A Denotational Investigation of Defunctionalization

Lasse R. Nielsen

BRICS Report Series RS-00-47

ISSN 0909-0878 December 2000

A Denotational Investigation of Defunctionalization

Lasse R. Nielsen
BRICS [*]
Department of Computer Science, University of Aarhus [†]

June 1999, revised in July 2000, and reformatted in May 2001

Abstract

Defunctionalization was introduced by John Reynolds in his 1972 article *Definitional Interpreters for Higher-Order Programming Languages*. Defunctionalization transforms a higher-order program into a first-order one, representing functional values as data structures. Since then it has been used quite widely, but we observe that it has never been proven correct.

We formalize defunctionalization denotationally for a typed functional language, and we prove that it preserves the meaning of any terminating program. Our proof uses logical relations.

Keywords: defunctionalization, program transformation, denotational semantics, logical relations.

[*]Basic Research in Computer Science (`http://www.brics.dk/`), Centre of the Danish National Research Foundation.

[†]Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark. E-mail: `lrn@brics.dk`

Contents

List of Figures

1 Introduction

In a language with lexical scope and higher-order functions, evaluating a function abstraction naturally yields a function value, a *closure* [Lan64], that contains information about both the definition of the function and the denotation of its free variables. Functions taking functions as arguments are common in functional programming with *map*- and *fold*-like functions. Functions returning functions occur when applying curried functions like iterators, or, e.g., a function implementing function composition, which takes two functions as arguments and returns a function as result.

Defunctionalization is a program transformation that turns a program with higher-order functions into one with only first-order functions, thus removing the need to have closures as expressible values, using Strachey's terminology [Str00]. The method was described by Reynolds in his influential '72 paper, "Definitional Interpreters for Higher-Order Programming Languages" [Rey98]. It is based on (1) using algebraic data-types to represent the functional values as data values, at each declaration point; and (2) decoding the representation with an *apply* function defined by cases over the data-type, at each application point. Reynolds only applies defunctionalization to one class of programs, definitional interpreters, but his method is generally applicable to any closed program.

Reynolds's method became an instant classic, one that has been used in many contexts, not only for implementing interpreters for higher-order languages in first-order languages [Bel93, CJW00]. Defunctionalization can be used in program transformers, such as compilers and partial evaluators, either to simplify the language (implicit in Bondorf's work [Bon90] section 5.6: Representing Higher-Order Values) or to support higher-order functions if the target language of the transformation does not support such [CJW00]. Recently, Bell, Bellegarde, and Hook have applied defunctionalization to typed languages [BH94, BBH97]. In the latter of these papers they formalize a defunctionalizing translation and prove that it preserves typability; however they provide no formal proof that defunctionalization is also a meaning preserving transformation, although there is no real doubt that it is one.

1.1 This work

In this article, we choose a prototypical simply-typed functional language, and define a defunctionalizing transformation from an annotated version of this source language to a target language without higher-order functions. This language and approach resemble the ones in Bell, Bellegarde, and

Hook's work [BBH97].

The target language is chosen as a subset of the one considered by Bell et al., which is itself very standard:

- It is typed.
- It has top-level, mutually recursive function declarations, but not first-class functions. That is, the function identifiers can only occur in function position in applications, and only such function identifiers can occur there. "Ordinary" variables cannot denote function values and expressions cannot evaluate to such, and the language only supports functions with two arguments. This is sufficient for the output of the defunctionalization transformation, and we restrict the language to avoid unnecessary features.
- It has user-defined algebraic data-types, with a syntax resembling that of Standard ML. The datatype constructors are used for the representation of function values when defunctionalizing, The data-types denote sets constructed by products, disjoint sums, and recursive definitions, and as such they can be implemented in any language with these type constructions.
- Its set of variable names includes the ones of the source language.
- It has the same operations on base type (natural numbers) as the source language, making translation of these immediate.

Programs in the target language can be translated easily to any language with top-level functions, product and sum constructions and recursive types, and operations on integers, e.g., Standard ML or even C, using tuples and inductive datatypes in the former and pointers to structures and tagged unions in the latter [TO98].

We give the semantics of the source and target languages, and formalize a defunctionalization transformation. We then prove that the translation preserves the semantics of terminating programs.

1.2 Defunctionalization

Reynolds's transformation works on untyped expressions. It turns function abstractions into constructors applied to the denotation of the free variables, and it turns applications into applications of a first-order *apply* function to a value and an argument.

We define a translation on both *types* and (typing derivations of) expressions that maps function types into names of data-types, and expressions into equivalent expressions, with only the above changes. In other words, expressions that are not function abstractions or applications are merely traversed to translate the subexpressions. We are translating typing derivations, so we can only translate well-typed programs.

Translating the types makes it clear how to proceed. Any function type is translated to a data-type, so any expression having a functional type must be translated into a new expression having the type of the data-type. Function-type introduction (function abstractions) is then translated to data-type construction, i.e., a constructor application. Function-type elimination, i.e., application, must then be turned into data-type deconstruction, i.e., a case construct. Each case then corresponds to the body of each function that could have flowed there in the original program.

Implementing this directly, using syntax that is introduced in Section 3, would give

$$(\mathsf{fun}\ f\ x : \tau_1 \rightarrow \tau_2 = \ldots x_2 \ldots x \ldots x_7 \ldots)$$

which is the syntax we use for a recursive function value where $x_1 \ldots x_n$ are free variables. Transforming this function definition yields

$$C\ \langle v_1 = x_1 \rangle \ldots \langle v_n = x_n \rangle$$

which is the syntax used for applying a data-type constructor to a tuple indexed by the names v_1 through v_n (reminiscent of, e.g., ML records).

Likewise, function application

$$f(y)$$

is transformed into a case dispatch matching the constructor that represents the function definition:

$$\mathsf{case}\ f\ \mathsf{of}\ C\ \langle v_1 = x_1 \rangle \ldots \langle v_m = x_n \rangle \Rightarrow\ \ldots x_2 \ldots y \ldots x_7 \ldots$$

This immediate implementation falls short for recursive functions. A recursive function can call itself inside the body of the function, so if the transformation places the body of the function at its possible application points, it would have to expand the application infinitely. This is where we need the "apply" function mentioned earlier. An application thus induces a global function definition for each data-type

$$\mathsf{fun\ app}\ \mathit{ff}\ \mathit{xx} = \mathsf{case}\ \mathit{ff}\ \mathsf{of}\ C_l\ \langle v_1 = x_1 \rangle \ldots \langle v_m = x_n \rangle \Rightarrow\ \ldots$$

$$\mathsf{Program}\ (\mathsf{fun}\ f\ x : \mathsf{nat} \to \mathsf{nat} = \mathsf{ifz}(x, 0, y.\mathsf{succ}(\mathsf{succ}(f\ y))))_l^{\langle\rangle}\ (\mathsf{succ}(\mathsf{succ}(0)))$$

is translated into

```
datatype $c_{\tau_1 \to \tau_2}$ = $C_l\ \langle\rangle$
fun $\mathsf{app}_{\tau_1 \to \tau_2}$ $f\!f$ $xx$ =
    case $f\!f$ of $C_l\ \langle\rangle$ $\Rightarrow$ $\mathsf{ifz}(xx, 0, y.\mathsf{succ}(\mathsf{succ}(\mathsf{app}_{\tau_1 \to \tau_2}\ f\!f\ y)))$
in
$\mathsf{app}_{\tau_1 \to \tau_2}$ $(C_l\ \langle\rangle)$ $\mathsf{succ}(\mathsf{succ}(0))$
```

Figure 1: Example program

and the application itself becomes just

$$\mathsf{app}\ f\ y$$

To preserve well-typedness, we also translate the types. We give a data-type declaration defining one data-type for each function type occurring in the program, such that each translated type is defined in the target program. For each function abstraction we add, to the datatype corresponding to the type of the abstraction, a constructor with an argument that contains the values in the environment of the abstraction.

Notice that this data-type declaration might define data-types with no constructors. Such declarations are allowed, but they are only needed if the program contains unreachable code or a diverging expression, since it is a symptom of there being an expression with a type for which no values are ever created. On the other hand it is easy to write a well-typed program with this property, so the translation must treat it properly.

We then define a group of top-level application functions, one for each functional type occurring in an application. These functions take two arguments, a function representation and an argument, and dispatch on the function representation. The case contains a match for each constructor of that type, with the body of the corresponding function as expression (and a renaming of the arguments of the apply function to the name of the formal parameters of the original function). We place the translation of the expression of the original program in the scope of these declarations. See Figure 1 for an example.

1.3 Scope of the results

We prove the *weak* correctness of the defunctionalization translation of a prototypical *typed* functional language with higher-order functions. In other words, *if* the original program terminates, *then* so does the defunctionalized one, and with the same result. Eventually, we want to prove full correctness, i.e., semantic equivalence of the two programs.

The source language is *monomorphic*, whereas Bell, Bellegarde, and Hook treat a polymorphic language, albeit by turning it into a monomorphic one by code duplication. Reynolds's programs are untyped, and only implicitly group the occurring functions into "continuations" and "abstractions" in the interpreters.

Where Reynolds stores only the values of the free variables of a function in the constructed value, our transformation stores the entire environment. Only storing the value of the free variables is an optimization which does not change the correctness of the translation, and we have omitted it from the present work.

Reynolds does not translate all function abstractions into values, only the ones that are actually used as arguments or results of higher-order functions. He leaves the globally defined functions, such as the one called *eval*, as functions. The present transformation, on the other hand, turns *all* abstractions in the source program into constructed values in the target program.

Reynolds can avoid transforming such functions because he knows that the function is not used as an argument or result of a function in the translated program, so there is no need to turn it into a non-functional value. In this paper we do not assume such knowledge, though it could be added with a prior analysis step. There is also another consideration at work; the function *eval* does occur as a free variable in another function abstraction which *is* translated into a constructor. If the value of *eval* is to be used as part of the argument to this constructor, being a free variable in the body of the function abstraction, then that value should have been translated too. Here Reynolds recognizes that it is not necessary to put the *eval* function into the "closure" he creates because it is defined at top level, and therefore *still* in scope for the code of the body of the abstraction when the body is moved into the *apply* function. This optimization is similar to Johnsson's λ-lifting which generates recursive equations [Joh85].

An analysis that detects whether the value of a function abstraction is used as an argument or result is trivial for the source language we use.

To summarize, if the value of a function definition is not being passed around, and the function is closed, then it can be made into a global (top-level) function. If it is not closed, but not being passed around and not free in some other function that is being converted, then it can be left in place. Reynolds at least uses the latter rule, and his program already has as many functions global as possible. The present transformation does neither of these optimizations. Rather, it makes the pessimistic, but safe, assumption that any function can possibly be used as argument to another function, so all function abstractions must be translated, just as in super-combinator conversion [Pey85].

1.4 Related subjects

Defunctionalization is closely related to two other transformations: Closure Conversion and Lambda Lifting.

Closure Conversion is a program transformation that turns all function abstractions into pairs of *closed* code (no free variables) and a representation of the values of the free variables in the original code. The closed code is still a function, so this is not defunctionalization, but it shares the concept of turning abstractions into values containing the values of the free variables at the point of definition. Closure conversion has also been studied in a typed setting by Harper, Morrisett, and Minamide [MMH96], and used in, e.g., a the compiler for Standard ML of New Jersey [AJ89].

Lambda Lifting is a program transformation that bypasses the need to represent closures by removing the lexical nesting. It "lifts" abstractions out of the program and into "global" top-level declarations (traditionally a set of mutually recursive equations), capturing the *non-functional* free variables by abstracting over them. Their denotation must then be passed at each application. The functions themselves are then all defined at the same level, and need not be captured in the same way. They can still be passed as arguments if needed, though [Joh85, DS00, FH00].

1.5 Prerequisites and notation

The reader is assumed to be familiar with Reynolds's "Definitional Interpreters for Higher-Order Programming Languages" [Rey98], and the concepts and notation of denotational semantics, as presented in, e.g., Winskel's textbook [Win93].

1.5.1 Indexed products

A product space, $A \times B$, is characterized by, amongst other things, the existence of the associated projection functions, $\pi_1 : A \times B \to A$ and $\pi_2 :$

$A \times B \to B$, i.e., the product is implicitly indexed by the numbers 1 and 2. That is, $A \times B$ is isomorphic to the subset of the function space $\{1,2\} \to A \cup B$ that maps 1 to something in A and 2 to something in B.

Generalizing this to products over arbitrary finite sets, if for each $s \in S$ we can construct a set A_s, then $\prod_{s \in S} A_s$ is a product space with projections $\pi_s : (\prod_{s \in S} A_s) \to A_s$ for each s in S. If the A_s are all equal to some set M, the product can be written M^S. The set of indices is called the "index set".

Let $A \overset{\text{def}}{=} \prod_{s \in S} A_s$ be an arbitrary product space, and α a tuple in it. The notation for elements of such products will be $\{s_1 = a_1, \ldots, s_n = a_n\}$ where $a_i \in A_{s_i}$, reminiscent of, e.g., records in ML.

An element of a product space is uniquely determined by its values under the projections. That is, we can define an element of a product space, $\prod_{s \in S} A_s$, by giving an element $a_s \in A_s$ for each $s \in S$.

Dom(A)

The domain of a product space is the index set of the product. In this case $\text{Dom}(A) = S$.

The Dom function may also be used on elements of the product, i.e., $\text{Dom}(\alpha)$. In that case it is just a shorthand for "$\text{Dom}(A)$ where $\alpha \in A$".

Projection

Instead of writing $\pi_s(\alpha)$, we write $\alpha(s)$, i.e., we interpret the tuple as the corresponding function in $S \to \cup_{s \in S} A_s$.

If it is not obvious from the context that $s \in \text{Dom}(\alpha)$ then writing $\alpha(s) = x$ implicitly means $s \in \text{Dom}(\alpha) \wedge \alpha(s) = x$.

This notation is also used on the product spaces themselves, i.e., $A(s)$ is the set A_s, corresponding to the usual notation for applying a function to a set of arguments:

$$A(s) = \pi_s(A) = \{\pi_s(\alpha) \,|\, \alpha \in A\} = A_s$$

Extending a product

If A is a product space then $A\{x = M\}$ is the product space

$$\prod_{y \in \text{Dom}(A) \cup \{x\}} A'_y$$

where $A'_x = M$ and if $y \neq x$ then $A'_y = A_y$.

This definition allows extending the product both if the new index is in the domain of the old product and if it is not.

The same notation will also be used for elements of the products, i.e., if $\alpha \in A$ and $v \in M$ then $\alpha\{x{=}v\} \in A\{x{=}M\}$, and $\alpha\{x{=}v\}\,(x) = v$.

Restricting a product

If A is a product then $A\setminus\{x\}$ is the product with domain $\mathrm{Dom}(A)\setminus\{x\}$, s.t. for $y \in \mathrm{Dom}(A) \setminus \{x\}$ we get $(A \setminus \{x\})(y) = A(y)$.

Again, we will use the same notation on the elements of the product.

Terminology: environment

We use the word "environments" about the elements of a product space of denotable values where the index set is a set of identifiers.

1.5.2 Disjoint indexed unions

A disjoint sum of two sets is usually defined as $A{+}B = \mathit{inl}(A){\cup}\mathit{inr}(B)$ where *inl* and *inr* are injection functions s.t. $\mathit{inl}(A) \cap \mathit{inr}(B) = \emptyset$. Traditionally these injections are defined as $\mathit{inl} : A \to (\{1\} \times A)$ and $\mathit{inr} : B \to (\{2\} \times B)$ with $\mathit{inl}(a) = (1, a)$ and $\mathit{inr}(b) = (2, b)$, but any pair of bijective functions with disjoint images can be used.

Binary disjoint sums can be generalized to sums over arbitrary finite index sets. If for each $s \in S$ we can construct a set A_s, then $\sum_{s\in S} A_s = \bigcup_{s\in S}\{s\} \times A_s$ is a sum with injections $\mathit{in}_s : A_s \to \{s\} \times A_s$ for each s in S.

