CHALMERS
UNIVERSITY OF TECHNOLOGY

Formal neighbourhoods, combinatory Bohm trees, and untyped normalization by evaluation

Annals of Pure and Applied Logic 163 (2012) 122–131

Contents lists available at SciVerse ScienceDirect

Annals of Pure and Applied Logic

journal homepage: www.elsevier.com/locate/apal

Formal neighbourhoods, combinatory Böhm trees, and untyped normalization by evaluation

Peter Dybjer [a,*], Denis Kuperberg [b]

[a] Chalmers University of Technology, Göteborg, Sweden
[b] ENS Lyon, France

ARTICLE INFO

Article history:
Available online 30 July 2011

ACM classification categories:
F.4.1
F.3.2
F.3.1
D.3.1

Keywords:
Combinatory logic
Normalization by evaluation
Formal neighbourhoods
Lazy evaluation
Böhm trees

ABSTRACT

We prove the correctness of an algorithm for normalizing untyped combinator terms by evaluation. The algorithm is written in the functional programming language Haskell, and we prove that it lazily computes the combinatory Böhm tree of the term. The notion of combinatory Böhm tree is analogous to the usual notion of Böhm tree for the untyped lambda calculus. It is defined operationally by repeated head reduction of terms to head normal forms. We use formal neighbourhoods to characterize finite, partial information about data, and define a Böhm tree as a filter of such formal neighbourhoods. We also define formal topology style denotational semantics of a fragment of Haskell following Martin-Löf, and let each closed program denote a filter of formal neighbourhoods. We prove that the denotation of the output of our algorithm is the Böhm tree of the input term. The key construction in the proof is a "glueing" relation between terms and semantic neighbourhoods which is defined by induction on the latter. This relation is related to the glueing relation which was earlier used for proving the correctness of normalization by evaluation algorithm for typed combinatory logic.

1. Introduction

The correctness of an algorithm for normalizing typed combinatory terms was proved by Coquand and Dybjer [5]. They showed that this algorithm can be used for deciding whether two terms are convertible (provably equal by equational reasoning from the axioms for K and S) by checking whether their normal forms are identical.

However, the algorithm can be applied to all combinatory terms, also to those which do not have a type. We show that in this case it computes the combinatory Böhm tree of the input under lazy evaluation. Although the Böhm tree may be infinite the output of the algorithm can approximate any finite part of it. In particular, the Böhm tree of a normalizing combinatory term is a (tree representation) of its normal form, irrespectively of whether the input term has a type or not.

We shall here use *formal neighbourhoods* to represent such finite approximations of data. On the one hand we define the Böhm tree of a combinatory term as a *filter* of formal neighbourhoods which approximate the successive head normal forms of the term, the subterms of the head normal form, etc. On the other hand we will write our algorithm in the lazy functional programming language Haskell. We define the *denotational semantics* of this Haskell program as a *filter* of formal neighbourhoods. Finally, we prove that a formal neighbourhood approximates the output of the algorithm iff it approximates the Böhm tree of the input term.

Background. This paper combines two techniques which are both inspired by constructive thinking.

* Corresponding author.
E-mail address: peterd@chalmers.se (P. Dybjer).

0168-0072/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.apal.2011.06.021

The first is *normalization by evaluation* or *normalization by intuitionistic model construction*, whereby the meaning of a term is first computed and then the normal form is extracted from the meaning. The resulting algorithm can be extracted from a constructive proof of weak normalization using Tait's *reducibility* predicates [16,17,3].

The second is the idea of expressing the *denotational semantics* of a program using formal neighbourhoods which approximate the *canonical form* of a program under lazy evaluation [18]. Martin-Löf used this technique for the domain interpretation of the language of terms underlying intuitionistic type theory. Martin-Löf's interpretation was in turn based on Scott's reformulation of domains as *neighbourhood systems* [25] and *information systems* [26].

Martin-Löf's work on formal neighbourhood semantics in 1983–84 was a starting point for his and Sambin's development of a more general formal topology [23,24].

Normalization by evaluation was originally applied to compute normal forms of terms in languages (or logical systems) where such normal forms always exist. However, several researchers have later applied the technique to languages where normal forms do not always exist. For example, Mogensen [19] defined a *self-reducer* for the untyped lambda calculus using normalization by evaluation. Then Danvy and Filinski [7,12] used the technique for *partially evaluating* programs in languages with general recursion and non-terminating computations.

Aehlig and Joachimski [1] analysed normalization by evaluation algorithm for the untyped lambda calculus. They considered an untyped β-normalizing version of Berger and Schwichtenberg's algorithm [4] for the simply typed lambda calculus which produced long $\beta\eta$-normal forms. Using operational techniques, they showed that this algorithm computed the Böhm tree (in the ordinary sense of the lambda calculus) of a term. Aehlig and Joachimski also wrote a Haskell program which implemented the Böhm tree computation using lazy evaluation.

A Standard ML version of this program was then analysed by Filinski and Rohde [13] who instead used denotational semantics (and a proof technique due to Pitts [21,20]) in their proof that the algorithm outputs Böhm trees. They argued that in this way they could prove the correctness of the precise Standard ML program rather than of a (subtly different) operational model, as Aehlig and Joachimski. Devautour (in a summer internship project supervised by the first author [8]) adapted Filinski and Rohde's proof to the case of combinatory logic.

Our goal here is to provide an alternative and we hope more perspicuous proof of this result by using Martin-Löf's formal neighbourhood semantics of lazy functional programs [18].

