CPE: A Calculus for Proving Equivalence of Expressions in a Non-Strict Functional Language*

Manfred Schmidt-Schauß
Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
D-60054 Frankfurt
Germany
E-mail: schauss@informatik.uni-frankfurt.de

May 14, 1996

Abstract

We propose a tableau-based calculus CPE for proving behavioural equivalence of expressions in a non-strict higher-order functional language. We consider two expressions as equivalent, if plugged into any program context, the following two situations do not occur: i) The two contexts reduce to different constants ii) one context reduces to a constant while the other has no weak head normal form. This definition is co-inductive, however, it our behavioural equivalence is different from the equivalences, which consider expression as equivalent, if the contexts are tested for having a weak head normal form or not.

The main interest is in comparing functions that operate on infinite lists, streams or other infinite data structures. The calculus is also able to prove extensional equivalence of functions in some cases. It includes also a non-trivial non-termination analysis.

The calculus is designed such that it can easily be automated. Comparing it with the take-lemma of Bird and Wadler, it can prove stronger equivalences.

The calculus can be used for verification, program construction, in particular for proving that program transformations in a compiler for a non-strict functional language are correct without any preconditions such as termination conditions.

1 Introduction

We are interested in behavioural equality of expressions in a non-strict, higher-order and lazily evaluated functional language. We test expressions d by checking the reduction behaviour of $C[d]$ where $C[.]$ is a program context. We consider two expressions d and e equal, if they cannot be distinguished using the following criterion: if there is no program context C such

*internal report 2/96, Fachbereich Informatik, Johann Wolfgang Goethe-Universität Frankfurt

that $C[d]$ and $C[e]$ reduce to different constant constructors, or one of them is undefined, while the other reduces to a constant constructor. We call this equality ω-equality ($=_\omega$).

For example, we consider $\lambda\, x.\, \bot$ and $\bot$ as equivalent, since these functions cannot be distinguished using our criterion.

Our ω-equality is different from the equality as defined in [Mor68] or [Abr90], since they consider these two functions as different, as their criterion is the distinction between termination and non-termination. But if two expressions are equivalent in their sense, then they are also ω-equal (see Theorem 7). ω-equality is also different from the co-inductively defined equality of expressions ([Gor94]). However, our equivalence relation and some of its properties can be obtained citing [How89] after translating our language into a lambda-calculus with constructors.

Since we show that ω-equality is the largest relation that satisfies substitutivity and that keeps different constant distinct and does not identify constant with $\bot$, we have a an analogue to the full abstraction theorem. That the relations based on the λ-calculus are different from ω-equality is a hint that the a translation into the lambda-calculus where constructors and destructors are simply translated into lambda-expressions instead of special constant does not reflect the inherent properties of equivalence of functional languages with algebraic data types.

Proving equivalence of programs in a functional language for input like the set of finite normal forms is usually done by induction ([BM88]). This method breaks down in a non-strict semantics, i.e, if expressions may be partially undefined, or if equality of stream-processing programs is to be proved, since an infinite list or an infinite stream has no normal form. There are some known methods for proving equivalence of programs where also infinite lists or other infinite data objects are taken into account. For example the type-based method in [Wad89]. The take-lemma ([BW88]) is a proof method designed for streams modelled by infinite lists. Another, more general method to define equivalent behaviour is based on co-induction (see [MT91, Pit94], for a tutorial see ([Gor94]).

In this paper, we develop a simple and automatable proof method for showing the behavioural equivalence of functions as defined in this paper. This may be a helpful tool for verification, program construction and verifying the correctness of program transformations in a compiler. The calculus CPE is intended for a typed functional programming language, but we design it for a combinator language with algebraic data types. This is no restriction on the expressiveness. Instead of using type information for variables, we shall use set-expressions to restrict the range of quantifications. We treat typing by considering non-well-typed expressions as undefined ($\bot$).

The general equivalence problem is not recursively enumerable, hence it is acceptable that the calculus does not prove all equivalences, but nevertheless it covers a lot of cases of practical interest. It is more general than the take-Lemma of Bird and Wadler since we have implicit and explicit non-termination checks. CPE is able to prove an example that is used to demonstrate the power of the methods from co-inductive equality ([Gor94]).

An advantage of our method compared to other ones is that it can easily be mechanised and that it works for all algebraic types, not only for infinite lists. A further distinguishing feature is that it includes an explicit and implicit non-termination check that is able to detect expressions without weak head normal form. It furthermore is able to check equivalence even in the presence of functional objects in the outcome of a program, since the calculus is able to prove extensional equivalence of functions. It can also be used to compute strictness information for a function f by plugging in an `Bot` as argument and checking with the calculus, whether the resulting term is equal to `Bot`.

In a larger example, we will demonstrate that the calculus is able to prove that $filter\ p\ (map\ f\ xs)$ is ω-equal to $map\ f\ (filter\ (p \circ f)\ xs)$ for all list-expressions including infinite lists. This example is taken from ([Gor94]) and is used there to demonstrate that there are simple examples of stream-processing functions that cannot be proved equivalent by the Take-Lemma, since the Take-Lemma method lacks a method to detect the $\bot=\bot$-situation.

A simple example (see Example 9) that can only be proved if the calculus CPE is extended by some inductive prover using size-arguments is that $length\ l =_\omega length\ (reverse\ l)$ for the standard functions *length* and *reverse* on lists. For infinite lists or partially undefined lists, the calculus is able to handle this example (see Example 9). For finite lists or other finite data structures, the method can be improved by extended it by a special treatment of nodes (see section 6 on extensions), or by calling an automated induction prover

2 The Language

We define Λ, a simple functional core language with algebraic data types. It resembles closely the language used in [PJL91] and [SSPS95]. We will use a version of the language with minimal syntax. Nevertheless, the language is sufficiently expressive to transform functional programs into Λ using standard methods, like lambda-lifting ([PJ87]).

Algebraic data types consist a finite number of corresponding data constructors together with their arities. Different algebraic data types have disjoint sets of data constructors. Usually, there is also a type constructor and the data constructors have a user-defined type. In this paper, we do not make use of an explicit type system, hence we omit a description.

The following constants are in Λ:

i) For every algebraic data type A there are finitely many constructor constants $\mathtt{con}_{A,1}, \ldots, \mathtt{con}_{A,n_i}$, where every constructor comes with an arity. The number of constructors of the algebraic type A is denoted by $con\#(A)$.

ii) For every algebraic type A one case constant $\mathtt{case}_A$ of arity $con\#(A) + 1$. The first argument is the expression to be cased. The other arguments are functions taking m_k arguments for $1 \leq k \leq n$, where m_k is the arity of the k^{th} constructor of A.

iii) User-defined supercombinators

iv) The constant `Bot` standing for a semantically undefined expression.

An example of an algebraic data type is `Bool` with the constants `true`, `false`, and case-constant `if`. Another example is `List` with the constants `Nil`, binary `cons`, also denoted by an infix binary ":", and case-constant `lcase`, which requires three arguments: the list expression, the expression for the `Nil`-case and a function taking the head and tail as arguments in the non-`Nil`-case.

Λ-expressions are built in the standard way, i.e., expressions may be variables x, constants `c`, and applications $(d\ e)$. The set of free variables in an expression t is denoted as $Var(t)$. We assume that application is left associating, i.e., $e_1\ e_2\ e_3$ is equivalent to $((e_1\ e_2)\ e_3)$. An expression without variables is called a *ground expression*. Sometimes expressions that contain free variables are also called abstract expressions.

Recursion is possible via defined supercombinators. An expression e with n free variables $x_1, \ldots, x_n$ may be defined as a supercombinator: $sc\ x_1\ \ldots\ x_n = e$, where sc is treated as a constant. We will assume that `Bot` is a non-terminating supercombinator.

As notational convention $D[t]$ will denote an expression which has at some position the single subexpression t. $D[.]$ is called a (linear) term-context. We will also use term-contexts with more than one argument. Term-contexts with n arguments can be viewed as supercombinators defined with n arguments, where in the body of the definition, every argument variable appears exactly once.

A `program` consists of a set of definitions of algebraic data types and of definitions of supercombinators. It may also contain some annotations that a supercombinator is strict in some of its arguments (see next subsection 2.1).

Note that the syntax permits expressions that are non-well-typed in the usual sense.

