdot X \multimap Y \qquad \Sigma \mid x\colon \mathsf{Mem} \cdot X,\, y\colon \mathsf{Mem} \cdot Y \vdash t\colon X}{\Sigma \mid \Gamma, x\colon (\mathsf{Mem} \times \mathsf{Mem}) \cdot X,\, y\colon (\mathsf{Mem} \times \mathsf{Mem}) \cdot Y \vdash s\,t\colon Y}}{\Sigma \mid \Gamma, x\colon \mathsf{Mem} \cdot X,\, y\colon (\mathsf{Mem} \times \mathsf{Mem}) \cdot Y \vdash s\,t\colon Y}}{\Sigma \mid \Gamma, x\colon \mathsf{Mem} \cdot X,\, y\colon \mathsf{Mem} \cdot Y \vdash s\,t\colon Y}$$

The low-level programs obtained from derivations maintaining this invariant on subexponentials are close to the implementation method described by Mackie [17].

Unfortunately, this simple approach yields low-level programs that use the pairing and unpairing functions arising from the use of STRUCT with $(\mathsf{Mem} \times \mathsf{Mem}) \lhd \mathsf{Mem}$ in many places, which makes optimisation of the low-level programs harder. One would like to avoid inserting such encoding functions and ideally avoid using recursive types at all.

Let us outline a more structured way of obtaining subexponential annotations, which comes from [3, 25]. We outline the method without the rules for polymorphism and rule DIRECT at first. The idea is to maintain a similar invariant as with Mem above, but to use fresh type variables instead of Mem and to collect the $\lhd$-constraints that arise from the typing rules. For the above example we then have the following derivation, which is subject to the constraints $(\alpha_1 + \alpha_2) \lhd \alpha_4$ and $(\alpha_1 + \alpha_3) \lhd \alpha_5$ in the two uses of STRUCT.

$$\dfrac{\dfrac{\dfrac{\Sigma \mid \Gamma \vdash s\colon \alpha_1 \cdot X \multimap Y \qquad \Sigma \mid x\colon \alpha_2 \cdot X,\, y\colon \alpha_3 \cdot Y \vdash t\colon X}{\Sigma \mid \Gamma, x\colon (\alpha_1 \times \alpha_2) \cdot X,\, y\colon (\alpha_1 \times \alpha_3) \cdot Y \vdash s\,t\colon Y}}{\Sigma \mid \Gamma, x\colon \alpha_4 \cdot X,\, y\colon (\alpha_1 \times \alpha_3) \cdot Y \vdash s\,t\colon Y}}{\Sigma \mid \Gamma, x\colon \alpha_4 \cdot X,\, y\colon \alpha_5 \cdot Y \vdash s\,t\colon Y}$$

The constraints that one obtains from any derivation in this way all have the form $B \lhd \beta$, i.e. with a variable as upper bound. Such constraints are easy to solve [25]: Suppose γ is a variable and $B_1 \lhd \gamma, \dots, B_n \lhd \gamma$ are all constraints with upper bound γ. If γ does not occur in any of the B_i, then $B_1 + \dots + B_n$ can be used as a solution for γ, otherwise $\mu\gamma.B_1 + \dots + B_n$ is a solution, and the variable γ has thus been eliminated. In this way, the constraints can be solved variable by variable.

Since in rule DIRECT one may choose the type X arbitrarily, its conclusion may not be such that all subexponential annotations are type variables. If all the subexponential annotations in positive position in X are type variables (e.g. $A \cdot (\alpha \cdot Y \multimap Y) \multimap Y$, but not $A \cdot (\mathsf{unit} \cdot Y \multimap Y) \multimap Y$), then an η-expansion of the term may be annotated with fresh type variables for subexponential annotations, and subexponential inference may be used as above. The restriction to allow only type variables as subexponential annotations in positive positions in X is not severe. It means that one is not allowed to make any assumption about the stack usage of the functions that are passed as arguments.

This outlines one way of inferring subexponential annotations, which also shows that subexponential annotations are not unique. Since these annotations control the organisation of the stack, it may be the case that one may want to solve the constraints for them according to different criteria, e.g. to minimise re-encoding operations, to minimise stack size etc. For the experiments in Section 10, the above simple inference procedure has been used.

7. Examples

We give a few simple examples to show the intended use of INT, e.g. as a compiler intermediate language. The aim is to explain INT and to illustrate its use for the construction low-level programs. Except for `corout`, similar combinators have appeared in [4]. Here we focus on their application to compilation. For the combinator `tailrec` we show that its use yields low-level programs that are close to ones one would write by hand.

To typeset the example INT programs, we use a concrete syntax where $\lambda x{:}A{\cdot}X.\, t$ is written as `fun (x: A.X) -> t` or just `fun x -> t` when the type is clear from the context. Likewise, $\mathsf{fn}\, x{:}A.\, t$ is written as `fn(x: A){ t }` or just `fn(x){ t }`. The programs in this section are accepted by an experimental implementation and types are inferred automatically.

7.1 Blocks

As a first sanity check we observe that one does not lose anything by moving to INT and that low-level programs can be written in the style of the given low-level language.

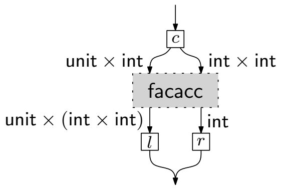
Closed terms of the form $\mathsf{fn}\, x{:}A.\, b$, where b is a series of let-definitions, amount to a block definition in the low-level language. We give an example to illustrate how low-level programs can be defined in INT. Consider the following term `facacc`.

`// facacc:` $\mathsf{unit}\cdot(\mathsf{int}\times\mathsf{int}\to T\mathsf{int})\multimap(\mathsf{int}\times\mathsf{int}\to T\mathsf{int})$

```
facacc =
  fun (f : unit . (int * int -> T int)) ->
    fn(x: int * int) {
      let (n, acc) = x in
      let b = eq(n, 1) in
      case b of
        inl() -> n
      | inr() ->
           let n1 = add(n, -1) in
           let acc1 = mul(acc, n) in
           f(n1, acc1)
    }
```

If we write X for the type of `facacc`, then by definition we have $X^- = (\mathsf{unit}\times\mathsf{int})+(\mathsf{int}\times\mathsf{int})$ and $X^+ = (\mathsf{unit}\times(\mathsf{int}\times\mathsf{int}))+\mathsf{int}$.

The translation of `facacc` in the empty context is then a low-level program of type $\mathsf{unit}\times(0+X^-)\to(0+X^+)$. If one simplifies this to $X^-\to X^+$ and applies optimisations described above, then one obtains the following low-level program.