We also have a more general aim: to investigate formal topology style proof principles for proving the correctness of lazy functional programs in their full generality. Many (perhaps most) practically useful proofs of functional programs can be done in first order logic where the proper axioms are the recursion equations of the programs and axioms for inductively defined predicates [10,11]. Moreover, coinduction can be useful for reasoning about infinitely proceeding computations [9]. However, there are cases where we genuinely seem to need the ínsights of domain theory such as the example in the present paper. There are two key difficulties. The first is that we need to reason about partially defined data. The second is that we use a *reflexive type*, a recursive type where the type variable appears negatively in the defining type equation. Such types do not have induction principles in the usual sense, although they have *mixed induction principles* in the sense of Pitts [20]. Proof systems based on domain theory have a long history going back to Scott's ideas about a Logic of Computable Functions and their implementation by Milner and his coworkers in the influential LCF-system [14]. But this system was based on classical domain theory where the use of "admissible" predicates sometimes caused complications. One of our aims here is to demonstrate an approach based on direct reasoning about the formal neighbourhoods of programs, which we propose as an alternative to principles based on classical domain theory and on abstract reasoning principles based on category theory [21].

2. Combinatory expressions and their neighbourhoods

2.1. The type of combinatory expressions

Our task is to write a program which computes the Böhm tree of a term in combinatory logic. Such terms are generated by the following grammar:

$$e \ ::= \mathtt{K} \mid \mathtt{S} \mid e@e.$$

We use the @-sign to denote application in combinatory logic, our *object language*, and reserve juxtaposition for application in Haskell, our *meta programming language*. (We use the usual notational conventions such as left-associativity of application, etc.)

The type of combinatory logic terms is implemented as a recursive data type `Exp` in Haskell:

```
data Exp = K | S | App Exp Exp.
```

Semantically, this type can be understood as the domain of lazy expressions $\mathcal{D}$ which is a least solution of the domain equation

$$\mathcal{D} \cong 1 + 1 + \mathcal{D} \times \mathcal{D}.$$

Here 1 is a one-point domain, + is the separated sum of domains, and $\times$ is the cartesian product of domains.

However, we shall not use classical domain theory here, but the neighbourhood system approach introduced by Scott [25,26] and in particular the *formal neighbourhoods* of Martin-Löf [18].

Our algorithm will do case analysis on the type Exp. Although Haskell provides case analysis for arbitrary data types, the formal analysis will be facilitated by using a special combinator for case analysis of expressions:

```
expcase :: a -> a -> (Exp -> Exp -> a) -> Exp -> a

expcase c d f K          = c
expcase c d f S          = d
expcase c d f (App e e') = f e e'.
```

Note that Haskell uses double colon :: for the typing relation.

Remark. We would like to emphasize that the discussion in this paper is not specific to the Haskell programming language. We only use a clean subset of Haskell which deserves to be called the "canonical lazy functional programming language": the simply typed lambda calculus with polymorphic types and recursive function and data type definitions. Our normalization algorithm will be written in this subset.

Since Haskell is a lazy language, the evaluation of a program `e` of type Exp, written `e :: Exp`, may give rise to a computation which goes on indefinitely. It may not produce any output at all, or it may produce infinite output. For example, using Haskell's fixed point combinator

```
fix :: (a -> a) -> a

fix f = f (fix f)
```

we can write the term

```
fix (\x -> App K x) :: Exp
```

which will produce infinite output when evaluated:

```
App K (App K (App K ... )).
```

A formal neighbourhood (we will later drop "formal") represents a finite piece of information about the value of an expression of a certain type. We have a special neighbourhood Δ which represents no information at all. Other examples of expression neighbourhoods are K which represents that we know that the expression is K, and $\Delta@\Delta$ which represents that we know that an expression is lazily evaluated to $e@e'$, but where we have no information about the canonical forms of e and e'.

The grammar for expression neighbourhoods is as follows:

$$U ::= \Delta \mid \mathtt{K} \mid \mathtt{S} \mid U@U.$$

To characterize the combinatory Böhm tree it is sufficient to use the following special *normal form neighbourhoods*:

$$U ::= \Delta \mid \mathtt{K} \mid \mathtt{K}@U \mid \mathtt{S} \mid \mathtt{S}@U \mid (\mathtt{S}@U)@U.$$

The terminology *neighbourhood* comes from the fact that each neighbourhood characterizes a set of programs (of type Exp) according to what we know about their (lazy) value. For example, Δ characterizes the set of all programs, K characterizes the set of programs which evaluate to K, and $\mathtt{K}@\Delta$ characterizes the set of all programs which evaluate to $\mathtt{K}@e$ for some e.

Dually, each program of type Exp determines the set of those neighbourhoods which approximate its lazy value.

Remark. We can now inductively define the notion of *(formal) inclusion* $U \supseteq U'$ of expression neighbourhoods: $\Delta \supseteq U$ for all U; $\mathtt{K} \supseteq \mathtt{K}$, $\mathtt{S} \supseteq \mathtt{S}$, and if $U \supseteq V$ and $U' \supseteq V'$ then $U@U' \supseteq V@V'$. It follows easily that formal inclusion is a partial order.

Remark. The *(formal) intersection* $U \cap U'$ of two neighbourhoods can be defined as the greatest lower bound of U and U' with respect to formal inclusion. Note that a lower bound may not exist, so in our setting, formal intersection is a partial operation.

Remark. Here we deviate in an inessential way from Martin-Löf [18] whose formal neighbourhoods are closed under finite intersection: if $(U_i)_{i\in I}$ is a family of formal neighbourhoods, then so is $\cap_{i\in I} U_i$. In this way formal intersection becomes a total operation, but the price is that *inconsistent* neighbourhoods such as $\mathtt{K} \cap \mathtt{S}$ are introduced. See also Scott [26].