2.1 Operational Semantics

A ground expression is in *weak head normal form* (WHNF), if it is of the form:

- `c` $e_1\ \ldots\ e_n$, where `c` is a constructor, and the arity of `c` is $\geq n$. This form is also called *constructor weak head normal form* (CWHNF). If $n = arity($`c`$)$, then we have a *saturated CWHNF* (SCWHNF).
- `f` $e_1\ \ldots\ e_n$, where `f` is a supercombinator or a `case` constant that requires more than n arguments.

The operational semantics for ground expressions is defined through $\delta-$reductions, with respect to a program P, i.e., a fixed set of algebraic data types and supercombinator definitions.

- If `sc` $x_1\ \ldots\ x_n = e$ is a supercombinator definition, then `sc` $t_1\ \ldots\ t_n \rightarrow_\delta e[x_1/t_1, \ldots, x_n/t_n]$.

If the supercombinator sc is defined to be strict in its j^{th} argument, then t_j should be in WHNF.

- For case constants we have: $\mathtt{case}_A\ (\mathtt{con}_j\ t_1 \ldots t_m)\ e_1 \ldots e_n \rightarrow_\delta (e_j\ t_1 \ldots t_m)$, if n is the number of data constructors for A, and the constructor $\mathtt{con}_j$ is a constructor belonging to A, and the constructor $\mathtt{con}_j$ has arity m.
- There are some reduction rules for the special supercombinator Bot, which are intended to inherit the undefinedness and to capture dynamic typing errors:
  - An expression of the form $(\mathtt{case}_A\ t)$ reduces to Bot, if either t is equal to Bot, or t is in WHNF, but not in CWHNF, or it is in CWHNF, i.e., $t = \mathtt{con}\ t_1 \ldots t_m$), but con is not a constructor for A, or the arity of con is not equal to m.
  - $(\mathtt{con}\ t_1 \ldots t_m)$ reduces to Bot, if m is strictly greater than the arity of con.
  - (Bot t) reduces to Bot
  - Bot reduces to Bot.

A ground expression t is in *normal form* (NF for short) if t cannot be reduced.

A *redex* is a position in an expression where a reduction can be performed. A *necessary redex* is defined as follows.

- If the root is a redex in t, then it is a necessary redex in t.
- If t is a necessary redex in d, then t is also a necessary redex in $(d\ e)$.
- If t is of the form $(\mathtt{case}_A\ e_0\ e_1\ \ldots\ e_n)$ and t is not in WHNF, then every necessary redex of e_0 is also a necessary redex of t.
- If t is of the form $(\mathtt{sc}\ t_1 \ldots t_n)$, and sc is (defined to be) strict in the j^{th} argument, then a necessary redex of t_j is also a necessary redex of t.

A *necessary reduction* is one which reduces a necessary redex. A *normal order reduction* is a reduction sequence that performs only necessary reductions.

The operational semantics has the usual properties: It is Church-Rosser (confluent), and whenever an expression t can be reduced to a WHNF t', there is a normal order reduction from t to some t'' that is in WHNF, and can be reduced to t'.

It is possible to write functions in Λ that behave the same as arithmetic functions. Hence we can and do assume that natural numbers and arithmetic functions are available.

3 Equivalence of Expressions

In this section, we give rigorous definitions of the basic notion of ω-equality, based upon a given program P.

Definition 1. Let d and e be two ground Λ-expressions. Then

- $d =_0 e$ for all expressions d and e.
- Let $k > 0$. Then $d =_k e$, iff one of the following conditions hold:
  - d and e have no WHNF
  - d and e have a CWHNF ($\texttt{con}\ d_1 \ldots d_m$) and ($\texttt{con}\ e_1 \ldots e_m$), respectively, and $d_i \ =_{k-1}\ e_i$ for all $i = 1, \ldots, m$.
  - d or e have a WHNF, but neither d nor e has an SCWHNF, and for all ground expressions a: $(d\ a) =_{k-1} (e\ a)$.

We say two expressions d and e are ω-equal ($d =_\omega e$), iff $d =_k e$ for all $k \geq 0$.

We write $\neq_k$ or $\neq_\omega$, respectively, if the corresponding relation $=_k$ or $=_\omega$ does not hold.

Note that this equality identifies all terms without WHNF.

This equality does not distinguish the function f with definition $f\ x = f\ x$ from `Bot`. If these are applied to some arguments, then they both yield non-termination, and hence they are ω-equal. In the notation of the λ-calculus, this means that $\lambda\, x.\ \bot$ and $\bot$ are not distinguished.

The definition permits that an expression in CWHNF is ω-equal to one in WHNF, but not in CWHNF, but does not permit ω-equality of an SCWHNF and another expression in WHNF.

Example 1. Consider the supercombinator f with $f\ x = f$. Then from the definition of ω-equality we can infer that the two expressions `Bot` and f are ω-equal. If we use a type system, then we would get a type error for f, since the equation $\alpha \to \beta = \beta$ has no solution in the sets of types. (For a description of types, see for example [BW88].)

Lemma 2. *Let a and b be ground expressions such that $a =_k b$, and let d be another ground expression.*

Then $(a\ d) =_k (b\ d)$

Proof. If both a and b have no WHNF, then this is obvious. Otherwise, assume that neither a nor b have an SCWHNF. Then we can use definition 1 to show the relation. If both have an SCWHNF, then addition of an argument yields `Bot` for both expressions.

As a preparation for the proof of substitutivity of $=_\omega$, we borrow a lemma from [SSPS95]. For every term t in Λ with a WHNF, let $nr\#(t)$ be the minimal number of reductions from t to its WHNF using only necessary reduction steps (see also [SSPS95]).

Lemma 3. *Let t be a ground Λ-expression with a WHNF and let t be reducible to s in one step. Then we have*

i) $nr\#(t) \geq nr\#(s)$

ii) *If the reduction was a necessary one, then* $nr\#(t) > nr\#(s)$

iii) *If* t' *is a necessary redex in* t*, and* t' *is different from* t*, then* $nr\#(t') > nr\#(t)$

For a proof of this lemma see ([SSPS95]).

The defined notion of ω-equality is a sensible one, i.e., it is an equivalence relation and substitutivity holds:

Theorem 4. *The relation* $=_k$ *and* $=_\omega$ *have the following properties*

i) *We have* $=_k \supseteq =_{k+1}$

ii) *All* $=_k$ *are equivalence relations. Thus* $=_\omega$ *is an equivalence relation*

iii) *We have substitutivity: Let* d *and* e *be ground expressions. Then* $d =_\omega e$ *implies that for all term contexts* $C[.]$ *we have* $C[d] =_\omega C[e]$

Proof. We prove only substitutivity: Suppose this is false, then there is an n-ary term context $C[.]$, some expressions d_i and $e_i, i = 1, \ldots, n$, and a number k such that $d_i =_k e_i$ for all i, and $C[d_1, \ldots, d_n] \neq_k C[e_1, \ldots, e_n]$.

We select a counterexample, such that k is minimal.

We have $k > 0$. We can assume that $C[d_1, \ldots, d_n]$ has a WHNF, since it is not possible by assumption that both $C[e_1, \ldots, e_n]$ and $C[d_1, \ldots, d_n]$ do not have a WHNF. We select a counterexample with smallest number $nr\#(C[d_1, \ldots, d_n])$.

We can assume that the body of the supercombinator that represents $C[.]$ contains only the argument variables as symbols.

If $C[d_1, \ldots, d_n]$ is not in WHNF, then consider some necessary redex.

If this redex r_d is not the whole term $C[d_1, \ldots, d_n]$, then this redex is not within some d_j, since then we can construct a smaller counterexample by simply reducing d_j. The interesting case is that the redex r_d can be viewed as constructed by some term context with arguments among the d_i, and we can identify a corresponding subexpression r_e in $C[e_1, \ldots, e_n]$. Lemma 3 shows that $nr\#(r_d)$ is strictly smaller than that $nr\#(C[d_1, \ldots, d_n])$, hence we derive that $r_e =_k r_d$. We can construct a new term context C' instead of C that has a new hole at the redex r_d. Using the induction assumption and reducing once r_d, we get a smaller counterexample.

Hence $C[d_1, \ldots, d_n]$ is a (necessary) redex.

There is some j, such that d_j is the head of a necessary redex. Without loss of generality, we assume that $j = 1$. By definition and assumption, we have $C[d_1, e_2, \ldots, e_n] =_k C[e_1, e_2, \ldots, e_n]$. Hence by transitivity of $=_k$, we can choose as new counterexample the

expressions $C[d_1, d_2, \ldots, d_n]$ and $C[d_1, e_2, \ldots, e_n]$. Thus we can assume that d_1 and e_1 are identical expressions. If there are some strictness restrictions that arguments of d_1 must be in WHNF, then we can argue as above that there are corresponding subexpressions in $C[d_1, \ldots, d_n]$ and $C[d_1, e_2 \ldots, e_n]$. These form a smaller counterexample, since the corresponding subexpression in $C[d_1, \ldots, d_n]$ must be in WHNF, since the expression itself is a redex. Now the reduction is a supercombinator application without strictness restrictions, and the same reduction can be performed in both expressions. We perform one reduction step, and get a new context and a smaller counterexample.