$c(x\colon (\mathsf{unit}\times\mathsf{int})+(\mathsf{int}\times\mathsf{int}))\{$
$\quad\mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}(y)\Rightarrow entry_2(y)$
$\qquad\qquad ;\ \mathsf{inr}(z)\Rightarrow entry_1(z)$
$\}$
$l(x{:}\mathsf{int})\{exit(\mathsf{inr}(x))\}$
$r(x{:}\mathsf{unit}\times(\mathsf{int}\times\mathsf{int}))\{exit(\mathsf{inl}(x))\}$

The (gray) body of the program is given by the following blocks:

$entry_1(x\colon \mathsf{int}\times\mathsf{int})\{$
$\quad\mathsf{let}\ \langle n, acc\rangle = x\ \mathsf{in}$
$\quad\mathsf{let}\ b = \mathsf{eq}(\langle n, 1\rangle)\ \mathsf{in}$
$\quad\mathsf{case}\ b\ \mathsf{of}\ \mathsf{inl}(\langle\rangle)\Rightarrow exit_1(n)$
$\qquad\qquad ;\ \mathsf{inr}(\langle\rangle)\Rightarrow h(\langle n, acc\rangle)$
$\}$

$h(x\colon \mathsf{int}\times\mathsf{int})\{$
$\quad\mathsf{let}\ \langle n, acc\rangle = x\ \mathsf{in}$
$\quad\mathsf{let}\ n_1 = \mathsf{add}(\langle n, -1\rangle)\ \mathsf{in}$
$\quad\mathsf{let}\ acc_1 = \mathsf{mul}(\langle acc, n\rangle)\ \mathsf{in}$
$\quad exit_2(\langle\langle\rangle, \langle n_1, acc_1\rangle\rangle)$
$\}$

$entry_2(x{:}\mathsf{unit}\times\mathsf{int})\{\mathsf{let}\ \langle s, x'\rangle = x\ \mathsf{in}\ exit_1(x')\}$

These low-level blocks are almost identical to the original INT-program. This example shows that low-level blocks can be defined in INT just as in the low-level language and that the translation is essentially the identity.

Using the higher-order structure of INT it is now possible to organise such low-level blocks. For example, we can write a combinator for chaining two blocks:

`// comp:` $\alpha\cdot(\alpha\to T\beta)\multimap(\beta\times\alpha)\cdot(\beta\to T\gamma)\multimap\alpha\to T\gamma$

```
comp = fun f -> fun g ->
          fn(x) { let y = f(x) in let z = g(y) in z }
```

The subexponential annotations in the type of `comp` tell us that whenever the program jumps to `f`, then the stack contains a value of type α (namely `x`) and when the program jumps to `g` then the stack contains a value of type $\beta\times\alpha$ (namely $\langle$`y`, `x`$\rangle$)). The variables inside the fn-Block are thus scoped much like in C. If one wants to compose blocks without keeping the variables on the stack, then one can define a combinator `comp1` of type $\mathsf{unit}\cdot(\alpha\to T\beta)\multimap\mathsf{unit}\cdot(\beta\to T\gamma)\multimap\alpha\to T\gamma$ using direct as follows:

`// cimp:` $\mathsf{unit}\times\beta+(\mathsf{unit}\times\gamma+\alpha)\to T(\mathsf{unit}\times\alpha+(\mathsf{unit}\times\beta+\gamma))$

```
cimp = fn(x) {
  case x of
    Inl((), b) -> Inr(Inl((), Inr(b)))
  | Inr(Inl((), Inr(c))) -> Inr(c)
  | Inr(a) -> Inl((), a)
}
comp1 = direct(cimp)
```

7.2 Tail Recursion

Of course, one needs to be able to connect blocks into loops, so that recursive functions can be implemented. To this end, one can define a combinator for tail recursion.

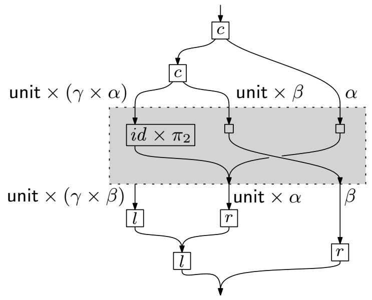
$$\texttt{tailrec}\colon \mathsf{unit}\cdot(\gamma\cdot(\alpha\to T\beta)\multimap\alpha\to T\beta)\multimap\alpha\to T\beta$$

It is defined using a direct definition.

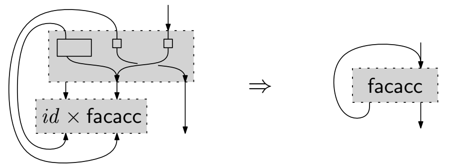
`// trimp:` $\mathsf{unit}\times(\gamma\times\alpha+\beta)+\alpha\to T(\mathsf{unit}\times(\gamma\times\beta+\alpha)+\beta)$

```
trimp = fn(x) {
  case x of
    Inl((), Inl(g, a)) -> Inl((), Inr(a))
  | Inl((), Inr(b)) -> Inr(b)
  | Inr(a) -> Inl((), Inr(a)))
}
tailrec = direct(trimp)
```

The function `trimp` specifies a low-level program that may be depicted as shown below. By definition using direct, `tailrec` is a term that translates to this low-level program.



To understand why we have used this particular definition of `tailrec`, consider the application `tailrec facacc` Its type is $\mathsf{int}\times\mathsf{int}\to T\mathsf{int}$. With the above definition of `tailrec`, it represents a factorial function with accumulator, meaning that (`tailrec facacc`)$(\langle n, 1\rangle)$ returns the factorial of n. Translating the application `tailrec facacc` amounts to connecting the program implementing `facacc` with the above program for tail recursion. If we simplify pairs of c and l and r boxes, then the resulting program may be depicted as on the left in the following figure. Even the simple optimisation described above is sufficient to perform this simplification. In fact the optimisation further untangles this program, giving us the program on the right.



But this is just the low-level program one gets from `facacc` by keeping only the blocks with label $entry_1$ and h and by renaming the occurrence of $exit_2$ in h to $entry_2$. This is a factorial program as one might write by hand.

Mutually tail recursive calls between n blocks can similarly be realised using an n-ary `tailrec`-combinator. This means it is possible to write low-level programs in INT like in the low-level language itself, i.e. one does not lose intensional expressiveness.