3. Combinatory Böhm trees

3.1. Head normal form

The notion of *combinatory Böhm tree* for combinatory logic is analogous to the usual notion of Böhm tree for the untyped lambda calculus [2]. It is obtained by first computing the head normal form of a term, then computing the head normal form of the subterms of the first head normal form, and so on possibly ad infinitum. To define it formally, we first introduce the notion of a *head normal form* v of a combinatory expression e. It is an inductively defined relation $e \Rightarrow^{\mathrm{h}} v$ generated by the following rules:

$$\mathtt{K} \Rightarrow^{\mathrm{h}} \mathtt{K} \qquad \mathtt{S} \Rightarrow^{\mathrm{h}} \mathtt{S}$$

$$\frac{e \Rightarrow^{\mathrm{h}} \mathtt{K}}{e@e' \Rightarrow^{\mathrm{h}} \mathtt{K}@e'} \qquad \frac{e \Rightarrow^{\mathrm{h}} \mathtt{K}@e' \qquad e' \Rightarrow^{\mathrm{h}} v}{e@e'' \Rightarrow^{\mathrm{h}} v}$$

$$\frac{e \Rightarrow^{\mathrm{h}} \mathtt{S}}{e@e' \Rightarrow^{\mathrm{h}} \mathtt{S}@e'} \qquad \frac{e \Rightarrow^{\mathrm{h}} \mathtt{S}@e'}{e@e'' \Rightarrow^{\mathrm{h}} \mathtt{S}@e'@e''}$$

$$\frac{e \Rightarrow^{\mathrm{h}} \mathtt{S}@e'@e'' \qquad (e'@e''')@(e''@e''') \Rightarrow^{\mathrm{h}} v}{e@e''' \Rightarrow^{\mathrm{h}} v}$$

Corollary. *It follows immediately, by case analysis, that*

$$\mathtt{K}@e@e' \Rightarrow^{\mathrm{h}} v \text{ iff } e \Rightarrow^{\mathrm{h}} v$$
$$\mathtt{S}@e@e'@e'' \Rightarrow^{\mathrm{h}} v \text{ iff } (e'@e''')@(e''@e''') \Rightarrow^{\mathrm{h}} v.$$

3.2. The Böhm tree of a combinatory expression

The relation $e \rhd^{\mathrm{Bt}} U$ expresses that the combinatory Böhm tree of an expression e is approximated by the expression neighbourhood U. It is inductively generated by the following rules:

$$e \rhd^{\mathrm{Bt}} \Delta$$

$$\frac{e \Rightarrow^{\mathrm{h}} \mathtt{K}}{e \rhd^{\mathrm{Bt}} \mathtt{K}} \qquad \frac{e \Rightarrow^{\mathrm{h}} \mathtt{K}@e' \qquad e' \rhd^{\mathrm{Bt}} U'}{e \rhd^{\mathrm{Bt}} \mathtt{K}@U'}$$

$$\frac{e \Rightarrow^{\mathrm{h}} \mathtt{S}}{e \rhd^{\mathrm{Bt}} \mathtt{S}} \qquad \frac{e \Rightarrow^{\mathrm{h}} \mathtt{S}@e' \qquad e' \rhd^{\mathrm{Bt}} U'}{e \rhd^{\mathrm{Bt}} \mathtt{S}@U'}$$

$$\frac{e \Rightarrow^{\mathrm{h}} \mathtt{S}@e'@e'' \qquad e' \rhd^{\mathrm{Bt}} U' \qquad e'' \rhd^{\mathrm{Bt}} U''}{e \rhd^{\mathrm{Bt}} \mathtt{S}@U'@U''}.$$

The Böhm tree of an expression e in Exp is the set

$$BT(e) = \{U \mid e \rhd^{\mathrm{Bt}} U\}.$$

The following properties are easy to prove:

1. $BT(\mathtt{K}@e@e') = BT(e)$
2. $BT(\mathtt{S}@e@e'@e'') = BT((e'@e''')@(e''@e'''))$
3. Böhm trees are upward closed with respect to formal inclusion: if $U \supseteq U'$ and $e \rhd^{\mathrm{Bt}} U'$ then $e \rhd^{\mathrm{Bt}} U$
4. $e \rhd^{\mathrm{Bt}} \Delta$
5. Böhm trees are closed under formal intersection $e \rhd^{\mathrm{Bt}} U$ then $e \rhd^{\mathrm{Bt}} U'$ then $e \rhd^{\mathrm{Bt}} U \cap U'$.

The three last properties state that a Böhm tree is a *filter* of neighbourhoods.

It is however not immediate to define application of combinatory Böhm trees and prove that they form a combinatory algebra, see Barendregt [2] for a construction of the Böhm tree model of the lambda calculus.

In the following section we shall give an alternative definition of the combinatory Böhm tree of a term: it will be lazily computed by a *normalization by evaluation (nbe)* algorithm. As usual for nbe-algorithms, it will be easy to prove that the algorithm maps convertible expressions to equal results (Böhm trees). Hence it will also follow easily that Böhm trees (defined by nbe) form a combinatory algebra.

4. A program which lazily computes combinatory Böhm trees

The reader is referred to [5] for an introduction to the algorithm (with extensions to natural number and Brouwer ordinal types). In the algorithm for typed combinatory logic we interpret a combinatory expression of function type $A \to B$ as a pair consisting of its normal form and a function which maps the meaning of A to the meaning of B. In this way we keep track not only of normal forms but also of how an expression maps normal forms to normal forms, and how such mappings extend to higher types.