Now we have the case that $C[d_1, \ldots, d_n]$ is in WHNF.

Without loss of generality we can assume that d_1 is the head of the expression. We can use transitivity of $=_k$ and generate (as above) a new counterexample, $C[d_1, d_2 \ldots, d_n]$ and $C[d_1, e_2, \ldots, e_n]$.

If $C[d_1, \ldots, d_n]$ is in CWHNF, then we have also that $C[d_1, e_2 \ldots, e_n]$ is in CWHNF, and there are corresponding arguments in $C[d_1, \ldots, d_n]$ and $C[d_1, e_2, \ldots, e_n]$ that form a smaller counterexample for $k - 1$.

If $C[d_1, \ldots, d_n]$ is in WHNF, but not in CWHNF, then neither $C[d_1, \ldots, d_n]$ nor $C[d_1, e_2, \ldots, e_n]$ are in SCWHNF and we have the application case. There is some a, such that $(C[d_1, \ldots, d_n]\ a) \neq_{k-1} (C[d_1, e_2, \ldots, e_n]\ a)$. Thus we can construct a new term context for smaller k, and thus get a contradiction to minimality of k.

The cases are exhausted, and we have reached the final contradiction.

Analysing the proof, we have also shown that the following holds:

Lemma 5. *For every n-ary term context $C[.]$, and expressions d_i, e_i, $i = 1, \ldots, n$, and every number k, we have that $d_i =_k e_i$ for all i implies $(C[e_1, \ldots, e_n]) =_k (C[d_1, \ldots, d_n])$.*

Now we can prove a characterisation of ω-equality. Let s and t be expressions. We say that s and t can be distinguished by contexts, $s \neq_{con} t$, iff there is some context $C[.]$, such that one of the following holds:

$C[s]$ reduces to 1, whereas $C[t]$ reduces to 0.

$C[s]$ reduces to 1, whereas $C[t]$ is undefined.

$C[s]$ is undefined, whereas $C[t]$ reduces to 1.

Theorem 6. *Let s and t be two ground expressions. Then $s =_\omega t$ iff not $s \neq_{con} t$*

Proof. If $s =_\omega t$, then Theorem 4 shows that s and t cannot be distinguished by contexts.

For the other direction assume that there are s and t, such that not $s \neq_{con} t$, but $s \neq_\omega t$ holds. Then there is some counterexample with minimal k, such that not $s \neq_{con} t$, and $s \neq_k t$.

We can assume that one of s or t has a WHNF. If neither s nor t has a SCWHNF, then we can find some ground expression x, such that $(s\ x) \neq_{k-1} (t\ x)$. Since these expressions also satisfy $(s\ x) \neq_{con} (t\ x)$, we have a smaller counterexample. Hence one of s or t has a SCWHNF.

If both s and t have an SCWHNF $\mathtt{c}\ s_1 \ldots s_n$ and $\mathtt{c}\ s_1 \ldots s_n$, respectively, then there is some index j, such that $s_j \neq_{k-1} t_j$. It is obvious, that the constructor and the number of arguments must be identical. Using a context of the form $\mathtt{case}_T\ s \ldots$, such that the corresponding function in the `case`-expression projects the j^{th} component, we see that also $(s_j\ x) \neq_{con} (t_j\ x)$, hence we would have a smaller counterexample.

Without loss of generality, the last case is that s has a SCWHNF, whereas t has no SCWHNF. But then we can construct a context that exhibits a different behaviour: We construct a `case`-expression, where s is cased, but the function corresponding to the constructor yields a 1. Then our operational semantics shows that application of this expression to t yields `Bot`.

We consider the relation of ω-equality to the observational equality of ([Mor68], [Abr90]). Let $=_{trm}$ be the equality that is defined as follows: s and t are equal w.r.t. $=_{trm}$, if for all contexts $C[.]$, $C[s]$ has a WHNF iff $C[t]$ has a WHNF.

Theorem 7. *Let s and t be ground expressions. Then $s =_{trm} t$ implies that $s =_\omega t$.*

Proof. Assume the theorem is false. Then there are two expressions s and t, such that for all contexts $C[.]$, we have that $C[s]$ has a WHNF iff $C[t]$ has a WHNF, but there is some context D, such that the condition on distinguishing expressions for $=_{con}$ does not hold. The only possibility is that $D[s]$ reduces to 1, whereas $D[t]$ reduces to 0. However, it is no problem to construct a context using a `case` that terminates for 0, but does not terminate for the 1. This is a contradiction.

Now we have also an analogue to the "full abstraction theorem" in [Abr90]

Corollary 8. *Let $\sim$ be the largest binary relation on ground expressions, such that substitutivity holds, i.e., for all program contexts $C[.]$ and all expressions d, e, $d \sim e$ implies $C[d] \sim C[e]$. Furthermore, $1 \sim 0$ and $1 \sim$ `Bot` is false. Then $\sim = =_\omega$.*

Proof. Let $d \sim e$. Then substitutivity shows that $d \neq_{con} e$, holds, too. But then we have $d =_\omega e$ by Theorem 6.

The results of [How89] cannot be directly applied, since a WHNF may start with a supercombinator, and this form is not compatible with the results in [How89], since it is not possible

to extract the arguments. If we translate our language into the lambda-calculus with constructors, and define a different notion of canonical, then the observational equivalence in [How89] appears to be the same as ω-equality.

For the calculus CPE we need a definition for comparing ω-equal expressions with respect to their length of normal-order reductions to WHNF. In order to have substitutivity, the comparison has to be done arbitrarily deep.

Definition 9. Let s and t be ground expressions, such that $s =_\omega t$. Then

- $s \geq_{nr,0} t$ if either s and t have no WHNF, or s and t have a WHNF and $nr\#(s) \geq nr\#(t)$.
- $s \geq_{nr,k} t$ if one of the following holds
  - s and t have no WHNF
  - $s \geq_{nr,0} t$ and both s and t have a CWHNF $\texttt{con}\ s_1 \ldots s_n$, and $\texttt{con}\ t_1 \ldots t_n$, respectively, and $s_i \geq_{nr,k-1} t_i$ for all $i = 1, \ldots, n$.
  - $s \geq_{nr,0} t$ and both s and t have a WHNF, but no SCWHNF, and $(s\ x) \geq_{nr,k-1} (t\ x)$ for all ground expressions x.
- $s \geq_{nr,\omega} t$ iff $s \geq_{nr,k} t$ for all $k \geq 0$.

In a later section, we will give a non-trivial sufficient criterion for two terms being in this relation using a variant of the tableau calculus.

Lemma 10. *The definition above defines a sensible ordering:*

- *The relation $\geq_{nr,\omega}$ is transitive.*
- *Furthermore, for every program context $C[.]$, we have that $s \geq_{nr,\omega} t$ implies $C[s] \geq_{nr,\omega} C[t]$*

Proof. It is easy to prove transitivity, hence we omit the proof. We prove the second claim.

Assume, the second claim is false. Then the following holds. There are some d_i and c_i, and some k, such that $d_i \geq_{nr,k} e_i$ for all i, but not $C[d_1, \ldots, d_n] \geq_{nr,k} C[e_1, \ldots, e_n]$.

Theorem 4 shows that ω-equality holds. Definition 9 shows that both $C[d_1, \ldots, d_n]$ and $C[e_1, \ldots, e_n]$ have a WHNF. We select k as small as possible, as second and third component of a measure we select $nr\#(C[d_1, \ldots, d_n]$ and then $nr\#(C[e_1, \ldots, e_n]$ minimal.

We can proceed in almost the same way as in the proof of Theorem 4 to reach a contradiction.

4 Set Descriptions

The calculus needs a description method for sets of expressions that restricts the possible instantiations of variables. We separate the complications that stem only from the set description method from the calculus part and thus give some abstract properties that a set description methods should have.

Examples for set descriptions are types, where for example $list(\alpha)$ denotes the set of all list-expressions. Another example for a set description method are the evaluation contexts in [SSS96], which can represent the set of all expressions without WHNF, the set of all lists, and also the set of all finite lists.