Product spaces come with an associated tupling operation, and similarly a disjoint sum has an associated deconstructor. To deconstruct a sum, we use the associated *case* function, that for binary sums has the "type scheme"

$$\mathit{case}_{A+B} : (A \to M) \times (B \to M) \to A + B \to M$$

and satisfies $\mathit{case}_{A+B}(f, g)(\mathit{inl}(x)) = f(x)$ and $\mathit{case}_{A+B}(f, g)(\mathit{inr}(y)) = g(y)$.

Generalizing to more than two summands we have the function

$$\mathit{case} : \left(\prod_{s\in S} A_s \to M\right) \to \left(\sum_{s\in S} A_s\right) \to M$$

defined as $\mathit{case}(p)(\mathit{in}_s(a)) = \mathit{case}(p)(s, a) = p(s)(a)$.

1.5.3 Lifting CPOs

If $(D, \sqsubseteq)$ is the chain-complete partial order (CPO) with carrier set D and partial ordering relation $\sqsubseteq$, then $(D_\perp, \sqsubseteq_\perp)$ is the "lifted" partial order with carrier set $\{\mathrm{up}(x) \,|\, x \in D\} \cup \{\perp\}$

If a CPO has a least element, the CPO is called pointed, and its least element is called bottom, written $\perp$. A lifted CPO is always pointed, so the use of $\perp$ is consistent.

If f is a function in the CPO $D \to E$ with E pointed then $(f)^\dagger : D_\perp \to E$ is defined as $(f)^\dagger\,(\perp) = \perp$ and $(f)^\dagger\,(\mathrm{up}(x)) = f(x)$.

Both the $\mathrm{up}(\ldots) : D \to D_\perp$ and $(\ldots)^\dagger : (D \to E) \to (D_\perp \to E)$ are continuous as functions from CPOs to CPOs, and the latter also preserves continuity [Win93, Chapter 8].

1.6 Overview

Section 1 describes Reynolds's defunctionalization and related work. Sections 2 through 4 formalize the source and target languages, and define the defunctionalizing program transformation. While lengthy, these three sections uses only elementary constructions and concepts to establish the basis for the following sections. Sections 5 and 6 show that the transformation preserves typability and that it is weakly semantics preserving, i.e., it preserves the semantics of all terminating programs. Finally Section 7 describes possible extensions of the proof and suggests several directions for further work.

2 The source language

In order to formalize defunctionalization, we give a formal definition of both the source language and the target language with their denotational semantics.

As a source language we use a prototypical higher-order functional language.

2.1 Syntax

The syntax is given by the grammar in Figure 2, where x and f ranges over a set of identifiers.

$$
\begin{array}{rcl}
\tau & ::= & \mathsf{nat} \\
 & | & \tau \to \tau \\
 & & \\
e & ::= & x \\
 & | & 0 \\
 & | & \mathsf{succ}(e) \\
 & | & \mathsf{ifz}(e, e, x.e) \\
 & | & (\mathsf{fun}\ f\ x : \tau \to \tau = e) \\
 & | & (e\ e) \\
 & & \\
p & ::= & \mathsf{Program}\ e
\end{array}
$$

Figure 2: Syntax of the source language

2.2 Typing judgments

The typing rules in Figure 3 define well-typed expressions. The judgments are of the form $\Gamma \vdash e : \tau$, where Γ is a type assignment (an environment mapping variables to types), e is an expression and τ a type, or of the form $\vdash p$ where p is a program ("$\mathsf{Program}\ e$" for some expression e of type nat).

If we can derive $\vdash \mathsf{Program}\ e$ then we say that the program is well-typed, and the derivation is called a *typing derivation of the program*.

Since all function abstractions include their type syntactically, for any type assignment Γ and expression e, there is at most one τ such that there exists a typing derivation of $\Gamma \vdash e : \tau$, and there is at most one derivation of any judgment (easy proof by induction on structure of program omitted). Therefore there can be no ambiguity when referring to *the* derivation of $\Gamma \vdash e : \tau$.

2.3 Denotational semantics

The denotational semantics used is a standard call-by-value semantics. It is given as a inductively defined function over the structure of typing derivations, so it only makes sense to talk about the denotation of an expression or a program if the expression or program is well-typed.

The denotation of a derivation is written as a function of the *conclusion* only, but this is just for ease of representation. The function actually takes the entire derivation as argument, and references to sub-derivations (again

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad\qquad \frac{}{\Gamma \vdash 0 : \mathsf{nat}}$$

$$\frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{succ}(e) : \mathsf{nat}}$$

$$\frac{\Gamma \vdash e : \mathsf{nat} \quad \Gamma \vdash e_1 : \tau \quad \Gamma\{x{=}\mathsf{nat}\} \vdash e_2 : \tau}{\Gamma \vdash \mathsf{ifz}(e, e_1, x.e_2) : \tau}$$

$$\frac{\Gamma\{f{=}\tau_1 \to \tau_2\}\{x{=}\tau_1\} \vdash e : \tau_2}{\Gamma \vdash (\mathsf{fun}\ f\ x : \tau_1 \to \tau_2 = e) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}$$

$$\frac{\{\} \vdash e : \mathsf{nat}}{\vdash \mathsf{Program}\ e}$$

Figure 3: Typing judgments for the source language

represented by their conclusions) are still compositional.

The semantic functions are shown in Figure 4. The arrow($\to$) in the denotation of a function type represents the *continuous* function space between the CPOs.

2.4 Annotated source language

The defunctionalizing transformation is inherently nonlocal, since it moves the bodies of functions from their original position in the program and into the application functions. To reference specific subexpressions of the program, we annotate each expression with a unique *label* taken from some countably infinite set of label identifiers.

Also, we annotate function abstractions with the variables bound by the type assignment of the expression. This annotation is of the form:

$$\begin{array}{rcl} env & ::= & \cdot \\ & | & env\langle x : \tau\rangle \end{array}$$

That such an annotation really represents the type assignment is captured

$$
\begin{array}{rcl}
\mathcal{S}[\![\mathsf{nat}]\!] & = & \mathrm{N} \\
\mathcal{S}[\![\tau_1 \to \tau_2]\!] & = & \mathcal{S}[\![\tau_1]\!] \to \mathcal{S}[\![\tau_2]\!]_\perp \\
\\
\mathcal{S}[\![\Gamma]\!] & = & \displaystyle\prod_{x \in \mathrm{Dom}(\Gamma)} \mathcal{S}[\![\Gamma(x)]\!] \\
\\
\mathcal{S}[\![\Gamma \vdash e : \tau]\!] & : & \mathcal{S}[\![\Gamma]\!] \to \mathcal{S}[\![\tau]\!]_\perp \\
\mathcal{S}[\![\Gamma \vdash x : \tau]\!]\rho & = & \mathrm{up}(\rho(x)) \\
\mathcal{S}[\![\Gamma \vdash 0 : \mathsf{nat}]\!]\rho & = & \mathrm{up}(0) \\
\mathcal{S}[\![\Gamma \vdash \mathsf{succ}(e) : \mathsf{nat}]\!]\rho & = & (\lambda x.\mathrm{up}(x+1))^\dagger\,(\mathcal{S}[\![\Gamma \vdash e : \mathsf{nat}]\!]\rho) \\
\mathcal{S}[\![\Gamma \vdash \mathsf{ifz}(e, e_1, x.e_2) : \tau]\!]\rho & = & (\Psi)^\dagger\,(\mathcal{S}[\![\Gamma \vdash e : \mathsf{nat}]\!]\rho) \\
\text{where } \Psi t = & \multicolumn{2}{l}{\begin{cases} \mathcal{S}[\![\Gamma \vdash e_1 : \tau]\!]\rho & \text{if } t = 0 \\ \mathcal{S}[\![\Gamma\{x{=}\mathsf{nat}\} \vdash e_2 : \tau]\!]\rho\{x{=}t_1\} & \text{if } t > 0 \end{cases}} \\
\multicolumn{3}{l}{\mathcal{S}[\![\Gamma \vdash (\mathsf{fun}\ f\ x : \tau_1 \to \tau_2 = e) : \tau_1 \to \tau_2]\!]\rho} \\
& = & \mathrm{up}(\mathrm{fix}(\Psi)) \\
\multicolumn{3}{l}{\text{where } \Psi F = \lambda X.\mathcal{S}[\![\Gamma\{f{=}\tau_1 \to \tau_2\}\{x{=}\tau\} \vdash e : \tau_2]\!]\rho\{f{=}F\}\{x{=}X\}} \\
\mathcal{S}[\![\Gamma \vdash (e_1\ e_2) : \tau_2]\!]\rho & = & \\
\multicolumn{3}{r}{\Big(\lambda f.((\lambda x.fx)^\dagger\,(\mathcal{S}[\![\Gamma \vdash e_2 : \tau_1]\!]\rho))\Big)^\dagger\,(\mathcal{S}[\![\Gamma \vdash e_1 : \tau_1 \to \tau_2]\!]\rho)} \\
\\
\mathcal{S}[\![\vdash p]\!] & : & \mathrm{N}_\perp \\
\mathcal{S}[\![\vdash \mathsf{Program}\ e]\!] & = & \mathcal{S}[\![\{\} \vdash e : \mathsf{nat}]\!]\{\}
\end{array}
$$

Figure 4: Call-by-value denotational semantics of well-typed programs

by the following judgment.

$$\frac{}{\{\} \vdash \cdot}$$

$$\frac{\Gamma(x) = \tau \quad \Gamma \setminus \{x\} \vdash \mathit{env}'}{\Gamma \vdash \mathit{env}'\langle x : t\rangle}$$

Let L be the set of labels occurring in the annotated program p. Since labels are unique, we can define a function mapping labels of L to the sub-derivations of $\vdash p$ of the expression labeled by that label.

Definition 1 *The $\mathit{PROOF}(p)$ relation*

We define a function recursively on the derivation of (annotated) expressions by.

$$\mathit{PROOF}\left(\frac{\mathcal{D}}{\Gamma \vdash e_l : \tau}\right) = \{(l, \Gamma \vdash e_l : \tau)\} \cup \bigcup_{d \in \mathcal{D}} \mathit{PROOF}(d)$$

We can then show by an easy induction proof that for every $\Gamma \vdash e_l : \tau$ the set $\mathit{PROOF}(\Gamma \vdash e_l : \tau)$ defines the graph of a function from labels to derivations, since no l occurs more than once in a well-annotated program, and also that any sub-derivation of $\Gamma \vdash e : \tau$ is in the image of $\mathit{PROOF}(\Gamma \vdash e : \tau)$.

Extending this function to programs, we define

$$\mathit{PROOF}(\vdash \mathsf{Program}\ e_l) = \mathit{PROOF}(\{\} \vdash e_l : \mathsf{nat})$$

It follows from the definition that the domain of $\mathit{PROOF}(p)$ is exactly L.

We use functions for recognizing and deconstructing syntax: isfun(e) and isappl(e) are true if e is syntactically a function abstraction and respectively an application expression. If $e = (\mathsf{fun}\ f\ x : \tau_1 \rightarrow \tau_2 = e')^{env}$ then isfun(e) is true and

$$\begin{array}{rcl} \text{funvar}(e) & = & f \\ \text{argvar}(e) & = & x \\ \text{argtype}(e) & = & \tau_1 \\ \text{restype}(e) & = & \tau_2 \\ \text{body}(e) & = & e' \\ \text{envof}(e) & = & \mathit{env} \end{array}$$

Likewise, if $e = (e_1\ e_2)$ then isappl(e) is true, funpart(e) $= e_1$, and argpart(e) $= e_2$. We do not define function deconstructing the remaining kinds of expressions, since we have no need for them.

The functions on expressions are extended to work on derivations too, by applying to the expression of the conclusion. Where the functions on expressions return subexpressions, the extended functions return the subderivation corresponding to that subexpression, e.g., $\mathrm{funpart}(\Gamma \vdash (e_1\ e_2) : \tau_2) = \Gamma \vdash e_1 : \tau_1 \to \tau_2$. Also, we define the functions $\mathrm{typeof}(\Gamma \vdash e : \tau) = \tau$ and $\mathrm{envof}(\Gamma \vdash e : \tau) = \Gamma$.

To collect the set of all types occurring in the program p, we define $\mathrm{typesof}(p)$ as the set

$$\{\tau \mid \tau \neq \mathsf{nat} \wedge (l, D) \in \mathrm{PROOF}\,(p) \wedge (\mathrm{typeof}(D) = \tau \vee (\mathrm{isfun}(D) \wedge \\ (\mathrm{argtype}(D) = \tau \vee \mathrm{restype}(D) = \tau)))\}$$

i.e., any type, other than `nat`, that is the type of an expression, or is the type of an argument or a result of a function. This set could equally well have been defined recursively in the structure of the derivation of p, in the same way as $\mathrm{PROOF}\,(p)$.

3 The target language

The language we want to transform our annotated source language programs into must necessarily have user-defined algebraic data-types for the translation to follow Reynolds's method [Rey98]. It also needs mutually recursive functions, and operations on natural numbers corresponding to the ones in the source language. We do not need reflexive data-types (ones where the types themselves can occur negatively, i.e., as domain of a function type, in their own definition), since in the output of defunctionalization, the datatype constructors are first order, just as the functions.

The target language will consist of a set of mutually recursive top-level data-type definitions followed by a set of mutually recursive function definitions (with a fixed arity of two, which is all we need for the output of defunctionalization). Finally there is one expression that is evaluated in the scope of these declarations to give the result of the program.

3.1 Syntax

The syntax is more complex than for the source language, mainly due to the extra syntactic categories introduced by the top-level data-type and function declarations. It is displayed in Figure 5, where c ranges over type names (of user-defined types), C ranges over constructor names, f ranges over function names, x and y range over variable names, and v ranges over indices of records.

type	τ	$::=$	$\mathsf{nat} \mid c$
datadecl-list	ddl	$::=$	$\cdot \mid dd\ \mathsf{and}\ ddl$
datadecl	dd	$::=$	$c\ =\ cdl$
condecl-list	cdl	$::=$	$\cdot \mid cd \parallel cdl$
condecl	cd	$::=$	$C\ ct$
con-tuple	ct	$::=$	$\langle\rangle \mid ct\langle v : \tau\rangle$
fundecl-list	fdl	$::=$	$\cdot \mid fd\ \mathsf{and}\ fdl$
fundecl	fd	$::=$	$f\ x\ y : \tau \rightarrow \tau \rightarrow \tau = exp$
expression	exp	$::=$	x
		$\mid$	0
		$\mid$	$\mathsf{succ}(exp)$
		$\mid$	$\mathsf{ifz}(exp, exp, x.exp)$
		$\mid$	$f\ exp\ exp$
		$\mid$	$C\ at$
		$\mid$	$\mathsf{case}\ exp\ \mathsf{of}\ ml$
		$\mid$	$\mathsf{let}\ x = exp\ \mathsf{in}\ exp$
arg-tuple	at	$::=$	$\langle\rangle \mid at\langle v = exp\rangle$
match-list	ml	$::=$	$\cdot \mid m \parallel ml$
match	m	$::=$	$C\ mt \Rightarrow exp$
match-tuple	mt	$::=$	$\langle\rangle \mid mt\langle v = x\rangle$
program	p	$::=$	$\mathsf{datatype}\ ddl$
			$\mathsf{fun}\ fdl\ \ \mathsf{in}\ exp$

Figure 5: Syntax of the target language

3.2 Typing judgments

The target language is also typed. There will be a typing judgment for each syntactical category.

In the following, $\mathcal{O} \subseteq$ TypeID is a set of type names (ranged over by $c \in$ TypeID in the syntax), Ω is an environment from type names to a Δ, where Δ is an environment from constructor names ($C \in$ ConID) to tuple representations, $\mathcal{V}$, where $\mathcal{V}$ again is an environment from label identifiers ($v \in$ LabelID) to types. Also, Φ is a map from function-names to triples of types and Γ is a map from variable-names to types, i.e., a type assignment.