In the untyped setting these pairs are represented by elements of the Haskell type

```
data Sem = Gl Exp (Sem -> Sem).
```

In other words, `Sem` is the solution of the domain equation (in mathematical notation):

$$D \cong Exp \times (D \to D).$$

The constructor

```
Gl :: Exp -> (Sem -> Sem) -> Sem
```

is a curried version of the isomorphism from left to right. Note that D is a *reflexive domain* in Scott's sense: D appears negatively on the right hand side.

The following Haskell function interprets an expression in `Exp` in the semantic domain `Sem`:

```
eval :: Exp -> Sem

eval K = Gl K (\x -> Gl (App K (reify x)) (\y -> x))
eval S = Gl S (\x -> Gl (App S (reify x))
                (\y -> Gl (App (App S (reify x)) (reify y))
                 (\z -> appsem (appsem x z) (appsem y z))))
eval (App e e') = appsem (eval e) (eval e')
```

where

```
appsem :: Sem -> Sem -> Sem

appsem (Gl e f) x = f x
```

is the application function on the semantic domain. The function

```
reify :: Sem -> Exp

reify (Gl e f) = e
```

returns the first component (the normal form) of a pair in the semantics. The normal form of an expression can now be computed by first interpreting and then reifying:

```
nbe :: Exp -> Exp

nbe e = reify (eval e).
```

Our task is to prove that this algorithm computes the combinatory Böhm tree using formal neighbourhoods.

5. Denotational semantics of the nbe program

5.1. A fragment of Haskell

We shall now define the denotation of the nbe program as a set of neighbourhoods (actually a filter). Since it is written in Haskell we shall define the denotational semantics of a fragment of Haskell which contains the nbe program. In doing so we shall follow Martin-Löf's semantics of the terms of the untyped language underlying intuitionistic type theory as *denotational neighbourhood filters of formal neighbourhoods* [18]. We only deviate from his presentation in inessential ways.

Note that we are only using the core of Haskell, that is, the simply typed lambda calculus with polymorphic types (in the sense of Hindley–Milner) and recursive data type and recursive function definitions. For the purpose of the presentation we shall only use a simply typed lambda calculus where the base types are `Exp` and `Sem`. (It would be straightforward, but verbose, to extend our definitions to recursive types in general.)

The terms of our Haskell subset are generated by the following grammar:

```
e ::= x | e e | \x -> e
    | K | S | App e e | expcase e e e e
    | Gl e e | reify e | appsem e e
    | fix e
```

As before, we will use $e@e'$ for `App e e'`. We will also use $\lambda x.e$ for `\x -> e`.

We can redefine the recursive function `eval` given in the previous section so that it is a term in the above fragment of Haskell:

```
eval :: Exp -> Sem

eval = fix (\ev e -> expcase
       (Gl K (\x -> Gl (App K (reify x)) (\y -> x)))
       (Gl S (\x -> Gl (App S (reify x))
              (\y -> Gl (App (App S (reify x)) (reify y))
               (\z -> appsem (appsem x z) (appsem y z))))))
       (\e' e'' -> appsem (ev e') (ev e''))
       e
       ).
```

We will associate to each program t with at most n free variables an $n+1$-ary relation $[\![t]\!]$ between neighbourhoods (of appropriate types). One can prove that $[\![t]\!]$ is an *approximable mapping* in Scott's sense [25,26]. In particular, we associate to each closed program t, a set $[\![t]\!]$ which is a filter of neighbourhoods.

5.2. Function neighbourhoods

We have already defined the neighbourhoods of type `Exp`, but also need to define those of the type `Sem` of semantic elements. Since `Sem` is a recursive type using a function space in its definition, we first introduce the notion of a function neighbourhood. The reader is referred to Plotkin [22], Scott [25,26], Martin-Löf [18], and Hedberg [15] for more discussion of function neighbourhoods qua finite elements of function domains.

The formal neighbourhoods of function type are constructed as follows: if U_i $(i \in I)$ is a finite set of neighbourhoods of type α and V_i $(i \in I)$ is a finite set of neighbourhoods of type β, then

$$\bigcap_{i\in I}[U_i; V_i]$$

is a neighbourhood of type $\alpha \to \beta$. We let $\Delta = \bigcap_{i\in\emptyset}[U_i; V_i]$ and define the notion of formal inclusion and formal intersection following Martin-Löf. See also Hedberg's [15] machine-checked proof that a category of lower semi-lattices (of formal neighbourhoods) and approximable mappings is cartesian closed.

$$\bigcap_{i\in I}[U_i; V_i] \supseteq \bigcap_{j\in J}[U'_j; V'_j]$$

iff

$$\forall i \in I.\exists J' \subseteq J.\bigcap_{j\in J'} U'_j \supseteq U_i \& V_i \supseteq \bigcap_{j\in J'} V'_j.$$

The idea is that a program f is approximated by a function neighbourhood $\bigcap_{i\in I}[U_i; V_i]$ iff the output of f is approximated by V_i whenever the input is approximated by U_i, for all $i \in I$. There are however two ways of interpreting this idea:

operationally: if an arbitrary input x is approximated by U_i, then the output $f\ x$ is approximated by V_i. Formally, one bases this notion of approximation on the lazy operational semantics of programs.

denotationally: one drops the reference to an element x of input type α, and instead directly models the effect of f as a relation between an input neighbourhood and the neighbourhoods which approximates the output computed by f on an input approximated by the input neighbourhood.

The two notions differ because of the full abstraction problem. The neighbourhoods of a program with input type `Bool` $\to$ `Bool` $\to$ `Bool` will depend on the result when evaluating it on the neighbourhood "parallel or" [22] which is not inhabited in Haskell or any other *sequential* programming language.