Abstract set descriptions are assumed to be unary predicates. The following properties should hold.

- For a set description S and a term s, If $S(s)$ holds and $s =_\omega t$, then $S(t)$ holds, too.
- There are the predicates `Bot` and `Top` with the meaning that for some expression s, `Bot`(s) is true, iff s has no WHNF, and that `Top`(s) is always true.

Example 2. Some interesting set descriptions are those, which describe certain kinds of lists. For example, the set of all lists, finite lists, or infinite lists, respectively, can be recursively described as follows

$$lists = \texttt{Bot} + \texttt{Nil} + \texttt{Top} : lists.$$
$$finlists = \texttt{Nil} + \texttt{Top} : lists.$$
$$inflists = \texttt{Bot} + \texttt{Top} : lists.$$

The mechanism of set descriptions shall also be used for abstract expressions. In general, the variables in the abstract expressions have to be restricted in some way. This will be done by *set-environments* which are written in the form $\{S_1(x_1), \ldots, S_n(x_n)\}$.

Definition 11. We define substitutions and instances.

- A *raw substitution* σ is a mapping from variables to Λ-expressions. This mapping will also be applied to abstract Λ-expressions by $\sigma\ \texttt{c} = \texttt{c}$ if `c` is a constant, and $\sigma\ (f\ a) = (\sigma(f))\ (\sigma(a)))$
- A *ground substitution* σ is a mapping from variables to ground Λ-expressions.
- A *ground instance* of a Λ-expression e is a ground Λ-expression that is derived by applying a ground substitution to the expression.
- Given a set-environment U, a substitution σ is said to be compatible with U (U-compatible), iff for all variables x, if $S(x)$ is in U, then we have $S(\sigma(x))$.

- A U-compatible ground instance of some Λ-expression e is a ground term $\sigma(e)$, where σ is a U-compatible substitution.

Now we can define the semantics of an abstract expression s given a set-environment U. This is defined as the set $\gamma_U(s) = \{\sigma(s) \mid \sigma \text{ is a } U\text{-compatible ground substitution.}\}$

We apply the set-description S also to abstract expressions t given a set-description U. We say $S(t)$ holds w.r.t U, if for all U-compatible ground instances t' of t, we have $S(t')$.

A substitution σ that instantiates abstract Λ-terms for variables is said to be U-compatible for a given set-environment U, if for all variables x: $S(x)$ in U implies that $S(\sigma(x))$ holds w.r.t. U.

We also require an algorithm for computing a term covering on the set descriptions: $tcov(S)$ should return a set of expressions R (the covering), and a set-environment U for the new variables contained in the expressions in R. The expressions in R may be `Bot`, or expressions of the form $(\texttt{c}\ x_1 \ldots x_n)$, where `c` is a constructor and the variables are new ones. The following condition should hold for $tcov(S) = (R, U)$:

$$\{t | t \text{ is ground and } S(t)\} =_\omega \bigcup_{r \in R} \gamma_U(r).$$

The set-environment `Bot` has as term covering only the constant `Bot`. We also assume that whenever a variable is constrained by `Bot`, this variable can be replaced by `Bot`. The set-environment `Top` has a term covering consisting of `Bot`, and of all possible terms of the form $(\texttt{con}\ x_1 \ldots x_n)$, where `con` is a constructor of arity n, and the new variables have a set-environment `Top`.

5 Equality Tableaux

In this section we will define the deduction calculus, which we will call **CPE** (**c**alculus for **p**rogram **e**quivalence).

In general we are not interested in comparing non well-typed expressions. A type systems ala Milner/Mycroft for functional languages could be used as a safe method to recognise well-typed expressions and functions.

Definition 12. An *EQ-tableau* is a pair (T, U) of a tree T and a set-environment U. The nodes of the tree are marked with pairs (s, t) where s and t are Λ-expressions. The pairs are denoted as $s \doteq t$.

Furthermore, the edges are either unmarked or marked with "R" (for necessary reduction of the right hand expression) or "L" (for necessary reduction of the left hand expression) or "C" (for constructor reduction) or "F" (function equality test). Leaves may have a second label "loop" or "bot=bot".

Let s and t be abstract expressions. An equation $s \doteq t$ at some node in an EQ-tableau *holds* (w.r.t. ω-equality), iff for all U-compatible ground substitutions σ: $\sigma(s) =_\omega \sigma(t)$ holds.

An EQ-tableau is *sound* iff for every (inner) non-leaf node N: If all the equations at the sons hold, then the equation at the node N holds.

An EQ-tableau is *complete* iff for every (inner) non-leaf node N and all U-compatible ground substitutions: If $\sigma(s) =_\omega \sigma(t)$ for the equation at the father node, then there is some son with equation $s' =_\omega t'$, and a U-compatible substitution σ', such that $\sigma(x) = \sigma'(x)$ for all variables $x \in Var(s,t)$, and $\sigma'(s') =_\omega \sigma'(t')$.

Definition 13. If in an EQ-tableau all leaves are either marked with an equation of syntactically equal expressions or have a label "loop" or "bot=bot", then it is called *closed*.

Now we present the rules of the non-deterministic calculus CPE that is intended to construct closed EQ-tableaux using rules which transforms EQ-tableaux into new ones. A new EQ-tableau will always originate from an old one by extending a path with new leaves, and perhaps the set-environment may be extended. Note that we do never extend leaves that are marked with "loop" or "bot=bot". An expansion rule is called *sound* (*complete*), if it transforms sound (complete) EQ-tableaux into sound (complete) ones. In the following we will present the expansion rules for the EQ-tableaux.

The important property is soundness, since this allows us to conclude that a closed tableau is a description of a proof of ω-equality of the root equation. Completeness of a tableau can be used, if we want to derive that the root equation is not ω-equal.

5.1 Reduction Rules

The common way of extending an EQ-tableau is to perform a reduction step on a leaf and to add the result as a direct son. This enables us to reduce expressions with the same $\delta-$reduction as in the ground case. The general situation for the reductions is that there is some leaf and in one of the terms there is some redex that will be reduced. The EQ-tableau is then extended with a new leaf marked by an equation, where the redex is replaced by its reduct. The new edge is marked with "R" or "L", if the redex is a necessary one in the right or left expression of the equation, respectively.

Lemma 14. *The expansion rule in this section is sound.*

Proof. This holds, since ω-equality is defined based on the reductions using the operational semantics and the supercombinator definitions.

5.2 Casing a Free Variable

Suppose we have a leaf marked with $C[(\mathtt{case}_A\ x)]$, where x is a variable, and $S(x)$ occurs in U. Let $(R, U_S) = tcov(S)$ be the term covering of S.

Then a new EQ-tableau is constructed by extending the EQ-tableau at this path with a leaf for every term r in R. The sons are constructed by replacing x by r. The new set-environment U_S is added to the global set-environment.

The edges will not be marked.

Lemma 15. *The expansion rule in this subsection is sound and complete.*

Proof. Soundness: Assume that the equations at the sons hold, then we have to show that the equation at the father node also holds. Let $s \doteq t$ be the equation at the father node, let σ be a U-compatible ground substitution. Consider the instantiation $\sigma(x)$ for x. Let $tcov(S) = (R, U_x)$ be the term covering, and let $U' = U \cup U_x$ and there is some $r \in R$, such that there is an extension σ' of σ, such that σ' and σ differ only in the (new) variables of r, σ' is u'-compatible, and $\sigma(x) =_\omega \sigma'(r)$. There is some son with equation $s' \doteq t'$, where s' and t' are constructed by replacing x by r. We have $\sigma(s) =_\omega \sigma'(s')$ and $\sigma(t) =_\omega \sigma'(t')$. Since the equation at the son holds, the equation at the father node holds, too. This can be done for all U-compatible substitutions σ, hence the equation at the father holds.

Completeness: Assume the equation $s \doteq t$ at the father node holds. Let σ be a U-compatible ground substitution, such that $\sigma(s) =_\omega \sigma(t)$. Then using the properties of a term covering, we can find some r, and an extension σ' of σ, such that $\sigma(x) =_\omega \sigma'(r)$. The equation at the corresponding son then holds under σ'.

5.3 Adding Arguments

> If there is an equation $d \doteq e$, and for d and e we have that they are either in WHNF, but not SCWHNF, or equal to `Bot`. Then add a new leaf marked by $(d\ x) \doteq (e\ x)$ and add $\texttt{Top}(x)$ to U, where x is a new variable. The edge has to be marked with "F".

Adding arguments is sound and complete, which follows from the definition of ω-equality.