7.3 Recursion

The definition of the tail recursion combinator is easily generalised to a full recursion combinator.

$$\texttt{fix}\colon \mathsf{list}\langle\alpha\rangle \cdot (\alpha \cdot X \multimap X) \multimap X$$

Its type shows that if the argument function puts a value of type α on the stack whenever it calls its argument, then the fixed point of this function keeps a list of α-values on the stack whenever it interacts with the step function.

```
fiximp = fn(x) {
  case x of
    Inl(l, Inl(b, q)) -> Inl(Cons(b, l), Inr(q))
  | Inl(Nil, Inr(a)) -> Inr(a)
  | Inl(Cons(hd, tl), Inr(a)) -> Inl(tl, Inl(hd, a)))
  | Inr(q) -> Inl(Nil, Inr(q))
}
fix = direct(fiximp)
```

For example, we can write `fix facacc` to define the same function as `tailrec facacc`, but now with full recursion instead of tail recursion. As another example, a naive implementation of the Fibonacci function can be defined as follows (using some syntactic sugar):

```
fib1 = fix (fun fib ->
  fn(i: int){ if i < 2 then 1 else fib(i-1) + fib(i-2) }
)
```

The combinators `fix` and `tailrec` may also be used to mix recursion and tail recursion easily. For example, a compiler might optimise the tail recursive call `fib(i-2)` into a tail recursion (current C compilers apply such a transformation when compiling a naive implementation of `fib`):

```
fib2 = fix (fun fib ->
  fn(i: int) {
     tailrec (fun tr ->
        fn (i, acc) {
           let acc1 = fib(i-1) + acc in
           if i < 2 then acc else tr (i-2, acc1)
        }) (i, 1)
  })
```

7.4 Direct Stack Access

With the subexponential annotations it is possible to control the stack content. Suppose we would like to translate a language with mutable variables into the low-level language. Since the low-level language does not allow mutation, the only possibility of doing so is to thread the current state of the mutable variable through different low-level variables. A back-end for the low-level language is then free to store the value of the variable in a register or on the stack.

This idea can be made formal in INT by means of a simple combinator that provides a single “mutable variable” of type α.

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

This combinator is much like `newvar` from Idealised Algol [22]. It may be used as `(alloc (fun write -> fun read -> t))(a)`, where `write`: $(\alpha \to T\mathsf{unit})$ and `read`: $T\alpha$ allow writing and reading of the variable and `a` is its initial value.

The combinator `alloc` is implemented as follows:

```
// allocimp:  $\alpha \times (\gamma \times \mathsf{unit} + (\delta \times \mathsf{unit} + \beta)) + \alpha$
//        $\to\ T(\alpha \times (\gamma \times \alpha\ +\ (\delta \times \alpha + \mathsf{unit})) + \beta)$
allocimp = fn(x) {
  case x of
    Inl(mem, Inl(c, newmem)) -> Inl(newmem, Inl(c, ()))
  | Inl(mem, Inr(Inl(d, ()))) -> Inl(mem, Inr(Inl(d, mem)))
  | Inl(mem, Inr(Inr(b)) -> Inr(b)
  | Inr(a) -> Inl(a, Inr(Inr()))
}
alloc = direct(allocimp)
```

For example, the following program uses one ‘mutable’ variable to control the iteration of a loop. It prints the numbers from `12` to `1` in decreasing order.

```
main = fn() {
   alloc (fun write -> fun read ->
     copy write as write1, write2 in
       let () = write1(12) in
       tailrec (fun loop -> fn () {
          let v = read in
          if 0 < v then
            let () = print(v) in
            let () = write2(v - 1) in
            loop ()
          else ()
       }) ()
   ) 0
}
```

Although this program appears to use mutable state, it does not use any allocation. Indeed, the combinator `alloc` works by using the local environment. (In the experimental implementation the loop variable ends up in a register.)

Another example where explicit stack control is important is given by coroutines. A combinator for running two processes as coroutines, with explicit yield functions, can also easily be defined using direct to have the following type

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

where $X_1 = \delta_1 \cdot (\alpha \to T\beta) \multimap T\gamma$ and $X_2 = \delta_2 \cdot (\beta \to T\alpha) \multimap \alpha \to T\gamma$ are the types of the two processes.

The argument of type $\alpha \to T\beta$ in X_1 represents a ‘yield’ function that can be used to send a message of type α to the second process and wait for a value of type β to be returned. The other process gets a symmetric yield function. The process of type X_1 is run first, which means that the other process can only be started if the first process sends a message to it. This initial message corresponds to the additional parameter α in X_2.

Notice the appearance of the types δ_1 and δ_2 in the type of `corout`. They make explicit that the stack content and instruction pointer of the waiting process is preserved while the other process is run. The type of `corout` shows that there is no stack overflow from the use of coroutines as a form of iteration.

An example application for `corout` is:

```
// proc1: $(\delta_1 \cdot (\mathsf{int} \to T\mathsf{int}) \multimap T\mathsf{unit})$
proc1 = fun yield ->
     tailrec (fun proc1 ->
      fn(j: int) {
         print "p1:"; print j;
         let i = yield(j+2) in
         proc1(i)
       }) 3
```

```
// proc2: (δ₂ · (int → Tint) ⊸ int  → Tunit)
proc2 = fun yield ->
   copy yield as yield1, yield2 in
   tailrec (fun proc2 ->
     fn(j: int) {
       print "p2:"; print j;
       let k = yield1(j-1) in
       let i = yield2(k-1) in
       proc2(i)
     })
main = corout proc1 proc2
```

The main program prints p1:3, p2:5, p1:4, p1:5, p2:7, p1:6, p1:7, p2:9, ...

These examples illustrate how the INT type system allows access to the stack. We believe that making the stack content accessible should also be useful for supporting garbage collection by making it easy to identify the GC roots.

It seems that INT is a flexible, conceptually simple language that allows one to write low-level programs in a structured way. The approach of using combinators rather than a more complex syntax aligns with the intended use of INT as a compiler intermediate language. In Section 10 we give some evidence that the translation of INT to the low-level language produces efficient programs.

8. Separate compilation

The translation from the higher-order language INT to the first-order low-level language is related to defunctionalization [25], so one may be concerned that it can be used only for whole-program compilation. Here we show that separate compilation is simple.

Suppose we have two terms

$$- \mid - \vdash t \colon X, \qquad - \mid x_1 \colon A_1 \cdot X, \dots, x_n \colon A_n \cdot X \vdash s \colon Y.$$

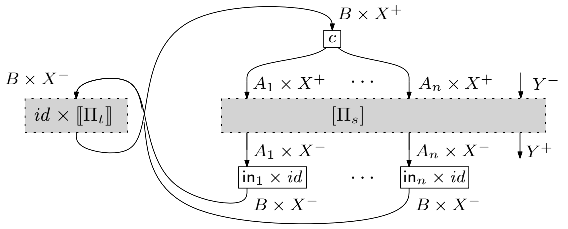
We would like to compile them separately and link t to each variable x_i. To do this, we translate separately the terms $- \mid - \vdash t \colon X$ and

$$- \mid - \vdash \lambda x{:}B{\cdot}X.\underbrace{\mathsf{copy}\ x\ \mathsf{as}\ x_1, \dots, x_n\ \mathsf{in}\ s}_{s'} \colon B \cdot X \multimap Y$$

where $B = A_1 + \cdots + A_n$.