Accordingly, Martin-Löf [18] associated two sets of neighbourhoods to each program, its *operational neighbourhood filter* and its *denotational neighbourhood filter*. He also proved an adequacy theorem: a denotational neighbourhood is always an operational neighbourhood, but not vice versa.

We remark that some function neighbourhoods are inconsistent, e g $[\Delta, \mathtt{K}] \cap [\Delta, \mathtt{S}]$. Such neighbourhoods are of course always uninhabited operationally.

5.3. Neighbourhoods of Sem

We recall the type Sem of semantic values:

```
data Sem = Gl Exp (Sem -> Sem).
```

The neighbourhoods of Sem are given by the following inductive definition:

- Δ is a neighbourhood.
- if U is a neighbourhood of type Exp, and V is a neighbourhood of type `Sem -> Sem`, then `Gl` U V is a neighbourhood of Sem.

5.4. Denotational semantics

To each program e (in our Haskell subset) with at most n free variables, we will assign an $n+1$-ary relation between neighbourhoods, and write $U_1, \ldots, U_n \, [\![e]\!] \, V$.

We first define the semantics of typed lambda terms

- $U_1, \ldots, U_n \, [\![x_i]\!] \, V$ iff $U_i \subseteq V$.
- $U_1, \ldots, U_n \, [\![\lambda x_{n+1}.e]\!] \, \bigcap_i [V_i; W_i]$ iff for all i, $U_1, \ldots, U_n, V_i \, [\![e]\!] \, W_i$.
- $U_1, \ldots, U_n \, [\![e \, e']\!] V$ iff there exists U such that $U_1, \ldots, U_n [\![e']\!] U$ and $U_1, \ldots, U_n [\![e]\!] [U; V]$.

Then we define the semantics of the combinators. For simplicity, we assume that the terms are closed, so that the denotations are sets of neighbourhoods. (It is of course straightforward to extend this to open terms.)

- $[\![\mathtt{K}]\!] = \{\mathtt{K}, \Delta\}$.
- $[\![\mathtt{S}]\!] = \{\mathtt{S}, \Delta\}$.
- $[\![t@t']\!] = \{U@U' \mid U \in [\![t]\!] \text{ and } U' \in [\![t']\!]\} \cup \{\Delta\}$.
- $[\![\mathtt{expcase}\ c\ d\ f\ e]\!] = \{V \mid (\mathtt{K} \in [\![e]\!] \,\&\, V \in [\![c]\!]) \vee (\mathtt{S} \in [\![e]\!] \,\&\, V \in [\![d]\!]) \vee (U@U' \in [\![e]\!] \,\&\, [U; [U', V]] \in [\![f]\!])\} \cup \{\Delta\}$.
- $[\![\mathtt{Gl}\ e\ f]\!] = \{\mathtt{Gl}\ U\ V \mid U \in [\![e]\!], V \in [\![f]\!]\} \cup \{\Delta\}$.
- $[\![\mathtt{reify}\, e]\!] = \{U \mid \mathtt{Gl}\ U\ V \in [\![e]\!]\} \cup \{\Delta\}$.
- $[\![\mathtt{appsem}\, e\, e']\!] = \{V \mid \exists U \in [\![e']\!], \exists V, W.\mathtt{Gl}\ W\ [U; V] \in [\![e]\!]\} \cup \{\Delta\}$.

Finally, the semantics of the fixed point combinator is [25]:

- $[\![\mathtt{fix}\, f]\!] = \{V \mid \Delta \, [\![f^n]\!] \, V \text{ for some } n\}$.

One can now show that:

- $[\![\mathtt{eval}\, \mathtt{K}]\!] = [\![\mathtt{Gl}\ \mathtt{K}\ (\lambda x.\mathtt{Gl}\ (\mathtt{K@(reify}\, x))\ (\lambda y.x))]\!]$.
- $[\![\mathtt{eval}\, \mathtt{S}]\!] = [\![\mathtt{Gl}\, \mathtt{S}\, \lambda x.\mathtt{Gl}(\mathtt{S@(reify}\, x))\ (\lambda y.\mathtt{Gl}\ (\mathtt{S@(reify}\, x))@(\mathtt{reify}\, y)\ (\lambda z.\mathtt{appsem}\ (\mathtt{appsem}\, x\, z)\ (\mathtt{appsem}\, y\, z))]\!]$ if $\mathtt{S} \in [\![e]\!]$.
- $[\![\mathtt{eval}\ (e@e')]\!] = [\![\mathtt{appsem}\ (\mathtt{eval}\, e)\ (\mathtt{eval}\, e')]\!]$.

We are now ready to prove our theorem: that the nbe-algorithm computes combinatory Böhm trees. In doing so we shall write $e \in U$ for $U \in [\![e]\!]$. We can read $e \in U$ as "e formally belongs to the neighbourhood U".