Notice that τ is used for types in both the source and the target language. It will be obvious from the context which is meant by an instance of τ. The typing judgment schema are as given below.

type: $\mathcal{O} \vdash \tau$

$$\frac{}{\mathcal{O} \vdash \mathsf{nat}} \qquad \frac{c \in \mathcal{O}}{\mathcal{O} \vdash c}$$

datadecl-list: $\mathcal{O} \vdash ddl : \Omega$

$$\frac{}{\mathcal{O} \vdash \cdot : \{\}} \qquad \frac{\mathcal{O} \vdash dd : (c, \Delta) \quad \mathcal{O} \vdash ddl : \Omega}{\mathcal{O} \vdash dd \;\mathsf{and}\; ddl : \Omega\{c{=}\Delta\}}$$

datadecl: $\mathcal{O} \vdash dd : (c, \Delta)$

$$\frac{\mathcal{O} \vdash cdl : \Delta}{\mathcal{O} \vdash c = cdl : (c, \Delta)}$$

condecl-list: $\mathcal{O} \vdash cdl : \Delta$

$$\frac{}{\mathcal{O} \vdash \cdot : \{\}} \qquad \frac{\mathcal{O} \vdash cd : (C, \mathcal{V}) \quad \mathcal{O} \vdash cdl : \Delta}{\mathcal{O} \vdash cd \| cdl : \Delta\{C{=}\mathcal{V}\}}$$

condecl: $\mathcal{O} \vdash cd : (C, \mathcal{V})$

$$\frac{\mathcal{O} \vdash ct : \mathcal{V}}{\mathcal{O} \vdash C\ ct : (C, \mathcal{V})}$$

con-tuple: $\mathcal{O} \vdash ct : \mathcal{V}$

$$\frac{}{\mathcal{O} \vdash \langle\rangle : \{\}} \qquad \frac{\mathcal{O} \vdash ct : \mathcal{V} \quad \mathcal{O} \vdash \tau}{\mathcal{O} \vdash ct\langle v : \tau\rangle : \mathcal{V}\{v{=}\tau\}}$$

fundecl-list: $\Omega, \Phi \vdash fdl : \Phi'$

$$\frac{}{\Omega, \Phi \vdash \cdot : \{\}} \qquad \frac{\Omega, \Phi \vdash fd : (f, (\tau_1, \tau_2, \tau_3)) \quad \Omega, \Phi \vdash fdl : \Phi'}{\Omega, \Phi \vdash fd \;\mathsf{and}\; fdl : \Phi'\{f{=}(\tau_1, \tau_2, \tau_3)\}}$$

fundecl: $\Omega, \Phi \vdash fd : (f, (\tau_1, \tau_2, \tau_3))$

$$\frac{\Omega, \Phi, \{x = \tau_1, y = \tau_2\} \vdash e : \tau_3}{\Omega, \Phi \vdash f\ x\ y : \tau_1 \to \tau_2 \to \tau_3 = e : (f, (\tau_1, \tau_2, \tau_3))}$$

expression: $\Omega, \Phi, \Gamma \vdash exp : \tau$

$$\frac{\Gamma(x) = \tau \quad \mathrm{Dom}(\Omega) \vdash \tau}{\Omega, \Phi, \Gamma \vdash x : \tau}$$

$$\frac{}{\Omega, \Phi, \Gamma \vdash 0 : \mathsf{nat}}$$

$$\frac{\Omega, \Phi, \Gamma \vdash e : \mathsf{nat}}{\Omega, \Phi, \Gamma \vdash \mathsf{succ}(e) : \mathsf{nat}}$$

$$\frac{\Omega, \Phi, \Gamma \vdash e : \mathsf{nat} \quad \Omega, \Phi, \Gamma \vdash e_1 : \tau \quad \Omega, \Phi, \Gamma\{x{=}\mathsf{nat}\} \vdash e_2 : \tau}{\Omega, \Phi, \Gamma \vdash \mathsf{ifz}(e, e_1, x.e_2) : \tau}$$

$$\frac{\Phi(f) = (\tau_1, \tau_2, \tau_3) \quad \Omega, \Phi, \Gamma \vdash e_1 : \tau_1 \quad \Omega, \Phi, \Gamma \vdash e_2 : \tau_2}{\Omega, \Phi, \Gamma \vdash f\ e_1\ e_2 : \tau_3}$$

$$\frac{\Omega(c)(C) = \mathcal{V} \quad \Omega, \Phi, \Gamma \vdash at : \mathcal{V}}{\Omega, \Phi, \Gamma \vdash C\ at : c}$$

$$\frac{\Omega, \Phi, \Gamma \vdash e : c \quad \Omega, \Phi, \Gamma \vdash ml : c \Rightarrow \tau}{\Omega, \Phi, \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ ml : \tau}$$

$$\frac{\Omega, \Phi, \Gamma \vdash e_1 : \tau_1 \quad \Omega, \Phi, \Gamma\{x{=}\tau_1\} \vdash e_2 : \tau_2}{\Omega, \Phi, \Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

arg-tuple: $\Omega, \Phi, \Gamma \vdash at : \mathcal{V}$

$$\frac{}{\Omega, \Phi, \Gamma \vdash \langle\rangle : \{\}} \qquad \frac{\Omega, \Phi, \Gamma \vdash at : \mathcal{V} \setminus \{v\} \quad \mathcal{V}(v) = \tau \quad \Omega, \Phi, \Gamma \vdash e : \tau}{\Omega, \Phi, \Gamma \vdash at\langle v = e\rangle : \mathcal{V}}$$

match-list: $\Omega, \Phi, \Gamma \vdash ml : c \Rightarrow \tau$

$$\frac{}{\Omega, \Phi, \Gamma \vdash \cdot : c \Rightarrow \tau} \qquad \frac{\Omega, \Phi, \Gamma \vdash ml : c \Rightarrow \tau \quad \Omega, \Phi, \Gamma \vdash m : c \Rightarrow \tau}{\Omega, \Phi, \Gamma \vdash m \parallel ml : c \Rightarrow \tau}$$

match: $\Omega, \Phi, \Gamma \vdash m : c \Rightarrow \tau$

$$\frac{\Omega(c)(C) = \mathcal{V} \quad \Gamma, \mathcal{V} \vdash mt : \Gamma' \quad \Omega, \Phi, \Gamma' \vdash e : \tau}{\Omega, \Phi, \Gamma \vdash C\ mt \Rrightarrow e : c \Rightarrow \tau}$$

match-tuple: $\Gamma, \mathcal{V} \vdash mt : \Gamma'$

$$\frac{}{\Gamma, \{\} \vdash \langle\rangle : \Gamma} \qquad \frac{\Gamma, \mathcal{V} \setminus \{v\} \vdash mt : \Gamma' \quad \mathcal{V}(v) = \tau}{\Gamma, \mathcal{V} \vdash mt\langle v = x\rangle : \Gamma'\{x{=}\tau\}}$$

program: $\vdash\ p$

$$\frac{\mathrm{Dom}(\Omega) \vdash ddl : \Omega \quad \Omega, \Phi \vdash fdl : \Phi \quad \Omega, \Phi, \{\} \vdash e : \mathsf{nat}}{\vdash \mathsf{datatype}\ ddl\ \mathsf{fun}\ fdl\ \ \mathsf{in}\ e}$$

3.3 Denotational semantics

Again we give a denotation to the derivations for each syntactic category and to Ω, Φ, etc.

We will let **Set** be the class of sets, ordered by inclusion. This ordering is a pointed partial order (has a least element: the empty set) and is closed under least upper-bounds (unions) of ω-chains, but it is not a partially ordered *set*.

Working in something that is not a set is not in itself a problem, since we will only referring to its elements, which *are* sets, and functions mapping sets to sets, but we will need a fixed point of some sort on this construction. The semantic domains are as follows.

$$\begin{aligned} \mathcal{D}[\![\mathcal{O}]\!] &= \prod_{c \in \mathcal{O}} \mathbf{Set} = \mathbf{Set}^{\mathcal{O}} \\ \mathcal{D}[\![\mathcal{O} \vdash \Omega]\!] &: \mathcal{D}[\![\mathrm{Dom}(\mathcal{O})]\!] \to \mathcal{D}[\![\mathrm{Dom}(\mathcal{O})]\!] \end{aligned}$$

Notice that the meaning of a user-defined type is a *set*.

When we write $\mathcal{D}[\![\mathcal{O} \vdash \Omega]\!]$, the function is actually defined on the entire derivation of the conclusion $\mathcal{O} \vdash \Omega$. For ease of reading, we omit the derivation. Also, if the argument of a semantic function matches the format of the conclusion of a rule exactly, then only the expression is given (e.g., instead of $[\![\Omega, \Phi, \Gamma \vdash e : \tau]\!]$ we just write $[\![e]\!]$). This shorthand is still unambiguous, since there is exactly one rule corresponding to each element of each syntactic category.

If $o \in \mathcal{D}[\![\mathcal{O}]\!]$ for the $\mathcal{O}$ in $\mathcal{O} \vdash \Omega$ then

$$\begin{aligned} \mathcal{D}[\![\Omega]\!]^o &= \prod_{c \in \mathrm{Dom}(\Omega)} \mathcal{D}_s[\![\Omega(c)]\!]^o \\ \mathcal{D}[\![\Delta]\!]^o &= \prod_{C \in \mathrm{Dom}(\Delta)} \mathcal{D}[\![\Delta(C)]\!]^o \end{aligned}$$

$$\begin{array}{rcl}
\mathcal{D}_s[\![\Delta]\!]^o & = & \sum_{C \in \mathrm{Dom}(\Delta)} \mathcal{D}[\![\Delta(C)]\!]^o \\
\mathcal{D}[\![\mathcal{V}]\!]^o & = & \prod_{v \in \mathrm{Dom}(\mathcal{V})} \mathcal{D}[\![\mathcal{V}(v)]\!]^o \\
\mathcal{D}[\![\Phi]\!]^o & = & \prod_{f \in \mathrm{Dom}(\Phi)} \mathcal{D}[\![\tau_1]\!]^o \to \mathcal{D}[\![\tau_2]\!]^o \to \mathcal{D}[\![\tau_3]\!]^o_\perp \\
 & & \text{where } (\tau_1, \tau_2, \tau_3) = \Phi(f) \\
\mathcal{D}[\![\Gamma]\!]^o & = & \prod_{x \in \mathrm{Dom}(G)} \mathcal{D}[\![\Gamma(x)]\!]^o
\end{array}$$

The first argument, $o \in \mathcal{D}[\![\mathrm{Dom}(\Omega)]\!]$, is written in superscript only to indicate that it will be passed unchanged in any recursive calls, i.e., it can be considered a constant for the expression.

Expanding the definition, we see that $\mathcal{D}[\![\Omega]\!]^o$, the meaning of the type declarations, can be written more readably as follows.

$$\mathcal{D}[\![\Omega]\!]^o = \prod_{c \in \mathrm{Dom}(\Omega)} \sum_{C \in \mathrm{Dom}(\Omega(c))} \prod_{x \in \mathrm{Dom}(\Omega(c)(C))} \mathcal{D}[\![\Omega(c)(C)(x)]\!]^o$$

In the following, we use the same convention as in the definition of the denotation of the source language: the semantic functions take full derivations as arguments, even though only the conclusion is written, and references to the premises are allowed and considered compositional.

type: To each type expression in the language we assign a set.

$$\begin{array}{rcl}
\mathcal{D}[\![\mathcal{O} \vdash \tau]\!] & : & \mathcal{D}[\![\mathcal{O}]\!] \to \mathbf{Set} \\
\mathcal{D}[\![\mathcal{O} \vdash \mathsf{nat}]\!]^o & = & \mathrm{N} \\
\mathcal{D}[\![\mathcal{O} \vdash c]\!]^o & = & o(c)
\end{array}$$

datadecl-list

$$\begin{array}{rcl}
\mathcal{D}[\![\mathcal{O} \vdash ddl : \Omega]\!] & : & \mathcal{D}[\![\Omega]\!] \\
\mathcal{D}[\![\mathcal{O} \vdash \cdot : \{\}]\!]^o & = & \{\} \\
\mathcal{D}[\![\mathcal{O} \vdash dd \ \mathsf{and} \ ddl : \Omega]\!]^o & = & (\mathcal{D}[\![ddl]\!]^o)\{c{=}\delta_s\} \\
 & & \text{where } (c, \delta_s) = \mathcal{D}[\![dd]\!]^o
\end{array}$$

datadecl

$$\begin{array}{rcl}
\mathcal{D}[\![\mathcal{O} \vdash dd : (c, \Delta)]\!]^o & : & \mathrm{TypeID} \times \mathcal{D}_s[\![\Delta]\!]^o \\
\mathcal{D}[\![\mathcal{O} \vdash c = cdl : (c, \Delta)]\!]^o & = & (c, \sum_{C \in \mathrm{Dom}(\delta)} \delta(C)) \\
 & & \text{where } \delta = \mathcal{D}[\![cdl]\!]^o
\end{array}$$

condecl-list

$$\begin{array}{rcl}
\mathcal{D}[\![\mathcal{O}\vdash dcl : \Delta]\!]^{o} & : & \mathcal{D}[\![\Delta]\!]^{o} \\
\mathcal{D}[\![\mathcal{O}\vdash \cdot : \{\}]\!]^{o} & = & \{\} \\
\mathcal{D}[\![\mathcal{O}\vdash cd \parallel cdl : \Delta]\!]^{o} & = & \mathcal{D}[\![cdl]\!]^{o}\{C = v\} \\
 & & \text{where } (C, v) = \mathcal{D}[\![cd]\!]^{o}
\end{array}$$

condecl

$$\begin{array}{rcl}
\mathcal{D}[\![\mathcal{O}\vdash C\ \ ct : (C, \mathcal{V})]\!]^{o} & : & \text{ConID} \times \mathcal{D}[\![\mathcal{V}]\!]^{o} \\
\mathcal{D}[\![\mathcal{O}\vdash C\ \ ct : (C, \mathcal{V})]\!]^{o} & = & (C, \mathcal{D}[\![ct]\!]^{o})
\end{array}$$

con-tuple

$$\begin{array}{rcl}
\mathcal{D}[\![\mathcal{O}\vdash ct : \mathcal{V}]\!]^{o} & : & \mathcal{D}[\![V]\!]^{o} \\
\mathcal{D}[\![\mathcal{O}\vdash \langle\rangle : \{\}]\!]^{o} & = & \{\} \\
\mathcal{D}[\![\mathcal{O}\vdash ct\langle v : \tau\rangle : \mathcal{V}]\!]^{o} & = & \mathcal{D}[\![ct]\!]^{o}\{v = \mathcal{D}[\![\tau]\!]^{o}\}
\end{array}$$

fundecl-list

$$\begin{array}{rcl}
\mathcal{D}[\![\Omega, \Phi\vdash fdl : \Phi']\!]^{o} & : & \mathcal{D}[\![\Phi]\!]^{o} \rightarrow \mathcal{D}[\![\Phi']\!]^{o} \\
\mathcal{D}[\![\Omega, \Phi\vdash \cdot : \{\}]\!]^{o\phi} & = & \{\} \\
\mathcal{D}[\![\Omega, \Phi\vdash fd \ \mathsf{and}\ fdl : \{\}]\!]^{o\phi} & = & \mathcal{D}[\![fdl]\!]^{o\phi}\{f = F\} \\
 & & \text{where } (f, F) = \mathcal{D}[\![fd]\!]^{o\phi}
\end{array}$$

fundecl

$$\begin{array}{rcl}
\mathcal{D}[\![\Omega, \Phi\vdash fd : (\tau_1, \tau_2, \tau_3)]\!]^{o\phi} & : & \mathcal{D}[\![\tau_1]\!]^{o} \rightarrow \mathcal{D}[\![\tau_2]\!]^{o} \rightarrow \mathcal{D}[\![\tau_3]\!]^{o}_{\perp} \\
\mathcal{D}[\![\Omega, \Phi\vdash f\ \ x\ \ y : \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 = e : (\tau_1, \tau_2, \tau_3)]\!]^{o\phi} & = & \\
\multicolumn{3}{r}{\lambda X_1.\lambda X_2.\mathcal{D}[\![e]\!]^{o\phi}\{x = X_1, y = X_2\}}
\end{array}$$