5.4 Approximations

> Let t be a subexpression of some term of the equation in some leaf node. Then the new leaf of the EQ-tableau can be derived from the old leaf as follows: Let x be a new variable. Replace all subexpressions in the pair which are syntactically equal to t by x. If we can compute some set description S, such that $S(t)$ holds, then we add $S(x)$ to U.
>
> The edge will not be marked.

The soundness of approximation is obvious, since all instances of the old leaf are instances of the new leaf. However, completeness may be lost.

The computation of $S(t)$ is not made explicit in this paper. It has to be done in the description of the module that handles the set descriptions.

5.5 Constructor Decomposition

If there is a leaf marked by $\texttt{con}\ d_1 \ldots d_n \doteq \texttt{con}\ e_1 \ldots e_n$, then add n leaves marked by the equations $d_i \doteq e_i$. The edges will be marked with "C".

It is obvious by the definition of ω-equality, that this rule is sound and complete.

5.6 Function Decomposition

If there is a leaf marked by $f\ d_1 \ldots d_n \doteq f\ e_1 \ldots e_n$, where f is some expression, then add n leaves marked by the equations $d_i \doteq e_i$. The edge is not marked.

This rule is sound, but in general not complete.

5.7 Expression replacement

If there is a leaf marked by $d \doteq e$, where d can be written as $C[d']$, and there is an expression d'', such that $d' =_\omega d''$, and for all U-compatible substitutions σ, we have $\sigma(d') \geq_{nr,\omega} \sigma(d'')$. Then a new leaf can be added, where d' in d is replaced by d''

This rule is sound and complete. The extra condition on the length of a normal order reduction is not effective, but we will show in section 5.13 how this property can be established using a variant of the CPE-calculus.

5.8 Loop-detection

There is a rule called *loop-detection* for discovering loops in the evaluation.

> Suppose we have a leaf that is marked with $t_1 \doteq t_2$, such that t_1 or t_2 is not in WHNF. If there is some ancestor node marked with $e_1 \doteq e_2$, such that there is some U-compatible abstract substitution σ and $\sigma(e_1 \doteq e_2)$ is syntactically equal to $t_1 \doteq t_2$, and on the path from that node to the leaf one of the following possibilities holds:
>
> - There is an edge that is marked with "L" and an edge marked with "R".
> - There is some edge marked with "C".
> - There is some edge marked "F".

Furthermore, if a function decomposition is on the path, then there must also be an edge marked "F" or "C" on the path.

Then the leaf will be marked "loop".

We will call the upper node for a leaf the corresponding *looping node*.

5.9 Bot Detection

The rule in this subsection is intended to detect situations of the form $\perp=\perp$.

Suppose we have a node N, such that for all descendant leaves the following holds. For every leaf (marked with $t_1 \doteq t_2$), there is a ("looping") node marked with $e_1 \doteq e_2$ that is on the path from N to that leaf and the following holds:

The expressions t_1 and t_2 are not in WHNF. Furthermore there are necessary redexes r_i in t_i for i = 1,2, such that there is some U-compatible abstract substitution σ and $\sigma(e_i) = r_i$ for $i = 1, 2$. Furthermore the following holds:

i) On the path from the looping node to the leaf there is an edge that is marked with "L", or r_1 is a proper subterm of t_1, or t_1 is equal to `Bot`.

ii) On the path from the looping node to the leaf there is an edge that is marked with "R", or r_2 is a proper subterm of t_2, or t_2 is equal to `Bot`.

The only permitted expansion rule on this path are reduction, casing free variables, expression replacement and approximation.

If this holds for all the leaves below N, then all leaves below N will be marked "bot=bot".

5.10 Failure Rules

There are situations, where the EQ-calculus is unable to prove equivalence by further expanding the tableau:

- There is a leaf with equation $d \doteq e$ and d and e are in CWHNF, and the constructors are different, or the number of arguments are different.
- There is a leaf with equation $d \doteq$ `Bot` or `Bot` $\doteq d$ and d is in CWHNF.
- There is a leaf with an equation where one term is in SCWHNF and the other one is in WHNF.

5.11 Soundness of CPE

In this section we prove the soundness of the calculus, i.e., we prove that if an EQ-tableau is closed, then the two terms at the root are ω-equal. In the following the notation p_L and p_R is used for the left and right term in an equation p.

Lemma 16. *Let p be an inner node in an EQ-tableau. Assume, that the equations at the sons hold w.r.t. $=_k$.*

If the sons are constructed using constructor decomposition or adding arguments, then the equation at the father holds w.r.t. $=_{k+1}$. Otherwise, the equation at the father holds w.r.t. $=_k$.

Proof. In the case of constructor decomposition this follows from the definition of $=_k$. In the case of adding arguments, the preconditions of this rule guarantee that the definition of ω-equality can be applied. In the other cases, the claim follows by refining the soundness proofs for the rules. In the case of function decomposition, we have to apply Lemma 5.

Theorem 17. *Let p be the equation $p_L \doteq p_R$ in Λ, and let U be a set-environment for the variables in p. Let there be a closed EQ-tableau with the equation p at the root.*

Then the equation $p_L =_\omega p_R$ is valid w.r.t U.

Proof. Assume that the theorem is false. Then the equation at the root does not hold, i.e., there is a U-compatible ground substitution τ, such that $\tau(p_L) \neq_\omega \tau(p_R)$. We choose a node with the smallest possible k, such that the equation at this node does not hold w.r.t. $=_k$. Lemma 16 shows that for every non-leaf node, there is some son with an equation that is also $\neq_k$ for some instance. Hence there is some leaf in the tree, such that the equation at this leaf does not hold w.r.t. $=_k$. Since the tableau is closed, this leaf must be marked "loop" or "bot=bot".

First we treat the case that the label is "loop".

Let q be the equation at this leaf and let σ be a U-compatible ground substitution, such that $\sigma(q_L) \neq_k \sigma(q_R)$.

We can select σ, such that one of the instances $\sigma(q_L)$ or $\sigma(q_R)$ has a WHNF. We make a further minimisation among the witnessing leaves such that as second and third component of the lexicographical ordering, we minimise the number of necessary reductions to a WHNF of the left term and then of the right term. If one of the terms has no WHNF, then we assume the component is ∞. As fourth component we take a leaf with minimal distance of its looping node from the root.

Since the chosen leaf has a label "loop", $\sigma(q_L) \doteq \sigma(q_R)$ is also a concretisation of the equation in the looping node N of this leaf. There is a path starting with N, such that the equations at every node on this path do not hold w.r.t. $=_k$. Since the proof requires that the minimality

is preserved, we must have a closer look at the different rules that may be applied along the path. There cannot be any C-edges or F-edges on this path, since otherwise we could find a node (and hence also a leaf) with a concretisation that is not equal w.r.t. $=_{k-1}$, which is a contradiction to minimality of k. Approximation and casing free variables do not change the instances, and also uniquely determine the next node on the path. Since a function decomposition on a looping path enforces that there is also a "C" or an "F"-edge, a function decomposition does not appear on the path.

This path leads to a further leaf N_r marked with the pair r that has instances that are not $=_k$-equal. If the leaf N_r has as label "bot=bot", then we can use the arguments for this case (see below).

If the leaf N_r has as label "loop", then we have to check the length of normal order reductions of concretisations. All the now possible expansion rules do not increase the number of normal order reductions, but may decrease them. For the rule "replacement of expressions, this is shown in Lemma 10. Since an L-edge or an R-edge corresponds to a strict decrease of one of the normal order reduction lengths, there are either no R-edges or no L-edges on the path from N to N_r, since otherwise we have a smaller witness w.r.t. the length of normal order reductions. By definition of looping node, the new leaf N_r must thus have a looping node that is above the node N. But then we have a contradiction to the minimality of the fourth component of our measure.

Now we treat the case that the label at a minimal leaf is "bot=bot".

We use the same notation as in the other case. The assumption that the theorem is false implies that we can find a U-compatible instance of the equation at the leaf, such that the equation does not hold, and hence one expression has a WHNF. Then we minimise the number of necessary reductions to a WHNF of the left term and then of the right term by selecting among all such leaves. If one of the terms has no WHNF, then we assume the component is ∞. Now the pair of necessary redexes at the leaf is a concretisation of the equation at the looping node. The preconditions for applying the Bot-detection rule enforce that there a path leading from the looping node to a further leaf also labelled "bot=bot". Lemma 3 shows that the minimality assumption cannot hold. The only possibility is that both terms are semantically $\bot$, which is the intended contradiction.