6. Proofs of correctness

6.1. Nbe maps convertible terms into equal Böhm trees

Let e `conv` e' mean that e and e' are convertible combinatory expressions, where convertibility is the least congruence (w.r.t. application) which contains the axioms for K and S. We prove that the nbe-algorithm maps convertible terms into equal Böhm trees. This follows immediately from the fact that e `conv` e' implies `eval` e = `eval` e', which is proved by a straightforward induction on the derivation of e `conv` e' :

- The K-conversion rule.

$$\begin{aligned}[\![\mathtt{eval}\ (\mathtt{K@}e\mathtt{@}e')]\!] &= [\![\mathtt{appsem}\ (\mathtt{appsem}\ (\mathtt{eval}\ \mathtt{K})\ (\mathtt{eval}\, e))\ \mathtt{eval}\, e']\!] \\ &= [\![\mathtt{appsem}\ (\mathtt{Gl}\ (\mathtt{K@(reify}\ (\mathtt{eval}\, e)))\ (\lambda y.(\mathtt{eval}\, e)))\ (\mathtt{eval}\, e')]\!] \\ &= [\![\mathtt{eval}\, e]\!].\end{aligned}$$

- The S-conversion rule follows by a similar calculation.
- Application preserves conversion. Assume as induction hypothesis that $[\![\mathtt{eval}\, e_0]\!] = [\![\mathtt{eval}\, e_0']\!]$ and $[\![\mathtt{eval}\, e_1]\!] = [\![\mathtt{eval}\, e_1']\!]$. Then

$$\begin{aligned}[\![\mathtt{eval}\ (e_0\mathtt{@}e_1)]\!] &= [\![\mathtt{appsem}\ \ (\mathtt{eval}\, e_0)\ (\mathtt{eval}\, e_1)]\!] \\ &= [\![\mathtt{appsem}\ \ (\mathtt{eval}\, e_0')\ (\mathtt{eval}\, e_1')]\!] \\ &= [\![\mathtt{eval}\ (e_0'\mathtt{@}e_1')]\!].\end{aligned}$$

It follows immediately that e `conv` e' implies $[\![\mathtt{nbe}\ \ e]\!] = [\![\mathtt{nbe}\ \ e']\!]$.

6.2. Completeness of nbe

We now prove that the nbe-algorithm is complete, that is, that it returns any finite part of the Böhm tree:

$e \triangleright^{\mathrm{Bt}} U$ implies $\mathtt{nbe}\, e \in U$.

The proof is by induction on the derivation of $e \triangleright^{\mathrm{Bt}} U$.

- $U = \Delta$ is immediate.
- $U = \mathtt{K}$ where $e \Rightarrow^{\mathrm{h}} \mathtt{K}$.
  Since convertible terms have equal normal forms, it follows that $[\![\mathtt{nbe}\ e]\!] = [\![\mathtt{nbe}\ \mathtt{K}]\!]$. Since $\mathtt{nbe}\,\mathtt{K} \in \mathtt{K}$ we conclude $\mathtt{nbe}\, e \in \mathtt{K}$.
- $U = \mathtt{K@}U'$ where $e \Rightarrow^{\mathrm{h}} \mathtt{K@}e'$ and $e' \triangleright^{\mathrm{Bt}} U'$. It follows that $[\![\mathtt{nbe}\, e]\!] = [\![\mathtt{nbe}\,(\mathtt{K@}e')]\!] = [\![\mathtt{K@}(\mathtt{nbe}\, e')]\!]$. By induction hypothesis, $\mathtt{nbe}\, e' \in U'$. Hence $\mathtt{nbe}\, e \in \mathtt{K@}U'$.
- The three cases for S (applied to 0, 1, and 2 arguments) are analogous.

6.3. Soundness of nbe

We then prove soundness, that is, that the nbe-algorithm only returns approximations of the Böhm tree:

$\mathtt{nbe}\, e \in U$ implies $e \triangleright^{\mathrm{Bt}} U$.

We shall prove this as a corollary of the following main lemma (cf. the reducibility/glueing method [5]):

$\mathtt{eval}\, e \in V$ implies $e \triangleright^{\mathrm{Gl}} V$.

where $e \triangleright^{\mathrm{Gl}} V$ is defined by (generalized) induction on V: it holds if either

- $V = \Delta$;
- or $V = \mathtt{Gl}\ U\ (\bigcap_i [V_i; W_i])$ where $e \triangleright^{\mathrm{Bt}} U$ and for all i and e', $e' \triangleright^{\mathrm{Gl}} V_i$ implies $e@e' \triangleright^{\mathrm{Gl}} W_i$.

The following properties are consequences of the analogous properties of $\triangleright^{\mathrm{Bt}}$ in section 3.2:

1. If $V \supseteq V'$ and $e \triangleright^{\mathrm{Gl}} V'$ then $e \triangleright^{\mathrm{Gl}} V$.
2. $\mathtt{K@}e@e' \triangleright^{\mathrm{Gl}} V$ iff $e \triangleright^{\mathrm{Gl}} V$.
3. $\mathtt{S@}e@e'@e'' \triangleright^{\mathrm{Gl}} V$ iff $e@e''@(e'@e'') \triangleright^{\mathrm{Gl}} V$.

The main lemma is then proved by induction on e.

Case K. We prove that $\mathtt{eval}\,\mathtt{K} \in V$ implies $\mathtt{K} \triangleright^{\mathrm{Gl}} V$ by case analysis on the neighbourhoods of $\mathtt{eval}\,\mathtt{K}$.
The case $V = \Delta$ is immediate, so let $V = \mathtt{Gl}\ U\ (\bigcap_i [V_i; W_i])$. It follows that $U \supseteq \mathtt{K}$ and

$\lambda x.\mathtt{Gl}\ (\mathtt{K@}(\mathtt{reify}\, x))\ (\lambda y.x) \in [V_i; W_i]$.