exp

$$\begin{array}{rcl}
\mathcal{D}[\![\Omega, \Phi, \Gamma\vdash e : \tau]\!]^{o} & : & \mathcal{D}[\![\Phi]\!]^{o} \rightarrow \mathcal{D}[\![\Gamma]\!]^{o} \rightarrow \mathcal{D}[\![\tau]\!]^{o}{}_{\perp} \\
\mathcal{D}[\![\Omega, \Phi, \Gamma\vdash x : \tau]\!]^{o\phi}\rho & = & \text{up}(\rho(x)) \\
\mathcal{D}[\![\Omega, \Phi, \Gamma\vdash 0 : \mathsf{nat}]\!]^{o\phi}\rho & = & \text{up}(0) \\
\mathcal{D}[\![\Omega, \Phi, \Gamma\vdash \mathsf{ifz}(e, e_1, x.e_2) : \tau]\!]^{o\phi}\rho & = & (\psi)^{\dagger}\,(\mathcal{D}[\![\Omega, \Phi, \Gamma\vdash e : \mathsf{nat}]\!]^{o\phi}\rho) \\
\text{where } \psi(n) = & \Bigg\{ & \begin{array}{ll} \mathcal{D}[\![e_1]\!]^{o\phi}\rho & \text{if } n = 0 \\ \mathcal{D}[\![e_2]\!]^{o\phi}\rho\{x = n-1\} & \text{if } n > 0 \end{array} \\
\mathcal{D}[\![\Omega, \Phi, \Gamma\vdash f\ \ e_1\ \ e_2 : \tau]\!]^{o\phi}\rho & = & \\
\multicolumn{3}{r}{\Big(\lambda X_1.\,(\lambda X_2.\phi(f)\ \ X_1\ \ X_2)^{\dagger}\,(\mathcal{D}[\![e_2]\!]^{o\phi}\rho)\Big)^{\dagger}\,(\mathcal{D}[\![e_1]\!]^{o\phi}\rho)} \\
\mathcal{D}[\![\Omega, \Phi, \Gamma\vdash C\ \ at : c]\!]^{o\phi}\rho & = & (\text{in}_C())^{\dagger}(\mathcal{D}[\![at]\!]^{o\phi}\rho) \\
\mathcal{D}[\![\Omega, \Phi, \Gamma\vdash \mathsf{case}\ e\ \mathsf{of}\ ml : \tau]\!]^{o\phi}\rho & = & \\
\multicolumn{3}{r}{\Big(\lambda X.case(\mathcal{D}[\![\Omega, \Phi, \Gamma\vdash ml : c \Rightarrow \tau]\!]^{o\phi}\rho)(X)\Big)^{\dagger}\,(\mathcal{D}[\![e]\!]^{o\phi}\rho)} \\
\mathcal{D}[\![\Omega, \Phi, \Gamma\vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2]\!]^{o\phi}\rho & = & \\
\multicolumn{3}{r}{\Big(\lambda X.\mathcal{D}[\![e_2]\!]^{o\phi}\rho\{x = X\}\Big)^{\dagger}\,(\mathcal{D}[\![e_1]\!]^{o\phi}\rho)}
\end{array}$$

arg-tuple

$$\begin{array}{rcl}
\mathcal{D}[\![\Omega, \Phi, \Gamma \vdash at : \mathcal{V}]\!]^{o}\rho & : & \mathcal{D}[\![\Phi]\!]^{o} \to \mathcal{D}[\![\Gamma]\!]^{o} \to \mathcal{D}[\![\mathcal{V}]\!]^{o}_{\perp} \\
\mathcal{D}[\![\Omega, \Phi, \Gamma \vdash \langle\rangle : \{\}]\!]^{o\phi}\rho & = & \mathrm{up}(\{\}) \\
\mathcal{D}[\![\Omega, \Phi, \Gamma \vdash at\langle v = e\rangle : \mathcal{V}]\!]^{o\phi}\rho & = & \\
\multicolumn{3}{r}{\Big(\lambda X.\, (\lambda V.\mathrm{up}(V\{x{=}X\}))^{\dagger}\, (\mathcal{D}[\![at]\!]^{o\phi}\rho)\Big)^{\dagger}\, (\mathcal{D}[\![e]\!]^{o\phi}\rho)}
\end{array}$$

match-list

$$\begin{array}{rcl}
\mathcal{D}[\![\Omega, \Phi, \Gamma \vdash ml : c \Rightarrow \tau]\!]^{o} & : & \\
\multicolumn{3}{r}{\mathcal{D}[\![\Phi]\!]^{o} \to \mathcal{D}[\![\Gamma]\!]^{o} \to \prod_{C \in \mathrm{Dom}(o(c))} (o(c)(C) \to \mathcal{D}[\![\tau]\!]^{o}_{\perp})} \\
\mathcal{D}[\![\Omega, \Phi, \Gamma \vdash \cdot : c \Rightarrow \tau]\!]^{o\phi}\rho & = & \Psi \\
& & \text{where } \mathrm{Dom}(\Psi) = \mathrm{Dom}(o(c)) \text{ and} \\
& & \qquad \Psi(C) = \lambda X.\perp \\
\mathcal{D}[\![\Omega, \Phi, \Gamma \vdash m \;[\!]\; ml : c \Rightarrow \tau]\!]^{o\phi}\rho & = & (\mathcal{D}[\![ml]\!]^{o\phi}\rho)\{C{=}F\} \\
& & \text{where } (C, F) = \mathcal{D}[\![m]\!]^{o\phi}\rho
\end{array}$$

match

$$\begin{array}{l}
\mathcal{D}[\![\Omega, \Phi, \Gamma \vdash C\ mt \Rrightarrow\ e : c \Rightarrow \tau]\!]^{o} : \\
\qquad\qquad \mathcal{D}[\![\Phi]\!]^{o} \to \mathcal{D}[\![\Gamma]\!]^{o} \to \mathrm{ConID} \times (o(c)(C) \to \mathcal{D}[\![\tau]\!]^{o}_{\perp}) \\
\mathcal{D}[\![\Omega, \Phi, \Gamma \vdash C\ mt \Rrightarrow\ e : c \Rightarrow \tau]\!]^{o\phi}\rho = (C, \lambda V.\mathcal{D}[\![e]\!]^{o\phi}(\mathcal{D}[\![mt]\!]^{o}\rho V))
\end{array}$$

match-tuple

$$\begin{array}{rcl}
\mathcal{D}[\![\Gamma, \mathcal{V} \vdash mt : \Gamma']\!]^{o} & : & \mathcal{D}[\![\Gamma]\!]^{o} \to \mathcal{D}[\![\mathcal{V}]\!]^{o} \to \mathcal{D}[\![\Gamma']\!]^{o} \\
\mathcal{D}[\![\Gamma, \mathcal{V} \vdash \langle\rangle : \Gamma]\!]^{o}\rho\nu & = & \rho \\
\mathcal{D}[\![\Gamma, \mathcal{V} \vdash mt\langle v = x\rangle : \Gamma]\!]^{o}\rho\nu & = & (\mathcal{D}[\![mt]\!]^{o}\rho)\{x{=}\nu(v)\}
\end{array}$$

program

$$\begin{array}{rcl}
\mathcal{D}[\![\vdash p]\!] & : & \mathrm{N}_{\perp} \\
\mathcal{D}[\![\vdash \mathsf{datatype}\ ddl\ \mathsf{fun}\ fdl\ \ \mathsf{in}\ e]\!] & = & \mathcal{D}[\![e]\!]^{o\phi} \\
& & \text{where } o = \mathrm{fix}(\lambda o.\mathcal{D}[\![ddl]\!]^{o}) \\
& & \qquad\quad \phi = \mathrm{fix}(\lambda\phi.\mathcal{D}[\![fdl]\!]^{o\phi})
\end{array}$$

3.4 Comments

One can check that for any fixed element $o \in \mathcal{D}[\![\mathcal{O}]\!]$, and for any element of any of the syntactic categories, S, $\mathcal{D}[\![S]\!]^{o}$ actually has the "type" stated next to its definition. Whereas most of the above definitions are given using standard notation for writing continuous functions between domains (as in [Win93, section 8]), and as such the fixed point used to find the recursive closure of the meaning of the function definitions is known to exist, there is one exception.

The part that is a source of possible problems is the fixed point of the function $\mathcal{D}[\![ddl]\!]$, which has type $\mathcal{D}[\![\Omega]\!] \to \mathcal{D}[\![\Omega]\!]$. If $\mathcal{D}[\![\Omega]\!]$ had been a chain-complete partial order, we would just have used the usual fixed point theorems, but it is not even a set. It is defined to be a product of sets, and with no further restrictions, for all we know it belongs to something at least as "big" as the class of all sets.

We will not be using domain-theoretic methods then. Instead the meaning is purely set-theoretic. Since we are defining algebraic, not reflexive, data-types, the recursive set equations we need to solve are all defined "positively" (nothing that is being defined is used in a negative position, e.g., as the domain of a function space, since there are no function spaces at all), so we don't need the methodology of domains for this case. All we define are countable sets.

What we mean when we write "fix" is then the union of the chain of (products of) countable sets, which is ordered by inclusion. It is easy to show that given a product (over the associated index set) of sets, o, the denotation $\mathcal{D}[\![C_p]\!]o$ is again a product of sets (actually a product of sums of products of sets), since the only operations we use are products and disjoint sums of sets, which again define sets.

Iterating $\mathcal{D}[\![C_p]\!]$ on the empty set defines the chain ordered by inclusion:

$$\mathcal{D}[\![C_p]\!]^n(\prod_{c_\tau \in \mathrm{Dom}(\Omega_p)} \emptyset)$$

(i.e., $\mathcal{D}[\![C_p]\!]^n(\emptyset)$), and as unions over sets of sets are always defined

$$\bigcup_{n \in \mathrm{N}} \mathcal{D}[\![C_p]\!]^n(\emptyset)$$

is again a well-defined and countable set. The elements form a chain because product and sum construction are monotone in their arguments. We will let this be the definition of $\mathrm{fix}(\lambda o.\mathcal{D}[\![ddl]\!]^o)$.

4 The defunctionalizing transformation

The transformation we have been aiming at translates well-typed programs in the (annotated) source language to well-typed programs in the target language, while preserving the meaning of the program. The transformation turns all function declarations into constructions of a value containing information about the environment at the abstraction point. Applications of such values are transformed into case-constructs that decomposes the value and evaluates the appropriate function body in an environment corresponding to the environment in which the value was constructed. The transformation is given as a function of the typing derivations of a program in the source language.

In the remainder of the article, the program p is fixed, allowing us to write PROOF (l) as shorthand for PROOF (p) (l), and everywhere we write $\Gamma \vdash e : \tau$ it will not only mean that $\Gamma \vdash e : \tau$ is derivable, but also that the derivation of that judgment is part of the derivation of $\vdash p$, i.e., the judgment is in the image of PROOF (p). Also, we choose two identifiers, call them *ff* and *xx*, that do not occur in p.

4.1 Definition of the transformation

The functions are defined using the usual shorthand for functions on derivations, and they even abbreviate the argument. Instead of writing $\mathcal{T}[\Gamma \vdash e : \tau]$ we will leave out everything except e if it corresponds verbatim to the conclusion of the rule schema. We create index identifiers from normal identifiers by underlining, representing some injective mapping from normal identifiers into the set of index identifiers.

First, we transform the expressions.

$$\begin{array}{rcl}
\mathcal{T}[\tau] & : & \textbf{type} \\
\mathcal{T}[\mathsf{nat}] & = & \mathsf{nat} \\
\mathcal{T}[\tau_1 \rightarrow \tau_2] & = & c_{\tau_1 \rightarrow \tau_2} \\
\\
\mathcal{T}_v\,[env] & : & \textbf{arg-tuple} \\
\mathcal{T}_v\,[\langle\rangle] & = & \langle\rangle \\
\mathcal{T}_v\,[env\langle x : \tau\rangle] & = & \mathcal{T}_v\,[env]\langle \underline{x} = x\rangle \\
\\
\mathcal{T}[e] & : & \textbf{exp} \\
\mathcal{T}[x] & = & x \\
\mathcal{T}[0] & = & 0 \\
\mathcal{T}[\mathsf{succ}(e)] & = & \mathsf{succ}(\mathcal{T}[e]) \\
\mathcal{T}[\mathsf{ifz}(e, e_1, x.e_2)] & = & \mathsf{ifz}(\mathcal{T}[e]\,, \mathcal{T}[e_1]\,, x.\mathcal{T}[e_2]) \\
\mathcal{T}[(\mathsf{fun}\ f\ x : \tau_1 \rightarrow \tau_2 = e)_l^{env}] & = & C_l\ \mathcal{T}_v\,[env] \\
\mathcal{T}[(e_1\ e_2)] & = & app_{\tau_1 \rightarrow \tau_2}\ \mathcal{T}[\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2]\ \mathcal{T}[e_2]
\end{array}$$

Then we build the data-type declarations, C_p, and function declarations,

F_p.

In the following we use a translation from finite sets to lists, listof(S), where lists are either *nil* or $x :: xs$ where xs is again a list. It is not important how this translation works, but to make it a function, let us just decide that it gives us the sorted list of the set elements with regard to some total ordering. The two types of sets we work on are sets of types in the source language (elements of the syntactic category τ) and labels (ranged over by l). We can impose a total order on the syntax of types by, e.g., saying that nat is less than everything else, and $\tau_1 \rightarrow \tau_2$ is less than $\tau_1' \rightarrow \tau_2'$ if τ_2 is less than τ_2' or $\tau_2 = \tau_2'$ and τ_1 is less than τ_1'.

That is, we can define listof(S) as

$$\begin{array}{rcll} \mathrm{listof}\,(\emptyset) & = & \mathit{nil} & \\ \mathrm{listof}\,(S) & = & (\min(S)) :: \mathrm{listof}\,(S \setminus \{\min(S)\}) & \text{if } S \neq \emptyset \end{array}$$

as any finite and totally ordered set has a least element.

We will say that y *member of* $(x :: xs)$ if $x = y$ or y *member of* xs, and then we have the property that $x \in S \iff x$ *member of* listof(S).