5.12 Strategies and Heuristics

Since the calculus CPE is non-deterministic, in general there are many different possibilities to continue the construction of the tableau. We will give some hints on a strategy and some heuristics that have a good chance to result in a closed tableau.

- In the rule "reducing an expression", it is sensible to reduce the expression only, if it is not in WHNF. Furthermore, it is preferable to reduce only necessary redexes. There may be a choice whether the left or right term is to be reduced first. It is not clear which one should be reduced faster. The examples show that a fair strategy gives good

results.

- Casing a free variable is in general ony sensible, if the case expression becomes a necessary redex after the instantiation.
- Approximation for a term t should be applied only in the case that a necessary redex is of the form : $\mathtt{case}_A\ t\ e_1 \ldots e_n$. In general this rule is applied in connection with casing free variables.
- constructor decomposition should be done immediately.
- function decomposition should be done with care, since it has a high potential of being imcomplete.
- The detection rules should be applied as early as possible, though there may be a run time penalty.
- Expression replacement requires some external control.

The failure rules may also be used to show that the expressions at the root are not ω-equal, which is only the case, if all rule applications in the tableau are complete, and all leaves are failure leaves.

5.13 A calculus for $\geq_{nr,\omega}$

It is obvious that if an expression s can be reduced to t, then $s \geq_{nr,\omega} t$, and we can apply expression replacement.

A method to prove $s \geq_{nr,\omega} t$ is to construct a closed CPE-tableau for the two equation $s \doteq t$ and to analyse the tableau. For this we have to make some restrictions on the construction of the tableau. There should be no function decomposition. Furthermore, on the right hand side, there should be only necessary reductions and no expression replacements.

In the tableau we have to identify special nodes, called M-nodes. These are defined as: i) the root, ii) nodes where left and right term are in WHNF, iii) nodes where a constructor decomposition has been applied, and iv) EQ-leaves that are not labelled $\mathtt{Bot} \doteq \mathtt{Bot}$. For the paths we assume that for "loop"-leaves, the path is continued at the looping node. A path is called cyclic, iff there is no M-node between the "loop"-leaf and its looping node.

The following paths should obey the restriction that the number of "L"-marks on edges is not less the number of "R"-edges:

- All non-cyclic paths from an M-node to all the descendent M-nodes.
- For a cyclic path, the part between the leaf and the looping node.

We give a proof of correctness of this method:

Proposition 18. *Let s and t be two expressions and let U be a set environment, such that there is a closed EQ-tableau, and the conditions above hold.*

Then for all U-compatible instances s' and t' of s and t, respectively, the relation $s' \geq_{nr,\omega} t'$ holds.

Proof. Assume that the proposition is false. Then there is a counterexample with root $s \doteq t$, and U-compatible instances s' and t' of s and t, respectively. We select the counterexample, such that $s =_\omega t$, but not $s' \geq_{nr,k} t'$. The instances s' and t' have a WHNF, since otherwise they are in the relation $\geq_{nr,\omega}$. As second and third components we select $nr\#(s')$ and then $nr\#(t')$ as minimal. Now consider a path from the root, such that there is a corresponding path for the instances s' and t'. If the path in the EQ-tableau reaches a WHNF for the right and left hand side or an EQ-leaf, then the conditions above show that a smaller counterexample w.r.t. the lengths of reductions can be constructed. The path cannot reach a "bot=bot"-leaf, since the instances have a WHNF. A cyclic path is not possible, since then we can find a smaller counterexample.

There are no other cases, hence the proposition is proved.

Example 3. We give a non-trivial example for two expressions that are in the $\geq_{nr,\omega}$-relation. We use the following definitions:

$$append\ xs\ ys = lcase\ xs\ ys\ (consappend\ ys)$$
$$consappend\ ys\ z\ zs = z : (append\ zs\ ys)$$

We show that $append\ xs\ ys \geq_{nr,\omega} xs$, if it is known that xs is an infinite list.

We build a closed tableau for the equation $append\ xs\ ys \doteq xs$ where the set environment consists of $inflist(xs)$.

$append\ xs\ ys$	$\doteq$	xs
$lcase\ xs\ ys\ (consappend\ ys)$	$\doteq$	xs
– $xs = \texttt{Bot}$		
$\texttt{Bot}$	$\doteq$	$\texttt{Bot}$
– $xs = z : zs$, $inflist(zs)$		
$consappend\ ys\ z\ zs$	$\doteq$	$z : zs$
$z : (append\ zs\ ys)$	$\doteq$	$z : zs$
– one leaf is $z \doteq z$		
$append\ zs\ ys$	$\doteq$	zs

This closed tableau satisfies the conditions and thus shows that $append\ xs\ ys$ can be replaced by xs in any CPE-tableau, if xs is an infinite list.

5.14 Examples

We demonstrate the effect of the calculus CPE on small examples.

Example 4. Assume we have the function f defined as $f\ x\ y = x : y$. Then we can easily construct a tableau that proves that f and $:$ are equal: We start with $f \doteq :$. Then we add an argument getting $f\ x \doteq (x :)$ and as set-environment $\texttt{Top}(x)$. The next step is $f\ x\ y \doteq (x : y)$ with set-environment $\{\texttt{Top}(x), \texttt{Top}(y)\}$. One reduction step gives $x : y \doteq (x : y)$, which is an equation of syntactically equal terms.

Example 5. Terms of a different type can be equivalent: Assume we have the functions f, g defined as $f\ x = f\ (f\ x)$ and $g\ x = g\ x$. The initial equation $f\ x \doteq g\ x$ gives a new leaf $f\ (f\ x) \doteq g\ x$. Approximating $(f\ x)$ by y gives $f\ y \doteq g\ x$, which in turn again gives a further leaf $f\ (f\ y) \doteq g\ x$. Since we have L-edges and R-edges, we can close the tableau.

Example 6. We show that two differently defined fixpoint combinators Y and F are ω-equal. Let the definitions be:

$$
\begin{array}{l}
Y\ f = f(Y\ f) \\
D\ f\ x = f(x\ x) \\
F\ f = (D\ f)(D\ f)
\end{array}
$$

We start the tableau with $Y \doteq F$.

$$
\begin{array}{lcl}
Y & \doteq & F \\
Y\ f & \doteq & F\ f \\
Y\ f & \doteq & (D\ f)(D\ f) \\
f(Y\ f) & \doteq & f\ ((D\ f)(D\ f)) \\
(Y\ f) & \doteq & (D\ f)(D\ f)
\end{array}
$$

The last equation has an identical ancestor, and there are L- and R-edges on the path. The global environment is $U = \{\texttt{Top}(f)\}$, which has no influence. Thus this tableau shows that Y and F are ω-equal. It is noteworthy that D cannot be typed in the Milner/Mycroft type system. As a generalisation, it is obvious that all fixpoint combinators are ω-equal.

Example 7. This example should demonstrate that the use of set environments is necessary for the correctness of the calculus. Let ID be the combinator defined by $ID\ x = x$. If we consider the initial equation $ID\ xs \doteq lcase\ xs\ \texttt{Nil}\ (:)$, then a naïve version of CPE might conclude as follows:

$ID\ xs$ $\doteq$ $lcase\ xs$ `Nil` (:)
xs $\doteq$ $lcase\ xs$ `Nil` (:)

There are three cases.

i)

`Bot` $\doteq$ $lcase$ `Bot Nil` (:)
`Bot` $\doteq$ `Bot`

ii)

`Nil` $\doteq$ $lcase$ `Nil Nil` (:)
`Nil` $\doteq$ `Nil`

iii)

$y : ys$ $\doteq$ $lcase\ y : ys$ `Nil` (:)
$y : ys$ $\doteq$ (:) $y\ ys$
$y : ys$ $\doteq$ $y : ys$

This would prove that $ID\ xs$ is ω-equal to $lcase\ xs$ `Nil` (:), which is not true, since ID accepts also other arguments than lists. The calculus CPE prevents this error by requiring for example a set environment $\{lists(xs)\}$, which would make the tableau above correct. The interpretation is then that for all the correct instances of xs, i.e., for all lists, the equation holds.

Example 8. We give an example demonstrating the power of the calculus CPE. The computation does not show the labels at the edges. Since almost all reduction steps are at necessary positions, every reduction will mark the edges with label "L" or "R". In several cases, we make more than one reduction in one step. We do not show the set-environment for l or for y. It is possible to use `Top` for l and y, since the covering expressions that lead to a non-well-typed term behave equivalent for the right and left expression, and the $\perp$-case subsumes these non-well-typed cases.