We need to prove two things:

- $\mathtt{K} \triangleright^{\mathrm{Bt}} U$, which follows directly from $U \supseteq \mathtt{K}$.
- For all i, e', $e' \triangleright^{\mathrm{Gl}} V_i$ implies $\mathtt{K@}e' \triangleright^{\mathrm{Gl}} W_i$.
  The case $W_i = \Delta$ is immediate so we let $W_i = \mathtt{Gl}\ U_i\ (\bigcap_j [V_{ij}; W_{ij}])$. It follows that $U_i \supseteq \mathtt{K@}U_i'$ and $\lambda y.x \in [V_{ij}; W_{ij}]$. We need to prove two things:
  - $\mathtt{K@}e' \triangleright^{\mathrm{Bt}} U_i$.
    Case $V_i = \Delta$. It follows that $U_i \supseteq \mathtt{K@}\Delta$ and hence $\mathtt{K@}e' \triangleright^{\mathrm{Bt}} U_i$.
    Case $V_i = \mathtt{Gl}\ U_i'\ (\bigcap_j [V_{ij}'; W_{ij}'])$. We know $e' \triangleright^{\mathrm{Bt}} U_i'$ and hence $\mathtt{K@}e' \triangleright^{\mathrm{Bt}} U_i$.
  - For all j, $e'' \triangleright^{\mathrm{Gl}} V_{ij}$ implies $\mathtt{K@}e'@e'' \triangleright^{\mathrm{Gl}} W_{ij}$. But, we know that $\lambda y.x \in [V_{ij}; W_{ij}]$ and hence that $W_{ij} \supseteq V_i$. Hence $e' \triangleright^{\mathrm{Gl}} V_i$ by property 1 and thus $\mathtt{K@}e'@e'' \triangleright^{\mathrm{Gl}} W_{ij}$ by property 2 of $\triangleright^{\mathrm{Gl}}$ above.

Case S. The proof that $\mathtt{eval}\,\mathtt{S} \in V$ implies $\mathtt{S} \triangleright^{\mathrm{Gl}} V$ begins with a case analysis which is analogous to the case analysis for K, except that the last case is different:

- ...
  - For all j, $e'' \triangleright^{\mathrm{Gl}} V_{ij}$ implies $\mathtt{S@}e'@e'' \triangleright^{\mathrm{Gl}} W_{ij}$.
    The case $W_{ij} = \Delta$ is immediate so we let $W_{ij} = \mathtt{Gl}\ U_{ij}\ (\bigcap_k [V_{ijk}; W_{ijk}])$. It follows that $U_{ij} \supseteq \mathtt{S@}U_i'@U_{ij}'$ and $\lambda z.xz(yz) \in [V_{ijk}; W_{ijk}]$. We need to prove two things:
    - $\mathtt{S@}e'@e'' \triangleright^{\mathrm{Bt}} U_{ij}$.
      Case $V_{ij} = \Delta$. It follows that $U_{ij} \supseteq \mathtt{S@}\Delta@\Delta$ and hence $\mathtt{S@}e'@e'' \triangleright^{\mathrm{Bt}} U_{ij}$.
      Case $V_{ij} = \mathtt{Gl}\ U_{ij}'\ (\bigcap_k [V_{ijk}'; W_{ijk}'])$. It follows that $U_{ij} \supseteq \mathtt{S@}U_i'@U_{ij}'$. We know $e'' \triangleright^{\mathrm{Bt}} U_{ij}'$ and hence $\mathtt{S@}e'@e'' \triangleright^{\mathrm{Bt}} U_{ij}$.

∗ For all k, $e''' \triangleright^{\mathrm{Gl}} V_{ijk}$ implies $\mathrm{S}@e'@e''@e''' \triangleright^{\mathrm{Gl}} W_{ijk}$. But we know that $\lambda z.xz(yz) \in [V_{ijk}; W_{ijk}]$ which together with previous assumptions about the neighbourhoods of x and y, the assumptions about e', e'', e''', and the definition of $\triangleright^{\mathrm{Gl}}$ entail that $e'@e'''@(e''@e''') \triangleright^{\mathrm{Gl}} W_{ijk}$. Thus $\mathrm{S}@e'@e''@e''' \triangleright^{\mathrm{Gl}} W_{ijk}$ by property 3 of $\triangleright^{\mathrm{Gl}}$ above.

Case $e@e'$**.** We prove that $\mathtt{eval}\,(e@e') \in V$ implies $e@e' \triangleright^{\mathrm{Gl}} V$ from the induction hypotheses that $\mathtt{eval}\,e \in U$ implies $e \triangleright^{\mathrm{Gl}} U$ for all U and $\mathtt{eval}\,e' \in U'$ implies $e' \triangleright^{\mathrm{Gl}} U'$ for all U'.

It follows that either

- $V = \Delta$ and we are done.
- Or there exists U such that $\mathtt{eval}\,e \in \mathtt{Gl}\ \Delta\ [U; V]$ and $\mathtt{eval}\,e' \in U$. In this case the induction hypotheses tells us that $e' \triangleright^{\mathrm{Gl}} U$ and $e \triangleright^{\mathrm{Gl}} \mathtt{Gl}\ \Delta\ [U; V]$. But then it follows immediately from the definition of the latter that $e@e' \triangleright^{\mathrm{Gl}} V$.

7. Conclusion and further research

A new approach to Böhm trees. There is already much work which shows how normalization by evaluation provides a new approach to normalization with simpler and sometimes more general proofs. Similarly, we here get a new approach to the theory of Böhm trees defined as the output of a lazy functional programs. By usual standards such Böhm trees generated by a program are perhaps rather unconventional mathematical objects. But we hope to have demonstrated that formal neighbourhoods provide a suitable framework for their analysis. We would for example like to investigate whether we can get a simpler proof of Wadsworth's theorem [27] which relates equality of Böhm trees to equality in Scott's D_∞-model, if Böhm trees are defined by nbe.

A formal topology style logic of lazy functional programs. As we mentioned in the introduction we are interested in isolating the principles needed for a logic of lazy functional programs which can be used for reasoning about partial elements and reflexive types. The aim is to provide a formal topology style alternative to the logic of computable functions in the sense of Scott and Milner.