Using this way of ordering our types, we can define the datatype declarations corresponding to functions in the source program in a fixed order.

$$\begin{array}{rcl} C_p & : & \textbf{datadecl-list} \\ C_p & = & \mathcal{C}_{\mathrm{c}}[\mathrm{listof}\,(\{\tau \,|\, \tau \in \mathrm{typesof}(p)\,\})] \\ & & \\ \mathcal{C}_{\mathrm{c}}[\ldots] & : & \text{type list} \rightarrow \textbf{datadecl-list} \\ \mathcal{C}_{\mathrm{c}}[\mathit{nil}] & = & \cdot \\ \mathcal{C}_{\mathrm{c}}[\tau :: tl] & = & c_\tau = \mathcal{C}_{\mathrm{C}}[\mathrm{listof}\,(\{l \,|\, \mathrm{PROOF}\,(l) = D \wedge \mathrm{isfun}(D) \wedge \mathrm{typeof}(D) = \tau\,\})] \\ & & \mathsf{and}\ \mathcal{C}_{\mathrm{c}}[tl] \\ & & \\ \mathcal{C}_{\mathrm{C}}[\ldots] & : & \text{label list} \rightarrow \textbf{condecl-list} \\ \mathcal{C}_{\mathrm{C}}[\mathit{nil}] & = & \cdot \\ \mathcal{C}_{\mathrm{C}}[l :: ls] & = & C_l\ \mathcal{C}_{\mathrm{v}}[\mathrm{envof}(\mathrm{PROOF}\,(l))]\ \| \mathcal{C}_{\mathrm{C}}[ls] \\ & & \\ \mathcal{C}_{\mathrm{v}}[\ldots] & : & \mathit{env} \rightarrow \textbf{contuple} \\ \mathcal{C}_{\mathrm{v}}[\langle\rangle] & = & \langle\rangle \\ \mathcal{C}_{\mathrm{v}}[ct\langle x : \tau\rangle] & = & \mathcal{C}_{\mathrm{v}}[ct]\langle \underline{x} : \mathcal{T}[\tau]\rangle \end{array}$$

These functions construct a list of data-type declarations. We define one for each function type occurring in the program. The declarations of the associated application functions are generated as follows.

$$
\begin{array}{rcl}
F_p & : & \textbf{fundecl-list} \\
F_p & = & \mathcal{F}_{\mathrm{c}}[\mathrm{listof}\,(\{\tau\,|\tau \in \mathrm{typesof}(p)\,\})]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{F}_{\mathrm{c}}[\ldots] & : & \text{type list} \rightarrow \textbf{fundecl-list} \\
\mathcal{F}_{\mathrm{c}}[nil] & = & \cdot \\
\mathcal{F}_{\mathrm{c}}[\tau_1 \rightarrow \tau_2 :: tl] & = & app_{\tau_1 \rightarrow \tau_2}\ \mathit{ff}\ xx : \mathcal{T}[\tau_1 \rightarrow \tau_2] \rightarrow \mathcal{T}[\tau_1] \rightarrow \mathcal{T}[\tau_2] = \\
 & & \quad \mathsf{case}\ \mathit{ff} \\
 & & \qquad \mathsf{of}\ \mathcal{F}_{\mathrm{C}}\left[\mathrm{listof}\left(\left\{ l \,\middle|\, \begin{array}{l} \mathrm{PROOF}\,(l) = D \wedge \mathrm{isfun}(D) \\ \wedge\ \mathrm{typeof}(D) = \tau \end{array} \right\}\right)\right] \\
 & & \quad \mathsf{and}\ \mathcal{F}_{\mathrm{c}}[tl]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{F}_{\mathrm{C}}[\ldots] & : & \text{label list} \rightarrow \textbf{match-list} \\
\mathcal{F}_{\mathrm{C}}[nil] & = & \cdot \\
\mathcal{F}_{\mathrm{C}}[l :: ls] & = & C_l\ \ (\mathcal{F}_{\mathrm{v}}[env]) \Rrightarrow (\mathsf{let}\ f = \mathit{ff}\ \mathsf{in}\ \mathsf{let}\ x = xx\ \mathsf{in}\ \mathcal{T}[e]) \parallel \mathcal{F}_{\mathrm{C}}[ls] \\
 & & \text{where } \mathrm{PROOF}\,(l) = \Gamma \vdash (\mathsf{fun}\ f\ x : \tau_1 \rightarrow \tau_2 = e)_l^{env} : \tau_1 \rightarrow \tau_2
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{F}_{\mathrm{v}}[\ldots] & : & env \rightarrow \textbf{match-tuple} \\
\mathcal{F}_{\mathrm{v}}[\langle\rangle] & = & \langle\rangle \\
\mathcal{F}_{\mathrm{v}}[vt\langle x : \tau\rangle] & = & \mathcal{F}_{\mathrm{v}}[vt]\langle \underline{x} = x\rangle
\end{array}
$$

where ff and xx are “fresh” variables, i.e., variables that does not appear in the program, p, at all.

Finally, we transform the program by adding the datatype- and function declarations we constructed above to the transformation of the main expression.

$$\mathcal{T}[\mathsf{Program}\ e] = \mathsf{datatype}\ C_p\ \mathsf{fun}\ F_p\ \ \mathsf{in}\ \mathcal{T}[e]$$

4.2 Comments on the transformation

Notice that we create data-types for any type appearing in the program (typesof(p) is any type, other than nat, mentioned in the program, i.e., as either domain, codomain, or function type of a function declaration), but not all of them will have constructors. In the case

$$
\begin{array}{l}
\mathsf{Program}((\mathsf{fun}\ f\ x : ((\mathsf{nat} \rightarrow \mathsf{nat}) \rightarrow \mathsf{nat}) \rightarrow \mathsf{nat} = 0)_l^{\langle\rangle} \\
\qquad\qquad (\mathsf{fun}\ g\ y : (\mathsf{nat} \rightarrow \mathsf{nat}) \rightarrow \mathsf{nat} = (y\ 0))_{l'}^{\langle\rangle})_{l''}
\end{array}
$$

we have $\mathsf{nat} \rightarrow \mathsf{nat}$ occurring in the program, so we need to define $c_{\mathsf{nat} \rightarrow \mathsf{nat}}$ in the translated program. Otherwise the translation of the above program would not type check, since the translation of $\mathsf{nat} \to \mathsf{nat}$ would not be a defined type. In this program there are no values of type $\mathsf{nat} \to \mathsf{nat}$, and there will not be any constructors of type $c_{\mathsf{nat} \to \mathsf{nat}}$ in the translated program either. There is also an application-point where the function has type $\mathsf{nat} \to \mathsf{nat}$, so we define a function $app_{\mathsf{nat} \to \mathsf{nat}}$ with a case-expression with an empty **match-list**.

5 Type preservation

The translation given in the preceding section translates programs in the source language into programs in the target language. In order for the translation to preserve the semantics of programs, it necessarily have to generate only well-typed programs, since the semantics of a program in the target language is undefined unless the program is well-type. This section shows that the translation generates only well-typed programs.

Formally, since we have fixed a program p, we need to prove that if $\vdash \mathsf{Program}\ e$ then $\vdash \mathsf{datatype}\ C_p\ \mathsf{fun}\ F_p\ \ \mathsf{in}\ \mathcal{T}[e]$, where C_p and F_p are as defined by the translation.

To this end we show the premises of this rule: There exists Ω and Φ s.t. $\mathrm{Dom}(\Omega) \vdash C_p : \Omega$, $\Omega, \Phi \vdash F_p : \Phi$, and $\Omega, \Phi, \{\} \vdash \mathcal{T}[e] : \mathsf{nat}$.

The following sections define such Ω_p and Φ_p and show that they satisfy the requirements.

5.1 Defining solutions to typing judgments

To prove the existence of Ω and Φ satisfying the premises for the well-typedness of a program, we define two such mappings.

Define Ω_p by

- $\mathrm{Dom}(\Omega_p) = \{c_\tau \,|\, \tau \in \mathrm{typesof}(p)\}$
- $\mathrm{Dom}(\Omega_p(c_\tau)) = \{C_l \,|\, \mathrm{PROOF}\,(l) = D \wedge \mathrm{isfun}(D) \wedge \mathrm{typeof}(D) = \tau\}$
- $\mathrm{Dom}(\Omega_p(c_\tau)(C_l)) = \{\underline{x} \,|\, x \in \mathrm{Dom}(\mathrm{envof}(\mathrm{PROOF}\,(l)))\}$
- $\Omega_p(c_\tau)(C_l)(\underline{x}) = \mathcal{T}[\mathrm{envof}(\mathrm{PROOF}\,(l)\,(x))]$

The above rules uniquely define Ω_p, since it gives both the domain of each indexed product as well as the value at each projection. It defines an object of the right "type" for an Ω (an environment from type names to environments from constructor names to environments from label names to types), and we can also prove that it satisfies $\mathrm{Dom}(\Omega_p) \vdash C_p : \Omega_p$,

Define Φ_p by

- $\mathrm{Dom}(\Phi_p) = \left\{ app_{\tau_1 \to \tau_2} \;\middle|\; \begin{array}{l} \mathrm{PROOF}(l) = D \wedge \mathrm{isappl}(D) \\ \wedge\ \mathrm{funtype}(D, \tau_1 \to \tau_2) \end{array} \right\}$
  where
  $$\begin{array}{rcl} \mathrm{funtype}(D,\tau) & = & (\mathrm{isappl}(D) \wedge \mathrm{typeof}(\mathrm{funpart}(D)) = \tau) \vee \\ & & (\mathrm{isfun}(D) \wedge \mathrm{typeof}(D) = \tau) \end{array}$$
- $\Phi_p(app_{\tau_1 \to \tau_2}) = (\mathcal{T}[\tau_1 \to \tau_2], \mathcal{T}[\tau_1], \mathcal{T}[\tau_2])$

Again the above defines something of the right type (environment from function-names to type triples), and we can prove that $\Omega_p, \Phi_p \vdash F_p : \Phi_p$.

Most of the proofs will be omitted from this article, since the main point is the semantic equivalence, not the preservation of typability, and the proofs are not themselves technically challenging. Also, Bell et al. [BBH97] have already shown that their translation preserves typability, so type preservation should not come as a surprise.

5.2 Data-type declarations are well-typed

The construction of Ω_p allows us to prove

$$\mathrm{Dom}(\Omega_p) \vdash C_p : \Omega_p$$

by induction on the definition of C_p. It is a simple proof by unfolding the definition and checking each part, with induction proofs over the lists that are arguments to $\mathcal{C}_{\mathrm{c}}[\ldots]$ and $\mathcal{C}_{\mathrm{C}}[\ldots]$. The proof has been omitted for brevity.

5.3 Expressions are well-typed

We can prove that if $\Gamma \vdash e : \tau$, then $\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}[e] : \mathcal{T}[\tau]$ for any Γ' that extends Γ, where "extends" is defined as follows.

Definition 2 Γ' *extends* Γ
We will say that a type assignment in the target language Γ' extends *a type assignment in the source language* Γ*, if* $\mathrm{Dom}(\Gamma) \subseteq \mathrm{Dom}(\Gamma')$ *and* $\forall x \in \mathrm{Dom}(\Gamma).\Gamma'(x) = \mathcal{T}[\Gamma(x)]$.

We prove that for any sub-derivation, $\Gamma \vdash e : \tau$, of the derivation of p that

$$(\Gamma \vdash e : \tau \wedge \Gamma' \textit{ extends } \Gamma) \Rightarrow \Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}[\Gamma \vdash e : \tau] : \mathcal{T}[\tau]$$

Proof.

The proof is by structural induction on the structure of the derivation (or, equivalently, on the structure of the expression e). The details can be seen in Appendix A.

■

5.4 Function declarations are well-typed

The last premise to show is

$$\Omega_p, \Phi_p \vdash F_p : \Phi_p$$

Showing this is done using the same methods as for Ω_p, i.e., by showing that it works for any part of Φ_p, expanding its definition, and using inductions over the list-arguments to $\mathcal{F}_{\mathrm{c}}[\ldots]$ and $\mathcal{F}_{\mathrm{C}}[\ldots]$. There is nothing subtle in this proof either, and it too has been omitted for brevity.

5.5 Summary and conclusion

We have shown that if the source program, $p = \mathsf{Program}\ e$, is well-typed, and if Ω_p and Φ_p are chosen as described then $\mathrm{Dom}(\Omega_p) \vdash C_p : \Omega_p$, $\Omega_p, \Phi_p \vdash F_p : \Phi_p$, and $\Omega_p, \Phi_p, \{\} \vdash \mathcal{T}[e] : \mathcal{T}[\mathsf{nat}]$. From these premises, we can infer that $\vdash \mathsf{datatype}\ C_p\ \mathsf{fun}\ F_p\ \ \mathsf{in}\ \mathcal{T}[e]$, i.e., that $\vdash \mathcal{T}[p]$.

This result guarantees that the denotation of the translated program is defined. In the next section we show that the translation preserves the meaning of any terminating program.

6 Meaning preservation

In this section we show that the transformation is weakly correct, i.e., *if* the original program terminates (evaluates to a non-bottom value), *then* so does the translated program, and with the same value. It makes sense to use equality as the desired relation between the results, since the denotation of both source and destination programs have the same type: lifted natural numbers.

The proof will be by logical relations as described in, e.g., Mitchell's textbook [Mit93].

We consider a single given program, so C_p and F_p are given. In the following we will let $o \overset{\mathrm{def}}{=} \mathrm{fix}(\lambda o.\mathcal{D}[\![C_p]\!]^o)$ and $\phi \overset{\mathrm{def}}{=} \mathrm{fix}(\lambda\phi.\mathcal{D}[\![F_p]\!]^{o\phi})$.

The following subsections relate the functions in ϕ to the functions in the original program, define the logical relation, prove that the translation of any expression gives expressions that are related by the logical relation, and finally extend this result to entire programs.

6.1 Properties of the semantics of the program

We need a few lemmas for the main proof.

Lemma 1 *If* $PROOF(l) = \Gamma \vdash (\mathsf{fun}\ f\ x : \tau_1 \to \tau_2 = e)^{env} : \tau_1 \to \tau_2$ *and* Γ' *extends* Γ *and* $\rho' \in \mathcal{D}[\![\Gamma']\!]^o$ *then*

$$\mathcal{D}[\![\mathcal{T}_v\,[env]]\!]^{o\phi}\rho' = \mathrm{up}(\nu)$$

where $\mathrm{Dom}(\nu) = \{\underline{x}\,|x \in \mathrm{Dom}(\Gamma)\}$ *and* $\nu(\underline{x}) = \rho'(x)$.

Proof of Lemma 1 (sketched). Simple induction on the length of *env*, since $\mathcal{T}_v\,[env\langle x : t\rangle] = \mathcal{T}_v\,[env]\langle \underline{x} = x\rangle$ so all expressions are variables, and as such cannot denote bottom, and then

$$\mathcal{D}[\![\mathcal{T}_v\,[env]\langle \underline{x} = x\rangle]\!]^{o\phi}\rho' = ((\lambda E.\mathrm{up}(E\{\underline{x}{=}\rho'(x)\})))^{\dagger}\,(\mathcal{D}[\![\mathcal{T}_v\,[env]]\!]^{o\phi}\rho')$$

■

Lemma 2 *If* $PROOF(l) = \Gamma \vdash (\mathsf{fun}\ f\ x : \tau_1 \to \tau_2 = e)^{env} : \tau_1 \to \tau_2$ *and* Γ' *extends* Γ *and* $\rho' \in \mathcal{D}[\![\Gamma']\!]^o$ *then*

$$\phi(app_{\tau_1\to\tau_2})(in_{C_l}(\nu))b = \mathcal{D}[\![\mathcal{F}_{\mathrm{C}}[ll]]\!]^{o\phi}\{f\!f{=}\,in_{C_l}(\nu), xx = b\}(C_l)(\nu)$$

where $\mathrm{up}(\nu) = \mathcal{D}[\![\mathcal{T}_v\,[env]]\!]^{o\phi}\rho'$ *(which by Lemma 1 isn't bottom) and*

$$ll = listof(\{l\,|PROOF(l) = D \wedge isfun(D) \wedge typeof(D) = \tau_1 \to \tau_2\})$$

Proof of Lemma 2. By definition of being a fixed point $\phi(app_{\tau_1\to\tau_2}) = \mathcal{D}[\![F_p]\!]^{o\phi}(app_{\tau_1\to\tau_2})$, and by simple induction on the way F_p is constructed it follows that

$$\mathcal{D}[\![F_p]\!]^{o\phi}(app_{\tau_1\to\tau_2}) = \lambda F.\lambda X.\mathcal{D}[\![\mathsf{case}\ f\!f\ \mathsf{of}\ \mathcal{F}_{\mathrm{C}}[ll]]\!]^{o\phi}\{f\!f{=}\,F, xx = X\}$$

where $ll = \mathrm{listof}\,(\{l\,|\mathrm{PROOF}\,(l) = D \wedge \mathrm{isfun}(D) \wedge \mathrm{typeof}(D) = \tau_1 \to \tau_2\})$.