Linking these terms then amounts to forming the application $s'\ t$ from $[\![\Pi_{s'}]\!]$ and $[\![\Pi_t]\!]$ alone. This may be done by a simple linker.

Let us look at the resulting linked program:



Consider what happens when $[\Pi_s]$ sends a message $\langle s, q\rangle \in A_k \times X^-$ to $[\![\Pi_t]\!]$ along the connection with type $A_k \times X^-$. The first component of the pair is injected into $B = A_1 + \cdots + A_n$, i.e. $\langle \mathsf{in}_k(s), q\rangle$ is formed, and the program jumps to $[\![\Pi_t]\!]$. When $[\![\Pi_t]\!]$ returns, the message $\langle \mathsf{in}_k(s), a\rangle \in B \times X^+$ is inspected. The number k identifies the return address of the value; the program sends $\langle s, a\rangle$ along the connection with type $A_k \times X^+$.

This way of implementing calls to t corresponds just to the way calls are realised on the machine level. To make a call, one first pushes the return address on the stack, then one jumps to the target and to return one pops the return address and jumps to it. Here, i plays the role of an abstract address. On a low level, the value $\mathsf{in}_k(s)$ of a sum type $A_1 + \cdots + A_n$ is typically encoded by a pair of a tag $k \in \{1, \dots, n\}$ and the code of a value of type A_k. Replacing $\langle s, q\rangle$ by $\langle \mathsf{in}_k(s), q\rangle$ therefore amounts to pushing k on the stack. Thus, even though we have assumed only direct jumps in the low-level language, the type system and interpretation of INT reconstruct the traditional implementation of procedure calls using jumps and the stack.

In an implementation it would be reasonable to replace the index k by an actual machine address, so as to avoid the case distinction on function return. It is possible to use the machine stack for calls and compile programs to assembly code that can be linked with the standard system linker.

9. Equational Reasoning

So far we have discussed INT as a language for constructing low-level programs. One main motivation for studying INT is also to develop a calculus for reasoning about low-level programs. Ongoing work on intended applications of such a calculus is reported in [26]. We would like to define a notion of equality for INT that is useful for reasoning about low-level programs. The notion of equality for INT should reflect a reasonable notion of equality for low-level programs.

One possible option would be to consider two terms equal when they translate to low-level programs with the same extensional behaviour. This definition would validate β-equality $(\lambda x{:}A{\cdot}X.\, t)\ s = t[s/x]$ when s is closed, but it would not validate other reasonable instances of it. For example, if t does not contain x, then s is dead code and the β-equality would correspond to a reasonable form of dead code elimination. However, if s has free variables, then it may be possible to distinguish the low-level programs interpreting $(\lambda x{:}A{\cdot}X.\, t)\ s$ and $t[s/x]$ by interacting with the dead code that would otherwise never be used. This shows that one should like to take a certain quotient of the generated low-level programs.

In this section we show that INT allows equality reasoning by relational parametricity [21]. This not only justifies full $\beta\eta$-equations for functions, but also allows us to use polymorphism as a reasoning principle [27]. Suppose, for example, we have a closed polymorphic term t of type $\forall\alpha.\, \mathsf{unit} \cdot (\alpha \to X) \multimap \alpha \to Y$, where α is not free in X or Y. In this case, if $x\colon A \vdash v\colon B$, then we should have $(\mathsf{fn}\, x{:}A.\, (t\ B\ s)(v)) = t\ A\ (\mathsf{fn}\, x{:}A.\, s(v))$, for closed s, for example. Such reasoning can be justified using relational parametricity.

We note that on closed types such as $T\mathsf{int}$ the notion of equality will be defined such that equal programs must return the same result and have the same output. This means that if one uses the equations to prove that two closed programs of base type are equal, these programs will indeed have the same behaviour.

9.1 Relational Parametricity

A *value type relation* is given by a triple (A, A', R) of two closed types A and A' and a binary relation R between the closed values of type A and those of type A'. For R we allow only *admissible* relations, which are finite intersections of relations of form $\langle f\rangle\langle g\rangle^{-1} = \{(v, w) \mid f(v) = g(w)\}$, where f and g are total programs $f\colon A \to B$ and $g\colon A' \to B$ for some B. We write $R \subseteq A \times A'$ for the triple (A, A', R).

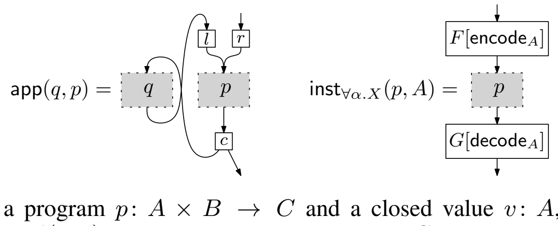
A *type environment* ρ is a mapping from type variables to value type relations. If σ and σ' are both mappings from type variables to closed value types, then we write $\rho \subseteq \sigma \times \sigma'$ if, for any α, $\rho(\alpha)$ is a relation $R_\alpha \subseteq \sigma(\alpha) \times \sigma'(\alpha)$.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

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

as usual:

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

We next define relations on programs, for which we introduce the notation $\mathsf{app}(p,q)$, $\mathsf{val}(p,v)$ and $\mathsf{inst}_{\forall\alpha.X}(p,A)$. For programs $p\colon A+B \to C+D$ and $q\colon C \to A$, we write $\mathsf{app}(p,q)$ for the following program of type $B \to D$.



For a program $p\colon A\times B \to C$ and a closed value $v\colon A$, we write $\mathsf{val}(p,v)$ for the program of type $B \to C$ obtained by pre-composing with the program of type $B \to A\times B$ with the single definition $\mathit{entry}(x{:}B)\{\mathit{exit}(\langle v,x\rangle)\}$. Finally, for a program $p\colon \overline{(\forall\alpha.\,X)^-} \to \overline{(\forall\alpha.\,X)^+}$ and a closed value type A, we write $\mathsf{inst}_{\forall\alpha.X}(p,A)$ for the program $G[\mathsf{decode}_A] \circ p \circ F[\mathsf{encode}_A]$, where $F[\,\cdot\,]$ and $G[\,\cdot\,]$ are the value type contexts $F[\,\cdot\,] = \overline{X^-[\,\cdot\,/\alpha]}$ and $G[\,\cdot\,] = \overline{X^+[\,\cdot\,/\alpha]}$. Notice in particular $F[\mathsf{encode}_A]\colon F[A] \to F[\mathsf{Mem}]$ and $F[\mathsf{Mem}] = \overline{X^-[\mathsf{Mem}/\alpha]} = \overline{(\forall\alpha.\,X)^-}$ by definition.