We show that $filter\ p\ (map\ f\ l)$ is ω-equal to $map\ f\ (filter\ (p \circ\ f)\ l)$. Using the Milner/Mycroft-type system, the type of both expressions as functions of three arguments p, f, l is $(\alpha \to Bool) \to (\beta \to \alpha) \to [\beta] \to [\alpha]$.

Let the function definitions be:

$$filter\ p\ as = \texttt{lcase}\ as\ \texttt{Nil}\ (fil2\ p)$$
$$fil2\ p\ a\ as = if\ (p\ a)\ (a\ :\ filter\ p\ as)\ (filter\ p\ as)$$
$$(f\ \circ\ g)\ x = f\ (g\ x)$$
$$map\ f\ as = \texttt{lcase}\ as\ \texttt{Nil}\ (map2\ f)$$
$$map2\ f\ b\ bs = (f\ b)\ :\ (map\ f\ bs)$$

For simplicity we assume that map is defined to be strict in its second argument. It is no problem to construct a tableau also for the map-function without defining the strictness, as can be seen by replacing the right hand side map in the tableau by `lcase...`

Now we develop the closed tableau.

$filter\ p\ (map\ f\ l)$ $\doteq$ $map\ f\ (filter\ (p \circ f)\ l)$

$filter\ p\ (\texttt{lcase}\ l\ \texttt{Nil}\ (map2\ f))$ $\doteq$ $map\ f\ (\texttt{lcase}\ l\ \texttt{Nil}\ (fil2\ (p \circ f)))$

$\texttt{lcase}\ (\texttt{lcase}\ l\ \texttt{Nil}\ (map2\ f))$ $\doteq$ ...
$\texttt{Nil}\ (fil2\ p)$

— Casing the free variable l.

---

1) $l = \texttt{Bot}$

$\texttt{lcase}\ (\texttt{lcase}\ \texttt{Bot}\ ...)\ ...$ $\doteq$ $map\ f\ \texttt{Bot}$

$\texttt{Bot}$ $\doteq$ $\texttt{Bot}$

---

2) $l = \texttt{Nil}$

$\texttt{lcase}\ (\texttt{lcase}\ \texttt{Nil}\ \texttt{Nil}\ ...)\ \texttt{Nil}\ ...$ $\doteq$ $map\ f\ \texttt{Nil}$

$\texttt{Nil}$ $\doteq$ $\texttt{Nil}$

---

3) $l = a : as$

$\texttt{lcase}\ (\texttt{lcase}\ (a : as)\ \texttt{Nil}$ $\doteq$ $map\ f\ (\texttt{lcase}\ (a : as)\ \texttt{Nil}$
$(map2\ f))\ \texttt{Nil}\ (fil2\ p)$ $fil2\ (p \circ f)))$

$\texttt{lcase}\ (map2\ f\ a\ as)\ \texttt{Nil}\ (fil2\ p)$ $\doteq$ $map\ f\ (if\ (p(f\ a))$
$(a\ :\ (filter\ (p \circ f)\ as))$
$(filter\ (p \circ f)\ as))$

$\texttt{lcase}\ (f\ a)\ :\ (map\ f\ as)$ $\doteq$ ...
$\texttt{Nil}\ (fil2\ p)$

$fil2\ p\ (f\ a)(map\ f\ as)$ $\doteq$ ...

$if\ (p\ (f\ a))$ $\doteq$ ...
$(f\ a)\ :\ (filter\ p\ (map\ f\ as))$
$(filter\ p\ (map\ f\ as))$

— Approximation $y = p(f\ a)$

$if\ y\ (f\ a)\ :\ (filter\ p\ (map\ f\ as))$ $\doteq$ $map\ f\ (if\ y\ (a\ :\ (filter\ (p \circ f)\ as))$
$(filter\ p\ (map\ f\ as))$ $(filter\ (p \circ f)\ as))$

Casing y:

1.1 $y = \texttt{Bot}$

$\texttt{Bot}$ $\doteq$ $map\ f\ \texttt{Bot}$

$\texttt{Bot}$ $\doteq$ $\texttt{Bot}$

1.2 $y = true$

$(f\ a) : (filter\ p\ (map\ f\ as))$ $\doteq$ $map\ f\ (a : (filter\ (p \circ f)\ as))$

$(f\ a) : (filter\ p\ (map\ f\ as))$ $\doteq$ $(f\ a) : (map\ f\ (filter\ (p \circ f)\ as))$

— constructor decomposition

$(filter\ p\ (map\ f\ as))$ $\doteq$ $map\ f\ (filter\ (p \circ f)\ as)$

closed with $l = as$

1.3 $y = false$

$(filter\ p\ (map\ f\ as))$ $\doteq$ $map\ f\ (filter\ (p \circ f)\ as)$

— closed with $l = as$

Example 9. We show how the example mentioned in the introduction can be reduced to an induction proof. Assume the following definitions for $length, rev$, and $append$:

$$
\begin{array}{l}
length\ xs = lcase\ xs\ 0\ conslength \\
conslength\ y\ ys = 1 + (length\ ys) \\
rev\ xs = lcase\ xs\ nil\ consrev \\
consrev\ y\ ys = append\ (rev\ ys)\ (y : \texttt{Nil}) \\
append\ xs\ ys = lcase\ xs\ ys\ (consappend\ ys) \\
consappend\ ys\ z\ zs = z : (append\ zs\ ys)
\end{array}
$$

We will start with the initial equation $(length\ (rev\ xs)) \doteq (length\ xs)$, where the set environment is $U = \{lists(xs)\}$. Let us distinguish the two cases that xs is a finite or an infinite list (see Example 2). Thus we can split the task into two different cases $U = \{finlists(ys)\}$ and $U = \{inflists(ys)\}$.

The case of a finite lists can be handed over to a module that is able to perform usual induction ([BM88, Wal94]).

Some experimentation with CPE shows that it seems to be impossible to prove the claim for finite lists using the built-in closure criteria. Hence for finite lists or other finite data structure, there are simple equations that can only be proved with an extension of CPE, where for parts of the tableau, other size measures are permitted, for example the length of the list (see also next section 6.)

Let us consider the case of infinite lists, where $U = \{inflists(xs)\}$.

$$(length\ (rev\ xs)) \quad \doteq \quad (length\ xs)$$

Computing the covering yields that there are the two cases $xs = \texttt{Bot}$ and $xs = y : ys$. The first case can easily be treated. The second yields the following equations, where ys is constrained to be an infinite list.

$$
\begin{array}{ll}
(length\ (rev\ (y : ys))) & \doteq \quad (length\ (y : ys)) \\
(length\ (append\ (rev\ ys)(y : \texttt{Nil}))) & \doteq \quad 1 + (length\ ys)
\end{array}
$$

It is easy to see that after two reduction steps, $(rev\ ys)$ becomes a necessary redex. The same holds for the expression $(length\ ys)$ on the right hand side, hence we can apply the Bot-detection rule, and get a "bot=bot"-leaf.

In summary we get a closed tableau, hence the equation holds.

6 Extensions of CPE

There are different extensions of the calculus CPE, which improve the strength of the calculus. We exhibit two extensions by induction and an improvement of case analysis, but do not give a formal treatment.

6.1 Induction using a size measure

Example 9 shows that there are simple inmduction proofs that cannot be constructed by CPE. The following extension permits to add such size-based induction proofs.

The idea is to mark certain nodes as inductive nodes. The subtableau rooted at this node requires a different treatment than the rest of the tableau. A requirement is that the set environment yields a measure for some free variables in the inductive node. For example $finlists(xs)$ implies that xs can be measured by the length of the list. This measure may be also be more complicated and may also be defined on more than one free variable. In order to apply the measure, the edges that are constructed for "Casing free variables" must remember the replacement for the free variable.

All expansion rules are permitted below the inductive node. There is a new expansion rule, where subexpressions are replaced using the induction hypothesis, which is the equation at the inductive node used as pattern. This "Application of induction hypothesis" is allowed, if some node in the inductive tableau contains somewhere a subexpression that is an instance of the right or left hand side of the induction hypothesis, and furthermore the instance is smaller than the expression at the inductive node w.r.t. the chosen ordering. The subexpression can then be replaced by the corresponding instance of the other side. Formally, this is like term rewriting.

For the closure rules there are now some restrictions:

The "bot=bot"-rule can only be applied, if there is no application of the induction hypothesis between the leaves and the looping node. The reason is that applying the induction hypothesis may conflict with the number of reductions.