If we let $g = \mathrm{in}_{C_l}(\nu)$ (remember that $\mathrm{in}_{C_l}(\nu) = (C_l, \nu)$) then this allows us to show

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{D}[\![F_p]\!]^{o\phi}(app_{\tau_1\to\tau_2})(g)(b)} \\
= & \mathcal{D}[\![\textsf{case}\ \mathit{ff}\ \textsf{of}\ \mathcal{F}_{\mathrm{C}}[ll]]\!]^{o\phi}\{\mathit{ff} = g, xx = b\} \\
= & \Big(\lambda X.case(\mathcal{D}[\![\mathcal{F}_{\mathrm{C}}[ll]]\!]^{o\phi}\{\mathit{ff} = g, xx = b\})(X)\Big)^{\dagger}(\mathcal{D}[\![\mathit{ff}]\!]^{o\phi}\{\mathit{ff} = g, xx = b\}) \\
= & case(\mathcal{D}[\![\mathcal{F}_{\mathrm{C}}[ll]]\!]^{o\phi}\{\mathit{ff} = g, xx = b\})(g) \\
= & (\mathcal{D}[\![\mathcal{F}_{\mathrm{C}}[ll]]\!]^{o\phi}\{\mathit{ff} = g, xx = b\})(\pi_1(g))(\pi_2(g)) \\
= & (\mathcal{D}[\![\mathcal{F}_{\mathrm{C}}[ll]]\!]^{o\phi}\{\mathit{ff} = g, xx = b\})(C_l)(\nu)
\end{array}$$

as needed. ■

Lemma 3 *If PROOF*$(l) = \Gamma \vdash (\textsf{fun}\ f\ x : \tau_1 \to \tau_2 = e)^{env} : \tau_1 \to \tau_2$, $\Gamma \vdash env$, Γ' *extends* Γ, $\rho' \in \mathcal{D}[\![\Gamma']\!]^o$ *and* ν *satisfies that* $\mathrm{Dom}(\nu) = \{\underline{x}\,|x \in \mathrm{Dom}(\Gamma)\}$ *and* $\nu(\underline{x}) = \rho'(x)$ *then*

$$\mathcal{D}[\![\mathcal{F}_{\mathrm{v}}[env]]\!]^{o\phi}\{\mathit{ff} = g, xx = b\}(\nu) = \rho''$$

implies $\mathrm{Dom}(\rho'') = \mathrm{Dom}(\Gamma) \cup \{\mathit{ff}, xx\}$ *and* $\forall x \in \mathrm{Dom}(\Gamma).\rho''(x) = \rho'(x)$, $\rho''(\mathit{ff}) = g$, *and* $\rho'(xx) = b$.

Proof of Lemma 3. The proof is (indirectly) by induction on the structure of *env*. The actual induction is on the size of $\mathrm{Dom}(\Gamma)$, which relates to *env* as we know $\Gamma \vdash env$ by inversion.

The details can be seen in Appendix B. ■

Lemma 4 *If PROOF*$(l) = \Gamma \vdash (\textsf{fun}\ f\ x : \tau_1 \to \tau_2 = e)^{env} : \tau_1 \to \tau_2$, Γ' *extends* Γ, $\rho' \in \mathcal{D}[\![\Gamma']\!]^o$ *and* $\mathrm{up}(\nu) = \mathcal{D}[\![\mathcal{T}_v\,[env]]\!]^{o\phi}\rho'$, *and if l member of ll then*

$$\mathcal{D}[\![\mathcal{F}_{\mathrm{C}}[ll]]\!]^{o\phi}\{\mathit{ff} = g, xx = b\}(C_l)(\nu) = \mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho''$$

where $\mathrm{Dom}(\rho'') = \mathrm{Dom}(\Gamma) \cup \{\mathit{ff}, xx, f, x\}$ *and* $\forall x \in \mathrm{Dom}(\Gamma) \setminus \{f, x\}.\rho''(x) = \rho'(x)$, $\rho''(\mathit{ff}) = \rho''(f) = g$, *and* $\rho'(xx) = \rho''(x) = b$.

Proof of Lemma 4. If *l member of ll* then we can easily show by induction on the list that

$$\begin{gathered}
\mathcal{D}[\![\mathcal{F}_{\mathrm{C}}[ll]]\!]^{o\phi}\{\mathit{ff} = g, xx = b\}(C_l)(\nu) \\
= \\
\mathcal{D}[\![\textsf{let}\ f = \mathit{ff}\ \textsf{in}\ \textsf{let}\ x = xx\ \textsf{in}\ \mathcal{T}[e]]\!]^{o\phi}(\mathcal{D}[\![\mathcal{F}_{\mathrm{v}}[env]]\!]^{o}\{\mathit{ff} = g, xx = b\}(\nu))
\end{gathered}$$

By Lemma 3 we know that

$$\mathcal{D}[\![\mathcal{F}_{\mathrm{V}}[env]]\!]^{o\phi}\{ff = g, xx = b\}(\nu) = \rho''$$

such that $\mathrm{Dom}(\rho'') = \mathrm{Dom}(\Gamma) \cup \{ff, xx\}$ and $\rho''(xx) = b$, $\rho''(ff) = g$, and for $x \in \mathrm{Dom}(\Gamma)$ we have $\rho''(x) = \rho'(x)$.

By expanding the definitions we can see that

$$\begin{array}{ll} \multicolumn{2}{l}{\mathcal{D}[\![\mathsf{let}\ f = ff\ \mathsf{in}\ \mathsf{let}\ x = xx\ \mathsf{in}\ \mathcal{T}[e]]\!]^{o\phi}\rho''} \\ = & \left(\lambda X.\mathcal{D}[\![\mathsf{let}\ x = xx\ \mathsf{in}\ \mathcal{T}[e]]\!]^{o\phi}\rho''\{f{=}X\}\right)^{\dagger}(\mathcal{D}[\![ff]\!]^{o\phi}\rho'') \\ = & \mathcal{D}[\![\mathsf{let}\ x = xx\ \mathsf{in}\ \mathcal{T}[e]]\!]^{o\phi}\rho''\{f{=}g\} \\ = & \mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho''\{f{=}g\}\{x{=}b\} \end{array}$$

and $\rho''\{f{=}g\}\{x{=}b\}$ exactly satisfies the requirements of the lemma. ■

Theorem 1 *If* $\Gamma \vdash (\mathsf{fun}\ f\ x : \tau_1 \to \tau_2 = e)_l^{env} : \tau_1 \to \tau_2$*,* Γ' *extends* Γ*,* $\rho \in \mathcal{D}[\![\Gamma']\!]^o$*, and* $\mathrm{up}(\nu) = \mathcal{D}[\![\mathcal{T}_v\,[env]]\!]^{o\phi}\rho'$ *then*

$$\phi(app_{\mathcal{T}})(in_{C_l}(\nu))b = \mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho''$$

where $\mathrm{Dom}(\rho'') = \mathrm{Dom}(\rho) \cup \{ff, xx, x, f\}$ *and* $\rho''(x) = \rho''(xx) = b$*,* $\rho''(f) = \rho''(ff) = in_{C_l}(\nu)$*, and* $y \in \mathrm{Dom}(\Gamma) \setminus \{f, x\} \Rightarrow \rho''(y) = \rho'(y)$*.*

Proof of Theorem 1. By Lemma 1 we know that if $\mathrm{up}(\nu) = \mathcal{D}[\![\mathcal{T}_v\,[env]]\!]^{o\phi}\rho'$ then $\forall x \in \mathrm{Dom}(\Gamma).\nu(\underline{x}) = \rho'(x)$, so we can use Lemma 2, from which we know that

$$\phi(app_{\mathcal{T}})(\mathrm{in}_{C_l}(\nu))b = \mathcal{D}[\![\mathcal{F}_{\mathrm{C}}[ll]]\!]^{o\phi}\{ff = \mathrm{in}_{C_l}(\nu), xx = b\}(C_l)(\nu)$$

By Lemma 4 we know that

$$\mathcal{D}[\![\mathcal{F}_{\mathrm{C}}[ll]]\!]^{o\phi}\{ff = \mathrm{in}_{C_l}(\nu), xx = b\}(C_l)(\nu) = \mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho''$$

where ρ'' is exactly the environment claimed in the theorem. ■

6.2 Definition of the logical relation

We define the type-indexed families of relations $\prec_\tau$ and $\preceq_\tau$ between $\mathcal{S}[\![\tau]\!]$ and $\mathcal{D}[\![\mathcal{T}[\tau]]\!]^o$, and between $\mathcal{S}[\![\tau]\!]_\perp$ and $\mathcal{D}[\![\mathcal{T}[\tau]]\!]^o{}_\perp$ respectively, as given in Figure 6.

$$\begin{array}{rll} n \prec_{\mathsf{nat}} m & \mathit{iff} & n = m \\ f \prec_{\tau_1 \to \tau_2} g & \mathit{iff} & \forall a \prec_{\tau_1} b. f a \preceq_{\tau_2} \phi(app_{\tau_1 \to \tau_2}) g b \\ x \preceq_{\tau} y & \mathit{iff} & x = \bot \vee (\exists x', y'. x = \mathrm{up}(x') \wedge y = \mathrm{up}(y') \wedge x' \prec_{\tau} y') \end{array}$$

Figure 6: The logical relation

These relations satisfy that for any $g \in \mathcal{D}[\![\mathcal{T}[\tau_1 \to \tau_2]]\!]^o$ and any $y \in \mathcal{D}[\![\mathcal{T}[\tau]]\!]^o{}_\bot$ the relations over $\mathcal{S}[\![\tau_1 \to \tau_2]\!]$ and $\mathcal{S}[\![\tau]\!]_\bot$, defined as $\{f \,|\, f \prec_{\tau_1 \to \tau_2} g\}$ and $\{x \,|\, x \preceq_\tau y\}$, are *inclusive*. That is, the relations contains the bottom of the corresponding domains and they are closed under least upper bounds of ω-chains.

This can be shown by well-founded induction on the structure of types and the dependencies between the relations, and noticing that the relations are all constructed as intersections and inverse images by strict continuous functions of smaller relations, which according to [Win93, Chapter 10] guarantees that the relations are inclusive if the smaller ones are. The base case will be $\preceq_{\mathsf{nat}}$, which follows directly, since the *discrete* set N is closed under least upper bound of ω-chains.

We further define a family of relations indexed by type assignments such that if Γ' *extends* Γ and $\rho \in \mathcal{S}[\![\Gamma]\!]$ and $\rho' \in \mathcal{D}[\![\Gamma']\!]^o$ then

$$\rho \prec_\Gamma \rho' \quad \mathit{iff} \quad \forall x \in \mathrm{Dom}(\Gamma). \rho(x) \prec_{\Gamma(x)} \rho'(x)$$

6.3 Translations of expressions

Now, we can show that the values of expressions are related to the values of the translated expression, which then directly implies the same for programs.

Lemma 5 *For any expression e, if $PROOF(l) = \Gamma \vdash e : \tau$ and $\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}[e] : \mathcal{T}[\tau]$ where Γ' extends Γ then for all $\rho \in \mathcal{S}[\![\Gamma]\!]$ and $\rho' \in \mathcal{D}[\![\Gamma']\!]^o$*

$$\rho \prec_\Gamma \rho' \Rightarrow \mathcal{S}[\![e]\!]\rho \preceq_\tau \mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi} \rho'$$

Proof of Lemma 5.

The proof is by structural induction on the structure of expressions.

The interesting cases are abstraction and application:

case $e = (\mathsf{fun}\ f\ x : \tau_1 \rightarrow \tau_2 = e')_l^{env}$**:** As defined

$$\mathcal{S}[\![e]\!]\rho = \mathrm{up}(\mathrm{fix}(\lambda F.\lambda X.\mathcal{S}[\![e]\!]\rho\{f = F\}\{x = X\}))$$

and

$$\mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho' = (\lambda v.\mathrm{up}(\mathrm{in}_{C_l}(v)))^{\dagger}\,(\mathcal{D}[\![\mathcal{T}_v\,[env]]\!]^{o\phi}\rho')$$

and we know from Lemma 1 that $(\mathcal{D}[\![\mathcal{T}_v\,[env]]\!]^{o\phi}\rho') = \mathrm{up}(\nu)$ for some ν, i.e., $\mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho' = \mathrm{up}(\mathrm{in}_{C_l}(\nu))$.

To show that these are $\preceq_{\tau_1 \rightarrow \tau_2}$-related we use fixed-point induction over the inclusive set

$$S \stackrel{\mathrm{def}}{=} \{f \,|\, f \prec_{\tau_1 \rightarrow \tau_2} \mathrm{in}_{C_l}(\nu)\}$$

to show that $\mathrm{fix}(\lambda F.\lambda X.\mathcal{S}[\![e]\!]\rho\{f = F\}\{x = X\})$ belongs to it. That is, we must show that the set is closed under the function

$$\mathcal{F} \stackrel{\mathrm{def}}{=} \lambda F.\lambda X.\mathcal{S}[\![e]\!]\rho\{f = F\}\{x = X\}$$

Let p be any element of S. Then

$$\mathcal{F}(p) \prec_{\tau_1 \rightarrow \tau_2} \mathrm{in}_{C_l}(\nu)$$

since for any $a \prec_{\tau_1} b$

$$\mathcal{F}(p)(a) = \mathcal{S}[\![e']\!]\rho\{f = p\}\{x = a\}$$

and from Theorem 1 we know that

$$\phi(app_{\tau_1 \rightarrow \tau_2})(\mathrm{in}_{C_l}(\nu))b = \mathcal{D}[\![\mathcal{T}[e']]\!]^{o\phi}\rho''$$

where $\rho\{f = p\}\{x = a\} \prec_{\Gamma} \rho''$ necessarily by the requirements on ρ'' (in combination the two properties, $\rho \prec_{\Gamma} \rho'$ and $\forall x \in \mathrm{Dom}(\Gamma).\rho''(x) = \rho'(x)$, implies $\rho \prec_{\Gamma} \rho''$)

By induction hypothesis (e' is smaller than e) the denotation of these expressions are related, i.e. $\mathcal{F}(p)(a) \preceq_{\tau_2} app_{\tau_1 \rightarrow \tau_2}(\mathrm{in}_{C_l}(v))(b)$ which proves that $\mathcal{F}(p) \prec_{\tau_1 \rightarrow \tau_2} \mathrm{in}_{C_l}(v)$.

Since $p \in S \Rightarrow \mathcal{F}(p) \in S$, and S is inclusive, we can conclude that $\mathrm{fix}(\mathcal{F}) \in S$, i.e.

$$\mathrm{fix}(\mathcal{F}) \prec_{\tau_1 \rightarrow \tau_2} \mathrm{in}_{C_l}(v)$$

We then conclude that

$$\mathrm{up}(\mathrm{fix}(\mathcal{F})) \preceq_{\tau_1 \rightarrow \tau_2} \mathrm{up}(\mathrm{in}_{C_l}(v))$$

as needed.

case $e = (e_1\ e_2)$**:** By induction hypothesis we know that

$$\mathcal{S}[\![e_1]\!]\rho \preceq_{\tau_1 \to \tau_2} \mathcal{D}[\![\mathcal{T}[e_1]]\!]^{o\phi}\rho'$$

and

$$\mathcal{S}[\![e_2]\!]\rho \preceq_{\tau_1} \mathcal{D}[\![\mathcal{T}[e_2]]\!]^{o\phi}\rho'$$

Also, by definition,

$$\mathcal{S}[\![(e_1\ e_2)]\!]\rho = \Big(\lambda f.\,(\lambda x.fx)^{\dagger}\,(\mathcal{S}[\![e_2]\!]\rho)\Big)^{\dagger}\,(\mathcal{S}[\![e_1]\!]\rho)$$

and

$$\begin{aligned}\mathcal{D}[\![app_{\tau_1 \to \tau_2}\ \mathcal{T}[e_1]\ \ \mathcal{T}[e_2]]\!]^{o\phi}\rho' \quad &= \\ \Big(\lambda x_1.((\lambda x_2.\phi(app_{\tau_1 \to \tau_2})x_1x_2)^{\dagger}\,&(\mathcal{D}[\![\mathcal{T}[e_2]]\!]^{o\phi}\rho'))\Big)^{\dagger}\,(\mathcal{D}[\![\mathcal{T}[e_1]]\!]^{o\phi}\rho')\end{aligned}$$

If either $\mathcal{S}[\![e_1]\!]\rho$ or $\mathcal{S}[\![e_2]\!]\rho$ is $\perp$ then so is $\mathcal{S}[\![(e_1\ e_2)]\!]\rho$ (by strictness), and $\perp$ is $\preceq_{\tau_2}$ related to anything.