With this notation we can define when programs are related. For any interface type X and any type environment $\rho \subseteq \sigma\times\sigma'$, we define a relation $[\![X]\!]_\rho$ between programs of type $X^-\sigma \to X^+\sigma$ and $X^-\sigma' \to X^+\sigma'$ by induction on X.

For the base case, we define $p\ [\![TA]\!]_\rho\ p'$ to hold if and only if the following conditions hold.

1. If $\mathit{entry}_p(\langle\rangle) \xrightarrow{o}_p t$ then $\mathit{entry}_{p'}(\langle\rangle) \xrightarrow{o}_{p'} t'$ for some t'.
2. If $\mathit{entry}_p(\langle\rangle) \xrightarrow{o}_p \mathit{exit}_p(v)$ then $\mathit{entry}_{p'}(\langle\rangle) \xrightarrow{o}_{p'} \mathit{exit}_{p'}(v')$ 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:
$$\forall v, v'.\ v\ [\![A]\!]_\rho\ v' \Rightarrow \mathsf{val}(p,v)\ [\![X]\!]_\rho\ \mathsf{val}(p',v')$$
- $p\ [\![A\cdot X \multimap Y]\!]_\rho\ p'$ if and only if:
$$\forall p_1, p_1'.\ p_1\ [\![X]\!]_\rho\ p_1' \Rightarrow$$
$$\mathsf{app}(p, \mathit{id}_{A\sigma} \times p_1)\ [\![Y]\!]_\rho\ \mathsf{app}(p', \mathit{id}_{A\sigma'} \times p_1')$$
- $p\ [\![\forall\alpha.\,X]\!]_\rho\ p'$ if and only if:
$$\forall S \subseteq A{\times}A'.\ \mathsf{inst}_{(\forall\alpha.X)\sigma}(p,A)\ [\![X]\!]_{\rho[\alpha\mapsto S]}\ \mathsf{inst}_{(\forall\alpha.X)\sigma'}(p',A')$$

We first state a few general properties of $[\![-]\!]_\rho$.

For any value type environment ρ, we write ρ^{-1} for the value type environment defined by $\rho^{-1}(\alpha) = \{(v',v) \mid (v,v') \in \rho(\alpha)\}$.

Lemma 1. *If $v\ [\![A]\!]_\rho\ v'$ then $v'\ [\![A]\!]_{(\rho^{-1})}\ v$.*

Proof. The proof goes by induction on the size of v. If A is α, then $[\![A]\!]_\rho = \rho(\alpha)$ and $[\![A]\!]_{(\rho^{-1})} = (\rho^{-1})(\alpha)$, so the assertion follows immediately. In the cases for base types, the assertion is immediate and in all other cases it follows directly from the induction hypothesis. □

Lemma 2. *If $p\ [\![X]\!]_\rho\ q$ then $q\ [\![X]\!]_{(\rho^{-1})}\ p$.*

If σ is a mapping from type variables to closed value types, then we write Δ_σ for the value type environment mapping all type variables to the diagonal $\Delta_\sigma(\alpha) = \{(v,v) \mid v \in \sigma(\alpha)\}$.

Lemma 3. *If $\rho \subseteq \sigma\times\sigma'$ and $v\ [\![A]\!]_{\Delta_\sigma}\ w\ [\![A]\!]_\rho\ v'$ then $v\ [\![A]\!]_\rho\ v'$.*

Proof. The proof goes by induction on the size of v. If A is a type variable or a base type, then $v\ [\![A]\!]_{\Delta_\sigma}\ w$ implies that v and w are identical, so the assertion follows. If A is $\mu\alpha.B$, then v, w and v' must be $\mathsf{fold}(v_1)$, $\mathsf{fold}(w_1)$ and $\mathsf{fold}(v_1')$ respectively with $v_1\ [\![B[\mu\alpha.\,B/\alpha]]\!]_{\Delta_\sigma}\ w_1\ [\![B[\mu\alpha.\,B/\alpha]]\!]_\rho\ v_1'$. The induction hypothesis gives $v_1\ [\![B[\mu\alpha.\,B/\alpha]]\!]_\rho\ v_1'$, from which we get the required $v\ [\![A]\!]_\rho\ v'$. □

Lemma 4. *If $\rho \subseteq \sigma\times\sigma'$ and $p\ [\![X]\!]_{\Delta_\sigma}\ q\ [\![X]\!]_\rho\ q'$ then $p\ [\![X]\!]_\rho\ q'$.*

Proof. The proof goes by induction on X. We show the case for $X = A\cdot Z \multimap Y$. Suppose $r\ [\![Z]\!]_\rho\ r'$. Then we have $r[\![Z]\!]_{\Delta_\sigma}r$, as programs are deterministic (strictly, this follows by a nested induction). By assumption, we get $\mathsf{app}(p, \mathit{id}\times r)\ [\![Y]\!]_{\Delta_\sigma}\ \mathsf{app}(q, \mathit{id}\times r)$ and $\mathsf{app}(q, \mathit{id}\times r)\ [\![Y]\!]_\rho\ \mathsf{app}(q', \mathit{id}\times r')$. The induction hypothesis yields $\mathsf{app}(p, \mathit{id}\times r)\ [\![Y]\!]_\rho\ \mathsf{app}(q', \mathit{id}\times r')$, which means we have shown $p\ [\![A\cdot Z \multimap Y]\!]_\rho\ q'$, as required. □

We extend the relation $[\![-]\!]_\rho$ to contexts in the evident way. If Σ is $x_1\colon A_1, \dots, x_n\colon A_n$, then $\vec{v}\ [\![\Sigma]\!]_\rho\ \vec{w}$ states that $\vec{v}$ and $\vec{w}$ are vectors of length n and $v_i\ [\![A_i]\!]_\rho\ w_i$ holds for all $i = 1,\dots,n$. If Γ is $y_1\colon B_1\cdot Y_1, \dots, y_n\colon B_n\cdot Y_n$, then $\vec{p}\ [\![\Gamma]\!]_\rho\ \vec{q}$ states that $\vec{p}$ and $\vec{q}$ are vectors of length n and $p_i\ [\![Y_i]\!]_\rho\ q_i$ (no typo, compare the definition of $[\![A\cdot X \multimap Y]\!]_\rho$) holds for all $i = 1,\dots,n$.