In our proof above we have used several inductively defined relations: the relation $e \Rightarrow^{\mathrm{h}} v$ which relates total elements of the type Exp, the relation $e \triangleright^{\mathrm{Bt}} V$ which relates partial elements of the type Exp and neighbourhoods of type Exp. The denotational semantics is given by a relation between programs and neighbourhoods of a given type. In our proofs we have used induction on the structure of neighbourhoods, and induction on the inductively defined relations. We would like to investigate in future work whether we can provide a general theory of inductive definitions of properties of programs and neighbourhoods. We would also like to understand the connection between this logic and abstract reasoning principles better.

Intersection types. Formal neighbourhoods are closely related to intersection types [6]. It would be interesting to see how the above development could be rephrased inside that framework.

Acknowledgement

The authors wish to thank Thierry Coquand for useful advice on this work.

References

[1] Klaus Aehlig, Felix Joachimski, Operational aspects of untyped normalisation by evaluation, Mathematical Structures in Computer Science 14 (4) (2004).
[2] Henk P. Barendregt, The Lambda Calculus, North-Holland, 1984, (Revised edition).
[3] Ulrich Berger, Program extraction from normalization proofs, in: Proceedings of the International Conference on Typed Lambda Calculi and Applications, Utrecht, March 1993.
[4] Ulrich Berger, Helmut Schwichtenberg, An inverse to the evaluation functional for typed λ-calculus, in: Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science, Amsterdam, July 1991, pp. 203–211.
[5] Thierry Coquand, Peter Dybjer, Intuitionistic model constructions and normalization proofs, Mathematical Structures in Computer Science 7 (1997) 75–94.
[6] Mario Coppo, Mariangiola Dezani-Ciancaglini, A new type-assignment for lambda terms, Archiv für Mathematische Logik 19 (1978) 139–156.
[7] Olivier Danvy, Type-directed partial evaluation, in: Proceedings of POPL96, the 1996 ACM Symposium on Principles of Programming Languages, 1996.
[8] Martin Devautour, Untyped normalization by evaluation. Technical report, École Normale Superieure de Lyon, 2004. Report for the training period June–August 2004, Magistère d'Informatique et de Modelisation, 2nd year, directed by Peter Dybjer.
[9] Peter Dybjer, Herbert Sander, A functional programming approach to the specification and verification of concurrent systems, Formal Aspects of Computing 1 (1989) 303–319.
[10] Peter Dybjer, Program verification in a logical theory of constructions, in: J.-P. Jouannaud (Ed.), Functional Programming Languages and Computer Architecture, in: LNCS, vol. 201, Springer-Verlag, September 1985, pp. 334–349.
[11] Peter Dybjer, Comparing integrated and external logics of functional programs, Science of Computer Programming 14 (1990) 59–79.
[12] Andrzej Filinski, A semantic account of type-directed partial evaluation, in: PPDP, 1999, pp. 378–395.
[13] Andrzej Filinski, Henning Korsholm Rohde, Denotational aspects of untyped normalization by evaluation, Theoretical Informatics and Applications 39 (3) (2005) 423–453.
[14] Michael J. C. Gordon, Robin Milner, Christopher P. Wadsworth, Edinburgh LCF, in: Lecture Notes in Computer Science, vol. 78, Springer, 1979.
[15] Michael Hedberg, A type-theoretic interpretation of constructive domain theory, Journal of Automated Reasoning 16 (1996) 369–425.
[16] Per Martin-Löf, About models for intuitionistic type theories and the notion of definitional equality, in: S. Kanger (Eds.), Proceedings of the 3rd Scandinavian Logic Symposium, 1975, pp. 81–109.

[17] Per Martin-Löf, An intuitionistic theory of types: Predicative part, in: H.E. Rose, J.C. Shepherdson (Eds.), Logic Colloquium '73, North-Holland, 1975, pp. 73–118.
[18] Per Martin-Löf, The domain interpretation of type theory, lecture notes, in: Kent Karlsson and Kent Petersson (Eds.), Workshop on Semantics of Programming Languages, Abstracts and Notes, Chalmers University of Technology and University of Göteborg, August 1983. Programming Methodology Group.
[19] Torben Æ Mogensen, Efficient self-interpretation in lambda calculus, Journal of Functional Programming 2 (3) (1992) 345–364.
[20] Andrew M. Pitts, Computational adequacy via "mixed" inductive definitions, in: MFPS, 1993, pp. 72–82.
[21] Andrew M. Pitts, Relational properties of domains, Information Computation 127 (2) (1996) 66–90.
[22] Gordon D. Plotkin, LCF considered as a programming language, Theoretical Computer Science 5 (3) (1977) 225–255.
[23] Giovanni Sambin, Intuitionistic formal spaces — a first communication, in: D. Skordev (Ed.), Mathematical Logic and its Applications, Plenum, 1987, pp. 187–204.
[24] Giovanni Sambin, Some points in formal topology, Theoretical Computer Science 305 (2003) 347–408.
[25] Dana S. Scott, Lectures on a mathematical theory of computation. Technical Report PRG-19, Oxford University Programming Research Group, May 1981.
[26] Dana S. Scott, Domains for denotational semantics, in: Automata, Languages and Programming, Proceedings of the 9th International Colloquium, in: LNCS, vol. 140, Springer-Verlag, July 1982, pp. 577–613.
[27] Christopher P. Wadsworth, The relation between computational and denotational properties for Scott's D_∞-models of the lambda-calculus. Semantics and correctness of programs, SIAM Journal on Computing 5 (3) (1976) 488–521.