If neither is $\perp$, then their values are up(v_1) and up(v_2) respectively (for some v_1 and v_2), so $\mathcal{S}[\![(e_1\ e_2)]\!]\rho = v_1v_2$. By definition of $\preceq_\tau$ there exist v_1' and v_2' such that $\mathcal{D}[\![\mathcal{T}[e_1]]\!]^{o\phi}\rho' = \text{up}(v_1')$ and $v_1 \prec_{\tau_1 \to \tau_2} v_1'$ and $\mathcal{D}[\![\mathcal{T}[e_2]]\!]^{o\phi}\rho' = \text{up}(v_2')$ and $v_2 \prec_{\tau_1} v_2'$, meaning that $\mathcal{D}[\![\mathcal{T}[(e_1\ e_2)]]\!]^{o\phi}\rho' = \phi(app_{\tau_1 \to \tau_2})v_1'v_2'$

Now, $v_1 \prec_{\tau_1 \to \tau_2} v_1'$ means that for any $a \prec_{\tau_1} b$, especially $a = v_2$ and $b = v_2'$, we know that

$$v_1v_2 \preceq_{\tau_2} \phi(app_{\tau_1 \to \tau_2})v_1'v_2'$$

but that is exactly the definition of

$$\mathcal{S}[\![(e_1\ e_2)]\!]\rho \preceq_{\tau_2} \mathcal{D}[\![\mathcal{T}[(e_1\ e_2)]]\!]^{o\phi}\rho'$$

The entire proof can be seen in Appendix C. ■

Theorem 2 *For any program p,*

$$\mathcal{S}[\![p]\!] = \text{up}(n) \Rightarrow \mathcal{D}[\![\mathcal{T}[p]]\!] = \text{up}(n)$$

Proof of Theorem 2.

$$\mathcal{S}[\![\textsf{Program } e]\!] = \mathcal{S}[\![e]\!]\{\}$$

and

$$\mathcal{D}[\![\mathcal{T}[\textsf{Program } e]]\!] = \mathcal{D}[\![\textsf{datatype } C_p \textsf{ fun } F_p \textsf{ in } \mathcal{T}[e]]\!] = \mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\{\}$$

and these are $\preceq_{\textsf{nat}}$-related by lemma 5. That is, if $\mathcal{S}[\![p]\!] = \text{up}(n)$ then $\mathcal{D}[\![\mathcal{T}[p]]\!] = \text{up}(m)$ s.t. $n \prec_{\textsf{nat}} m$, but being related at type $\textsf{nat}$ is the same as being equal, i.e., $n = m$. ■

6.4 Summary and conclusion

We have shown that the translation preserves the meaning of a program if this meaning is up(n) for some $n \in \text{N}$, i.e., if the program terminates. Section 7 discusses possible extensions of both the setting and the proof that would make the result stronger or more widely applicable.

7 Applicability and conclusion

The previous sections have presented a proof that for a simple, typed functional language, the naïve defunctionalization algorithm (no analysis or optimization) into a fairly restricted target language preserves the meaning of terminating programs.

This section discusses possible extensions of the translation and the source language that would increase the applicability of the result.

7.1 Extensions of the target language

There is no problem in extending the target language – the translation still only translate into the same subset of it. If we use a more general target language, e.g., Standard ML then we have a different semantics. For the present result to be applicable, it is sufficient to show that the semantics of the target language and of ML agree on the subset that is the image of the translation.

7.2 Extensions of the source language

The source language is chosen to be simple. It could easily be extended with a number of features without significantly increasing the difficulty of the proof.

More Base Types: If the source language had booleans or strings or other base types, and the target language had the same types and operations on them, then they could be translated directly, just like the operations on natural numbers.

Data-types: The source language could have data-types like the ones of the target language, and these could even allow function types as arguments to the constructors. Such data-types would be treated as base types by the translation, i.e., the name of the type and the operations (case) on it would be copied verbatim. The types of the arguments of the constructors would be converted if they were functional, just like the arguments to functions. That is, the constructors are treated as functions. Even allowing reflexive datatypes in the source language would not necessitate changes to the target language, though the semantics of the source language would of course have to take that into account.

Global Declarations: The source language could have global function declarations in the same way as the current target language. By design they can never be used as arguments to functions or be the value of an expression, since they can only occur in a specific position. Therefore they can be translated merely by translating their bodies and types.

Polymorphic Types: The source language is simply typed. If we added let-polymorphism then we could use the same method as Bell, Hook, and Bellegarde [BBH97] to translate a program with polymorphic functions into a monomorphic program.

This translation, however, potentially gives an exponential blow-up in the size of the program, since each function is replicated at all the types it can possibly be used at.

7.3 Extensions of annotations: analysis

The translation as it is now is very naïve. It basically assumes that any function of a type can float to any application matching that type, which is of course not always true. In the case where only one function can reach a given application point, we could inline the body rather than putting it in an apply-function. If this is the only application point that function can reach, this will even be a saving. This is what Shivers calls "super-β" reduction [Shi90]. Generally, if the functions of a type can be split into groups such that no application point can be reached by functions from more than one group, then the data-types can be split into these groups too. This is not a saving in code size as much as in readability of the produced code, since such groups would usually be of functions that are used in similar ways, and it would improve the accuracy of most types of analysis on the translated program.

Also, functions that are never applied could be put in a group of their own, and there would not need to be an application function for these. The current approach is the crudest approximation to such grouping, but, e.g., a control/data flow analysis [Shi91] would make better approximations possible.

Another optimization is only storing the values of the *free* variables of a function abstraction in the constructed values, just as Reynolds does.

7.4 Conclusion

We have proven that naïve defunctionalization preserves the meaning of terminating programs in a typical simple, typed, functional language. The proof is by logical relations. We expect that the proof can easily be extended to a larger source language, and that the translation can be improved by using global information, e.g., in the form of a control-flow analysis.

Acknowledgments

Thanks to Andrzej Filinski for a timely consultancy and useful comments, and to John Reynolds for encouraging and insightful comments. Also thanks to Olivier Danvy for encouragement and supervision.

Postlude Since finishing the present work, I have become aware of Banerjee, Heintze, and Riecke's new work which provides a complete correctness proof for defunctionalization using operational semantics [BHR01].

A Proof of type preservation

We prove that for any sub-derivation, $\Gamma \vdash e : \tau$, of the derivation of p that

$$(\Gamma \vdash e : \tau \wedge \Gamma' \textit{ extends } \Gamma) \Rightarrow \Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}[\Gamma \vdash e : \tau] : \mathcal{T}[\tau]$$

Proof. The proof is by structural induction on the derivation of $\Gamma \vdash e : \tau$.
Proof by Structural Induction

Basis (Axioms): The basic derivations are of the variable and null expressions.

case $e = x$ **:** If $\Gamma \vdash x : \tau$ then it must be because $\Gamma(x) = \tau$. If Γ' *extends* Γ then $\Gamma'(x) = \mathcal{T}[\tau]$, and as $\mathcal{T}[x] = x$ we can also conclude that $\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}[x] : \mathcal{T}[\tau]$ from the corresponding rule for the destination language.

case $e = 0$ **:** If $\Gamma \vdash 0 : \tau$ then $\tau = \mathsf{nat}$, and as $\mathcal{T}[0] = 0$ and $\mathcal{T}[\mathsf{nat}] = \mathsf{nat}$, we can use the corresponding rule in the destination language to conclude $\Omega_p, \Phi_p, \Gamma' \vdash 0 : \mathsf{nat}$

Induction Hypothesis : Assume it holds for expressions structurally smaller than e.

Induction Step : case $e = \mathsf{succ}(e')$ If $\Gamma \vdash \mathsf{succ}(e') : \tau$ then $\tau = \mathsf{nat}$ and $\Gamma \vdash e' : \mathsf{nat}$

Since e' is structurally smaller than e, we know by induction hypothesis that $\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}[e'] : \mathsf{nat}$. By definition $\mathcal{T}[\mathsf{succ}(e')] = \mathsf{succ}(\mathcal{T}[e'])$ so it suffices that we can conclude $\Omega_p, \Phi_p, \Gamma' \vdash \mathsf{succ}(\mathcal{T}[e']) : \mathsf{nat}$.

case $e = \mathsf{ifz}(e', e_1, x.e_2)$**:** If $\Gamma \vdash e : \tau$ then we know, by inversion, that $\Gamma \vdash e' : \mathsf{nat}$, $\Gamma \vdash e_1 : \tau$, and $\Gamma\{x{=}\mathsf{nat}\} \vdash :$.

By induction hypothesis we can see that $\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}[e'] : \mathsf{nat}$, $\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}[e_1] : \mathcal{T}[\tau]$, and $\Omega_p, \Phi_p, \Gamma'\{x{=}\mathsf{nat}\} \vdash \mathcal{T}[e_2] : \mathcal{T}[\tau]$ (noticing that if Γ' *extends* Γ then $\Gamma'\{x{=}\mathcal{T}[\tau]\}$ *extends* $\Gamma\{x{=}\tau\}$).

As $\mathcal{T}[e] = \mathsf{ifz}(\mathcal{T}[e'], \mathcal{T}[e_1], x.\mathcal{T}[e_2])$ it suffices that we can derive, using the above, that

$$\Omega_p, \Phi_p, \Gamma' \vdash \mathsf{ifz}(\mathcal{T}[e'], \mathcal{T}[e_1], x.\mathcal{T}[e_2]) : \mathcal{T}[\tau]$$

case $e = (\mathsf{fun}\ f\ x : \tau_1 \to \tau_2 = e)_l^{env}$**:** If $\Gamma \vdash e_l : \tau$ then $\tau = \tau_1 \to \tau_2$. Note that $\tau_1 \to \tau_2 \in \text{typesof}(p)$ then.

We can see that for this to be derivable, the premise $(\Gamma \vdash env)$ must be derivable too.

Now, $\mathcal{T}[e] = C_l\ \mathcal{T}_v[env]$, so we must show that $\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}_v[env] : \Omega_p(c_\mathcal{T})(C_l)$ to get the result we need.

To this end, we show the more general statement

$$\Gamma \vdash env \wedge \Gamma' \ \textit{extends}\ \Gamma \Rightarrow \Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}_v[env] : \mathcal{V}_\Gamma$$

where $\mathrm{Dom}(\mathcal{V}_\Gamma) = \{\underline{x}\,|x \in \mathrm{Dom}(\Gamma)\,\}$ and $\mathcal{V}_\Gamma(\underline{x}) = \mathcal{T}[\Gamma(x)]$. This statement is shown by mathematical induction on the size of $\mathrm{Dom}(\Gamma)$.

Proof by Mathematical Induction

Basis $(n = 0)$**:** Assume $\Gamma \vdash \mathit{env}$ and Γ' *extends* Γ. If $|\mathrm{Dom}(G)| = 0$ then $\Gamma = \{\}$. If $\{\} \vdash \mathit{env}$ then $\mathit{env} = \langle\rangle$, and then $\mathcal{T}_v\,[\mathit{env}] = \langle\rangle$ too. Also $\mathcal{V}_\Gamma = \{\}$, since $\mathrm{Dom}(\mathcal{V}_\Gamma) = \emptyset$.
From all this we can directly derive

$$\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}_v\,[\mathit{env}] : \mathcal{V}_\Gamma$$

Induction Hypothesis : Assume it works for $|\mathrm{Dom}(\Gamma)| = k$

Induction Step $(n = k + 1)$**:** Assume $|\mathrm{Dom}(\Gamma)| = k + 1$, $\Gamma \vdash \mathit{env}$, and Γ' *extends* Γ.
Now, *env* must be of the form $\mathit{env}'\langle x : \tau\rangle$ for some env', x, and τ, s.t. $\Gamma \setminus \{x\} \vdash \mathit{env}'$ and $\Gamma(x) = \tau$. Then $T[v]\mathit{env} = \mathcal{T}_v\,[\mathit{env}']\langle \underline{x} : x\rangle$.
If Γ' *extends* Γ then Γ' *extends* $\Gamma \setminus \{x\}$, so by the mathematical induction hypothesis we know that $\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}_v\,[\mathit{env}'] : \mathcal{V}_{\Gamma\setminus\{x\}}$
As Γ' *extends* Γ we know that $\Gamma'(x) = \mathcal{T}[\Gamma(x)]$, so $\Omega_p, \Phi_p, \Gamma' \vdash x : \mathcal{T}[\tau]$.
From this we can derive

$$\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}_v\,[\mathit{env}] : \mathcal{V}_{\Gamma\setminus\{x\}}\{\underline{x} {=} \mathcal{T}[\tau]\}$$

and since $\mathcal{V}_\Gamma = \mathcal{V}_{\Gamma\setminus\{x\}}\{x {=} \mathcal{T}[\Gamma(x)]\}$, this is what we want.

By definition of Ω_p we can see that $\Omega_p(c_\tau)(C_l) = \mathcal{V}_\Gamma$, so $\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}_v\,[\mathit{env}] : \Omega_p(c_\tau)(C_l)$.
Then we can conclude that

$$\Omega_p, \Phi_p, \Gamma' \vdash C_l\ \mathcal{T}_v\,[\mathit{env}] : c_\tau$$

and as $\mathcal{T}[\tau_1 \to \tau_2] = c_{\tau_1 \to \tau_2}$ this shows that

$$\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}[e] : \mathcal{T}[\tau]$$

case $e = (e_1\ e_2)$**:** If $\Gamma \vdash (e_1\ e_2) : \tau_2$ then there exists τ_1 s.t. $\Gamma \vdash e_1 : \tau_1 \to \tau_2$ and $\Gamma \vdash e_2 : \tau_1$.
Then $\tau_1 \to \tau_2 \in \mathrm{typesof}(p)$ and

$$\tau_1 \to \tau_2 \in \{\tau\,|\exists l.\mathrm{PROOF}\,(l) = D \wedge \mathrm{isappl}(D) \wedge \mathrm{typeof}(\mathrm{funpart}(D)) = \tau\,\}$$

In other words, $c_{\tau_1 \to \tau_2} \in \mathrm{Dom}(\Omega_p)$ and $app_{\tau_1 \to \tau_2} \in \mathrm{Dom}(\Phi_p)$ with $\Phi_p(app_{\tau_1 \to \tau_2}) = (\mathcal{T}[\tau_1 \to \tau_2], \mathcal{T}[\tau_1], \mathcal{T}[\tau_2])$.
By induction hypothesis we can derive

$$\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}[e_1] : \mathcal{T}[\tau_1 \to \tau_2]$$

and

$$\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}[e_2] : \mathcal{T}[\tau_1]$$

and since $\mathcal{T}[(e_1\ e_2)] = app_{\tau_1 \to \tau_2}\ \mathcal{T}[e_1]\ \mathcal{T}[e_2]$ we can use the above to conclude

$$\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}[(e_1\ e_2)] : \mathcal{T}[\tau_2]$$

■

B Proof of Lemma 3

Lemma 3 is given as:

If PROOF$(l) = \Gamma \vdash (\mathsf{fun}\ f\ x : \tau_1 \to \tau_2 = e)^{env} : \tau_1 \to \tau_2$, $\Gamma \vdash env$, Γ' *extends* Γ, $\rho' \in \mathcal{D}[\![\Gamma']\!]^o$ and ν satisfies that $\mathrm{Dom}(\nu) = \{\underline{x} \,|x \in \mathrm{Dom}(\Gamma)\}$ and $\nu(\underline{x}) = \rho'(x)$ then

$$\mathcal{D}[\![\mathcal{F}_{\mathrm{v}}[env]]\!]^{o\phi}\{ff = g, xx = b\}(\nu) = \rho''$$

implies $\mathrm{Dom}(\rho'') = \mathrm{Dom}(\Gamma) \cup \{ff, xx\}$ and $\forall x \in \mathrm{Dom}(\Gamma).\rho''(x) = \rho'(x)$, $\rho''(ff) = g$, and $\rho'(xx) = b$.

Proof of Lemma 3. The proof will indirectly be by induction on the structure of *env*. The actual induction will be in the size of Dom(Γ), which relates to *env* since we know $\Gamma \vdash env$ by inversion.