We also write $\Sigma \to X$ and $\Gamma \multimap X$ for $A_1 \to \cdots \to A_n \to X$ and $B_1\cdot Y_1 \multimap \ldots \multimap B_n\cdot Y_n \multimap X$ respectively.

We similarly extend the operations app, val and inst to multiple arguments. Define $\mathsf{val}(p, v_1, v_2)$ by $\mathsf{val}(\mathsf{val}(p,v_1),v_2)$, and $\mathsf{app}(p,q_1,q_2)$ by $\mathsf{app}(\mathsf{app}(p,q_1),q_2)$, and $\mathsf{inst}_{\forall\alpha_1,\alpha_2.X}(p,A_1,A_2)$ by $\mathsf{inst}_{(\forall\alpha_2.X)[A_1/\alpha_1]}(\mathsf{inst}_{\forall\alpha_1,\alpha_2.X}(p,A_1,A_2),A_2)$ and extend this notation similarly to vectors.

Definition 5. *Suppose X is a type, Σ is a value context, Γ is an interface context, and $\vec{\alpha}$ is a list of free type variables in X, Σ and Γ. Then we write $\Sigma \mid \Gamma \models p = q\colon X$ for two programs p and q if: For any $\rho \subseteq \sigma\times\sigma'$, all $(\vec{v},\vec{w}) \in [\![\Sigma]\!]_\rho$ and all $(\vec{p},\vec{q}) \in [\![\Gamma]\!]_\rho$ we have that $\mathsf{app}(\mathsf{val}(\mathsf{inst}_{(\forall\vec{\alpha}.\Sigma\to\Gamma\multimap X)\sigma}(p,\vec{\alpha}[\sigma]),\vec{v}),\vec{q})$ and $\mathsf{app}(\mathsf{val}(\mathsf{inst}_{(\forall\vec{\alpha}.\Sigma\to\Gamma\multimap X)\sigma'}(q,\vec{\alpha}[\sigma']),\vec{w}),\vec{r})$ are $[\![X]\!]_\rho$-related.*

In words: If one instantiates all type variables with related types, chooses related values for the variables in Σ and plugs in related programs for the variables in Γ, then one obtains related programs.

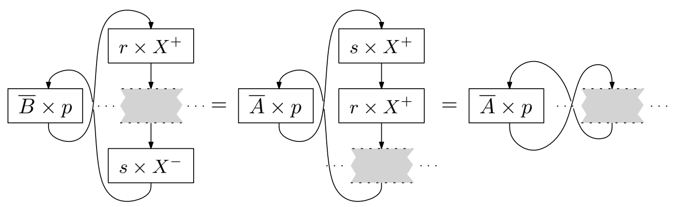
Notice that closed terms of type TA are equal if and only if they translate to low-level programs with the same extensional behaviour.

With Definition 5, we can now define the side condition $(*)$ on rule DIRECT. It requires $\Sigma \mid \Gamma \models [\![\Pi]\!] = [\![\Pi]\!]\colon X^- \to TX^+$, where $[\![\Pi]\!]$ is the interpretation of the derivation of the premise $\Sigma \mid \Gamma \vdash t\colon X^- \to TX^+$. Depending on the definition of t, it may not be easy to establish this side condition. However, rule DIRECT is intended as an extension mechanism for the definition of combinators. It can be expected that it will be used infrequently, for the definition of libraries and the like.

The parametricity lemma now takes the following form:

Lemma 6. *If Π derives the sequent $\Sigma \mid \Gamma \vdash t\colon X$, then we have $\Sigma \mid \Gamma \models [\![\Pi]\!] = [\![\Pi]\!] \colon X$.*

Proof. The proof goes by induction on Π. We just outline why the choice of section-retraction pair in rule STRUCT has no implication on the equality of a derivation. The figure below shows on the left a program of the form one obtains by applying rule STRUCT with $A \lhd B$ and then connecting a program p to this variable using app. If we know $r \circ s = id$, then the transformations in the figure below maintain extensional equality, which means that the choice of r and s has no influence on equality.



□

We have the following coherence lemma. Its proof goes along the lines of the coherence proof of Ghica & Smith in [7].

Lemma 7. *Let Π and Π' be two derivations of $\Sigma \mid \Gamma \vdash t\colon X$. Then we have $\Sigma \mid \Gamma \models [\![\Pi]\!] = [\![\Pi']\!] \colon X$.*

This lemma justifies writing just $\Gamma \models s = t\colon X$ for terms s and t, as all their derivations are equal. Moreover, Lemmas 2 and 4 imply that $[\![X]\!]_{\Delta_\sigma}$ is symmetric and transitive. As any two derivations of the same term are in relation $[\![X]\!]_{\Delta_\sigma}$, it makes sense to write just $s[\![X]\!]_{\Delta_\sigma} t$.

Since equality is defined by parametricity, identity extension can be proved by a standard argument.

Lemma 8 (Identity Extension)**.** *For any σ we have $\models s = t\colon X\sigma$ if and only if $s\ [\![X]\!]_{\Delta_\sigma}\ t$.*

Using Lemmas 2 and 4, we get symmetry and transitivity:

Lemma 9. *If $\Sigma \mid \Gamma \models s = t\colon X$ then $\Sigma \mid \Gamma \models t = s\colon X$.*

Lemma 10. *If $\Sigma \mid \Gamma \models s = t\colon X$ and $\Sigma \mid \Gamma \models t = u\colon X$ then $\Sigma \mid \Gamma \models s = u\colon X$.*

Substitution lemmas are proved by induction on derivations. They hold because the translation validates substitution of closed terms, see the extended version of [4].

Lemma 11. *If $\Sigma \mid \Gamma \models s = t\colon X$ and A is a closed type, then $\Sigma[A/\alpha] \mid \Gamma[A/\alpha] \models s[A/\alpha] = t[A/\alpha] \colon X[A/\alpha]$.*

Lemma 12. *If $\Sigma, x\colon A \mid \Gamma \models s = t\colon X$ then $\Sigma \mid \Gamma \models s[v/x] = t[v/x] \colon X$ for any value $v\colon A$.*

Lemma 13. *If $\Sigma \mid \Gamma, x\colon A \cdot X \models s = t\colon Y$ and $\Sigma \mid \Delta \vdash u\colon X$ then $\Sigma \mid \Gamma,\ A \cdot \Delta \models s[u/x] = t[u/x] \colon Y$.*

Lemma 14. *The following are true.*

1. *If $\Sigma, x\colon A \mid \Gamma \models s = t\colon X$ then $\Sigma \mid \Gamma \models \mathsf{fn}\, x{:}A.\, s = \mathsf{fn}\, x{:}A.\, t\colon A \to X$.*
2. *If $\Sigma \mid \Gamma, x\colon A \cdot X \models s = t\colon Y$ then $\Sigma \mid \Gamma \models \lambda x{:}A{\cdot}X.\, s = \lambda x{:}A{\cdot}X.\, t\colon A \cdot X \multimap Y$.*
3. *If $\Sigma \mid \Gamma \models s = t\colon X$ and α is not free in Σ or Γ, then $\Sigma \mid \Gamma \models \Lambda\alpha.\, s = \Lambda\alpha.\, t\colon \forall\alpha.\, X$.*

In the following lemma, we write just $s = t$ to mean that $\Sigma \mid \Gamma \models s = t\colon X$ holds for any Σ, Γ and X that make both terms well-typed.