The "loop"-rule requires that in case that the induction hypothesis is applied on the path between the leaf and the looping node, the conditions on "C" and "F"-labels holds.

Now let us try again the $length(rev\ xs)$-example:

Example 10. Assume that xs is finite, i.e. $U = \{finlists(xs)\}$, and that the inductive node is $length(rev\ xs) \doteq length\ xs$. Then there are two cases: $xs = \texttt{Nil}$ and $xs = y : ys$ with $finlists(ys)$. The first case reduces to equal expressions. We develop the path for the second case:

$$\begin{array}{lcl} (length\ (rev\ (y : ys))) & \doteq & (length\ (y : ys)) \\ (length\ (append\ (rev\ ys)(y : \texttt{Nil}))) & \doteq & 1 + (length\ ys) \end{array}$$

It is not hard to show using CPE and an extra tableau, that for arbitrary lists as and bs, we have $length(append\ as\ bs) =_\omega length(as) + length(bs)$.

We can continue the tableau:

$$\begin{array}{lcl} (length\ (rev\ ys) + length(y : \texttt{Nil}))) & \doteq & 1 + (length\ ys) \end{array}$$

Using induction:

$$\begin{array}{lcl} length\ ys + 1 & \doteq & 1 + (length\ ys) \end{array}$$

The two cases are: $ys = \texttt{Nil}$ and $ys = z : zs$. We consider the second one:

$$\begin{array}{ll} length\ (z : zs) + 1 & \doteq\ 1 + (length\ (z : zs)) \\ (1 + length\ (zs)) + 1 & \doteq\ 1 + (1 + length\ (zs)) \\ 1 + (length\ (zs) + 1) & \doteq\ 1 + (1 + length\ (zs)) \\ (length\ (zs) + 1) & \doteq\ (1 + length\ (zs)) \end{array}$$

6.2 Improvement of case analysis

The "Approximation" rule in connection with "Casing free variables" rule has sometimes the disadvantage, that on the same path two contradictory cases were selected, but CPE has no possibility to detect this and may fail at this path. If this could be detected, then CPE would have a better chance to close tableaux. The underlying idea of this improvement is to remember the expression and the chosen value for this expression, such that in case we have to make a case analysis for the same expression on the same path, we could use the information.

We demonstrate the usefulness of the ideas by giving an example for a proof that the merge of two finite sorted lists of integers is again sorted.

Example 11. We choose Haskell-like syntax [HPW+92] for the example.

$$\begin{array}{l} sorted\ [] = \texttt{True} \\ sorted\ [y] \ = \texttt{True} \\ sorted\ (y : (z : zs)) = (y \leq z)\ \&\&\ sorted(z : zs) \\ merge\ []\ ys = ys \\ merge\ xs\ [] = xs \\ merge\ (x : xs)\ (y : ys) = if(x \leq y)\ (x : (merge\ xs\ (y : ys)))\ (y : (merge\ (x : \\ xs)\ ys)) \end{array}$$

We assume that the calculus has some knowledge about integers and the $\leq$-relation.

The tableau for proving that the merge of two finite sorted lists of integers is again sorted starts with $sorted\ xs\ \&\&\ sorted\ ys \doteq sorted(merge\ xs\ ys)$. It is interesting to note that the claim is false for infinite lists. Consider $xs = [3, \ldots]$ and $ys = [2, 1, \ldots]$, which yields `Bot` on the left hand side and `False` on the right hand side.

We assume the appropriate set environment that ensures that xs and ys are finite lists of integers.

$sorted\ xs$ && $sorted\ ys$ $\doteq$ $sorted(merge\ xs\ ys)$

1) $xs = \texttt{Nil}$

$sorted\ \texttt{Nil}$ && $sorted\ ys$ $\doteq$ $sorted(merge\ \texttt{Nil}\ ys)$

$sorted\ ys$ $\doteq$ $sorted\ ys$

2) $xs = x : xss$

$sorted\ (x : xss)$ && $sorted\ ys$ $\doteq$ $sorted(merge\ (x : xss)\ ys)$

2.1) $ys = \texttt{Nil}$

$sorted\ (x : xss)$ && $\texttt{True}$ $\doteq$ $sorted(x : xss)$

Can be closed after some steps.

2.2) $ys = y : yss$

$sorted\ (x : xss)$ && $sorted\ (y : yss)$ $\doteq$ $sorted(merge\ (x : xss)\ (y : yss))$

2.2.1) $(x \leq y) = \texttt{True}$

$sorted\ (x : xss)$ && $sorted\ (y : yss)$ $\doteq$ $sorted(x : (merge\ xss\ (y : yss)))$

2.2.1.1) $xss = \texttt{Nil}$

$sorted\ (y : yss)$ $\doteq$ $sorted(x : (y : yss))$

Using the assumption 2.2.1):

$sorted\ (y : yss)$ $\doteq$ $sorted(y : yss)$

2.2.1.2) $xss = x_2 : xsss$

$sorted\ (x : x_2 : xsss)$ && $sorted\ (y : yss)$ $\doteq$ $sorted(x : (merge\ (x_2 : xsss)\ (y : yss)))$

2.2.1.2.1) $(x_2 \leq y) = \texttt{True}$

$sorted\ (x : x_2 : xsss)$ && $sorted\ (y : yss)$ $\doteq$ $sorted(x : x_2 : (merge\ xsss\ (y : yss)))$

After some steps, looping back to 2.2.1)

2.2.1.2.2) $(x_2 \leq y) = \texttt{False}$

$sorted\ (x : x_2 : xsss)$ && $sorted\ (y : yss)$ $\doteq$ $sorted(x : y : (merge\ (x_2 : xsss)\ yss))$

It is possible to deduce $(x \leq x_2) = \texttt{True}$, hence we have a loop.

2.2.2) $(x \leq y) = \texttt{False}$

$sorted\ (x : xss)$ && $sorted\ (y : yss)$ $\doteq$ $sorted(y : (merge\ (x : xss)\ yss))$

Using an analogous scheme as for the case 2.2.1), we can close the tableau.

7 Conclusion

We have described an automatable calculus CPE that is able to prove behavioural (co-inductive) equality of functions in a non-strict functional language. This shows that reasoning about program transformations and equality of functions processing infinite objects like

streams can be implemented, such that the tools can be used either by compilers to improve efficiency and/or safety, or by a programmer to get some feed back on the properties of the written functions.

The calculus can be used also for strictness analysis by abstract reduction. To show that a function f is strict, we simply start with f `Bot` $\doteq$ `Bot`. This method can be seen as an extension of the calculus described in [SSPS95], where we use free variables instead of `Top` constants in the case of more than one argument.

We plan to implement the calculus in the near future. The implementation of a strictness analyser described in [SSPS95] and the manually computed examples given in this paper show that an implementation shall be able to prove non-trivial examples.

References

[Abr90] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[BM88] R.S. Boyer and J.S. Moore. *A computational logic handbook*. Academic Press, London, 1988.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall International, London, 1988.

[Gor94] Andrew D. Gordon. A tutorial on co-induction and functional programming. In *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 78–95. Springer, 1994.

[How89] D. Howe. Equality in lazy computation systems. In *4th IEEE Symp. on Logic in Computer Science*, pages 198–203, 1989.

[HPW+92] Paul Hudak [ed.], Simon L. Peyton Jones [ed.], Philip Wadler [ed.], Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell. A non-strict purely functional language. version 1.2, 1992.

[Mor68] J.H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.

[MT91] R. Milner and M. Tofte. Co-induction in relational semantics. *J. Th. Computer Science*, 87:209–220, 1991.

[Pit94] A.M. Pitts. A co-induction principle for recursively defined domains. *J. Th. Computer Science*, 124:195–219, 1994.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International, London, 1987.

[PJL91] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: a Tutorial*. Prentice-Hall International, London, 1991.

[SSPS95] M. Schmidt-Schauß, S.E. Panitz, and M. Schütz. Strictness analysis by abstract reduction using a tableau calculus. In Alan Mycroft, editor, *Static Analysis Symposium '95*, number 983 in Lecture Notes in Computer Science, pages 348–365. Springer, 1995.

[SSS96] M. Schütz and M. Schmidt-Schauß. Constructing evaluation contexts. Technical Report 1/1996, Fachbereich Informatik, Universität Frankfurt, 1996.

[Wad89] P. Wadler. Theorems for free! In MacQueen, editor, *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. Addison-Wesley, 1989.

[Wal94] Ch. Walther. Mathematical induction. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 127–228. Oxford university press, 1994.