We prove the following property for any sub-derivation, $\Gamma \vdash e : \tau$, of the derivation of p:

$$(\Gamma \vdash e : \tau \wedge \Gamma' \textit{ extends } \Gamma) \Rightarrow \Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}[\Gamma \vdash e : \tau] : \mathcal{T}[\tau]$$

The proof is by mathematical induction on the size of Γ.

Proof by Mathematical Induction

Basis ($|\mathrm{Dom}(\Gamma)| = 0$)**:** Then $\Gamma = \{\}$ and if $\{\} \vdash env$ then $env = \langle\rangle$. Then $\mathcal{F}_{\mathrm{v}}[env] = \langle\rangle$, and $\mathcal{D}[\![\langle\rangle]\!]^{o\phi}\{ff = g, xx = b\}\nu = \{ff = g, xx = b\}$, which trivially satisfies the requirements.

Induction Hypothesis : Assume the property holds for $|\mathrm{Dom}(\Gamma)| = k$.

Induction Step ($|\mathrm{Dom}(\Gamma)| = k + 1|$)**:** If $\Gamma \neq \{\}$ and $\Gamma \vdash \mathit{env}$, then $\mathit{env} = \mathit{env}'\langle x : \tau\rangle$ such that $\Gamma(x) = \tau$ and $\Gamma \setminus \{x\} \vdash \mathit{env}'$.

Let $\rho'' = \mathcal{D}[\![\mathcal{F}_{\mathrm{v}}[\mathit{env}']]\!]^{o\phi}\{\mathit{ff} = g, xx = b\}(\nu \setminus \{\underline{x}\})$

Then, by induction hypothesis, $\mathrm{Dom}(\rho'') = \mathrm{Dom}(\Gamma \setminus \{x\}) \cup \{\mathit{ff}, xx\}$ and $x \in \mathrm{Dom}(\Gamma \setminus \{x\}) \Rightarrow \rho''(x) = \rho'(x)$ (and values at ff and xx are preserved).

Then $\mathcal{D}[\![\mathcal{F}_{\mathrm{v}}[\mathit{env}']\langle \underline{x} : x\rangle]\!]^{o\phi}\{\mathit{ff} = g, xx = b\}(\nu) = \rho''\{x = \nu(\underline{x})\}$

We know that $\nu(\underline{x}) = \rho'(x)$, and as x occurs in the program, it is not equal to ff or xx which are chosen fresh, so

$$\mathcal{D}[\![\mathcal{F}_{\mathrm{v}}[\mathit{env}]]\!]^{o\phi}\{\mathit{ff} = g, xx = b\}(\nu) = \rho''\{x = \rho'(x)\}$$

which satisfies the requirements of the lemma, since $\mathrm{Dom}(\rho''\{x = \rho'(x)\}) = \mathrm{Dom}(\rho'') \cup \{x\} = \{\mathit{ff}, xx\} \cup \mathrm{Dom}(\Gamma \setminus \{x\}) \cup \{x\} = \{\mathit{ff}, xx\} \cup \mathrm{Dom}(\Gamma)$ and if $y \in \mathrm{Dom}(\Gamma)$ then either $x = y$ and then $\rho''\{x = \rho'(x))\}(y) = \rho'(y)$ or $x \in \mathrm{Dom}(\Gamma \setminus \{x\})$ and then $\rho''\{x = \rho'(x))\}(y) = \rho''(y) = \rho'(y)$, and $\rho''\{x = \rho'(x))\}(\mathit{ff}) = \rho''(\mathit{ff}) = g$ and likewise xx is mapped to b.

■

C Proof of Lemma 5

Lemma 5 states that for any expression e, if $\mathrm{PROOF}\,(l) = \Gamma \vdash e : \tau$ and $\Omega_p, \Phi_p, \Gamma' \vdash \mathcal{T}[e] : \mathcal{T}[\tau]$ where Γ' *extends* Γ then for all $\rho \in \mathcal{S}[\![\Gamma]\!]$ and $\rho' \in \mathcal{D}[\![\Gamma']\!]^{o}$

$$\rho \prec_{\Gamma} \rho' \Rightarrow \mathcal{S}[\![e]\!]\rho \preceq_{\tau} \mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho'$$

Proof of Lemma 5.

The proof is by structural induction on the structure of expressions.

case $e = x$**:** Then $\mathcal{T}[e] = x$. As $\rho \prec_{\Gamma} \rho'$ we know that $\rho(x) \prec_{\Gamma(x)} \rho'(x)$. Then it follows that

$$\mathrm{up}(\rho(x)) \preceq_{\Gamma(x)} \mathrm{up}(\rho'(x))$$

which is exactly

$$\mathcal{S}[\![e]\!]\rho \preceq_{\Gamma(x)} \mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho'$$

case $e = 0$**:** The proof of this case follows directly from $\mathcal{S}[\![e]\!]\rho = \mathrm{up}(0)$ and $\mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho' = \mathrm{up}(0)$ and $\mathrm{up}(0) \preceq_{\mathsf{nat}} \mathrm{up}(0)$.

case $e = \mathsf{succ}(e')$**:** By induction hypothesis $\mathcal{S}[\![e']\!]\rho \preceq_{\mathsf{nat}} \mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho'$. That means that either $\mathcal{S}[\![e']\!]\rho = \bot$, but then $\mathcal{S}[\![e]\!]\rho = \bot$ too, and by definition $\mathcal{S}[\![e]\!]\rho \preceq_{\mathsf{nat}} \mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho'$.

If $\mathcal{S}[\![e']\!]\rho \neq \bot$ then there must exist some $n \in \mathrm{N}$ s.t. $\mathcal{S}[\![e']\!]\rho = \mathrm{up}(n)$.

If $\mathrm{up}(n) \preceq_{\mathsf{nat}} \mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho'$ then $\mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho' = \mathrm{up}(n)$ too.

Then $\mathcal{S}[\![e]\!]\rho = \mathrm{up}(n+1)$ and $\mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho' = \mathrm{up}(n+1)$, which are $\preceq_{\mathsf{nat}}$-related.

case $e = \mathsf{ifz}(e', e_1, x.e_2)$**:** By an argument similar to the case for $\mathsf{succ}(e')$, either $\mathcal{S}[\![e']\!]\rho = \bot$ and we are finished, or $\mathcal{S}[\![e']\!]\rho\mathcal{D}[\![\mathcal{T}[e']]\!]^{o\phi}\rho' = \mathrm{up}(n)$.

If $n = 0$ then $\mathcal{S}[\![e]\!]\rho = \mathcal{S}[\![e_1]\!]\rho$ and $\mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho' = \mathcal{D}[\![\mathcal{T}[e_1]]\!]^{o\phi}\rho'$, which are $\preceq_\tau$-related by induction hypothesis.

If $n > 0$ then $\mathcal{S}[\![e]\!]\rho = \mathcal{S}[\![e_2]\!]\rho\{x{=}n-1\}$ and

$$\mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho' = \mathcal{D}[\![\mathcal{T}[e_2]]\!]^{o\phi}\rho'\{x{=}n-1\}$$

which, since $\rho\{x{=}n-1\} \prec_{\Gamma\{x{=}\mathsf{nat}\}} \rho'\{x{=}n-1\}$, are also $\preceq_\tau$-related by induction hypothesis.

case $e = (\mathsf{fun}\ f\ x : \tau_1 \to \tau_2 = e')_l^{env}$**:** As defined

$$\mathcal{S}[\![e]\!]\rho = \mathrm{up}(\mathrm{fix}(\lambda F.\lambda X.\mathcal{S}[\![e]\!]\rho\{f{=}F\}\{x{=}X\}))$$

and

$$\mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho' = (\lambda v.\mathrm{up}(\mathrm{in}_{C_l}(v)))^\dagger\,(\mathcal{D}[\![\mathcal{T}_v\,[env]]\!]^{o\phi}\rho')$$

and we know from Lemma 1 that $(\mathcal{D}[\![\mathcal{T}_v\,[env]]\!]^{o\phi}\rho') = \mathrm{up}(\nu)$ for some ν, i.e., $\mathcal{D}[\![\mathcal{T}[e]]\!]^{o\phi}\rho' = \mathrm{up}(\mathrm{in}_{C_l}(\nu))$.

To show that these are $\preceq_{\tau_1\to\tau_2}$-related we will use fixed-point induction over the inclusive set

$$S \stackrel{\mathrm{def}}{=} \{f \,|\, f \prec_{\tau_1\to\tau_2} \mathrm{in}_{C_l}(\nu)\}$$

to show that $\mathrm{fix}(\lambda F.\lambda X.\mathcal{S}[\![e]\!]\rho\{f{=}F\}\{x{=}X\})$ is in it. That is, we must show that the set is closed under the function

$$\mathcal{F} \stackrel{\mathrm{def}}{=} \lambda F.\lambda X.\mathcal{S}[\![e]\!]\rho\{f{=}F\}\{x{=}X\}$$

Let p be any element of S. Then

$$\mathcal{F}(p) \prec_{\tau_1\to\tau_2} \mathrm{in}_{C_l}(\nu)$$

as for any $a \prec_{\tau_1} b$

$$\mathcal{F}(p)(a) = \mathcal{S}[\![e']\!]\rho\{f\!=\!p\}\{x\!=\!a\}$$

and from Theorem 1 we know that

$$\phi(app_{\tau_1\to\tau_2})(\text{in}_{C_l}(\nu))b = \mathcal{D}[\![\mathcal{T}[e']]\!]^{o\phi}\rho''$$

where $\rho\{f\!=\!p\}xa \prec_\Gamma \rho''$ necessarily by the requirements on ρ'' (in combination the two properties, $\rho \prec_\Gamma \rho'$ and $\forall x \in \text{Dom}(\Gamma).\rho''(x) = \rho'(x)$, implies $\rho \prec_\Gamma \rho''$).

By induction hypothesis (e' is smaller than e) the denotation of these expressions are related, i.e., $\mathcal{F}(p)(a) \prec_{\tau_2} app_{\tau_1\to\tau_2}(\text{in}_{C_l}(v))(b)$ which proves that $\mathcal{F}(p) \prec_{\tau_1\to\tau_2} \text{in}_{C_l}(v)$.

As $p \in S \Rightarrow \mathcal{F}(p) \in S$, and S is inclusive, we can conclude that $\text{fix}(\mathcal{F}) \in S$, i.e.,

$$\text{fix}(\mathcal{F}) \prec_{\tau_1\to\tau_2} \text{in}_{C_l}(v)$$

Then we can then conclude that

$$\text{up}(\text{fix}(\mathcal{F}))\preceq_{\tau_1\to\tau_2}\text{up}(\text{in}_{C_l}(v))$$

as needed.

case $e = (e_1\ e_2)$**:** By induction hypothesis we know that

$$\mathcal{S}[\![e_1]\!]\rho \preceq_{\tau_1\to\tau_2} \mathcal{D}[\![\mathcal{T}[e_1]]\!]^{o\phi}\rho'$$

and

$$\mathcal{S}[\![e_2]\!]\rho \preceq_{\tau_1} \mathcal{D}[\![\mathcal{T}[e_2]]\!]^{o\phi}\rho'$$

Also, by definition,

$$\mathcal{S}[\![(e_1\ e_2)]\!]\rho = \Big(\lambda f.\,(\lambda x.fx)^\dagger\,(\mathcal{S}[\![e_2]\!]\rho)\Big)^\dagger\,(\mathcal{S}[\![e_1]\!]\rho)$$

and

$$\begin{aligned}&\mathcal{D}[\![app_{\tau_1\to\tau_2}\ \mathcal{T}[e_1]\ \ \mathcal{T}[e_2]]\!]^{o\phi}\rho' \ = \\ &\qquad\Big(\lambda x_1.((\lambda x_2.\phi(app_{\tau_1\to\tau_2})x_1x_2)^\dagger\,(\mathcal{D}[\![\mathcal{T}[e_2]]\!]^{o\phi}\rho'))\Big)^\dagger\,(\mathcal{D}[\![\mathcal{T}[e_1]]\!]^{o\phi}\rho')\end{aligned}$$

If either $\mathcal{S}[\![e_1]\!]\rho$ or $\mathcal{S}[\![e_2]\!]\rho$ is $\perp$ then so is $\mathcal{S}[\![(e_1\ e_2)]\!]\rho$ (by strictness), and $\perp$ is $\preceq_{\tau_2}$ related to anything.

If neither is $\perp$, then their values are $\mathrm{up}(v_1)$ and $\mathrm{up}(v_2)$ respectively (for some v_1 and v_2), so $\mathcal{S}[\![(e_1\ e_2)]\!]\rho = v_1 v_2$.

By definition of $\preceq_\tau$ there exists v_1' and v_2' s.t. $\mathcal{D}[\![\mathcal{T}[e_1]]\!]^{o\phi}\rho' = \mathrm{up}(v_1')$ and $v_1 \prec_{\tau_1 \to \tau_2} v_1'$ and $\mathcal{D}[\![\mathcal{T}[e_2]]\!]^{o\phi}\rho' = \mathrm{up}(v_2')$ and $v_2 \prec_{\tau_1} v_2'$.

From that we infer that $\mathcal{D}[\![\mathcal{T}[(e_1\ e_2)]]\!]^{o\phi}\rho' = \phi(app_{\tau_1 \to \tau_2})v_1'v_2'$

Now, $v_1 \prec_{\tau_1 \to \tau_2} v_1'$ means that for any $a \prec_{\tau_1} b$, especially $a = v_2$ and $b = v_2'$, we know that

$$v_1 v_2 \preceq_{\tau_2} \phi(app_{\tau_1 \to \tau_2})v_1'v_2'$$

which is exactly the definition of

$$\mathcal{S}[\![(e_1\ e_2)]\!]\rho \preceq_{\tau_2} \mathcal{D}[\![\mathcal{T}[(e_1\ e_2)]]\!]^{o\phi}\rho'$$

$\blacksquare$

References

[AJ89] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In Michael J. O'Donnell and Stuart Feldman, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, Texas, January 1989. ACM Press.

[BBH97] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 25–37, Amsterdam, The Netherlands, 9–11 June 1997.

[Bel93] Jeffrey M. Bell. An implementation of Reynolds' defunctionalization method for a modern functional language. Master's thesis, Oregon Graduate Institute of Science and Technology, November 1993.

[BH94] Jeffrey M. Bell and James Hook. Defunctionalization of typed programs. Technical Report CSE-94-024, Oregon Graduate Institute of Science and Technology, May 1994.

[BHR01] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Semantics-based design and correctness of control-flow analysis-based program transformations. Unpublished, March 2001.

[Bon90] Anders Bondorf. *Self-Applicable Partial Evaluation (Revised Version).* PhD thesis, DIKU, Computer Science Department, University of Copenhagen, December 1990. DIKU Rapport 90/17.

[CJW00] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In Gert Smolka, editor, *Proceedings of the Ninth European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 56–71, Berlin, Germany, March 2000. Springer-Verlag.

[DS00] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287, 2000.

[FH00] Adam Fischbach and John Hannan. Specification and correctness of lambda lifting. In W. Taha, editor, *Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2000)*, number 1924 in Lecture Notes in

Computer Science, pages 108–128, Montréal, Canada, September 2000. Springer-Verlag.

[Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 190–203, Nancy, France, September 1985. Springer-Verlag.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[Mit93] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1993.

[MMH96] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg Beach, Florida, January 1996. ACM Press.

[Pey85] Simon L. Peyton Jones. An introduction to fully-lazy supercombinators. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, number 242 in Lecture Notes in Computer Science, pages 176–208, Val d'Ajol, France, 1985. Springer-Verlag.

[Rey98] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[Shi90] Olin Shivers. Super-β: Copy, constant, and lambda propagation in Scheme, May 1990. Available at
`http://www.cc.gatech.edu/fac/Olin.Shivers/citations.html#superbeta`.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[Str00] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):1–49, 2000.

[TO98] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages.* Foundation of Computing Series. The MIT Press, 1993.