Lemma 15. *The following equalities are valid whenever the terms involved are well-typed in appropriate contexts.*

$$(\lambda x{:}A{\cdot}X.\, s)\ t = s[t/x] \qquad (\lambda x{:}A{\cdot}X.\, u) = u \text{ if } x \text{ is not free in } u$$
$$(\Lambda\alpha.t)\ A = t[A/\alpha] \qquad (\Lambda\alpha.u\ \alpha) = u \text{ if } \alpha \text{ is not free in } u$$
$$(\mathsf{fn}\, x{:}A.\, t)(v) = t[v/x] \qquad (\mathsf{fn}\, x{:}A.\, u(x)) = u \text{ if } x \text{ is not free in } u$$

Proof. The η-equalities follow directly by unfolding of definitions. For the β-equalities we use the substitution lemmas above. □

Equality has been defined to allow useful reasoning, without being too strong to exclude combinators defined by direct.

Lemma 16. *The terms* `tailrec`, `fix`, `alloc`, `corout` *satisfy* $(*)$.

Proof. We outline the proof for `tailrec`. Let $\rho \subseteq \sigma \times \sigma'$ be a value type environment.

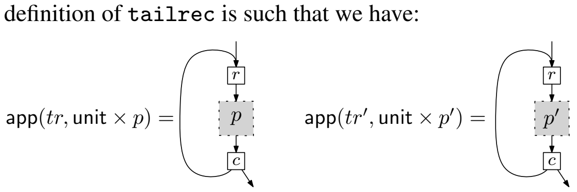
First one notes that using inst with `tailrec` amounts to substitution for the type variables. Let ρ be an environment with $\rho(\alpha) \subseteq A \times A'$, $\rho(\beta) \subseteq B \times B'$ and $\rho(\gamma) \subseteq C \times C'$. We have to show that $[\![\mathtt{tailrec}[A/\alpha, B/\beta, C/\gamma]]\!]$ and $[\![\mathtt{tailrec}[A'/\alpha, B'/\beta, C'/\gamma]]\!]$ of types $\mathsf{unit} \times (C \times A + B) + A \to \mathsf{unit} \times (C \times B + A) + B$ and $\mathsf{unit} \times (C' \times A' + B') + A' \to \mathsf{unit} \times (C' \times B' + A') + B'$ are $[\![\mathsf{unit} \cdot (\gamma \cdot (\alpha \to T\beta) \multimap \alpha \to T\beta) \multimap \alpha \to T\beta]\!]_\rho$- related.

Suppose $p\colon C \times B + A \to C \times A + B$ and $p'\colon C' \times B' + A' \to C' \times A' + B'$ are $[\![\gamma \cdot (\alpha \to T\beta) \multimap \alpha \to T\beta]\!]_\rho$-related. Consider the behaviour of p and p' on inputs of the form $\mathsf{inr}(v)$. First, if $v\ [\![\alpha]\!]_\rho\ v'$ and $p(\mathsf{inr}(v)) \xrightarrow{o} \mathsf{inr}(w)$, then $p'(\mathsf{inr}(v')) \xrightarrow{o} \mathsf{inr}(w')$ and $w\ [\![\beta]\!]_\rho\ w'$. This follows by taking q to be the always nonterminating program and observing that the assumption gives us $\mathsf{app}(p, id \times q)\ [\![\alpha \to T\beta]\!]_\rho\ \mathsf{app}(p', id \times q)$.

Second, if $v\ [\![\alpha]\!]_\rho\ v'$ and $p(\mathsf{inr}(v)) \xrightarrow{o} \mathsf{inl}(w)$, then also $p'(\mathsf{inr}(v')) \xrightarrow{o} \mathsf{inl}(w')$ and $w\ [\![\alpha]\!]_\rho\ w'$. To see this, suppose we have $v\ [\![\alpha]\!]_\rho\ v'$ and $p(\mathsf{inr}(v)) \xrightarrow{o} \mathsf{inl}(w)$. Let k be a number not appearing in o. Using an argument as in the first case, it can be shown that the other program cannot output k. Now suppose $\rho(\alpha)$ is $\langle f\rangle\langle g\rangle^{-1}$. Define programs $q\colon A \to B$ and $q'\colon A' \to B'$ as follows: On input $x{:}A$, the program q checks $f(x) = f(w)$ (which can be done as f is a program and equality can be defined) and prints k if true. Then it goes into an infinite loop. On input $y{:}A'$ the program q' checks $f(w) = g(y)$ and prints k if true. Then it goes into an infinite loop. One checks that these two programs are in relation $[\![\alpha \to T\beta]\!]_\rho$. Now, by definition, $\mathsf{app}(p, id \times q)$ outputs k on input v. As p and p' are related, then so must $\mathsf{app}(p', id \times q')$ on input v'. Hence, we have $p'(\mathsf{inr}(v')) \xrightarrow{o} \mathsf{inl}(w')$ for some w' with $f(w) = g(w')$, i.e. $w\ [\![\alpha]\!]_\rho\ w'$. This shows the assertion when $\rho(\alpha)$ is $\langle f\rangle\langle g\rangle^{-1}$. For a conjunction of such relations, the programs q and q' are modified to test a number of equalities.

Third, there is an analogous case when $p(\mathsf{inr}(v))$ does not terminate, which is handled like the first case.

Overall, we obtain that on input $\mathsf{inr}(v)$ both p and p' have the same effects and output related results. As outlined in Section 7.1, the definition of `tailrec` is such that we have:


$\mathsf{app}(tr, \mathsf{unit} \times p) =$
r
p
c
$\mathsf{app}(tr', \mathsf{unit} \times p') =$
r
p'
c

(in both cases the two lines into i go into the right input, corresponding to inr; the left input is unused) But with the established behaviour of p and p', the result can now be shown by a straightforward bisimulation-style argument. □

10. Experiments

In Section 7.3 we have seen that a simple program written using the combinator `tailrec` can be simplified to a low-level program that is close to what one would write by hand. It is reasonable to ask how much overhead arises from the use of direct-combinators and the use of higher-order functions.

We can assess such questions to a certain extent using an experimental implementation[2] of INT. It translates INT to LLVM bitcode, which can then be compiled to machine code using the LLVM tools. LLVM bitcode is similar to the first-order low-level language studied in this paper. However, it does not support disjoint sums or recursive data types. The former are easily represented using tags, e.g. $\mathsf{inl}(v)$ is represented by a pair $\langle 0, v\rangle$. Recursive data types are encoded using the heap, as LLVM does not give access to the system stack. Since the resulting memory allocations using `malloc` can be expensive in the combinator `fix`, this combinator has been modified to use a separate heap-allocated stack (see the appendix).

With this implementation, we have tested two slightly more complicated examples to test the compilation of combinators and of higher-order functions: the first example is a call-by-value CPS-translation of the Fibonacci function; a second example is a program to compute the digits of Euler's number. The latter example is adapted from the NoFib Benchmark Suite [20]; it uses functions of type $\mathsf{int} \to T\mathsf{int}$ to implement infinite lists of numbers. To check for any possible implementation overhead, we have also tested the two small examples of the Fibonacci function from Section 7.3.

The runtime of the compiled programs[2] is shown below. The Fib-example computes `fib 38` and Euler computes ten digits of e. To put these times into context, the runtime of roughly equivalent programs compiled with CLANG (a C compiler also using LLVM as its backend) and GHC (lazy evaluation perhaps corresponds closest to the call-by-name evaluation of $\multimap$-functions in INT) are also shown[3]. For such a few examples written in very different languages, a performance comparison is not meaningful, of course, and the numbers are only meant to put those for INT in context. The results indicate that further work towards assessing the suitability of INT for compilation could be worthwhile.

Program	CLANG	INT	GHC
Fib (naive)	.14s	.35s	.40s
Fib (recursion and tail recursion)	.14s	.18s	.20s
Fib (continuation passing style)	–	.30s	4.81s
Euler	–	.61s	.25s

11. Conclusion

We have presented the type system INT for organising low-level programs. It extends and simplifies earlier type systems [4, 24]. We have argued that it captures a simple and flexible way of working with fragments of low-level programs. Its subexponential type annotations express problems of stack layout analysis in the type system. The type system gives structured access to the stack, while managing administrative details. It supports separate compilation and its structure can be seen as reconstructing calls from jumps.

We have developed a variant of relational parametricity for INT, which is useful to prove the soundness of embeddings of higher languages into INT, see [26] for a report on intended applications. In further work, it should be interesting to compare the approach to existing approaches to relational parametricity and control, such as [10, 18]. Since the low-level language can be seen as a calculus of continuations, such work may be relevant. In another direction, it may also be useful to consider bisimulation-based methods for reasoning about INT programs, perhaps in the style of [14].

[2] Available from `https://github.com/uelis/intc`

[3] System: Ubuntu 13.10, Intel Core i5-3320M (64-bit); INT with LLVM 3.4, option -O3, target x86-64; GHC 7.6.3, option -O3, clang 3.2, option -O3

References

[1] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000.

[2] H. Cejtin, S. Jagannathan, and S. Weeks. Flow-directed closure conversion for typed languages. In G. Smolka, editor, *ESOP*, volume 1782 of *LNCS*, pages 56–71. Springer, 2000.

[3] U. Dal Lago and U. Schöpp. Type inference for sublinear space functional programming. In K. Ueda, editor, *APLAS*, volume 6461 of *LNCS*, pages 376–391. Springer, 2010.

[4] U. Dal Lago and U. Schöpp. Functional programming in sublinear space. In A. D. Gordon, editor, *ESOP*, volume 6012 of *LNCS*, pages 205–225. Springer, 2010.

[5] J. Egger, R. E. Møgelberg, and A. Simpson. Enriching an effect calculus with linear types. In *CSL*, volume 5771 of *LNCS*, pages 240–254, Springer, 2009.

[6] O. Fredriksson and D. R. Ghica. Abstract machines for game semantics, revisited. In *LICS*, pages 560–569. IEEE, 2013.

[7] D. Ghica and A. I. Smith. Bounded linear types with generalised resources. In *ESOP 2014*, 2014.

[8] D. R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 363–375. ACM, 2007.

[9] G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In R. Sethi, editor, *POPL*, pages 15–26, 1992.

[10] M. Hasegawa. Relational parametricity and control. In *LICS*, pages 72–81. IEEE, 2005.

[11] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163:285–408, December 2000.

[12] S. L. P. Jones, N. Ramsey, and F. Reig. C–: A portable assembly language that supports garbage collection. In G. Nadathur, editor, *PPDP*, volume 1702 of *LNCS*, pages 1–28. Springer, 1999.

[13] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Math. Proc. Cambridge Philos. Soc.*, 119(3):447–468, 1996.

[14] S. B. Lassen and P. B. Levy. Typed normal form bisimulation for parametric polymorphism. In *LICS*, pages 341–352. IEEE, 2008.

[15] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[16] P. B. Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, 2004.

[17] I. Mackie. The geometry of interaction machine. In R. K. Cytron and P. Lee, editors, *POPL*, pages 198–208. ACM Press, 1995.

[18] R. E. Møgelberg and A. Simpson. Relational parametricity for computational effects. *Logical Methods in Computer Science*, 5(3), 2009.

[19] V. Nigam and D. Miller. Algorithmic specifications in linear logic with subexponentials. In *PPDP'09*, pages 129–140. ACM, 2009.

[20] W. Partain. The nofib benchmark suite of Haskell programs. In J. Launchbury and P. M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 195–202. Springer, 1992.

[21] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[22] J. C. Reynolds. The essence of algol. In P. W. O'Hearn and R. D. Tennent, editors, *ALGOL-like Languages, Volume 1*, pages 67–88. Birkhauser, 1997.

[23] U. Schöpp. Stratified bounded affine logic for logarithmic space. In *LICS'07*, pages 411–420. IEEE, 2007.

[24] U. Schöpp. Computation-by-interaction with effects. In H. Yang, editor, *APLAS*, volume 7078 of *LNCS*, pages 305–321. Springer, 2011.

[25] U. Schöpp. On interaction, continuations and defunctionalization. In M. Hasegawa, editor, *TLCA*, volume 7941 of *LNCS*, pages 205–220. Springer, 2013.

[26] U. Schöpp. Call-by-value in a basic logic for interaction. In APLAS, to appear, 2014.

[27] P. Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989.